To
Sumithra, Iniya, and Sathyan
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>List of Figures</strong></td>
<td>xiv</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 What does this book teach?</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Topics</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Some of the problems that we will solve (and their real-world applications)</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Fundamental Algorithm Design Techniques</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Background needed</td>
<td>5</td>
</tr>
<tr>
<td>1.6 Worked Exercises and Exercises</td>
<td>5</td>
</tr>
<tr>
<td>1.7 Advanced Material</td>
<td>6</td>
</tr>
<tr>
<td>1.8 References</td>
<td>6</td>
</tr>
<tr>
<td><strong>2 Problem Solving and Algorithms</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Two Warmup Problems</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1 Find the Celebrity</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2 Tiling a Board</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Problem: Prime Number (Primality) Checking</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1 Problem Solving Process</td>
<td>12</td>
</tr>
<tr>
<td>2.2.2 Understanding the Primality Checking Problem</td>
<td>13</td>
</tr>
<tr>
<td>2.2.3 Problem Solving Phase</td>
<td>13</td>
</tr>
<tr>
<td>2.2.4 Pseudocode for Algorithm IsPrime</td>
<td>14</td>
</tr>
<tr>
<td>2.2.5 Correctness of the Algorithm IsPrime</td>
<td>14</td>
</tr>
<tr>
<td>2.2.6 Speeding up the algorithm</td>
<td>14</td>
</tr>
<tr>
<td>2.2.7 FastIsPrime Algorithm</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Problem: Search</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1 n-Queens Problem</td>
<td>16</td>
</tr>
<tr>
<td>2.3.2 The Square Root Problem</td>
<td>20</td>
</tr>
<tr>
<td>2.4 A Model for Computation: Random Access Machine (RAM)</td>
<td>23</td>
</tr>
<tr>
<td>2.5 Asymptotic Notation: Big-O (O) notation, Big-Ω notation, and Big-Θ notation</td>
<td>24</td>
</tr>
</tbody>
</table>
## CONTENTS

2.5.1 Big-O notation ................................................. 25  
2.5.2 Big-Ω notation ............................................. 26  
2.5.3 Big-Theta (Θ) Notation .................................... 27  
2.5.4 Little-o and Little-ω notations ......................... 28  
2.5.5 Some common runtimes of algorithms ................. 29  

2.6 Worked Exercises .............................................. 31  
2.7 Exercises ....................................................... 35  

3 Mathematical Induction and Algorithms .................... 37  
3.1 Celebrity Problem Revisited ................................. 37  
3.2 Tiling Problem Revisited .................................... 38  
3.3 Worked Exercises .............................................. 40  
3.4 Exercises ....................................................... 41  

4 Recursion ........................................................... 43  
4.1 Problem: Greatest common divisor (gcd) ................. 44  
4.1.1 Euclid’s algorithm to compute gcd .................... 44  
4.1.2 Recursive function: Two key features ................. 45  
4.1.3 Termination, Correctness, and Running Time ......... 45  
4.2 Problem: Searching for the maximum element .......... 46  
4.2.1 A non-recursive algorithm ............................... 47  
4.2.2 A recursive algorithm: Divide and conquer strategy 47  
4.2.3 Correctness of Algorithm Maxr ......................... 48  
4.3 Techniques for Solving Recurrences ....................... 51  
4.3.1 Guess and Verify .......................................... 51  
4.3.2 A General Theorem for “Divide and Conquer” Recurrences .................................................. 53  
4.3.3 Handling Floors and Ceilings in Recurrences ...... 56  
4.4 Worked Exercises .............................................. 57  
4.5 Exercises ....................................................... 62  

5 More Divide and Conquer ....................................... 67  
5.1 Problem: Sorting ................................................. 67  
5.1.1 A Simple Sorting Algorithm ......................... 67  
5.1.2 MergeSort: A Divide and Conquer Sorting Algorithm ....... 69  
5.1.3 QuickSort: Another Divide and Conquer Algorithm ....... 74  
5.1.4 Average-Case Analysis of QuickSort .................. 78  
5.2 Problem: Selection .............................................. 80  
5.3 Problem: Multiplying two numbers ........................ 83
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>Problem: Matrix Multiplication</td>
<td>85</td>
</tr>
<tr>
<td>5.4.1</td>
<td>A Divide and Conquer Algorithm</td>
<td>86</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Strassen's Algorithm</td>
<td>87</td>
</tr>
<tr>
<td>5.5</td>
<td>Problem: Fourier Transform</td>
<td>88</td>
</tr>
<tr>
<td>5.6</td>
<td>Worked Exercises</td>
<td>97</td>
</tr>
<tr>
<td>5.7</td>
<td>Exercises</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>Dynamic Programming</td>
<td>105</td>
</tr>
<tr>
<td>6.1</td>
<td>Divide and Conquer vs. DP</td>
<td>105</td>
</tr>
<tr>
<td>6.2</td>
<td>Optimization problems</td>
<td>106</td>
</tr>
<tr>
<td>6.3</td>
<td>Illustrating the DP Technique using the Maxsum Problem</td>
<td>107</td>
</tr>
<tr>
<td>6.4</td>
<td>Problem: Matrix Chain Multiplication</td>
<td>115</td>
</tr>
<tr>
<td>6.4.1</td>
<td>DP solution</td>
<td>117</td>
</tr>
<tr>
<td>6.5</td>
<td>Problem: Aligning two Sequences</td>
<td>121</td>
</tr>
<tr>
<td>6.6</td>
<td>Problem: 0/1 Knapsack</td>
<td>123</td>
</tr>
<tr>
<td>6.7</td>
<td>A Fully Polynomial-Time Approximation Scheme for Knapsack</td>
<td>125</td>
</tr>
<tr>
<td>6.7.1</td>
<td>Approximation Algorithm and FPTAS</td>
<td>126</td>
</tr>
<tr>
<td>6.7.2</td>
<td>Alternate DP algorithm for Knapsack</td>
<td>127</td>
</tr>
<tr>
<td>6.7.3</td>
<td>FPTAS for Knapsack</td>
<td>128</td>
</tr>
<tr>
<td>6.8</td>
<td>DP Summary</td>
<td>130</td>
</tr>
<tr>
<td>6.9</td>
<td>Worked Exercises</td>
<td>131</td>
</tr>
<tr>
<td>6.10</td>
<td>Exercises</td>
<td>134</td>
</tr>
<tr>
<td>7</td>
<td>Greedy Algorithms</td>
<td>139</td>
</tr>
<tr>
<td>7.1</td>
<td>Problem: Fractional Knapsack</td>
<td>140</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Greedy Strategies</td>
<td>140</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Proof of Optimality</td>
<td>141</td>
</tr>
<tr>
<td>7.2</td>
<td>Problem: Caching or Demand Paging</td>
<td>143</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Proof of Optimality of LFD</td>
<td>144</td>
</tr>
<tr>
<td>7.3</td>
<td>Problem: Efficient Data Compression</td>
<td>145</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Huffman Coding Algorithm</td>
<td>147</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Optimality of Huffman Coding</td>
<td>149</td>
</tr>
<tr>
<td>7.4</td>
<td>Problem: Set Cover</td>
<td>153</td>
</tr>
<tr>
<td>7.4.1</td>
<td>A Greedy Algorithm</td>
<td>153</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Performance of Greedy Algorithm</td>
<td>154</td>
</tr>
<tr>
<td>7.5</td>
<td>Worked Exercises</td>
<td>155</td>
</tr>
<tr>
<td>7.6</td>
<td>Exercises</td>
<td>157</td>
</tr>
</tbody>
</table>
8 Randomization and Probabilistic Analysis

8.1 Randomization
8.2 Introducing a Randomized Algorithm
8.3 Verifying Matrix Multiplication
  8.3.1 A Randomized Algorithm
  8.3.2 Bounding the Error Probability
8.4 Randomized Quicksort
  8.4.1 Probabilistic Analysis of Deterministic Quicksort
8.5 Randomized algorithms vs. probabilistic (average-case) analysis
  8.5.1 Randomized Algorithm classification
8.6 Randomized Selection
8.7 High Probability Bounds: Showing Concentration near Expected Value
  8.7.1 Load Balancing: Balls into Bins Paradigm
8.8 High Probability Analysis of Randomized Quicksort
8.9 A Randomized Data Structure: Skip Lists
  8.9.1 Analysis
8.10 Probably Approximately Correct (PAC) Learning
  8.10.1 Learning Framework
  8.10.2 PAC Learning Theorems
  8.10.3 Examples
8.11 Worked Exercises
8.12 Exercises

9 Hashing and Fingerprinting

9.1 Hashing
  9.1.1 Hashing by Chaining
9.2 Universal Hashing
  9.2.1 Constructing universal hash functions
  9.2.2 An Application of Universal Hashing: Perfect Hashing
9.3 Hashing Strings and Fingerprinting
  9.3.1 Problem: Comparing Strings
  9.3.2 Fingerprinting Theorem
  9.3.3 Application to Pattern Matching
9.4 Worked Exercises
9.5 Exercises
CONTENTS

10 Algorithms for Numbers ................................................. 221
  10.1 Problem: Primality Testing ........................................ 221
    10.1.1 An Almost Correct Algorithm: Pseudoprimality Testing .... 222
    10.1.2 A Correct Primality Testing Algorithm ...................... 224
  10.2 Public Key Cryptography ........................................... 225
    10.2.1 RSA Cryptosystem .............................................. 226
    10.2.2 Why is RSA hard to break? .................................. 227
    10.2.3 Randomized RSA .............................................. 227
  10.3 Exercises .......................................................... 228

11 Fundamental Graph Algorithms ....................................... 229
  11.1 Representing a Graph .............................................. 230
    11.1.1 Adjacency List (Ad) Representation ......................... 230
    11.1.2 Adjacency Matrix Representation ............................ 232
  11.2 Depth-first Search (DFS) .......................................... 232
    11.2.1 Depth-first Search (DFS) Algorithm in an Undirected graph ... 232
    11.2.2 A key property of DFS ....................................... 235
    11.2.3 DFS Application: Finding connected components of a graph ... 236
    11.2.4 DFS Application: Finding cut vertices and cut edges of a graph ......................................................... 237
    11.2.5 Depth-first search in a directed graph ....................... 240
    11.2.6 Topological Sort .............................................. 242
  11.3 Breadth-first search (BFS) ........................................ 246
    11.3.1 BFS Algorithm ............................................... 246
  11.4 Worked Exercises .................................................. 249
  11.5 Exercises .......................................................... 251

12 Graph Algorithms: MST and Shortest Paths ....................... 257
  12.1 Minimum Spanning Tree (MST) Problem .......................... 257
    12.1.1 Kruskal's (Greedy) Algorithm ................................ 258
    12.1.2 Prim's Algorithm .............................................. 263
  12.2 Shortest Paths .................................................... 265
    12.2.1 Single Source Shortest Paths Problem ....................... 267
      Dijkstra's Algorithm .............................................. 267
      A* algorithm ..................................................... 270
      Bellman-Ford Algorithm ......................................... 272
    12.2.2 All-Pairs Shortest Paths .................................... 275
      The Floyd-Warshall algorithm ................................... 276
CONTENTS

E.5 Degree of a vertex ........................................... 367
E.6 Connectivity ................................................. 367
E.7 Spanning Tree (ST) ......................................... 368
  E.7.1 The Size of a Spanning Tree .............................. 368

Appendix F Matrices ............................................. 369
  E.1 Basic Facts .................................................. 369

Appendix G Commonly Used Math Formulas in Algorithm Analysis 371
  G.1 Geometric series .......................................... 371
  G.2 Combinatorial Inequalities ................................. 372

Bibliography ......................................................... 372
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Tiling problem (top) and its solution (bottom).</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Backtracking solution to the 4-queens problem. Note that the search proceeds in a &quot;depth-first&quot; manner. When the search is stuck, i.e., it is not possible to extend the solution (i.e., go deeper in the tree), we backtrack, i.e., go one level up and try again. For example, the leftmost branch is stuck and we backtrack to the previous level. It turns out that there is no solution from the leftmost branch. We do find a solution on the second branch from the top node. We stop when there is a solution. One can continue the backtracking till all solutions are found.</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Solution for $n = 6$ (top) and $n = 7$ (bottom).</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>Illustrating the Algorithm MySqrt on $n = 35$.</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>Random Access Machine (RAM) model. The RAM model consists of a computing unit plus a memory that allows random access. The memory is assumed to consists of (infinite number of) cells with each cell having a unique address. Each cell can be accessed in constant time (by specifying its address) regardless of its address in any step.</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>Illustration of the Big-O notation. In both figures, $g(n) = O(f(n))$. Left: $f(n) \geq g(n)$ for all $n \geq n_0$, where $n_0$ is a fixed constant. Note that what happens when $n &lt; n_0$ is irrelevant. Right: $g(n) \geq f(n)$, but $g(n) \leq 5f(n)$, for all $n$.</td>
<td>25</td>
</tr>
<tr>
<td>2.7</td>
<td>Figure for Exercise 2.4 Note that there is no solution for $n = 6$.</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Recursion flow diagram for gcd.</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Recursion flow diagram for Algorithm Maxr.</td>
<td>49</td>
</tr>
<tr>
<td>4.3</td>
<td>Illustration for proof of DC Recurrence Theorem.</td>
<td>54</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

5.1 An example to illustrate Algorithm SimpleSort on input set \( \{6, 5, 2, 3, 1, 4\} \). ........................................... 68

5.2 MergeSort example. The top figure shows the recursion as it unwinds in a top-down fashion. The bottom figure shows how the array gets sorted in a bottom-up fashion. ............. 71

5.3 Illustrating Procedure Merge. \( A \) and \( B \) are the two input sorted arrays and \( X \) is the output array that is overall sorted. 72

5.4 QuickSort recursion tree: worst-case (top) and best-case (bottom). The worst-case tree is very skewed (to the right). This happens when the input is already sorted; thus the pivot is always the smallest element in each recursive call and all elements are to the right of the pivot; hence one subproblem is empty and all the remaining elements are in the other subproblem. The depth of the tree is \( n \). The best-case is when the pivot is the middle element, i.e., the median, in every recursive call when the two subproblems are of almost equal size. The depth of the tree is at most \( \log_2 n \). 77

5.5 Showing the 8th-roots of unity and how the recurrence proceeds on them. The top level represents the 8th-roots of unity, the second level 4th-roots of unity, and the third level the square roots of unity. At each level, the roots are grouped into positive-negative pairs; for each pair only one of them need to be solved in the next level of recursion. From the bottom, each level is obtained by taking successive square roots. At the top level we evaluate the polynomial at the 8 points corresponding to the 8th-roots of unity. This is accomplished by evaluating at the polynomial (recursively) at the 4 points corresponding to the 4th-roots of unity and so on. .......................................................... 94

7.1 Binary tree representation of the prefix code of Example 7.1. 146

7.2 Binary tree representation of the Huffman (prefix) code of Example 7.1. ................................. 148

7.3 Illustration of proof of Lemma 7.1 \( x \) and \( y \) are the two lowest frequency characters. They can be made siblings at the lowest depth without increasing the cost of the code. 150

7.4 Illustration of proof of Lemma 7.2 ................................. 152
8.1 Elements are listed in sorted order. The probability that a randomly chosen pivot lands in the middle one-third of the set is 1/3. .................................................. 168

8.2 A skip list data structure. The path taken to search for element 11 is shown in red. ........................................ 179

9.1 A hash function \( h \) maps elements from a large set \( U \) to a smaller set \( \{0, \ldots, m - 1\} \). ................................. 200

9.2 Two elements are hashed to the same value (2), causing collision. ................................................................. 201

9.3 Hashing with chaining. The elements that hash to the same value (2) are stored in a linked list. ......................... 203

9.4 ASCII code for characters. ........................................ 211

11.1 A graph and its representation. The left shows the adjacency list representation and the right the adjacency matrix representation. ......................................................... 231

11.2 An undirected graph and its DFS tree rooted at vertex \( a \) where the DFS is started. Thick lines show the tree edges and the dotted lines show the back edges. The \( \text{num} \) values of the vertices — which shows the order in which the vertices are visited — are also shown. ............................................... 233

11.3 A graph \( G \) (top) and its DFS tree (bottom). The \( \text{num} \) values of the vertices are shown in black and the \( \text{low} \) values are shown in red. The cut vertices of \( G \) are \( b, d \) and \( g \). Note that each of their \( \text{num} \) values is less than or equal to the \( \text{low} \) values of at least one their respective children. For example, \( d \) is a cut vertex because one of its children, \( e \), has its \( \text{low} \) value greater than or equal to \( d \)'s \( \text{num} \) value. The DFS tree of \( G \) can be used to find all its cut vertices. Note that a cut vertex will not have a back edge that goes from its descendant to its ancestor. For example, there is a back edge from \( d \)'s descendant \( f \), but it does not go to \( d \)'s ancestor, just \( d \). Hence \( d \) is a cut-vertex, since it is the only vertex that connects its descendant \( f \) to its ancestors. ........................................... 238

11.4 Example: A directed graph and its DFS tree. .................. 242

11.5 Example: A DAG. The top shows a topological ordering of the vertices of the DAG so that all edges are directed from left to right. The bottom shows the DFS tree of the DAG. Note that it has no back edges. ............................................ 243
11.6 An undirected graph and its BFS tree. .................................. 246

12.1 A connected weighted graph (left) and its MST (right). Note that MST is a spanning tree and hence will have exactly \( n-1 \) edges on a connected graph of \( n \) nodes. (Top) Kruskal’s algorithm, starting from an empty tree, adds edges in the order of increasing weight, starting from the smallest weighted edge, unless the added edge closes a cycle with edges already added. In this example, Kruskal adds edges in the following order (shown in red): \((a,b),(b,g),(b,d),(a,c),(d,e),(d,f)\). The cut that validates the addition of edge \((a,c)\) is the partition (shown by red line) — \((\{a,b,d,g\},\{c,e,f\})\), because \((a,c)\) is the lightest edge crossing this cut. (Bottom) Prim’s algorithm, starting from a tree with one vertex, adds edges so that the tree is extended by one additional vertex at a time. The edge that connects a non-tree vertex to a vertex in the tree via the least-weighted edge is added. In this example, Prim adds edges in the following order (shown in red) starting from a tree with the singleton vertex \(f\): \((f,d),(b,d),(a,b),(b,g),(a,c),(d,e)\). The cut that validates the addition of edge \((a,c)\) is the partition (shown by red line) — \((\{f,a,b,d,g\},\{c,e\})\) because \((a,c)\) is the lightest edge crossing this cut. .................................. 259

12.2 Illustration of proof of Kruskal’s algorithm. \( e = (u,v) \in T \) crosses the cut \((S,V-S)\). Note that \(S\) is the set of all nodes connected to \(u\) using edges already added in the tree \(T\) till now. If \(e \notin T'\), then the optimal tree \(T'\) should have another edge \(e'\) crossing the cut. This edge \(e'\) should have weight no less than the weight of \(e\), since Kruskal adds edges in increasing order of weight. .................................. 260

12.3 A weighted graph and its shortest path tree (SPT), rooted at source node \(c\). Dijkstra’s algorithm adds vertices in increasing order of distances from the source. In this example, the order (shown in red) in which vertices are added to the SPT is: \(c,a,b,d,g,e,f\). .................................. 267

12.4 Illustration of proof of Dijkstra’s algorithm. \(u\) is the node that has the smallest \(d\) value. Hence it is the one that is extracted from the heap and added to the tree (i.e., made part of \(S\)). The algorithm updates the \(d\) values of only the non-tree neighbors of \(u\) (i.e., part of \(V-S\)). .................................. 271
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.5</td>
<td>Illustration of proof of Bellman-Ford algorithm. (Top) $P = v_0, v_1, ..., v_k$, where $v_0 = s$ and $v_k = u$ is a shortest path from $s$ to $u$. (Bottom) $v_0, ..., v_k$ is a negative weight cycle, i.e., a cycle whose sum of weights is negative.</td>
</tr>
<tr>
<td>12.6</td>
<td>Illustration of proof of Floyd-Warshall algorithm. $P$ is a shortest path from $i$ to $j$ going through vertex $k$. It consists of a path $P_1$ from $i$ to $k$, followed by a path $P_2$ from $k$ to $j$. Since $P$ is a shortest path from $i$ to $j$, $P_1$ is a shortest path from $i$ to $k$ using only intermediate vertices in ${1, ..., k-1}$ and $P_2$ is a shortest path from $k$ to $j$ using only intermediate vertices in ${1, ..., k-1}$.</td>
</tr>
<tr>
<td>12.7</td>
<td>Negative edge graph</td>
</tr>
<tr>
<td>13.1</td>
<td>Figure A shows a flow network. Figure B shows the residual network after routing a flow of 1: $s \to a \to b \to t$. Note how new edges $(a, s)$, $(b, a)$, $(t, b)$ appear in the residual graph. For example, edge $(b, a)$ appears with a capacity of 1, because originally $(b, a)$ had capacity 0 and we push a flow of -1 from $b$ to $a$ (i.e., a flow of 1 from $a$ to $b$); hence the residual capacity of this edge is $0 - (-1) = 1$. Figure C shows the residual network after routing a flow of 1: $s \to b \to a \to t$. Figures D and E show residual graphs after routing a flow of 99 respectively. There is no directed (augmenting) path from $s$ to $t$ in the network of Figure E. Hence no more flow can be pushed. The total flow pushed overall is 200 which is the maximum flow.</td>
</tr>
<tr>
<td>13.2</td>
<td>Level graph of the flow graph of Figure A of Figure 13.1. Note that edge $(a, b)$ is deleted in the level graph.</td>
</tr>
<tr>
<td>13.3</td>
<td>A bipartite graph $G$ (top) and the directed graph $G'$ (bottom) which is a flow graph obtained from $G$. All capacities in $G'$ are 1.</td>
</tr>
<tr>
<td>13.4</td>
<td>Figure shows a graph $G$ (top) and the contracted graph $G'$ (bottom) obtained by contracting edge $(a, d)$ (labelled 4). Note that self-loop (shown in dotted line) is not present in the contracted graph, but multi-edges are retained.</td>
</tr>
<tr>
<td>B.1</td>
<td>A complete binary tree</td>
</tr>
<tr>
<td>B.2</td>
<td>A binary heap</td>
</tr>
<tr>
<td>B.3</td>
<td>Figure shows the indices of an array (from 0 to 9) corresponding to the nodes of a binary heap.</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>B.4</td>
<td>Illustrating Min-Heapify. The algorithm fixes the heap property at a node if it is violated. In this example, there is a violation at node 16 which is fixed.</td>
</tr>
<tr>
<td>B.5</td>
<td>Union-Find data structure as a collection of disjoint rooted trees. (Top) Make-Set creates ( n ) singleton trees, where ( n ) is the total number of elements in the set. (Middle) Two trees representing two sets after some union operations. Find operation finds the root of the tree that the node belongs to by parent pointer chasing to the root. (Bottom) Combing the two trees via union-by-rank heuristic attaches the tree with smaller height to a tree of larger height.</td>
</tr>
<tr>
<td>E.1</td>
<td>A connected undirected graph (left) and the same graph with weights (right).</td>
</tr>
<tr>
<td>E.2</td>
<td>A graph and an induced subgraph of the graph.</td>
</tr>
<tr>
<td>E.3</td>
<td>A disconnected graph.</td>
</tr>
</tbody>
</table>
1

Introduction

This book is for a course on algorithms. It focuses on the fundamentals of algorithms, and their design and mathematical analysis. Many real-world applications give rise to algorithmic problems. In fact, the modern computer revolution was fuelled by algorithms and is continue to be fuelled. Thus we will emphasize on the real-world significance of algorithms.

One can design many algorithms for computing a solution to a given problem. Various important questions arise:

- How do we design a “good” algorithm?
- What do we mean by a “good” algorithm?
- How do we analyze algorithms?
- How do we know that an algorithm is “good/efficient”?
- How to efficiently implement algorithms (say, using suitable data structures) in a high-level programming language such as C, C++, Java, or Python?

Algorithms are everywhere. They occur and play a key role in many areas not limited to computer science alone, but also to other sciences (like biology, economics, physics, chemistry etc.), and engineering (electrical and electronics engineering, mechanical engineering etc.). Listed below are some applications, predominantly from a computer science perspective. However, it is hard to think of an area in science and engineering where algorithms do not play a major role. Thus it is appropriate to say that algorithms (and computer science) have come to play a central and enabling role in other areas of science and engineering as well.

- Scientific Computation, Numerical Analysis.
1. INTRODUCTION

- Computational Biology, Bioinformatics.
- Communication Networks.
- Computer Systems, Operating Systems, Programming Languages, Compilers, Typesetting, Editors...
- Internet/Web Applications.
- Electronic Commerce; Cryptography
- Games.
- Optimization.
- Computer Graphics, Animation.
- Signal processing, Communications Technology.
- Machine Learning and Artificial Intelligence.

1.1 What does this book teach?

This book teaches how:

- To solve various problems that arise in wide range of applications by algorithmic thinking and techniques.
- To learn and apply various algorithmic design techniques.
- To design and analyze efficient algorithms for a variety of problems. The emphasis is on algorithmic thinking to solve problems and not just programming.
- To understand, appreciate, and explore how algorithms are applied in the real world.

1.2 Topics

The book covers the following topics (not necessarily in this order). The idea is to expose students to a variety of algorithmic problems that are related through one or more of the following themes: problem domain, algorithmic technique, and analysis technique.
1.3 SOME OF THE PROBLEMS THAT WE WILL SOLVE (AND THEIR REAL-WORLD APPLICATIONS)

- Algorithm design and algorithmic thinking (especially recursion).
- Mathematics of algorithm analysis.
- Algorithms for numbers.
- Recursion, Divide and Conquer.
- Randomized algorithms and techniques
- Algorithms for searching, sorting, and various other problems.
- Hashing and its applications.
- Data Structures.
- Greedy method.
- Dynamic programming.
- Graph algorithms.
- Matrix algorithms.
- Optimization algorithms — Linear Programming and applications.
- Distributed and parallel algorithms.
- Approximation Algorithms.

1.3 Some of the problems that we will solve (and their real-world applications)

Throughout the course, we will study algorithms for fundamental problems that also play an important role in real-world applications. Some of the problems that we will study are:

- **Prime number checking** (A basic tool in cryptosystems).
- **Searching a large database** (How does UH retrieve your record fast given your ID? How does Google retrieves webpages given a search keyword?).
- **Signal processing** (How to do Fourier transform — which is at the heart of many applications — fast?)
1. INTRODUCTION

- **Sorting** (How to arrange a given set of elements in some desired order very fast?).
- **Searching a graph** — (robot motion planning, graph exploration).
- **Finding the shortest route** in a map (Google maps!).
- **DNA similarity checking**. (How do you check for a DNA match?)
- **Compressing** a large file. (How does UNIX compress work?)
- **Ranking** Web pages. (How does Google rank Webpages?)
- **Learning to classify**. (How does your email filter spam?)
- **Electing a leader in a network.** (How do a set of nodes in a distributed network coordinate?)
- **Distributed/parallel processing of Big Data.** (How to process very large data (e.g., Web graph) that cannot even fit into a single machine?)

This current book draft (part 1) will focus on a subset of the above problems, especially in the context of fundamental algorithmic design techniques (next Section). A follow up version (part 2) will cover more advanced topics.

1.4 **Fundamental Algorithm Design Techniques**

We will learn various algorithm design techniques. These techniques are useful in solving a variety of problems. Mastering these techniques is the key to designing an algorithm for a new problem (that you haven’t seen before).

- **Search techniques** — exhaustive search, backtracking, binary search, hashing/fingerprinting.
- **Divide/Decrease and Conquer** — a **Recursive** technique.
- **Randomization and probabilistic analysis** — randomness in algorithms and analysis of algorithms under a probabilistic input. **Hashing/fingerprinting** is an important randomized technique used in search, cryptography, and other applications.
- **Greedy technique** — used in optimization.
1.5 Background needed

For this course, it is important to have a good background in discrete mathematics, taught typically in first or second year of undergraduate computer science. You will find some background material in the appendix of this book. It is highly recommended that you refresh your background by reviewing a textbook in discrete mathematics (e.g., Rosen’s book [3]). Alternatively, an online reference is the first year computer science undergraduate course on discrete structures at Stanford — see [http://web.stanford.edu/class/cs103x/cs103x-notes.pdf]. The algorithms textbook [1] by Cormen, Leiserson, Rivest and Stein has an appendix that reviews the basic discrete math concepts that we need. This book reviews some basic background in discrete mathematics and data structures in the appendices.

You also must have taken an undergraduate course in algorithms and data structures and should have good programming experience in at least one high-level programming language, e.g., C, C++, Pascal, Java, Python, Lisp, ML etc.

1.6 Worked Exercises and Exercises

Each chapter has numerous exercises which complement the material covered in the chapters. Solving the exercises is a good exposure to solving algorithmic problems. Each chapter also has several “worked exercises”; these are important in further understanding and applying the concepts covered in the respective chapters. Trying to solve the worked exercises before seeing their solutions is a good learning technique. The solutions to the worked exercises can provide insights in solving the exercises.
1.7 Advanced Material

Headings of some chapters and sections of the book are suffixed by stars (*). These are (somewhat) advanced material for undergraduate students and can be skipped during first reading. Exercises and Worked Exercises with stars are problems with above average difficulty for undergraduate students.

1.8 References

- The main reference for the course is this book that is available at [https://sites.google.com/site/gopalpandurangan/home/algorithms-course](https://sites.google.com/site/gopalpandurangan/home/algorithms-course). The book will continue to be updated throughout the semester. Note that not all topics in the book will be covered. The book is not a substitute for all things covered in class; hence it is highly recommended not to miss any class.

- There are lots of books on algorithms that can serve as useful references. These can give you a different perspective or cover topics not covered in this book. Some other references and texts are:
  - Algorithms for the Real World — Lecture notes available from my homepage. URL: [https://sites.google.com/site/gopalpandurangan/home/algorithms-for-the-real-world](https://sites.google.com/site/gopalpandurangan/home/algorithms-for-the-real-world)
  - Cormen, Leiserson, Rivest and Stein: Introduction to Algorithms (Third Edition)
  - Klienberg and Tardos: Algorithm Design
  - Dasgupta, Papadimitriou, and Vazirani: Algorithms
  - Horowitz, Sahni, Rajasekaran: Computer Algorithms
Problem Solving and Algorithms

The major goal of algorithms is to solve problems, i.e., using computer algorithms to solve problems. Problems arise in the real-world in virtually every domain, and many of them are of algorithmic nature. Some examples that we are familiar with are: (1) Given two numbers, how to perform some arithmetic operation involving them? (2) Given a number, how to check if it is prime? (3) Find a prime number that is large (say, having 500 digits) (4) Given two cities on a road map, find the shortest route between them? (5) Given a chess position, what is the “best” move to make? (6) How to search for a word in a text? (7) How to find the webpages that are most relevant to the query that we are interested in? One can list tens of thousands of such problems that occur in every day world (see also the problems listed in Chapter 1).

One may ask, why can’t we solve them manually, i.e., “hand solve” them. The main reasons for not doing that are: (1) It is too cumbersome or difficult, even if the problems involve “small” inputs and (2) it doesn’t usually scale when the problem size becomes large and (3) (perhaps the most important and obvious) it is too slow. A better alternative, in the computer age, is to design algorithms to (correctly) solve them. The main goal of this book is to show how one can correctly and efficiently design algorithms to solve problems. To argue about correctness and efficiency, we need mathematics. Understanding the mathematics behind algorithms is another main goal.

In this chapter, we start with some basic concepts in algorithm design, especially, recursion, divide and conquer, and algorithm analysis.
What is an Algorithm? An algorithm is a sequence of precise instructions that (usually) terminates. “Precise” means that the instructions should be unambiguous and mathematically rigorous. Termination criterion ensures that the algorithm finishes in finite time. Although most algorithms terminate in a finite number of steps, there are useful algorithms that go on forever (can you think of some example?). We will generally deal with algorithms that have a finite running time.

2.1 Two Warmup Problems

Let’s warmup by looking at some algorithmic puzzle-like problems that also illustrate a key algorithmic technique. It also illustrates “efficiency” and analysis of algorithms — two things that we will be part and parcel of every algorithm that we will study. “Efficient” means that we want to solve the problem with as few resources as possible, e.g., as fast as possible which typically translates to using as few “costly” operations as possible. One needs mathematical analysis to show precise bounds on the performance of the algorithm.

2.1.1 Find the Celebrity

Problem 2.1. A celebrity among a group of \( n \) people is a person who knows nobody but is known by everybody else. The task is to identify a celebrity by only asking questions to people of the following form: “Do you know this person?”. Design an “efficient” algorithm to identify a celebrity or determine that a group has no such person. What is the largest number of questions your algorithm needs to solve this problem?

Solution This problem can be solved by reducing the problem size by one — decrease and conquer strategy, a recursive technique that is useful in solving many problems. In the decrease and conquer strategy we reduce the problem size by one (or sometimes by more than one) and then recursively apply the same strategy to the reduced problem till the problem size becomes small (e.g., 1). Let’s illustrate this by solving the celebrity problem.

We start with \( n \) persons. Choose 2 persons, say A and B and ask A whether he/she knows B. If yes, then A cannot be a celebrity and can be eliminated. If no, then B cannot be a celebrity and can be eliminated. This reduces the problem size by one. We repeat the same strategy in among the group of \( n−1 \) persons.
2.1. TWO WARMUP PROBLEMS

After \( n - 1 \) questions, one person (say X) will remain. If there is a celebrity then it should be this person (why?)\footnote{A formal proof by mathematical induction on why this strategy is correct is given in Chapter 3.} To check whether this person is a celebrity we ask X whether he knows everybody else and whether everybody else knows X. Total number of questions = \( n - 1 + n - 1 + n - 1 = 3n - 3 \). (Actually, we can improve this to \( 3n - 4 \). How?)

A even better algorithm can be given! It can be shown that it can be done using \( 3n - \lfloor \log n \rfloor - 3 \) questions and this is the best possible!, i.e., in general, one cannot solve this problem using a lesser number of questions. (Throughout the book, unless otherwise stated, by \( \log \), we will mean logarithm to the base 2.)

Note that the performance or “complexity” in the above problem is naturally determined by the number of questions. The complexity of an algorithm is typically determined by the costliest operation that dominates the overall cost of the algorithm.

2.1.2 Tiling a Board

Problem 2.2. You are given a \( 2^n \times 2^n \) board missing one square. The missing square can be any of the board squares. See figure 2.1. Cover all the squares (except the missing square) by L-shaped tiles exactly with no overlaps.

Solution There are many ways to solve this problem. This can be viewed as a search problem (we will discuss about such problems in Section 2.3 where we want to search for a solution, i.e., a placement of L-shaped tiles to cover the \( 2^n \times 2^n \) grid. Such problems can be solved by using a backtracking strategy, which is essentially an exhaustive strategy. In this strategy, we start placing the L-shaped tile one by one starting from some position and continue till we get stuck (i.e., we cannot cover a particular square); if we get stuck we backtrack to the previously placed tile and continue the search by trying a different placement. Backtracking is discussed in Section 2.3 It is not clear whether backtracking always leads to a valid solution, and if it does, it seems difficult to bound the number of steps needed (a step is a placement of a tile).

Here, we approach this problem by a very different strategy which involves decomposing the problem into subproblems of smaller size. (This is unlike backtracking — which works directly on solving the original problem.) This important algorithmic strategy is called as divide and conquer. In this strategy, the bigger problem is decomposed into one or more
smaller problems of the same kind, i.e., they are exactly the same problem but on a strictly smaller input size. Divide and conquer, is a very powerful technique. It can be naturally implemented using recursion. Recursion is central to algorithmic thinking. In fact, we will emphasize on writing algorithms using recursion as much as possible (unless non-recursive algorithms are much easier to write). Typically, recursive algorithms are conceptually simpler to understand and easier to write.

We will show that the original problem can be decomposed to 4 sub-problems of the same kind (see Figure 2.1). That is each of the sub-problems is exactly of the same type as the original problem — its dimensions are a power of 2 (in fact, reduced by half) and it has one missing square.

It is important to note that in decrease/divide and conquer strategies that the resulting subproblems should be of the same kind, but strictly smaller in size. It is even better if the subproblem sizes go down by a constant factor, in which case the number of “recursive levels” (i.e., the depth of the recursion) will be logarithmic in the problem size.
2.2. PROBLEM: PRIME NUMBER (PRIMALITY) CHECKING

In the tiling problem, to apply the divide and conquer strategy, we need to figure out how to reduce a bigger problem into a smaller problem of the same kind. First, it is always instructive to look at small instances of the problem: this is very useful in understanding the problem and, in many cases, also in designing an algorithm. When $n = 1$, we have a $2 \times 2$ grid with one missing square; it is easy to see that this can be easily solved, no matter where the missing square is, by placing one L-shaped tile. Thus, if we can reduce the problem size (and apply it repeatedly), we have a way to solve the problem.

The larger problem has size $2^n \times 2^n$ and it is reasonable to have the smaller problem(s) to have size(s) $2^{n-1} \times 2^{n-1}$. Note that each of the smaller problem should have a missing square, hence it will be of the same kind. How can we accomplish this reduction? Suppose we put a L-shaped tile in the upper left corner, this will not work. Let’s try to divide the $2^n \times 2^n$ grid into 4 smaller grids each of size $2^{n-1} \times 2^{n-1}$ (see Figure 2.1). To get a missing square, we place a L-shaped tile in the center as shown in Figure 2.1 (bottom). This creates four subproblems each of size $2^{n-1} \times 2^{n-1}$ with one missing square (remove the 3 squares that are covered by the L-shaped tile). If we keep repeating this, we end up in a $2 \times 2$ instance which we can solve easily.

The above strategy demonstrates that not only a solution exists (wherever the missing square may be), but it also gives a way to find it using a divide and conquer strategy which is a naturally recursive strategy. Note that this is a top-down strategy — reducing the larger problem to a smaller one and finding a solution (placement of the tiles) in the process.

2.2 Problem: Prime Number (Primality) Checking

Let’s now look at a classical real-world problem, namely, primality checking, whose complete solution was given relatively recently (in the last decade), though the problem itself has been around for a long time.

**Problem 2.3** (Primality checking). Given a natural number, say, $n$, is it a prime number or not?

This is a problem that requires just a boolean answer: yes or no. It is central to number theory and also to algorithms. Later in the course we will give an efficient (randomized) algorithm for this problem, but first let’s look at some obvious algorithms. But before that, let’s look at how we can approach in solving a problem. This is generic and applies to any problem.
2.2.1 Problem Solving Process

Let’s explicitly list the major steps in solving an algorithmic problem. Usually we solve problems by going through these steps implicitly or sometimes even omitting them which, in many cases, leads to erroneous results.

- **Understand** the problem — what is the input and output for the problem? A very useful tip in understanding a problem is to see what the output is for a small-sized input. Usually, for many problems, this can be easily computed by hand and helps immensely understanding the problem and also in coming up with a solution for it. For example, in the celebrity problem, check what happens when \( n \) is small, say 2 or 3. This also helps in making sure that your solution works for trivial (degenerate) cases, e.g., \( n = 1 \).

- **Problem solving** phase — try to come up with an algorithm.
  - Many times, this involves understanding how to solve small instances of the problem. This usually suggests an algorithm, although that may not be the most efficient.
  - Try to see if one of the algorithmic design techniques help. For example, if you want to apply Divide and Conquer — break the problems into smaller subproblems and combine their solutions.
  - Express the algorithm in pseudocode format. More importantly, it is mandatory to write a description of your algorithm in English. In fact, if the description is precise and unambiguous, a pseudocode may not even be necessary. However, a pseudocode is always recommended since it is precise and makes the transition to a computer program easy (see last step).

- **Analysis** phase — Analyze the correctness and the performance of the algorithm.
  - This usually involves exploiting some mathematics behind the problem.
  - The first thing to do is to argue that the algorithm is correct, i.e., it produces the correct output for every possible input\(^2\). Generally speaking, correctness trumps performance, since there is no

\(^2\)Interestingly, there is a class of algorithms called randomized algorithms, where this need not be true always, but instead has to be correct with “high probability”.
point in having an algorithm that is extremely efficient but outputs wrong answers. Second, is to analyze the running time of the algorithm — this is called time complexity analysis. Sometimes, we also analyze the memory used by the algorithm — this is called as space complexity analysis.

- **Implementation phase** — Convert the pseudocode into a computer program.
  - This involves converting the pseudocode in some programming language, e.g. C++, Java, Python, Lisp etc.
  - We won’t address this aspect in this book, since our focus is on algorithm design and analysis techniques for solving problems, rather than their implementation.

Do not short-circuit the process by omitting the problem solving phase — a typical mistake of beginners!

### 2.2.2 Understanding the Primality Checking Problem

It is easy to understand the input and output for this problem.

- **Input**: A natural number \( n \).
- **Output**: Yes (if \( n \) is prime) or No (if \( n \) is composite).

### 2.2.3 Problem Solving Phase

How to come up with an algorithm?

For a start, let’s look at the mathematical definition of a prime number.

**Definition 2.2.1** (Prime number). A natural number (that is greater than 1) is prime if it is divisible only by itself and 1.

The definition suggests a method (algorithm) to check for primality:

- Given a number \( n \), check whether any number between 2 and \( n - 1 \) (both inclusive) divides \( n \) or not.
- If any of these divides \( n \), then output “No”.
- If none of them divides \( n \), then output “Yes”.

2.2.4 Pseudocode for Algorithm IsPrime

The pseudocode is given in Algorithm 1. It is quite straightforward: The for loop exhaustively checks whether each of the number between 2 and \( n - 1 \) (both included) divides \( n \). If any of them divides \( n \), we say that the number is “composite”. If none of them divides \( n \), then the number is “prime”.

**Algorithm 1** Algorithm IsPrime\((n)\). Input: A natural number \( n \). Output: Yes (if \( n \) is prime) or No (if \( n \) is composite).

1: for \( i := 2 \) to \( n - 1 \) do
2: if \( i \) divides \( n \) then
3: Output “No”
4: Halt
5: Output “Yes”

2.2.5 Correctness of the Algorithm IsPrime

Why is the Algorithm IsPrime correct?

The correctness follows because it correctly implements the mathematical definition of a prime number. The for loop checks for divisibility of \( n \) by every number between 2 and \( n - 1 \).

2.2.6 Speeding up the algorithm

Algorithm IsPrime is rather slow, especially if the input number \( n \) is prime. In this case, the for loop runs \( n - 2 \) times. Thus the running time of the algorithm is approximately proportional to \( n \), the magnitude of the input number.

Can we speed up IsPrime? For this, we use the following simple fact.

**Fact 2.1.** If \( i \) is a divisor of a number \( n \), then \( n/i \) is also a divisor of \( n \).

We need to check for divisors only till \( \max_{2 \leq i \leq n-1} (\min\{i, n/i\}) \). The maximum (among all \( i \), \( i \) taking values from 2 to \( n - 1 \)) is attained when \( i = n/i \), i.e., \( i^2 = n \) or \( i = \sqrt{n} \). Hence it is enough to check for divisors up to \( \sqrt{n} \).

2.2.7 FastIsPrime Algorithm

The pseudocode for FastIsPrime is given in Algorithm 2. The only change from IsPrime is that the for loops runs till \( \lfloor \sqrt{n} \rfloor \). The run time of
Algorithm 2 Algorithm FastIsPrime($n$). Input: An natural number $n$. Output: Yes (if $n$ is prime) or No (if $n$ is composite).

1: for $i := 2$ to $\lfloor \sqrt{n} \rfloor$ do
2:  if $i$ divides $n$ then
3:    Output “No”
4:  Halt
5: Output “Yes”

FastIsPrime is proportional (approximately) to $\sqrt{n}$ which is significantly smaller than $n$.

However, note that FastIsPrime is still quite slow as its running time is an exponential function in the size of the input (See Section 2.5.5). An important aspect of efficient algorithms is that their run time scales polynomially in the input size. Even with regards to polynomial, we ideally would like to keep the degree of the polynomial as small as possible, e.g., linear time algorithms are great, since their run time is proportional to the size of the input. Such algorithms are very helpful in handling “Big Data” applications, where the input sizes are very very large.

Before we go into the analyzing the performance of algorithms, we need a model for computation, which we discuss in Section 2.4

### 2.3 Problem: Search

Search is a fundamental theme that appears often when solving problems of various kinds. A problem can be cast as a search problem if it involves searching (finding) element(s) which satisfies some desired property (or properties) from an appropriate set $S$. The set $S$ can be called as the search space.

Many problems are indeed are (or can be cast as) search problems, e.g., finding the median in a given set of numbers, finding a particular word (if it exists) in a given text, finding non-trivial factors of a given number (if they exist), finding the “best” move in a given chess or Go position, finding the shortest route between two points in a map etc.

**Search Method** A straightforward method is “brute-force” or “exhaustive” search:

- Examine each element in the set $S$ and see if it satisfies the desired property or properties.
If it does, then output the element(s) as solution.

In terms of problem solving strategy (for solving puzzles etc.), exhaustive search tries all possible candidate solutions until a (correct) solution to a problem is found. Exhaustive search is costly (can take too much time, e.g., exponential in the input size) if the search space is large — which is typically the case. In such cases, more efficient search algorithms are needed.

2.3.1 \( n \)-Queens Problem

**Problem:** Place \( n \) queens on an \( n \times n \) chessboard so that no two queens attack each other by being in the same column, row, or diagonal.

An exhaustive search is to try all possible candidate solutions, i.e., all possible positions. For the \( n \)-Queens problem, the number of possible positions for the \( n \) queens is very large: \( \binom{n^2}{n} \geq n^n \), which grows exponentially in \( n \). A very good algorithmic idea that can drastically reduce the number of positions to be searched is called **backtracking**. The backtracking method can be summarized as follows:

- Construct candidate solutions incrementally, i.e., in a “step-by-step” fashion.
- Evaluate a partially constructed solution and see if it can be extended by one more “step” without violating the problem’s constraints.
- If no extension is possible from this partial solution, then backtrack to replace the previous step with another step. Repeat.

Let’s apply the backtracking method to the \( n \)-Queens problem.

\( n \)-Queens problem: Backtracking solution

Backtracking process can be represented as a **tree**. See Figure 2.2. We build the solution one column at a time. Initially the “root” node of the tree starts with the empty solution (no queens). The “depth” of the node will represent the number of queens placed (till now). A node’s **children** will represent all possible one-column extensions to consider: there will be \( n \) children per node — representing the \( n \) possible row positions corresponding to that particular column.

*How fast is backtracking compared to exhaustive search?* Exhaustive search examines \( \binom{n^2}{n} \) positions. On the other hand, backtracking examines only those positions where there is only one queen in each column.
2.3. PROBLEM: SEARCH

Figure 2.2: Backtracking solution to the 4-queens problem. Note that the search proceeds in a “depth-first” manner. When the search is stuck, i.e., it is not possible to extend the solution (i.e., go deeper in the tree), we backtrack, i.e., go one level up and try again. For example, the leftmost branch is stuck and we backtrack to the previous level. It turns out that there is no solution from the left most branch. We do find a solution on the second branch from the top node. We stop when there is a solution. One can continue the backtracking till all solutions are found.
and row: \( n! \) positions. Why is it \( n! \)? This is because, the first queen can be placed in any one of the \( n \) rows in the first column; the second queen can be placed in any one of the \( n - 1 \) rows in the second column (except the one that is attacked by the first queen); the third queen can be placed in any one of the \( n - 2 \) rows in the third column and so on. Hence the total number of possible positions that backtracking examines is (at most) 
\[
 n \times (n - 1) \times (n - 2) \times (n - 3) \ldots \times 2 \times 1 = n!.
\]
Actually, backtracking is even better, since it also cuts out diagonal squares attacked by previously placed queens; we didn’t take into account these, which will reduce the number of positions even further (it is difficult to compute this).

We note that \( n! \) is significantly smaller than \( \binom{n^2}{4} \), but \( n! \) is still exponential in \( n \) (why?). For \( n = 4 \), this is: \[
\binom{16}{4} = \frac{16!}{4!(16-4)!} = \frac{16 \times 15 \times 14 \times 13}{4 \times 3 \times 2} = 1820,
\]
whereas, \( 4! = 24 \).

**How fast can you solve n-queens?** Both exhaustive and backtracking search can take exponential time, i.e., exponential in \( n \). This is too slow. Even if \( n = 100 \), this can be beyond the scope of a typical laptop computer. It turns out that one can find a solution to the \( n \)-queens problem much faster — in linear time, i.e., in time propotional to \( n \). How can this be done? This is done analytically, i.e., without performing any search. Here we only give a partial solution, leaving the full solution as an exercise. The partial solution works only for certain types of \( n \) values.

The idea is to generalize the solution position that we had for \( n = 4 \) — see Figure 2.3. (Note that there are no solutions for \( n = 2 \) and \( n = 3 \). Since there is no queen in the diagonal, we can easily obtain a solution for \( n = 5 \), by placing the 5th queen on the last square of the 5th column (in the diagonal). For \( n = 6 \), we can construct the following solution, using the idea from \( n = 4 \): place the first three queens in the second, fourth and sixth rows of the first 3 columns respectively and then place the remaining three queens in the first, third, and fifth rows of the last three columns respectively (see Figure 2.3). (This, again can be easily extended for \( n = 7 \), by placing the queen on the diagonal as shown in Figure 2.3 (bottom).) However, note that this idea will not work for \( n = 8 \). However, you can check that it will work for all values of \( n \) that are of the form \( 6t \) (\( t = 1, 2, \ldots \)) as well as \( 6t+4 \) (\( t = 0, 1, 2, \ldots \)) and by extension also for values of \( n = 6t+1 \) and \( n = 6t + 5 \). Thus it works for all values of \( n \) that leaves a remainder \( 0, 1, 4 \) and \( 5 \) when divided by \( 6 \). For other values of \( n \) (i.e., those that leave a remainder of \( 2 \) and \( 3 \) when divided by \( 6 \)), we need to modify the above solution a bit. We leave this as an exercise (Exercise 2.9).
Figure 2.3: Solution for $n = 6$ (top) and $n = 7$ (bottom).
It is thus interesting to note that we can always find a solution to the $n$-queens problem, and that too, pretty fast. But to find all possible solutions to the $n$-queens problem is a very difficult problem (takes a long time, at least exponential in $n$). Currently, even the world’s fastest machines struggle to find all solutions to the 1000-queens problem.

### 2.3.2 The Square Root Problem

We next consider a number-theoretic problem, like primality checking, which turns out to be simpler, namely that of finding the square root of a given number. More precisely, the Square Root problem is, given a natural number $n$, find the largest integer that is less than or equal to $\sqrt{n}$. Recall that we needed to solve the above problem in Algorithm FastIsPrime.

**Finding the Square Root: A search Problem** This can be cast as a search problem where the search space $S$ is the set $\{1, \ldots, n\}$, and the number desired is $\lfloor \sqrt{n} \rfloor$, i.e., the largest integer that is less than or equal to $\sqrt{n}$.

**A naive (i.e., brute-force) algorithm** A naive algorithm (Algorithm 3) for the Square Root problem is as follows. Simply, search all numbers from 1 to $n$, one by one and check if they satisfy the desired property.

**Algorithm 3** Algorithm NaiveMySqrt($n$). **Input:** An natural number $n$. **Output:** $\lfloor \sqrt{n} \rfloor$.

1: for $i := 1$ to $n$ do
2: if $(i \times i \leq n)$ and $((i + 1) \times (i + 1) > n)$ then
3: Output $i$ // this is the answer

This algorithm is slow: it takes about $\sqrt{n}$ “runs” of the for loop, hence $O(\sqrt{n})$ time, since the number of operations per loop is constant (two multiplications and two comparisons).

**A Significantly Faster Algorithm: Binary Search** The main algorithmic idea behind the faster algorithm (Algorithm 4) is the technique of binary (or bisection) search, a fundamental technique that is useful in many algorithmic applications (arguably, the most used algorithmic technique in computer science!). Binary search reduces the search space by approximately half in each iteration. That is, if the search space is originally

---

^3Note that this algorithm is an iterative algorithm. However, it is easy to modify to make it as a recursive algorithm — see Exercises.
of size $k$, then in the first iteration it becomes at most $k/2$, and in the
next iteration it becomes at most $k/4 = k/2^2$, and then in the next it be-
comes $k/8 = k/2^3$ and so on; hence, by $\log_2 k$ iterations, the size is at most
$k/2^{\log_2 k} = 1$. Thus after $\log_2 k$ iterations, the search space has reduced to
at most 1, hence we would have found the answer by the end of $\log_2 k$ it-
erations. Thus the convergence to the desired answer happens in at most
$\log k$ number of steps, where $k$ is the size of the search space. Note that
$\log k$ is the logarithm of $k$ to base 2, i.e., $\log_2 k$.

In the square root problem, thus the algorithm finishes in time at most
$\log n$, since the initial search space is of size at most $n$. Figure 2.4 illustrates
the algorithm on input 35. The algorithm maintains two values $\text{low}$ and
$\text{high}$ which is initialized to 1 and $n$ respectively. The algorithm checks if
the mid point value, i.e., $\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor$ is the desired answer,
by checking whether $(\text{mid} \times \text{mid} \leq n)$ and $((\text{mid} + 1) \times (\text{mid} + 1) > n)$. If mid is not the answer, then the desired answer is either less than $\text{mid}$
or greater than $\text{mid}$; depending on whichever the situation is, the low
or high value are set to mid accordingly. It is important to note that the
following loop invariant, is maintained in every iteration of the while
loop: the desired answer, i.e., $\lfloor \sqrt{n} \rfloor$ is always between $\text{low}$ and $\text{high}$ at
the beginning of the iteration. The algorithm ends when $\text{mid}$ value is
the desired answer. Note that the loop invariant guarantees correctness
at the end: since the search keeps getting reduced (roughly by half), the
desired value which is always between $\text{low}$ and $\text{high}$ will eventually be
output. How to formally prove that the loop invariant is maintained? This
is done by mathematical induction which we will see in Chapter 3. Here
let’s present a quick argument that previews such a proof.

**Lemma 2.1.** In algorithm 4, the following loop invariant holds at the be-
ginning of every iteration of the while loop: $\text{low} \leq \lfloor \sqrt{n} \rfloor \leq \text{high}$.

**Proof.** The proof is via mathematical induction on the number of iter-
tions. We first show that the invariant is true at the beginning of the first
iteration — this is the base case. This is obviously true, since $\text{low} = 1$ and
$\text{high} = n$ at the beginning.

We assume that the invariant is true for some iteration $i$ (this is the
induction hypothesis). Under this assumption, we show that it remains
true for iteration $i + 1$ (assuming, of course, the algorithm has not halted
in iteration $i$). This is again easy to establish. If the algorithm didn’t
terminate in iteration $i$, then there are two cases:

1. Case 1: $(\text{mid} \times \text{mid} \leq n)$. In this case, clearly $\lfloor \sqrt{n} \rfloor$ should be larger
   than $\text{mid}$. Hence $\text{low}$ is set to $\text{mid}$. By induction hypothesis, $\lfloor \sqrt{n} \rfloor \leq$
Figure 2.4: Illustrating the Algorithm MySqrt on $n = 35$.

$high$ and hence the invariant is true at the end of the iteration as well.

2. Case 2: $(mid \times mid > n)$. In this case, clearly $\lfloor \sqrt{n} \rfloor$ should be smaller than $mid$. Hence $high$ is set to $mid$. By induction hypothesis, $\lfloor \sqrt{n} \rfloor \geq low$ and hence the invariant is true at the end of the iteration as well.

Hence the invariant is true at the end of iteration $i$, i.e., at the beginning of iteration $i + 1$. $\square$

Algorithm 4 Algorithm $\text{MySqrt}(n)$. **Input**: An natural number $n$. **Output**: $\lfloor \sqrt{n} \rfloor$.

1: $low = 1$ // Low value of guess
2: $high = n$ // High value of guess
3: **while** true **do**
4: \hspace{1cm} $mid = \lfloor (low + high)/2 \rfloor$ //current estimate for value of square root
5: \hspace{1cm} **if** $(mid \times mid \leq n)$ and $((mid + 1) \times (mid + 1) > n)$ **then**
6: \hspace{2cm} Output $mid$ and STOP // this is the answer
7: \hspace{1cm} **else if** $(mid \times mid \leq n)$ **then**
8: \hspace{2cm} $\lfloor \sqrt{n} \rfloor$
9: \hspace{2cm} $low = mid$
10: **else**
11: \hspace{2cm} $\lfloor \sqrt{n} \rfloor$
12: \hspace{2cm} $high = mid$
2.4. A MODEL FOR COMPUTATION: RANDOM ACCESS MACHINE (RAM)

We describe a simple computation model used to analyze running time of algorithms. This model captures computation that takes place in a single processor machine, where instructions are executed one after the other in a sequential manner. (Many modern computers are actually parallel machines where more than one processor operates in parallel for which we need to modify the RAM model.)

The Random Access Machine model, popularly called as the RAM model (see Figure 2.5), has the following key features:

1. A computing unit (processor) plus a memory.

2. Instructions are executed sequentially (i.e., one after the other, according to the order of the program).

3. Accessing (reading/writing) any one memory cell takes one step (one time unit). This is why it is called “random” access (although there
is no randomness). It merely means that regardless of the location, any memory cell (read piece of data stored in memory) can be accessed (read/write) in one time step.

4. Any “reasonable” one operation (e.g., arithmetic operations, comparison etc.) takes one step. It is important to clarify what a reasonable operation means. For example, adding/multiplying/subtracting/dividing two numbers consisting of small number of bits (e.g., 32-bit or 64-bit) is reasonable; this is usually determined by the “word size” in the memory. Word size is the number of bits that a memory cell can store; in many machines it can be 32 bits. Operations involving two numbers, each of word size (i.e., 32 bits) can be reasonably assumed to take constant time. However, assuming the same for \( n \)-bit numbers is not — this will take about \( n \) steps. Thus, one has to be careful in accounting for the cost of the underlying primitive operation. Note that in many algorithms, this abstraction of using unit cost for one primitive operation simplifies analysis tremendously, without which it is hard to quantify the performance of the algorithm in a clean way. This is indeed a powerful abstraction!

We note that in many cases in analyzing algorithms we even further simplify the assumptions of the RAM model. Typically, we end up only bounding the number of “costly” operations that the algorithm performs. Usually bounding the number of costly operations will bound the total running time as well (e.g., up to a small factor). For example, in analyzing sorting algorithms (Section 5), we typically only focus on the total number of comparisons made by the algorithm, while ignoring other operations. This is usually a good metric to measure the performance of comparison-based sorting algorithms (more on this in Chapter 5).

2.5 Asymptotic Notation: Big-O (\( O \)) notation, Big-\( \Omega \) notation, and Big-\( \Theta \) notation

Algorithmic analysis will not be as elegant as it is if not due to the fact that we emphasise on asymptotic and approximate analysis, rather than exact analysis for a given input size. In other words, we mainly care about how the algorithm’s performance scales with the problem size in a reasonably approximate manner that is approximate up to a constant factor (i.e., a constant that is independent of the problem size). This allows the algorithm analyzer to focus on quantifying the dominant function that deter-
2.5. ASYMPTOTIC NOTATION: BIG-O (O) NOTATION, BIG-Ω NOTATION, AND BIG-Θ NOTATION

mines the algorithm’s performance, while avoiding cumbersome details. We will see examples of this philosophy throughout the course. As an example, we will say that IsPrime takes $O(n)$ time (assuming that all arithmetic operations involving $\log(n)$-bit numbers take constant time) and FastIsPrime takes $O(\sqrt{n})$ time. We will see later that a better way of expressing these runtimes is with respect to the size of the input, which is $\log n$ bits (not $n$).

Consider two positive functions $g(n)$ and $f(n)$ on natural numbers. Think of $n$ as the input size of the algorithm; and $f$ (and $g$) as denoting the runtime (of an algorithm) as a function of the input size.

2.5.1 Big-O notation

A function $g(n)$ is “Big-O” of $f(n)$, i.e., $g(n) = O(f(n))$, if there is some positive constant $c > 0$ (independent of $n$), such that $cf(n)$ grows asymptotically (i.e., for large $n$) at least as “fast” as $g(n)$. (See Figure 2.6.) In other words, $g(n) \leq cf(n)$, for large $n$, i.e., for all $n \geq n_0$ ($n_0$ is some fixed number).

Informally, we will say that if $g(n) = O(f(n))$, then $f(n)$ (possibly, multiplied with a large-enough constant factor) “dominates” or “upper bounds” $g(n)$. Formally we have the following definition.

Figure 2.6: Illustration of the Big-O notation. In both figures, $g(n) = O(f(n))$. Left: $f(n) \geq g(n)$ for all $n \geq n_0$, where $n_0$ is a fixed constant. Note that what happens when $n < n_0$ is irrelevant. Right: $g(n) \geq f(n)$, but $g(n) \leq 5f(n)$, for all $n$. 

A function $g(n)$ is “Big-O” of $f(n)$, i.e., $g(n) = O(f(n))$, if there is some positive constant $c > 0$ (independent of $n$), such that $cf(n)$ grows asymptotically (i.e., for large $n$) at least as “fast” as $g(n)$. (See Figure 2.6.) In other words, $g(n) \leq cf(n)$, for large $n$, i.e., for all $n \geq n_0$ ($n_0$ is some fixed number).

Informally, we will say that if $g(n) = O(f(n))$, then $f(n)$ (possibly, multiplied with a large-enough constant factor) “dominates” or “upper bounds” $g(n)$. Formally we have the following definition.

A function $g(n)$ is “Big-O” of $f(n)$, i.e., $g(n) = O(f(n))$, if there is some positive constant $c > 0$ (independent of $n$), such that $cf(n)$ grows asymptotically (i.e., for large $n$) at least as “fast” as $g(n)$. (See Figure 2.6.) In other words, $g(n) \leq cf(n)$, for large $n$, i.e., for all $n \geq n_0$ ($n_0$ is some fixed number).

Informally, we will say that if $g(n) = O(f(n))$, then $f(n)$ (possibly, multiplied with a large-enough constant factor) “dominates” or “upper bounds” $g(n)$. Formally we have the following definition.
Definition 2.5.1. Big-O notation Given two (positive) functions $f(n)$ and $g(n)$, we say $g(n) = O(f(n))$ if there is a positive constant $c$ (independent of $n$), such that for any $n \geq n_0$, $g(n) \leq cf(n)$.

The above definition is usually cumbersome to use for showing how two functions are related. An alternate, but easier way, is to define Big-O using limits as follows:

Definition 2.5.2. Given two (positive) functions $f(n)$ and $g(n)$, we say $g(n) = O(f(n))$ if

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} \leq c,$$

where $c > 0$ is a fixed constant (assuming the limit exists). In other words, the limit is bounded above by a fixed constant.

The above definition assumes that the limit exists.

2.5.2 Big-Ω notation

Big-Ω notation can be considered a complement of the Big-O notation. If $f(n) = O(g(n))$ then we can say that $g(n) = \Omega(f(n))$. Informally, we will say that if $g(n) = \Omega(f(n))$, then $f(n)$ (possibly, multiplied by a small-enough constant factor) is “dominated by” or “lower bounds” $g(n)$.

Formally, we can define Big-Ω as we did for Big-O.

Definition 2.5.3. A (positive) function $g(n)$ is “Big-Ω” of (another positive) $f(n)$, i.e., $g(n) = \Omega(f(n))$, if there is a positive constant $c > 0$ (independent of $n$), such that for any $n \geq n_0$, $g(n) \geq cf(n)$.

Alternatively, using limits we have the definition:

Definition 2.5.4. A (positive) function $g(n)$ is “Big-Ω” of (another positive) $f(n)$ if

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} \geq c,$$

where $c > 0$ is a fixed positive constant (assuming the limit exists). In other words, the limit is bounded below by a fixed constant.

The above definition assumes that the limit exists (otherwise we can modify the definition to take the limit inf).

4Otherwise we can modify the definition to take the limit sup, e.g., see Example 2.1. However, in most of the functions that we will encounter, limits exist and we don’t have to worry about this technicality.
2.5.3 Big-Theta (Θ) Notation

Big-Theta notation is used to compare functions that are both Big-O and Big-Ω of each other, i.e., in other words have the same asymptotic behavior. More precisely, two positive functions \( g(n) \) and \( f(n) \) are said to be “Big-Theta” of each other if their growth is asymptotically within a constant factor of each other. We can formally define it as follows.

**Definition 2.5.5.** A (positive) function \( g(n) \) is “Big-Theta” of (another positive) \( f(n) \) if there are some fixed positive constants \( c_1 \) and \( c_2 \) such that,
\[
c_1 f(n) \leq g(n) \leq c_2 f(n), \quad \text{for all } n > n_0.
\]
Equivalently, \( g(n) = \Theta(f(n)) \) if \( g(n) = O(f(n)) \) AND \( g(n) = \Omega(f(n)) \).

Informally, we will say that if \( g(n) = \Theta(f(n)) \), then both \( f(n) \) and \( g(n) \) are of the “same order”, i.e., they are essentially the same function (as far as asymptotic growth is concerned).

Alternatively, using limits we have the definition:

**Definition 2.5.6.** A (positive) function \( g(n) \) is “Big-Theta” of (another positive) \( f(n) \) if
\[
\lim_{n \to \infty} \frac{g(n)}{f(n)} = c.
\]
where \( c > 0 \) is a fixed positive constant (assuming the limit exists).

We next illustrate with some examples. More examples can be found in Section “Worked Exercises”.

**Example 2.1.**
1. \( 2n = O(n) = \Theta(n) \). Constants are ignored! Big-O, Big-Omega, and Big-Theta notation “hides” constant factors.
2. \( n^3 + 2^{1000}n^2 + 2^{100000}n = O(n^3) = \Theta(n^3) \).
3. In general for any polynomial
\[
p(n) = \sum_{i=0}^{d} a_i n^i
\]
where \( a_i \) are constants and \( a_d > 0 \), we have \( p(n) = \Theta(n^d) \).
4. \( n = O(n^2) \), but \( n \neq \Theta(n^2) \). That is, \( n^2 \) grows “strictly” faster than \( n \).
5. \( \lg n = O(n) \) (where \( \lg n = \log_2 n \)). There are several ways to show this, but let’s use the limit definition: \( \lim_{n \to \infty} \frac{\lg n}{n} = \frac{\infty}{\infty} \), an indeterminate form, by allowing both the numerator and denominator to go to
∞. To get a finite limit, we use the L’Hopital’s rule (see Worked Exercises) by taking derivatives of the numerator and denominator and then applying the limit:

\[
\lim_{n \to \infty} \frac{\lg n}{n} = \lim_{n \to \infty} \frac{\left(\lg n\right)'}{\left(n\right)'} = \lim_{n \to \infty} \left(1/n\right) = 0.
\]

6. \(n^{1000} = O(2^n)\), but \(n^{1000} \neq \Theta(2^n)\).

7. IsPrime algorithm runs in \(O(n)\) time, where \(n\) is the (magnitude of the) input number.

8. 

\[
f(n) = \begin{cases} 
  n^2 & : n \text{ is even} \\
  n^3 & : n \text{ is odd}
\end{cases}
\]

\(g(n) = n^3\)

Note that \(\lim_{n \to \infty} f(n)/g(n)\) does not exist (i.e., it does not converge — it fluctuates between 0 and 1 depending whether \(n\) is even or odd). However, the limit is upper bounded by 1, this is all that we need to show that \(f(n) = O(g(n))\). Formally, this “limit upper bound” is captured by \(\lim \sup\):

\[
\lim \sup_{n \to \infty} \frac{f(n)}{g(n)} \leq 1.
\]

2.5.4 Little-o and Little-\(\omega\) notations

Sometimes, we would like asymptotic notation to capture a function that grows strictly slower than some other function. For this we use the Little-o notation which can be considered as a stronger form of the Big-O notation. If \(f(n)\) is \(o(g(n))\) then \(f(n) = O(g(n))\), but not vice-versa.

**Definition 2.5.7.** Given two functions, \(g(n)\) and \(f(n)\), we say that \(g(n) = o(f(n))\) if for any positive constant \(c\) there is a constant \(n_c\) such that for any \(n > n_c\), \(g(n) < cf(n)\).

The above definition says that if \(g(n)\) is \(o(f(n))\), i.e., \(g(n)\) grows strictly slower than \(f(n)\) or \(f(n)\) grows strictly faster than \(g(n)\), then no matter what constant \(c\) you choose (however small it may be), when \(n\) becomes large enough (i.e., \(n > n_c\), where \(n_c\) is a constant that depends on \(c\), \(g(n)\) is dominated by \(cf(n)\). In other words, \(cf(n)\) always starts dominating \(g(n)\), however small \(c\) is, after a certain point (when \(n\) becomes large).

Alternatively, using limits we have the definition:
2.5. ASYMPTOTIC NOTATION: BIG-O (O) NOTATION, BIG-Ω NOTATION, AND BIG-Θ NOTATION

**Definition 2.5.8.** A (positive) function \( g(n) \) is “Little-o” of (another positive) \( f(n) \) if
\[
\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0.
\]

Similarly, we would like asymptotic notation to capture a function that grows **strictly faster** than some other function. This is the Little-ω notation which can be considered as a complement of the Little-o notation. That is if \( f(n) = o(g(n)) \), then \( g(n) = \omega(f(n)) \).

**Definition 2.5.9.** Given two functions, \( g(n) \) and \( f(n) \), we say that \( g(n) = \omega(f(n)) \) if for any positive constant \( c \) there is a constant \( n_c \) such that for any \( n > n_c \), \( g(n) > cf(n) \).

Alternatively, the above can be expressed using limits as:

**Definition 2.5.10.** A (positive) function \( g(n) \) is “Little-ω” of (another positive) \( f(n) \) if
\[
\lim_{n \to \infty} \frac{g(n)}{f(n)} = \infty.
\]
(assuming the limit exists)

The above definitions assumes that the limit exists (otherwise we can modify the definitions to take the \( \lim sup \) or \( \lim inf \) for Little-o and Little-ω respectively).

**Example 2.2.**
- \( 1/n = o(1) \)
- \( n^{1000} = o(1.1^n) \)
- \( n^2 = \omega(n) \)
- \( n = (\text{?})n^{1+\sin n} \)
  \[
  \limsup_{n \to \infty} \frac{n}{n^{1+\sin n}} = \infty
  \]
  \[
  \liminf_{n \to \infty} \frac{n}{n^{1+\sin n}} = 0
  \]
  Hence the two functions are not asymptotically comparable via little-o or little-ω notation.

2.5.5 Some common runtimes of algorithms

We use the following terminology for some common run times of algorithms.
• $O(n)$ — Linear time.

• $O(n^2)$ — Quadratic time.

• $O(n^c)$ (for some constant $c > 0$) — Polynomial time.

• $O(2^n)$ or $O(c^n)$ (for some constant $c > 0$) — Exponential time.

• $O(\log n)$ — Logarithmic time.

Polynomial time algorithms are considered “efficient” — typically preferred compared to exponential time algorithms. This although has to be carefully interpreted, especially in practice. For example, a $n^{10}$ algorithm may not be preferable over a $(1.1)^n$ algorithm (although the latter is exponential), for even reasonably large-size values of $n$. However, a phenomenon that has been seen quite often in the theory of algorithms is that, once polynomial time solvability is established, the degree of the polynomial can be reduced, leading to more efficient algorithms. Hence breaching the exponential barrier has important consequences both in theory and in practice. In particular, the famous “P versus NP” problem (considered as one of the most important open problems in mathematics and computer science\footnote{Actually, the “P versus NP” problem is listed as one of the 7 millennium problems of the Clay Institute that carries a million dollar prize.}) is very much related to this. For many important algorithmic problems in the real-world the best known algorithms take exponential time and no polynomial time algorithms are known. However, verifying the solution to these algorithmic problems can be done in polynomial time; on the other hand computing a solution takes exponential time (by the best known methods). The “P versus NP” problem asks whether for such problems polynomial time algorithms exist for computing a solution, not just for verifying.

Even among polynomial time algorithms, we prefer ones with the smallest exponent $c$. For many “big data” problems with large input size, linear time algorithms are preferred.

Another key point that one should note about asymptotic notation is that it “hides” constants. It is quite possible that an $O(n)$ algorithm can be very impractical since the constant factor masked by the big-O notation can be very large. Such an algorithm might be worse compared to even to an $O(n^2)$ algorithm with a small hidden constant factor. These issues are important when implementing an algorithm in practice.
2.6. WORKED EXERCISES

How do you classify IsPrime? Is it a polynomial time algorithm? The input size consists of \( \log n \) bits. The run time is \( O(n) \). Hence, IsPrime is an exponential time algorithm.

How do you classify FastIsPrime? Note that FastIsPrime algorithm runs in \( O(\sqrt{n}) \) time (assuming \( \sqrt{n} \) can be calculated in constant time!) Hence FastIsPrime runs in \( O(2^{(\log n)/2}) = O(2^{\log n}) \), i.e., it also runs in exponential time.

2.6 Worked Exercises

Worked Exercise 2.1. Rank the following functions by order of growth; that is, find an arrangement \( g_1, g_2, ..., g_8 \) of the functions satisfying \( g_1 = O(g_2), g_2 = O(g_3), ..., g_7 = O(g_8) \). Justify your ordering.

\[
(\log n)^3, n^{\log \log n}, n^{1+1/\log n}, 4^{\log n}, n!, 2^{\sqrt{2 \log n}}, n^2, (1.1)^n
\]

Note: By \( \log n \), we mean \( \log_2 n \), unless otherwise stated. Note that \( \log^3 n \) is the usual way of writing \( (\log n)^3 \).

Solution:
The ordering (in increasing order of growth) is:

\( (\log n)^3, 2^{\sqrt{2 \log n}}, n^{1+1/\log n}, 4^{\log n} = n^{\log 4} = n^2, n^{\log \log n}, (1.1)^n, n! \).

We use two helpful ideas in comparing functions:

1. When comparing functions of the form \( f^a \) it might be helpful to compare the logarithm of those functions, i.e., \( \log f^a = g \log f \).

\[ n! = n(n-1) \cdots (2)(1) \text{ which you can use to lower or upper bound } n! \]

, e.g., \( 2^n < n! < n^n \), for all \( n \geq 4 \).

2. Use the limit definition of the Big-O notation. You might want to recall the L'Hopital's rule for computing the limit of the ratio of two functions. The L'Hopital's rule states the following: If \( \lim_{x \to a} f(x)/g(x) \) evaluates to an indeterminate form such as \( 0/0 \) or \( \infty/\infty \), then \( \lim_{x \to a} f(x)/g(x) = \lim_{x \to a} f'(x)/g'(x) \). Here \( a \) can be any number and \( f' \) and \( g' \) are the derivatives of \( f \) and \( g \) respectively.

To compare \( (\log n)^3 \) and \( 2^{\sqrt{2 \log n}} \), compare the log of both the functions — the first is \( \log(\log n)^3 = 3 \log \log n \) and the log \( 2^{\sqrt{2 \log n}} = \sqrt{2 \log n} \). The latter grows faster: you can see this by replacing \( \log n \) by \( x \): then the first will be \( 3 \log x \) and the second will be \( \sqrt{2x} \) — the latter grows faster, since a polynomial function grows faster than a log function.
Let \( f(n) = \frac{n^{1+1}}{\log n} \). Taking log on both sides we have, \( \log f(n) = (1 + 1/\log n) \log n = \log n + 1 \). Comparing with \( \log 2^{\sqrt{\log n}} = \sqrt{\log n} \), we see \( \log n \) grows faster than \( \sqrt{\log n} \). Also note that \( f(n) = 2^{\log n+1} = 2n \). Clearly \( n^2 \) grows faster than \( 2n \). To compare \( n^2 \) and \( n^{\log \log n} \), clearly the latter is growing faster, as the exponent is growing with \( n \), as opposed to being fixed (as in the case of \( n^2 \)).

Similarly, by comparing the logarithms of \( n^{\log \log n} \) and \( (1.1)^n \), we see that the latter is faster.

Finally, \( n! > (1.1)^n \), for large \( n \), because of the following: \( n! = n(n-1)(n-2) \cdots 1 \geq 3 \times 3 \times 3 \cdots 3 \times 2 \times 1 \geq 3^{n-2} = 3^n/9 \). It is easy to see that \( 3^n/9 > (1.1)^n \) for all \( n \geq 10 \) (say).

**Worked Exercise 2.2.** You are given a sorted array \( S[1, \ldots, n] \) of \( n \) (distinct) numbers, i.e., the numbers in the array are arranged in increasing order: \( S[1] < S[2] < \cdots < S[n] \).

Give a recursive algorithm to search for a given element (say \( x \)) in the array. If \( x \) is present in the array, the algorithm should return the index of the position where it is present, otherwise it should return 0.

Your algorithm should take \( O(\log n) \) comparisons. Give pseudocode of your algorithm and argue that your algorithm indeed takes \( O(\log n) \) comparisons (in the worst case).

**Solution:**

The algorithm is very similar to the binary search algorithm for the Square root problem. But care has to be taken to avoid “off-by-one” type errors (in setting the termination condition and the low and high values.).

**Algorithm 5** Algorithm Binary-Search\((S, x)\). **Input:** Sorted array \( S[1, \ldots, n] \) and an element \( x \). **Output:** If \( x \) is in \( S \), then \( i \) such that \( S[i] = x \) or output “null”.

\[
\begin{align*}
1: & \quad \text{low} = 1 \quad /\text{Low value of guess} \\
2: & \quad \text{high} = n \quad /\text{High value of guess} \\
3: & \quad \text{while low} \leq \text{high} \quad \text{do} \\
4: & \quad \text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor \\
5: & \quad \text{if } S[\text{mid}] = x \quad \text{then} \\
6: & \quad \quad \text{Output mid and STOP} \quad /\text{this is the answer} \\
7: & \quad \text{else if } S[\text{mid}] < x \quad \text{then} \\
8: & \quad \quad \text{low} = \text{mid} + 1 \\
9: & \quad \text{else} \\
10: & \quad \quad \text{high} = \text{mid} - 1 \\
11: & \quad \text{Output “null”}
\end{align*}
\]

The analysis is again very similar to that of Square root problem. The
search space decreases by at least a factor of 2, hence the number of iterations of the while loop is at most $\log n$. The correctness follows by mathematical induction — refer to the argument given in the book for the Square root problem.

**Worked Exercise 2.3.** Show that the celebrity problem can be solved by using $3n - \log n - 3$ questions (assume that $n$ is a power of 2). (Hint: Group people into pairs — similar to a knockout tournament, e.g., Wimbledon tennis tournament.)

**Solution:**

As in the hint, first group the $n$ people in pairs of two — i.e., we will have $n/2$ pairs. This will be the first "round". For each pair, say $(x, y)$, ask one question — if $x$ knows $y$. This question will eliminate one person — i.e., he/she can’t be a celebrity. The winner from this pair will be one who is not eliminated. All the winners from this round — there will be $n/2$ of them — will advance to the next round. In the next round, again repeat the process — group into $n/4$ pairs and question — $n/4$ will advance to the next round. This is exactly how a knockout tournament is organized. Finally, there will be one champion (the winner in the final round) — after $\log n$ rounds; because each round eliminates half the number of persons. How many questions were asked before the champion is found? This can be computed by summing up the questions asked in each round:

$$n/2 + n/4 + n/8 + \cdots + 1 = n(1/2 + 1/2^2 + 1/2^3 + \cdots + 1/2^{\log n})$$

The series inside the parantheses is a geometric sum (see Appendix G): Hence that can be evaluated as $1/2 + 1/2^2 + 1/2^3 + \cdots + 1/2^{\log n} = (1/2)(1-1/2^{\log n}) = 1 - 1/n$. Hence,

$$n/2 + n/4 + n/8 + \cdots + 1 = n(1/2 + 1/2^2 + 1/2^3 + \cdots + 1/2^{\log n}) = n(1-1/n) = n - 1.$$

If there is a celebrity he/she has to be the champion. This has to be checked by asking whether all others know the champion and whether the champion knows anybody else. This takes no more than $2n - 2$ questions. However, note that we don’t have to ask the same question (in one way) with the person who lost to the winner. This saves $\log n$ questions. Hence, the total number of questions is $n - 1 + 2n - 2 - \log n = 3n - \log n - 3$.

**Worked Exercise 2.4 (**). There are $n$ coins placed in a row (see Figure 2.7 for an example). The goal is to form $n/2$ pairs of them by a sequence of moves. On each move a single coin can jump left or right over two coins adjacent to it (either over two single coins or one previously formed pair)
to land on the next single coin — no triples are allowed. Note that the coin cannot land on an empty space (it has to land on exactly one coin). Empty space between adjacent coins is ignored.

Find out all values of \( n \) for which the puzzle has a solution and devise an algorithm (enough to explain your algorithm in English) that solves the puzzle in the minimum number of moves for such \( n \). Explain why your solution works in the minimum number of moves.

(Hint: Clearly, there is no solution when \( n \) is odd. Find the smallest \( n \) for which there is a solution. For larger \( n \) use a decrease and conquer strategy).

![Figure 2.7: Figure for Exercise 2.4. Note that there is no solution for \( n = 6 \).](image)

**Solution:**

Clearly, there is no solution for odd \( n \).

For even \( n \), let's first see if there is a solution for small values (this is always a good idea!).

It is easy to see that \( n = 2, 4, 6 \) have no solution.

For \( n = 8 \), there is a solution. Here it is:

Let the 8 coins be arranged as follows: 1 2 3 4 5 6 7 8.

Then the moves are: \( 5 \rightarrow 2 \) (i.e., \( 5 \) on top of \( 2 \)), \( 3 \rightarrow 7 \), \( 4 \rightarrow 1 \), \( 6 \rightarrow 8 \).

This solution can be found either by exhaustive search or by backtracking.

Clearly this is minimal, since 4 moves are needed.

For a general even number, say \( n = 8 + 2k \), we reduce the problem by “decrease-and-conquer” strategy, in particular “decrease-by-2”.

Assume the \( n \) coins are arranged in increasing order from 1 to \( n \) (as above).

The first move is: \( 4 \rightarrow 1 \). This reduces the problem to the same problem with size \( n - 2 \) which we solve again recursively.

The solution clearly uses the minimum (i.e., \( n/2 \)) number of moves because each move pairs up two coins (also follows easily by induction).
2.7 Exercises

Exercise 2.1. Does the following piece of pseudocode terminate? Justify your answer. The input to the pseudocode is a natural number n.

Algorithm foo(n)

1: i = 1
2: while i ≤ n do
3: if i = 5 then
4: i = 0
5: i = i + 1

Exercise 2.2. The algorithm Algorithm MySqrt (Algorithm 4) is iterative. Modify it to make it a recursive algorithm.

Exercise 2.3. Rank the following functions by order of growth; that is, find an arrangement g₁, g₂, g₃, ... of the functions satisfying g₁ = O(g₂), g₂ = O(g₃), ......

n², n/ log n, n log n, (1.001)n, 1/n², log₁₀₀₀ n, 1/ log n, 4log n, n!, n log n, n₁.₆, 2√log n

Note: log or log means logarithm to the base 2. Note that log₁₀₀₀ n is the usual way of writing (log n)₁₀₀₀

Exercise 2.4. Rank the following functions by order of growth; that is, find an arrangement g₁, g₂, ..., g₈ of the functions satisfying g₁ = O(g₂), g₂ = O(g₃), ..., g₇ = O(g₈). Justify your ordering.

n log log n, n, √n⁻¹ log n, e⁻log n, log(n!), 3(2 log n)⁻¹/₃, (log n)!, log₁₀₀ n

Note: By log n, we mean log₂ n, unless otherwise stated. Note that log₁₀₀ n is the usual way of writing (log n)¹₀₀.

Exercise 2.5. (a) Show that given any polynomial p(n) = \sum_{i=0}^{d} a_in^i = a₀ + a₁n + ... + a_dn^d

where aᵢ are constants and a_d > 0, we have p(n) = Θ(nᵈ).

(b) Show that n¹₀₀₀₀ = O(2ⁿ), but n¹₀₀₀₀ ≠ Θ(2ⁿ).

(Hint: See the hint for Exercise 2.4.)

Exercise 2.6. Prove or disprove each of the following statements:

i. f(n) = Θ(g(n)) implies n^f(n) = Θ(n^g(n))

ii. f(n) = Ω(f(n/2))
Exercise 2.7. Compute the number of L-shaped tiles needed to cover a $2^n \times 2^n$ board. Show that it is $(4^n - 1)/3$. Obtain the same answer by setting up a recurrence for $T(n)$, which is the number of tiles needed to cover a $2^n \times 2^n$ board, and solving for $T(n)$. (Hint: See next chapter on solving recurrences.)

Exercise 2.8. There are $n$ adults in town A and they all have to go to town B. There is only a motorbike available which is owned by two boys. The motorbike can carry only one adult or two boys at a time. Using the motorbike all the adults have to reach town B from town A. Show how this can be accomplished while, at the end, leaving the motorbike with the two boys in town A. How many trips are needed? (Hint: Use decrease and conquer to reduce the problem size.)

Exercise 2.9. Complete the solution (proposed in Section 2.3.1) to the $n$-queens problem by giving a solution for values of $n$ that leave a reminder of 2 or 3 when divided by 6. (Hint: Solve for $n = 8$ and try to generalize.)
Mathematical Induction and Algorithms

Before we delve deeply into designing and analyzing algorithms, we focus on a fundamental mathematical concept that is perhaps the single most useful mathematical tool that we will in use in this book to analyze algorithms, and in many cases, will also help in understanding the design of algorithms. That concept is mathematical induction. It is actually a powerful proof technique that is used to prove many mathematical theorems. However, as you will see throughout this book, we will use mathematical induction many times to prove correctness of algorithms as well as sometimes to analyze the performance of algorithms (e.g., show bound on the running time of algorithms). Mathematical induction is also useful in designing algorithms, and although this will not be explicitly clear in some cases, this will be apparent once you see the correctness proof in these cases and will lead you to appreciate how designing algorithms and coming up with their correctness proofs are intimately connected. If you are not familiar with mathematical induction, see Appendix A for a brief introduction.

3.1 Celebrity Problem Revisited

Let’s recall the celebrity problem (Section 2.1.1) and the algorithm that was proposed there for its solution. In particular, we will focus on the decrease and conquer strategy.

We start with \( n \) persons. Choose any 2 persons, say A and B and ask A whether he/she knows B. If yes, then A cannot be a celebrity and can be eliminated. If no, then B cannot be a celebrity and can be eliminated. This
reduces the problem size by one. We repeat the same strategy in among the group of \(n - 1\) persons. After \(n - 1\) questions, we will be left with one person (the potential celebrity), and if there is a celebrity, it has to be this person.

Although it is intuitively clear that this strategy is correct, we will now formally prove the correctness of the algorithm using mathematical induction.

**Theorem 3.1.** The above strategy is correct when applied to a group of \(n\) persons.

**Proof.** As usual, the proof consists of showing the following two steps.

**Base step:** \(n = 1\). Clearly, this is vacuously true, since there is only one person.

**Induction step:** Assume that the strategy is correct for \(k\) persons (this is the induction hypothesis). We will show that the strategy is correct for \(k + 1\) persons. Applying the above strategy for \(k + 1\) persons, we eliminate one person. It is clear that the eliminated person cannot be a celebrity. Thus, if there is a celebrity, then it has to be among the remaining \(k\) persons. By the induction hypothesis the strategy is correct for \(k\) persons, i.e., it will identify the potential celebrity.

Hence the strategy is correct for \(k + 1\) persons. \(\square\)

Note that the above proof is quite easy, but it crucially formalizes the correctness of the decrease and conquer strategy: showing the base case and showing the correctness of the decrease-and-conquer step is enough to show the correctness of the whole algorithm.

### 3.2 Tiling Problem Revisited

Let’s recall the tiling problem and its divide and conquer strategy (Section 2.1.2). On lines very similar to the celebrity problem, it is easy to show the correctness of the strategy using mathematical induction. We leave this as an exercise.

What we will show next is a bound on the number of L-shaped tiles needed to cover a \(2^n \times 2^n\) grid. How many such tiles are needed? Since we know that we can tile all the squares (except the missing square) using L-shaped tiles (each of which covers 3 squares), the number needed is \(\frac{(2^n)(2^n) - 1}{3} = 4^n - 1\). Let’s prove this fact, using an alternative technique,
3.2. TILING PROBLEM REVISITED

namely using *recurrences* which is a powerful and natural way to analyze divide and conquer (more generally, recursive) algorithms (this is discussed in more detail in the next chapter).

We first define a function that captures the number of tiles needed. This function, obviously, depends on $n$, the size parameter. Let $T(n)$ denote the number of tiles needed to cover a $2^n \times 2^n$ grid (with one missing square). By the divide and conquer strategy, we reduced the problem to four subproblems each of size $2^{n-1} \times 2^{n-1}$ by adding one L-shaped tile. Hence we can write a equation for $T(n)$ in terms of $T(n - 1)$ — this is called a recurrence:

$$T(n) = 4T(n - 1) + 1.$$

Note that $T(n - 1)$ denotes the number of tiles needed to cover a $2^{n-1} \times 2^{n-1}$ grid (with one missing square). The above recurrence says that the number of tiles needed to cover a $2^n \times 2^n$ grid is 4 times the number of tiles needed to cover a $2^{n-1} \times 2^{n-1}$ grid (because of the four subproblems of this size) plus one (because of the one tile placed in the center). The recurrence is similar to the induction step. Just like the base step of induction, recurrences need a base case; otherwise, they are not defined. We have $T(1) = 1$ which says that for a $2 \times 2$ grid we need only one L-shaped tile to cover it. Note how closely the recurrence mirrors the induction argument.

How to solve the above recurrence? One way to solve is to “unwind” the recurrence, i.e., write $T(n - 1)$ in terms of $T(n - 2)$ and write $T(n - 2)$ in terms of $T(n - 3)$ and so on and solve the sum directly (the next chapter has some examples of this technique). Another way is to use induction to solve the recurrence. For this we need to “guess” the solution first and then use induction to verify its correctness. We know that $4^n - 1$ tiles are required, just by a divisibility argument. We next show that this is indeed the case.

**Theorem 3.2.** The solution to the recurrence $T(n) = 4T(n - 1) + 1$, with $T(1) = 1$ is $T(n) = 4^n - 1$.

**Proof.** We will prove by induction on $n$.

Base step: $n = 1$. $T(1) = (4 - 1)/3 = 1$ which is true.

Induction step: Assume that the statement is true for $k$. We will prove that it holds for $k + 1$.

\[4^n - 1 = 4(4^{k+1} - 1) + 1 = 4^{k+1} - 1.\]
By the recurrence and the induction hypothesis,

\[ T(k + 1) = 4T(k) + 1 = 4\left(\frac{4^k - 1}{3}\right) + 1 = \frac{4^{k+1} - 4 + 3}{3} = \frac{4^{k+1} - 1}{3} \]

which is exactly what we need to show.

□

3.3 Worked Exercises

**Worked Exercise 3.1.** This problem is a variant of Worked Exercise 2.2.

You are given an array \( S[1, \ldots, n] \) of \( n \) (distinct) numbers, where the numbers in the array are arranged in **increasing-increasing** order, i.e., \( S[i] < S[i + 1] < \cdots < S[n] < S[1] < S[2] < \cdots S[i - 1] \), for some unknown index \( i \).

Give a recursive algorithm to search for a given element (say \( x \)) in the array \( S \). If \( x \) is present in the array, the algorithm should return the index of the position where it is present, otherwise it should return 0.

Your algorithm should take \( O(\log n) \) comparisons. Give pseudocode of your algorithm, argue its correctness using mathematical induction, and show that your algorithm takes \( O(\log n) \) comparisons (in the worst case).

**Solution:**

The algorithm is similar to the binary search algorithm but with modifications to account for the fact that the array is in increasing-increasing order which makes it a bit tricky. Another way to look at this order is that it is circularly sorted, i.e., take a sorted array and shift the elements in a circular fashion. The pseudocode is given in Figure 6. There are a couple of different approaches to search this order: one is to find the split, i.e., the maximum (or minimum) element in the array and search both the parts separately by the usual binary search, i.e., if \( S[i] \) is the maximum element in the array, then search \( S[1] < S[2] < \cdots < S[i] \) (first part) and \( S[i + 1] < S[i + 2] < \cdots < S[n] \) (second part) separately since they are both sorted in the usual sense. This is conceptually simpler, but requires first finding where the split is, i.e., the index of the maximum element. Another approach is to search directly without finding the split.

The correctness proof and analysis is very similar to the Binary-Search problem (Worked Exercise 2.2), except that there are 2 calls to Binary-Search each of which takes \( O(\log n) \) comparisons. The number of comparisons needed to find the max is \( O(\log n) \) as it also is like Binary-Search. Hence overall, \( O(\log n) \) comparisons.
3.4 Exercises

Exercise 3.1. Consider Worked exercise 2.2. Prove the correctness of your algorithm by using mathematical induction.

Exercise 3.2. Consider the problem of sorting n numbers (see Section 5.7 for a formal definition): Given array S of n (distinct) numbers (in some arbitrary order), we want to arrange them in increasing order, i.e., at the end, we want to rearrange S such that S[1] < S[2] < S[3] < \ldots < S[n]. The only operation that we will allow to distinguish numbers is comparison, i.e., given two distinct array elements S[i] and S[j], we can ask whether S[i] < S[j] or not. The goal is to design a sorting algorithm using as few comparisons as possible. Consider the following sorting algorithm known
as insertion sort, which is described inductively. If \( n = 1 \) then the array is trivially sorted. Now consider an array of size \( n > 1 \). We sort the first \( n - 1 \) elements of the array and then insert the \( S[n] \) element in the array in the appropriate place in the sorted order of the first \( n - 1 \) elements, by comparing \( S[n] \) with the rest of the elements. Prove the correctness of this algorithm by using mathematical induction and analyze the number of comparisons needed in the worst case. Show how to use binary search to improve over the number of comparisons.
Recursion is a fundamental paradigm in algorithm design. Recursion embodies a basic algorithmic strategy for solving problems. We saw several examples (such as the Celebrity problem) in the last chapter. Consider a problem of certain size. If the size is small, say a small constant (e.g., one), then (typically) it is easily solvable directly. Otherwise, if the problem size is large, then the strategy is to break the problem into smaller-sized problems of the same kind. How to solve the smaller-sized problems? Again break it into further problems of even smaller size and so on. When this strategy is repeatedly applied, eventually the sizes of the problems will become small and then the problems can be solved directly. This can be naturally implemented as a recursive strategy, i.e., solving a problem by decomposing it into one or more subproblems each of smaller size and then recursively solving the subproblems. Recursion ensures that we only have to worry about the decomposition part and the part where the solutions of the subproblems need to be combined to get the solution to the original problem. Recursion “automatically” takes care of solving the subproblems. This is indeed a powerful strategy that works for many problems as we will see in this chapter and subsequent chapters, especially in implementing divide and conquer (Chapter 5) and dynamic programming (Chapter 6) techniques.

A recursive algorithm solves a problem by solving one or more smaller instances of the same problem. A recursive function is one that invokes itself. A recursive algorithm or function will always have a termination condition, otherwise the function will keep calling itself forever.

Recursion can be considered as a natural “algorithmic counterpart” to mathematical induction in the sense that their correctness and analysis can (usually) be shown straightforwardly by induction. Furthermore,
recursive algorithms are naturally the preferred way to implement divide (or decrease) and conquer algorithmic strategy. Recursive algorithms are generally compact, elegant, and easier to show correctness, compared to non-recursive (i.e., iterative) algorithms. Generally, all in all, it is a good idea to think first on a recursive strategy to solve a problem.

In principle, any algorithm can be expressed using recursive functions — in many cases, the recursive version is much simpler to write. In particular, divide/decrease and conquer algorithms can be naturally implemented using recursion. We will next look at several problems that illustrate recursion and divide and conquer technique.

4.1 Problem: Greatest common divisor (gcd)

Consider the gcd computation problem which is central to many algorithmic applications involving numbers.

**Problem 4.1.** Given two integers $a$ and $b$ find the greatest common divisor (gcd) of $a$ and $b$, denoted by $\text{gcd}(a, b)$.

We will design a recursive algorithm for finding a gcd of two given numbers. We use the following mathematical fact.

**Fact 4.1.** The gcd of two integers $a$ and $b$ with $a > b$ is equal to the gcd of $b$ and $a \mod b$ (the remainder when $a$ is divided by $b$). In other words, $\text{gcd}(a, b) = \text{gcd}(b, a \mod b)$.

**Proof.** Assuming $a > b$, we can write:

$$a = b \cdot d + (a \mod b)$$

where $d$ is an integer (the quotient). Hence any number $x$ that divides both $a$ and $b$ should divide $a \mod b$ as well.

4.1.1 Euclid's algorithm to compute gcd

Fact 4.1 suggests an algorithm to calculate gcd recursively which is attributed to Euclid, dating back over 2000 years and is one of the the oldest algorithms known!

---

Iterative algorithms, unlike recursive algorithms, typically have “loops” that do the non-trivial computation, e.g., *for* and *while* loops. In recursive algorithms, there will be typically no loops, but only recursive calls (and if statements).
Algorithm 7 Algorithm $gcd(a, b)$. Input: Non-negative integers $a$ and $b$, with $a > b$. Output: $gcd(a, b)$.

1: if $b = 0$ then
   // Termination condition
2:   Output $a$
3: Output $gcd(b, a \mod b)$ // Recursive call

Figure 4.1: Recursion flow diagram for gcd.

4.1.2 Recursive function: Two key features

1. Termination condition: A recursive algorithm should have one or more termination condition(s).

2. Recursive call: It should call itself with a strictly smaller value of its arguments.

The recursion flow diagram Consider invoking Algorithm $gcd$ on inputs $a = 8$ and $b = 5$. The sequence of recursive calls can be represented as a recursion flow diagram which, in this case, will be just a path (Figure 4.1).

4.1.3 Termination, Correctness, and Running Time

Note that without a termination condition, the recursive algorithm may go on forever. The termination condition is sometimes called as a “base
(or basis) case” whose correctness is easy to check. In Algorithm gcd, it is clear that if \(b = 0\), then \(a\) is the gcd (trivially). The algorithm’s correctness follows from repeated application of Fact 4.1 and by the termination condition. Formally, this can be shown by a straightforward induction proof, which we leave as an exercise for the reader.

How long does Algorithm gcd take? Running time is determined by the number of recursive calls. Given inputs \(a, b\) (assume \(a \geq b > 0\)), it can be shown that the number of recursive calls is at most \(1 + 2\lceil \log_2(b) \rceil\). Thus the algorithm’s running time is \(O(\log b)\) (the logarithm of the smaller of the two numbers). That is the running time is essentially proportional to the number of bits of \(b\).

**Theorem 4.1.** The number of recursive calls generated by Algorithm gcd when invoked on inputs \(a\) and \(b\) \((a \geq b)\) is at most \(1 + 2\lceil \log_2(b) \rceil\), i.e., \(O(\log b)\).

**Proof:** The first call is \(gcd(a, b)\). Henceforth, we show that the second argument in the call decreases by at least a factor of 2 in at most two recursive calls.

The subsequent recursive call is \(gcd(b, a \mod b)\). Consider 2 cases:

- **Case 1:** \(a \mod b \leq b/2\). In this case, the second argument decreases by a factor of 2 by assumption.

- **Case 2:** \(a \mod b > b/2\). The next recursive call will be \(gcd(a \mod b, b \mod (a \mod b))\). The second argument \(b \mod (a \mod b) < b/2\), since \(a \mod b > b/2\). Thus in two calls the second argument has reduced by at least a factor of 2.

Thus, after at most every two calls we have the following changes to the second argument:

\[ b \rightarrow (\leq b/2) \rightarrow (\leq b/4) \rightarrow (\leq b/8) \rightarrow \cdots \rightarrow 0 \]

How many “steps” before the second argument (which is an integer) reaches 1? Let the number of steps be \(t\). Then \(2^t \geq b\), hence \(t \geq \log_2(b)\). Thus the number of steps is \(\lceil \log_2(b) \rceil\). One more step is needed to reach 0. Hence the overall number of recursive calls is at most \(1 + 2\lceil \log_2(b) \rceil = O(\log b)\).

### 4.2 Problem: Searching for the maximum element

Let’s consider another classic problem to illustrate the idea of recursion and its analysis.
4.2. PROBLEM: SEARCHING FOR THE MAXIMUM ELEMENT

Problem 4.2. \textbf{Input:} An array $S[1 \ldots n]$ of $n$ numbers. \textbf{Output:} The maximum number in $S$.

We can think of an array as an ordered set (or list) of elements.

4.2.1 A non-recursive algorithm

This algorithm simply compares each element with the current maximum in a sequential manner. This is called \textbf{sequential search}.

\textbf{Input:} Array $S[1 \ldots n]$ of size $n \geq 1$.
\textbf{Output:} maximum element in $S$.

\textbf{Algorithm} Max($S[1 \ldots n]$)

1: $\text{max} = S[1]$ // initialize max to be the first element.
2: \textbf{for} $i = 1$ to $n$ \textbf{do}
3: \hspace{1em} \textbf{if} $\text{max} < S[i]$ \textbf{then}
4: \hspace{2em} $\text{max} = S[i]$
5: \textbf{Output} max

In each step of the for loop the current max is compared with the $i$th element of $S$; if it is less then variable max is updated. Clearly, at the end of the for loop, max contains the maximum number in $S$. Thus, algorithm Max takes $O(n)$ comparisons (we will assume that comparison is the costly operation for this algorithm and hence this is a good measure of the running time. Nevertheless, in this example, even if you take the other operations into account, e.g., assignment etc., the overall time is $O(n)$).

4.2.2 A recursive algorithm: Divide and conquer strategy

There are many ways to design a recursive algorithm for finding the maximum element. One can adopt a decrease and conquer strategy whereby the size of the array (in the subproblem) is reduced by one in iteration. This strategy gives rise to one recursive subproblem (Can you see how?).

Alternatively, we adopt a divide and conquer strategy has typically 3 parts: Split, Solve (recursively), and Combine.

- **Split** the problem into several \textbf{smaller independent} subproblems.
- **Recursively solve** the subproblems.
- **Combine** the solutions of the subproblems to get the solution for the original problem.
Divide and conquer strategy for finding max:

- **Split**: Split the given set $S$ into two (almost) equal parts (first half and second half)\[^2\]
- **Solve**: Recursively find the maximum of the first part and the second part — let these be $\text{max}_1$ and $\text{max}_2$ respectively.
- **Combine**: Output the maximum of $S$ as the maximum of $\text{max}_1$ and $\text{max}_2$.

### Algorithm 8 Algorithm Maxr($S[a..b]$)

**Input**: Array $S[a..b]$, with left index $a$ and right index $b$. **Output**: Maximum element in $S$.

1. if $a = b$ then
2. Return $S[a]$
3. $m = \lfloor(a + b)/2\rfloor$ // middle point
4. $\text{max}_1 = \text{Maxr}(S[a..m])$ // first part
5. $\text{max}_2 = \text{Maxr}(S[m + 1..b])$ // second part
6. if $\text{max}_1 > \text{max}_2$ then
7. Return $\text{max}_1$
8. else
9. Return $\text{max}_2$

**A Divide and Conquer Algorithm** To solve the original problem of finding the maximum element in $S[1..n]$, we call Maxr($S[1..n]$).

**The recursion “flow diagram”** Consider invoking Algorithm Maxr on input $S[1..8]$.

The sequence of recursive calls can be represented as a “flow diagram” which will be a tree. This is usually called as a **Recursion Tree**. See Figure 4.2

### 4.2.3 Correctness of Algorithm Maxr

It is easy to establish the correctness of the above algorithm by using mathematical induction. The induction is on the size of the input set $S$.

This is done in two steps.

\[^2\]In many divide and conquer problems, the input is typically split into two (almost) equal disjoint parts giving rise to two independent problems; thus this is a good strategy to try for divide and conquer problems.
4.2. **PROBLEM: SEARCHING FOR THE MAXIMUM ELEMENT**

![Recursion flow diagram for Algorithm Maxr.](image)

**Proof by Mathematical Induction**

1. **Base case**: This is the trivial case when the size of $S$ is 1. Clearly, the algorithm is correct, since it outputs this single element.

2. **Induction step**: Assume that the algorithm is correct for all sets of size $< n$. This is called the Induction hypothesis.

**Establishing the Induction Step** Assumming the induction hypothesis, we will show that the algorithm is correct also for input set of size $n$, i.e., we will show that Maxr($S[1 \ldots n]$) is correct. By the induction hypothesis, $max_1$ and $max_2$ are correct maximum values of the sets $S[a \ldots m]$ and $S[m + 1, b]$ respectively (since they are of size $< n$). Hence the maximum of $max_1$ and $max_2$ should be the maximum of the set $S[1 \ldots n]$.

**Running Time** How long does Algorithm Maxr take? Running time is determined by the total number of recursive calls. The number of recursive calls is equal to the size of the recursion tree, i.e., the number of nodes in the tree.

In our example, when input is $S[1 \ldots 8]$, the size of the tree is $15 = 1 + 2 + 4 + 8 = 2^0 + 2^1 + 2^2 + 2^3 = 2^4 - 1$. Generalizing the above: when the input is $S[1 \ldots 2^k]$, the size of the tree is $2^0 + 2^1 + 2^2 + \ldots 2^k = 2^{k+1} - 1$.
$2(2^k) = 2n = O(n)$, where $n = 2^k$ is the size of the array. Thus the running time is proportional to the size of the array, i.e., $O(n)$.

**Analyzing using a Recurrence** Recurrence relations are a powerful way to analyze recursive algorithms. This is especially the case with divide/decrease and conquer algorithms. Let’s analyze Algorithm Maxr using a recurrence.

Instead of counting recursive calls, we will count the number of comparisons. Both are equivalent ways (asymptotically) to bound the run time.

Let the function $T(n)$ denote the number of comparisons required by Maxr on an input array of size $n$.

Then we can write the recurrence:

$$T(n) = 2T(n/2) + 1.$$ 

We also need a “base” case:

$$T(1) = 0.$$ 

How to solve such a recurrence?

**Unwinding the recurrence** For convenience, we will assume that $n$ is a power of 2, i.e., $n = 2^k$, for some $k$. This assumption will not change the asymptotic result\(^3\) Then:

$$T(2^k) = 2T(2^{k-1}) + 1 = 2(2T(2^{k-2}) + 1) + 1 =$$

$$= 2^2T(2^{k-2}) + 2 + 1$$

$$= 2^3T(2^{k-3}) + 2^2 + 2 + 1 = 2^kT(1) + 2^{k-1} + \cdots + 1$$

$$= 2^{k-1} + 2^{k-2} + 2 + 1 = \sum_{i=0}^{k-1} 2^i.$$ 

The above sum is obtained by using the formula for the sum of a geometric series:

$$\sum_{i=0}^{k-1} x^i = 1 + x + \cdots + x^{k-1} = \frac{x^k - 1}{x - 1}.$$ 

where, $x$ is called the (common) ratio.

\(^3\)The reason is as follows. If $n$ is not a power of 2, one can sandwich $n$ between two successive powers of 2, i.e., $2^k < n < 2^{k+1}$ and then apply the argument for $2^k$ and $2^{k+1}$ which are away by a factor of 2 from $n$. Since $T(n)$ is an increasing function of $n$, we have $T(2^k) \leq T(n) \leq T(2^{k+1})$ and thus: $2^k - 1 \leq T(n) \leq 2^{k+1} - 1$. 
Hence,
\[ T(n) = \sum_{i=0}^{k-1} 2^i = 2^k - 1. \]
Hence,
\[ T(2^k) = 2^k - 1 = n - 1 = n - 1 = O(n). \]

4.3 Techniques for Solving Recurrences

4.3.1 Guess and Verify

Alternative to the “unwinding” technique discussed above, another powerful way to solve recurrences is the “guess and verify” method. This uses mathematical induction to guess the correct solution to the recurrence and then prove that it is indeed correct, by using induction. Let’s solve the recurrence for Algorithm Maxr using this method.

The “real” recurrence is:
\[ T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1. \]

Guess \( T(n) = O(n) \). Let’s use induction to show that \( T(n) \leq cn \), for some (suitable) constant \( c \).

Induction hypothesis: Assume that \( T(k) \leq ck \) for all \( k < n \). Plugging the hypothesis in the recurrence gives:
\[ T(n) \leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1. \]
\[ = cn + 1, \]
which does not prove our hypothesis (note that it should be \( T(n) \leq cn \)).

Revising our Guess We revise our guess to: \( T(n) \leq cn - b \) and show this by induction. Not that this is called strengthening the guess (or the hypothesis), i.e., we now try to show a stronger bound. Strengthening (as opposed to weakening) the hypothesis might seem counterintuitive; but, in fact, this strengthened hypothesis allows the induction proof to go through.

(New) Induction hypothesis: Assume that \( T(k) \leq ck - b \) for all \( k < n \). Then
\[ T(n) \leq (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1. \]
= cn - 2b + 1 \leq cn - b

if \( b \geq 1 \).

How large should \( c \) and \( b \) be? This depends on the “base” case of the recurrence.

In \( \text{Maxr} \), \( T(1) = 0 \). Hence, we can choose \( c = 1 \) and \( b = 1 \), since it satisfies the base case. Thus, we have shown \( T(n) \leq n - 1 \) for all \( n \).

Another Example

**Example 4.1.** Solve the following recurrence:

\[
T(n) = 2T(n/2) + n.
\]

**Guess and verify** that \( T(n) \leq cn \log n \), for some constant \( c \). **Induction hypothesis:** Assume that \( T(k) \leq ck \log k \) for all \( k < n \).

Hence we have:

\[
T(n) \leq 2cn/2 \log n/2 + n.
\]

\[
= cn \log n - cn \log 2 + n.
\]

\[
= cn \log n - cn + n.
\]

\[
\leq cn \log n
\]

if \( c \geq 1 \). Note that \( c \) should be chosen large enough to satisfy the base condition.

**A Wrong Argument using Induction** If you try to show \( T(n) \leq cn \) by arguing:

\[
T(n) \leq 2cn/2 + n = (c + 1)n = O(n).
\]

This is incorrect! (why?)

**Solving Recurrences via Change of Variables**

**Example 4.2.** Solve

\[
T(n) = 2T(\sqrt{n}) + 1.
\]

Renaming \( m = \log n \), we have:

\[
T(2^m) = 2T(2^{m/2}) + 1.
\]
Again, renaming \( S(m) = T(2^m) \):

\[
S(m) = 2S(m/2) + 1.
\]

Thus, \( S(m) = O(m) \) and \( T(n) = T(2^m) = S(m) = O(m) = O(\log n) \).

### 4.3.2 A General Theorem for “Divide and Conquer” Recurrences

We now give a formula that can quickly solve certain types of recurrences. This type occurs most commonly in analysis of divide and conquer type algorithms and thus very useful.

Consider recurrences of the form:

\[
T(n) = aT(n/b) + f(n),
\]

where \( a \geq 1 \) and \( b > 1 \) are constants and \( f(n) \) is some function. Assume, without loss of generality, that \( n \) is a power of \( b \), i.e., \( n = b^L \) for some \( L \geq 0 \). (It can be shown that the asymptotic bounds as shown in the DC Recurrence Theorem below will hold, even when \( n \) is not a power of \( b \). Exercise 4.2 asks you to show it.) Assume that \( T(1) = O(1) \), i.e., the time for solving an instance of size 1 is a constant. Under these assumptions, we prove the following theorem.

**Theorem 4.2 (DC Recurrence Theorem).** The solution for the above recurrence \( T(n) \) is:

1. If \( af(n/b) = cf(n) \) for some constant \( c < 1 \) then \( T(n) = \Theta(f(n)) \).
2. If \( af(n/b) = cf(n) \) for some constant \( c > 1 \) then \( T(n) = \Theta(n^{\log_b a}) \).
3. If \( af(n/b) = f(n) \) then \( T(n) = \Theta(f(n) \log_b n) \).

**Proof.** Draw a “recursion tree” for \( T(n) \): \( T(n) \) stands for the root node (level 0) and it has \( a \) children each of which is a root of the recursion tree for \( T(n/b) \). One can associate a value with each node: the value is the cost that is paid at that level to go to the next recursive level. The value associated with the root node \( T(n) \) is \( f(n) \) (this is the cost paid for going to the next level, i.e., level 1). A recursion tree is a complete \( a \)-ary tree (i.e., all levels are full — this follows from the assumption that \( n \) is a power of \( b \), where each node at level (depth) \( i \) has the value \( f(n/b^i) \). The total value of all nodes in level \( i \) is \( a^i f(n/b^i) \). See Figure 4.3 for an illustration. The leaves of the tree contains the “base cases” of the recursion. By our assumption,
Figure 4.3: Illustration for proof of DC Recurrence Theorem.
4.3. TECHNIQUES FOR SOLVING RECURRENCES

$T(1) = f(1) = \Theta(1)$, Thus we have the “total cost” of the recursion which is the sum of all the values (costs) in all levels:

$$T(n) = f(n) + af(n/b) + a^2f(n/b^2) + \cdots + a^L f(n/b^L)$$

where $L$ is the depth of the recursion tree. Note that $n/b^L = 1$.

Substituting, $af(n/b) = cf(n)$ in the above formula for $T(n)$ we have

$$T(n) = f(n) + cf(n) + c^2f(n) + \cdots + c^L f(n).$$

Note that $L = \log_b n$ and since $f(1) = \Theta(1)$, $a^L f(n/b^L) = \Theta(n^{\log_b a}).$

1) If $f(n)$ is a constant factor larger than $af(n/b)$ then $T(n)$ is a geometric series (see Appendix[4] with largest term $f(n)$. Hence $T(n) = \Theta(f(n))$.

2) If $f(n)$ is a constant factor smaller than $af(n/b)$ then $T(n)$ is a geometric series with largest term $a^L f(n/b^L) = \Theta(n^{\log_b a})$.

3) If $af(n/b) = f(n)$ then there are $L + 1$ levels each level summing to $f(n)$ and hence $\Theta(f(n) \log_b n)$.

Example 4.3. 1. $T(n) = T(3n/4) + 2n$. Here $af(n/b) = 2(3n/4) = (3/4)f(n)$ Hence $T(n) = \Theta(n)$.

2. $T(n) = 7T(n/2) + \Theta(n^2)$. That is, $T(n) = 7T(n/2) + c_1 n^2$, for some positive constant $c_1$. $af(n/b) = 7c_1 (n/2)^2 = (7/4)c_1 n^2 = (7/4)f(n)$. Hence, $T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$.

3. MergeSort: $T(n) = 2T(n/2) + n$. Here $af(n/b) = f(n)$ and hence $T(n) = \Theta(n \log n)$.

Other Examples  Some recurrences do not exactly fit the DC Recurrence Theorem.

Example 4.4. $T(n) = 2T(n/2) + n/\log n$

We can't apply the DC Recurrence Theorem directly since $af(n/b) = n/(\log n - 1)$ is not equal to a constant factor times $n/\log n$. We compute the recurrence directly.

[4]Note that, we can write $a^2f(n/b^2) = a(af(n/b^2)) = a(cf(n/b)) = c(af(n/b)) = c^2f(n)$. Similarly, we can infer that $a^i f(n/b^i) = c^i f(n)$, for any $i$. 
The sum of the nodes in the $i$th level is $n/(\log n - i)$. Thus, the depth of recursion is at most $\log n - 1$.

$$T(n) = \sum_{i=0}^{\log n - 1} \frac{n}{\log n - i} = \sum_{j=1}^{\log n} \frac{n}{j} = nH_{\log n} = \Theta(n \log \log n),$$

where $H_k = \sum_{i=1}^{k} \frac{1}{i}$ is the harmonic function and $H_k = \Theta(\log k)$.

**Example 4.5.** $T(n) = T(3n/4) + T(n/4) + n$.

We can solve this by directly looking at the recursion tree for this recurrence. Each complete level in the tree adds up to $n$ and each leaf has depth between $\log_4 n$ and $\log_4/3 n$. Hence, $T(n) = \Theta(n \log n)$.

### 4.3.3 Handling Floors and Ceilings in Recurrences

Suppose we have the following recurrence (this arises in MergeSort, as we will see later)

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n.$$

We want to show that $T(n) = \Theta(n \log n)$.

We show the upper bound, lower bound is similar.

$$T(n) \leq 2T(\lfloor n/2 \rfloor) + n \leq 2T(n/2 + 1) + n.$$

To solve for $T(n)$ we do a “domain transformation” as follows. Let $S(n) = T(n + \alpha)$ where $\alpha$ is a constant chosen such that

$$S(n) \leq 2S(n/2) + \Theta(n).$$

To find $\alpha$, we compare the two recurrences — that of $T(n)$ and $S(n)$.

$$T(n + \alpha) \leq 2T((n + \alpha)/2 + 1) + n.$$

$$T(n + \alpha) \leq 2T(n/2 + \alpha) + \Theta(n).$$

which gives $\alpha = 2$.

By DC Recurrence Theorem, $S(n) = O(n \log n)$, and hence $T(n) = S(n - \alpha) = O(n \log n)$. 

4.4 Worked Exercises

Worked Exercise 4.1. A light bulb is controlled by \( n \) switches. Each switch can be in one of two possible states — “OFF” and “ON”. We can’t tell the status of a switch (i.e., whether it is ON or OFF) at any given time. We can only toggle it between the two states by pressing the switch. Only when all switches are in the ON state, the light bulb will glow, otherwise it will not glow. Given that the light bulb is not initially glowing, give a recursive algorithm to make it glow. What is the number of operations (as a function of \( n \)) needed by your algorithm in the worst case? (Count each toggle of a switch as one operation.)

Solution:
As usual, we will think recursively. We look at the base case, i.e., \( n = 1 \), of glowing the bulb with one switch. In this case, it takes one operation, simply toggle the switch if the bulb is not glowing. Now consider the recursive step where we have \( n > 1 \) switches. We can reduce this to a problem glowing \( n - 1 \) switches; with the \( n \)th switch taking two possibilities. The recursive function \( \text{glow}() \) is as follows. The pseudocode is given in Figure 9. Note the recursive calls — there are two recursive calls to \( \text{glow}(S[1, \ldots, n-1]) \) which is needed; the first one for one position of switch \( S[n] \) and other after toggling \( S[n] \) (if it was not glowing after the first recursive call). Intuitively what this is doing is that fixing one position of \( S[n] \) it explores all possibilities of the first \( n - 1 \) switches.

The number of operations (toggling) can be computed by a recurrence. Let \( T(n) \) be the number of operations needed to glow a bulb with \( n \) switches. Then from the algorithm, we have: \( T(1) = 1 \) and for \( n > 1 \), \( T(n) = 2T(n - 1) + 1 \). This can be solved by unwinding the recurrence as follows:

\[
T(n) = 2T(n - 1) + 1 = 2(2T(n - 2) + 1) + 1 = 2^2T(n - 2) + 2 + 1 = 2^3T(n - 3) + 2^2 + 2 + 1 \text{ and so. Thus, continuing the pattern: } T(n) = 2^{n-1}T(n - (n - 1)) + 2^{n-2} + \cdots + 2 + 1 = 2^{n-1}T(1) + 2^{n-2} + \cdots + 2 + 1 = 2^n - 1
\]

by using the formula for the geometric series (with common ratio 2).

Worked Exercise 4.2. Consider the problem of finding the largest and second largest elements from a set of \( n \) elements.

a) Give a simple iterative algorithm that takes no more than \( 2n - 3 \) comparisons.

b) Give a divide and conquer algorithm and show that it takes \( \frac{3n}{2} - 2 \) comparisons when \( n \) is a power of 2.
Algorithm 9 Algorithm glow(S[1,...,n]). Input: An array S of n switches. Output: Configuration of switches that make the bulb glow.

```
1: if n = 1 then
2:   if bulb not glowing then
3:     Toggle switch S[1] and Return
4:   else
5:     Return
6: else
7:   glow(S[1,...,n−1]) // recursive call
8:   if bulb not glowing then
9:     Toggle switch S[n] // one toggle operation
10:    glow(S[1,...,n−1]) // recursive call
11:   else
12:   Return
```

c) Show how you can find both these elements in $n + \log n - 2$ comparisons in the worst case (assume $n$ is a power of 2). (Hint: See hint in problem 2.3.)

Solution:

a) Find the max by a simple iterative algorithm (see Section 4.2.1 of the book). Then repeat the same, by finding the max among the rest of the elements — this will be the second max. The total number of comparisons is $n - 1 + n - 2 = 2n - 3$.

b) A divide and conquer algorithm: Divide the elements into two groups. Find the largest and second largest elements from each group. Then find the largest and second largest from these four elements. To find the largest and second largest from each group, proceed recursively.

A divide and conquer algorithm is given in Figure 10.

Analysis: Let $T(n)$ be the number of comparisons needed for Find2Largest(A[1..n]), when there are $n$ elements. Assuming $n$ is a power of 2, number of comparisons to solve each sub-problem is $T(n/2)$ and exactly 2 comparisons for combining the result. Thus

$$T(n) = 2T(n/2) + 2$$

Boundary condition: with two elements number of comparisons is $T(2) = 1$.

By induction on $n$, we show that $T(n) = 3n/2 - 2$. 
1) **Base case**: for \( n = 2 \), \( T(n) = 3(2/2) - 2 = 1 \).

2) **Induction step**: for \( n > 2 \)

\[
T(n) = 2T(n/2) + 2
\]
\[
= 2(3(n/2)/2 - 2) + 2
\]
\[
= 3n/2 - 2
\]

Alternate way to do this is to simply "unwind" the recurrence, i.e., use the recursion tree method.

c) The algorithm consists of two phases. In the first phase, we find the largest element by comparing all \( n \) elements. In the second phase, we find the second largest element, by comparing a particular subset of the elements (as described below).

For the first phase, compare the elements pairwise. From each pair, eliminate the smaller element and keep the larger element. If the number of elements is odd, there is an unpaired element; keep the unpaired element. Thus we are left with \( \lceil n/2 \rceil \) elements. Repeat the process on the remaining \( \lceil n/2 \rceil \) elements that leaves us with \( \lceil n/4 \rceil \) elements and so on. After exactly \( \lceil \lg n \rceil \) steps, we are left with exactly one element, which is the largest of the \( n \) elements. In these \( \lceil \lg n \rceil \) steps, we make exactly \( n - 1 \) comparisons. Because, each comparison eliminates one element and exactly \( n - 1 \) elements are eliminated. Notice that the second largest element can be eliminated only when it is compared to the largest element. Thus to find the second largest element, we need to check only the elements, say the candidate elements, that are compared with the largest element in the first phase.

To facilitate tracking of the candidate elements, in addition to the pairwise comparison in the first phase, build a binary tree. Initially, each element constitutes a leaf of the tree. When two elements \( a \) and \( b \) are compared, create an internal node having children \( a \) and \( b \), and label it with the larger of \( a \) and \( b \). For an unpaired element \( c \), create a node having only one child \( c \) and label it also with \( c \). The root of the tree is labeled with the largest element. The candidates for the second largest can be found by tracking the path of the largest element back from the root to the leaf labeled by the largest element. There are at most \( \lceil \lg n \rceil \) candidates requiring at most \( \lceil \lg n \rceil - 1 \) comparisons to get the second largest element. The total number of comparisons \( \leq (n - 1) + (\lceil \lg n \rceil - 1) = n + \lceil \lg n \rceil - 2 \).
Worked Exercise 4.3. Solve the following recurrences. Give the answer in terms of Big-Theta notation. Solve up to constant factors, i.e., your answer must give the correct function for \( T(n) \), up to constant factors. You can assume constant base cases, i.e., \( T(1) = T(0) = c \), where \( c \) is a positive constant. You can ignore floors and ceilings. You can use the DC recurrence theorem if it applies.

(a) \( T(n) = 3T(n/2) + n \)

(b) \( T(n) = 2T(n-1) + 2 \).

(c) \( T(n) = 4T(n/2) + n^3 \).

(j) \( T(n) = T(3n/4) + T(n/4) + n \)

(k) \( T(n) = T(7n/8) + n \)

Solution:

(a) \( T(n) = 3T(n/2) + n \)

\[ 3(n/2) = (3/2)n, \]

hence \( c = 3/2 > 1 \). Hence \( T(n) = \Theta(n^{\log_3 3}) \), by the DC recurrence theorem.

(e) \( T(n) = 2T(n-1) + 2 \).

Unwinding the recurrence, we have

\[
T(n) = 2(2T(n-2) + 2) + 2 = 2^2T(n-2) + 2^2 + 2 = 2^2(2T(n-3) + 2) + 2^2 + 2
\]

\[ = 2^3T(n-3) + 2^3 + 2^2 + 2 \]

and so on.

Continuing the pattern, we see that

\[ T(n) = 2^{n-1}T(n-\(n-1\)) + 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2. \]

Assuming \( T(1) = c \), where \( c \) is a positive constant, we have

\[ T(n) = 2^{n-1}c + 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2. \]

Using the formula for the geometric series we sum:

\[
2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2 = 2(1 + 2 + \cdots + 2^{n-2}) = 2(2^{n-1} - 1) = 2^n - 2.
\]
Hence $T(n) = 2^{n-1}c + 2^n - 2 = \Theta(2^n)$.

(f) $T(n) = 4T(n/2) + n^3$.
Using the DC Recurrence theorem:
$4(n/2)^3 = (4/2^3)n^3$, hence $c = 4/2^3 = 4/8 < 1$.
Hence $T(n) = \Theta(n^3)$.

(j) $T(n) = T(3n/4) + T(n/4) + n$.
There are a couple of different ways to do this:

First way: Build a recursion tree for $T(n)$.

The root has two children $T(3n/4)$ and $T(n/4)$ with a total cost of $n$. Continuing this, it is easy to see that at every level the total cost is $n$. The number of levels is at most $\log_{4/3} n$ since the longest path from root to leaf is via the leftmost path, i.e., path with $3/4$-fraction — this is because that this is the path where the size of the subproblem reduces the least. Hence total cost, $T(n)$, is $\Theta(n \log n)$.

Second way: Guess the solution is $\Theta(n \log n)$ and then verify this by induction.

First, we show the upper bound. Assume that, for some constant $a > 0$, $T(k) \leq ak \log k$ for all $k < n$ (induction hypothesis). Note that the base case is easily true, by choosing $a$ large enough (compared to $T(1)$ and $T(0)$). Next we show that the statement is true for $k = n$, i.e., $T(n) \leq an \log n$. Then

\[
T(n) \leq a(3n/4) \log(3n/4) + a(n/4) \log(n/4) + n
\]
\[
= a(3n/4)(\log n + \log(3/4)) + a(n/4)(\log n + \log(1/4)) + n
\]
\[
= \log(n)(3an/4 + an/4) - a(3n/4) \log(4/3) - a(n/4) \log(4) + n
\]
\[
= an \log n - an(3/4) \log(4/3) - a(n/4) \log(4) + n
\]
\[
= an \log n - an(3/4 \log(4/3) + (1/4) \log(4)) + n \leq an \log n.
\]

if $a$ is chosen sufficiently large.

Now we show the lower bound. Assume that, for some constant $a > 0$, $T(k) \geq ak \log k$ for all $k < n$ (induction hypothesis). Note that the base case is easily true, by choosing $a$ small enough. Next we show that the statement is true for $k = n$, i.e., $T(n) \geq an \log n$. Then, we have:

\[
T(n) = T(3n/4) + T(n/4) + n \geq a(3n/4) \log(3n/4) + a(n/4) \log(n/4) + n
\]
\[
= a(3n/4)(\log n + \log(3/4)) + a(n/4)(\log n + \log(1/4)) + n
\]
\[
= \log(n)(3an/4 + an/4) - an(3/4) \log(4/3) - an(4/4) \log(4) + n
\]
\[
= an \log n - an((3/4) \log(4/3) + (1/4) \log(4)) + n \geq an \log n
\]
if $a$ is chosen sufficiently small.

(k) (5 points) $T(n) = T(7n/8) + n$

Using the DC recurrence theorem,

$7n/8 = (7/8)(n)$, hence $c = 7/8 < 1$.

Hence $T(n) = O(n)$.

### 4.5 Exercises

**Exercise 4.1.** Prove the asymptotic bound for the following recurrences by using induction. Assume that base cases of all the recurrences are constants i.e., $T(n) = \Theta(1)$, for $n < c$ where $c$ is some constant.

(a) $T(n) \leq 2T(n/2) + n^2$. Then, $T(n) = O(n^2 \log n)$.

(b) $T(n) \leq T(5n/6) + O(n)$. Then, $T(n) = O(n)$.

(c) $T(n) = T(5n/6) + T(n/6) + n$. Then, $T(n) = O(n \log n)$.

**Exercise 4.2.** Show the asymptotic bounds in the DC recurrence theorem hold even when $n$ is not a power of $b$. (Hint: Sandwich $n$ between two successive powers of $b$, i.e., let $b^k \leq n \leq b^{k+1}$. Then proceed as in the proof of the DC recurrence theorem.)

**Exercise 4.3.** Solve the following recurrences. Give the answer in terms of Big-Theta notation. Solve up to constant factors, i.e., your answer must give the correct function for $T(n)$, up to constant factors. You can assume constant base cases, i.e., $T(1) = T(0) = c$, where $c$ is a positive constant. You can ignore floors and ceilings. You can use the DC recurrence theorem if it applies.

a) $T(n) = 3T(n/2) + n$.

b) $T(n) = 4T(n/2) + n \log n$.

c) $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

d) $T(n) = T(n/2) + T(n/3) + 1$.

e) $T(n) = 2T(n - 1) + 2$.

f) $T(n) = 4T(n/2) + n^3$.

g) $T(n) = T(n - 1) + 2^n$. 

4.5. EXERCISES

h) \( T(n) = 2T(n/2) + n / \log n \).

i) \( T(n) = 4T(n/3) + n^2 \).

j) \( T(n) = T(3n/4) + T(n/4) + n \).

k) \( T(n) = T(7n/8) + n \).

l) \( T(n) = T(3n/4) + O(n) \).

Exercise 4.4. You are given an array consisting of \( n \) numbers. A popular element is an element that occurs (strictly) more than \( n/2 \) times in the array. Give an algorithm that finds the popular element in the array if it exists, otherwise it should output “NO”. Your algorithm should take no more than \( 2n \) comparisons. (As usual, we only count the comparisons between array elements.) Give pseudocode, argue its correctness, and show that your algorithm indeed takes no more than \( 2n \) comparisons. (Hint: Use a decrease and conquer strategy, similar to the celebrity problem.)

Exercise 4.5. \( n \) marbles are placed in a line. A marble’s color is either red or blue. The colors can appear in any arbitrary manner (e.g., they can all be red or blue, or some may be red and the others blue). The goal is to remove all the marbles by a sequence of moves. In each move, a red marble can be removed; at the same time, we replace its neighbors (if any) by marbles of opposite color (i.e., Red becomes Blue and vice versa). A marble is a neighbor of another if it is adjacent to it in the original line (two marbles are not neighbors if there are ‘gaps’ between them, which can occur when in-between marbles are removed). To illustrate, see the following example.

Let RBBRBRBBB be the initial arrangement of marbles. Then the successive steps are shown as follows (a removed marble is shown as X):

\[
\begin{align*}
XRBBRBRBB & \rightarrow XRBBRRXRB \rightarrow XRBBRRXXR \rightarrow XRBBRRXX\rightarrow \\
XRBBXX\rightarrow XXRBBXX\rightarrow XXRBXX\rightarrow XXXRXX\rightarrow XXXX\rightarrow XXXXX.
\end{align*}
\]

1. It is clear that problem does not always have a solution (i.e., all marbles cannot be removed); for example, if all marbles are blue then nothing can be removed. Give a necessary and sufficient condition for the problem to have a solution. Prove that your condition is indeed necessary and also sufficient. “Necessary” means that without the condition, there is no solution; “Sufficient” means that with the condition there is a solution. (Hint: Consider small problem sizes to understand; red marbles are the key, in particular the number of red marbles.)
2. Assuming that there is a solution, give an algorithm to remove all the marbles. Prove that your algorithm indeed works. (Hint: It is important to remove the red marbles in a proper manner. Consider the example: the arrangement RRR clearly has a solution: XBR → XRX → XXX. However, if you remove the middle red first, then you will be stuck. Consider small problem sizes to understand what happens. For large sizes, use decrease/divide and conquer. Use induction in your proof.)

Exercise 4.6. Given two integers \( a \) and \( b \), we want to compute \( d = \gcd(a, b) \) and two integers \( x \) and \( y \) such that \( ax + by = d \). Modify Euclid’s algorithm to output \( x \) and \( y \) (besides \( d \)).

Exercise 4.7. You are given \( n \) stones (assume that \( n \) is a power of 2) each having a distinct weight. You are also given a two-pan balance scale (no weights are given). For example, given two stones, you can use the scale to compare which one is lighter by placing the two stones on the two different pans. The goal is to find the heaviest and the lightest stone by using as few weighings as possible. Give a divide and conquer strategy that uses only \( \frac{3n}{2} - 2 \) weighings. (Hint: See Worked Exercise 4.2.)

\begin{algorithm}
\begin{algorithmic}
\State \textbf{if} $\text{Left} = \text{Right}$ then
\State $\text{// only one element}$
\State \quad \textbf{return} ($A[\text{left}]$, NONE)
\State \textbf{if} $\text{Left} + 1 = \text{Right}$ then
\State $\text{// two elements}$
\State \quad \textbf{if} $A[\text{Left}] > A[\text{Right}]$ then
\State \qquad \textbf{Largest} $\leftarrow A[\text{Left}]$
\State \qquad \text{SecondLargest} $\leftarrow A[\text{Right}]$
\State \quad \textbf{else}
\State \qquad \text{Largest} $\leftarrow A[\text{Right}]$
\State \qquad \text{SecondLargest} $\leftarrow A[\text{Left}]$
\State \textbf{return} (\text{Largest}, \text{SecondLargest})
\State $\text{// Divide the input and solve the problem for each subset.}$
\State $\text{Middle} = \lfloor(\text{Left} + \text{Right})/2\rfloor$
\State $(P_1, P_2) \leftarrow \text{Find2Largest}(A[\text{Left}..\text{Middle}])$
\State $(Q_1, Q_2) \leftarrow \text{Find2Largest}(A[\text{Middle} + 1..\text{Right}])$
\State $\text{// Combine the results.}$
\State \textbf{if} $P_1 > Q_1$ then
\State \quad \text{Largest} $\leftarrow P_1$
\State \textbf{if} $P_2 > Q_1$ then
\State \quad \text{SecondLargest} $\leftarrow P_2$
\State \textbf{else}
\State \quad \text{SecondLargest} $\leftarrow Q_1$
\State \textbf{else}
\State \quad \text{Largest} $\leftarrow Q_1$
\State \textbf{if} $Q_2 > P_1$ then
\State \quad \text{SecondLargest} $\leftarrow Q_2$
\State \textbf{else}
\State \quad \text{SecondLargest} $\leftarrow P_1$
\State \textbf{return} (Largest, SecondLargest)
\end{algorithmic}
\end{algorithm}
4. RECURSION
In this chapter, we will see more problems which can be solved using the divide and conquer strategy. The problems that we will study are fundamental with variety of applications in the real world.

5.1 Problem: Sorting

Problem 5.1 (Sorting). Input: An array $X[1 \ldots n]$ of $n$ elements (coming from a totally ordered domain, e.g., integers, real numbers, strings). (A totally ordered domain is one where every pair of elements, say $a$ and $b$, in the domain can be assigned an ordering with respect to each other, i.e., either, $a < b$, or $a > b$ or $a = b$.) Output: An ordering of elements of $X$ such that $X[1] \leq X[2] \leq \ldots \leq X[n]$.

Sorting is one of the most fundamental problems in computing, perhaps the most well-studied. It arises as a basic subroutine in other problems. There are many different algorithms devised for sorting, but here we will focus on two Divide and Conquer algorithms namely, MergeSort and Quicksort. Note that we assume that the only way to infer the ordering between two elements (say $a$ and $b$) is by comparing them, i.e., asking the question “Is $a < b$?”. The goal is to devise an algorithm that does as few comparison operations as possible.

5.1.1 A Simple Sorting Algorithm

Here is an “obvious” algorithm for sorting. (This is not a divide and conquer algorithm.) First, find the smallest element in the array and put it in the first position. Then, find the second smallest element and place it in the second position. In general, find the $i$th smallest element and place
it in the $i$th position. To find the $i$th smallest element we have to only compare with elements that are in the positions $i + 1$ or greater. We can call this algorithm as SimpleSort — see figure 11 for the pseudocode. See figure 5.1 for an example run of this algorithm.


1: for $I = 1$ to $n$ do
2:   for $J = I + 1$ to $n$ do
3:      if $X[J] < X[I]$ then
4:         // Exchange $X[I]$ and $X[J]$ 
5:          TEMP = $X[I]$
6:          $X[I] = X[J]$
7:          $X[J] = TEMP$

Correctness of SimpleSort The correctness follows from the loop invariant of the outer for loop: after the $i$th iteration of the outer for loop, the $i$th smallest element is placed in the $i$th index of the array. It is then never moved again.

Running time of SimpleSort How to measure the running time of SimpleSort? One reasonable measure is to count the number of comparisons that the algorithm performs. Note that a “comparison” is an operation that compares one element with another. In Algorithm SimpleSort, there
is comparison operation in line 3. Measuring number of comparisons is a good way to measure the efficiency of any sorting algorithm. It typically dominates (or is at least of the same order compared to) other operations performed by a sorting algorithm.

We calculate the number of comparisons in SimpleSort. Let the array $X$ contain $n$ elements. In the first iteration of the outer for loop, the first element of $X$ is compared with the rest of the elements — this costs $n - 1$ comparisons. In the second iteration of the outer for loop, the (current) second element of $X$ is compared with elements in positions 3 onwards till the last — costs $n - 2$ comparisons. Thus, in the $i$th iteration of the outer for loop, the (current) $i$th element of $X$ is compared with elements in positions $i + 1$ onwards till the last — costs $n - i$ comparisons.

Thus the total number of comparisons performed by SimpleSort is

\[
\sum_{i=1}^{n-1} i + \cdots + 1 = \frac{(n - 1)(n)}{2},
\]

since this is the sum of the first $n - 1$ numbers.

We can write \(\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}\). The dominating term is $n^2/2$. Hence the number of comparisons of SimpleSort is $\Theta(n^2)$, i.e., it has quadratic run time.

SimpleSort took $(n - 1)n/2$ comparison operations. Note that $(n - 1)n/2 = \binom{n}{2}$, i.e., the total number of different pairs of elements. SimpleSort compares every distinct pair of elements, i.e., it compares every element with every other element once. Can we get a “better” sorting algorithm — one that takes substantially less number of comparisons? We next give an algorithm called MergeSort that takes only $O(n \log n)$ comparisons (assume $\log$ is to the base 2). Note that if $n = 2^{20}$, then $n^2 = 2^{40}$, whereas $n \log n$ is only $20 \times 2^{20}$. Hence an $O(n \log n)$ algorithm gives us substantial savings compared to an $\Theta(n^2)$ algorithm.

### 5.1.2 MergeSort: A Divide and Conquer Sorting Algorithm

We follow a divide and conquer strategy to design a sorting algorithm. As outlined in Chapter 2, the strategy consists of three parts.

- **Split**: Split the array into two (approximately) equal halves.
• **Solve**: Recursively sort the two subarrays independently.

• **Combine**: Combine the two sorted subarrays (by merging) to get an overall sorted array.

Using the above strategy, it is easy to write a high-level procedure for MergeSort as follows.

**MergeSort** \(A[1, \ldots, n]\):

1. If \(n = 1\) then output \(A\).
2. \(A_1 = \text{Merge-Sort}(A[1, \ldots, n/2])\) (Sort the first half).
3. \(A_2 = \text{Merge-Sort}(A[(n/2) + 1, \ldots, n])\) (Sort the second half).
4. Return Merge\((A_1, A_2)\) (Merge the two halves into an overall sorted array and return it. **Merge** takes two sorted arrays and combines them into one overall sorted array.)

See Figure 12 for the MergeSort recursion tree which illustrates the algorithm in action on an input of size 8.

**Merge Procedure** The only thing that is really non-trivial in MergeSort is the Merge procedure which we describe next. This procedure takes as input two sorted arrays \(A\) and \(B\) and outputs an overall sorted array \(X\) consisting of the elements of \(A\) and \(B\).

**Merge** procedure uses an auxiliary array \(X\). \(X\) will be initially empty. In each iteration one element from either \(A\) or \(B\) will be added to \(X\), depending on whichever is the smallest; it then considers the next element in the set from which the element was just added and repeats the process. Since both \(A\) and \(B\) are individually sorted themselves it is enough to consider the elements of both in increasing order (from left to right). The total time is linear in the total number of elements in \(A\) and \(B\). At the end, all the elements will be added and they will also be sorted. See Figure 5.3 that illustrates how the Merge procedure works on a example. Pseudocode of Merge is given in Section 5.1.2.

**Running time of Merge** We need to bound the running time of **Merge** to bound the running time of **MergeSort**. As done earlier (for **SimpleSort**), we will use the number of comparisons as an estimate of the running time. How many comparisons does **Merge** procedure take? Suppose the input arrays \(A\) and \(B\) are each of size \(k\). Then, **Merge** takes no more than
Figure 5.2: MergeSort example. The top figure shows the recursion as it unwinds in a top-down fashion. The bottom figure shows how the array gets sorted in a bottom-up fashion.
Figure 5.3: Illustrating Procedure Merge. \(A\) and \(B\) are the two input sorted arrays and \(X\) is the output array that is overall sorted.
2k comparisons, since at most one comparison is done to add one new element to the X array. More generally, if A and B are of sizes $k_1$ and $k_2$ respectively, then Merge takes no more than $k_1 + k_2$ comparisons.

**Running time of MergeSort** We write a recurrence. Let $T(n)$ be the number of comparisons taken by MergeSort on input size $n$. Assume $n$ is a power of 2 (without loss of generality). Then

$$T(n) \leq 2T(n/2) + n,$$

with the base case: $T(1) = 0$.

Using the DC recurrence theorem, we obtain the solution as $T(n) = O(n \log n)$. Hence MergeSort takes $O(n \log n)$ time.

**Correctness of MergeSort** It is easy to establish the correctness of the above algorithm by using mathematical induction. This is similar to the proof of the correctness of Maxr, another divide and conquer algorithm we saw earlier. The induction is on the size of the input set $S$. This is done in two steps.

1. **Base case**: This is the trivial case when the size of $S$ is 1. Clearly, the algorithm is correct, since it outputs this single element.

2. **Induction step**: Assume that the algorithm is correct for all sets of size $< n$. This is called the induction hypothesis.

The proof of establishing the induction step is left as an exercise for you (it is similar to what we did for Maxr in Chapter 2).

**Algorithm Merge-Sort: Pseudocode**

We present a more detailed pseudocode for MergeSort.

**Algorithm 12** Algorithm Merge_Sort($X, n$)

1: MergeSort($X, 1, n$)

**Merge: Pseudocode** We assume the following: **Input**: Array $X$ and indices $Left$, $Middle$, $Right$, such that the subarrays $X[Left \ldots Middle]$ and $X[Middle + 1 \ldots Right]$ are (already) sorted.

**Output**: Sorted array $X$.

Note that a temporary array Temp is used to store the overall sorted array.
5. MORE DIVIDE AND CONQUER

Algorithm 13 Procedure MergeSort(X, Left, Right)
1: if Left ≠ Right then
2: Middle = ⌈1/2(Left + Right)⌉
3: MergeSort(X, Left, Middle − 1)
4: MergeSort(X, Middle, Right)
5: Merge(X, Left, Middle, Right)

Algorithm 14 Algorithm Merge(X, Left, Middle, Right).
1: I := Left; J := Middle; K := 0 // I points to index in the first half and
2: while I ≤ Middle − 1 and J ≤ Right do
3: K := K + 1
4: if X[I] ≤ X[J] then
5: Temp[K] := X[I]; I := I + 1 // Copy from first half in Temp
6: else
7: Temp[K] := X[J]; J := J + 1 // Copy from second half in Temp
8: if J > Right then
9: for T := 0 to Middle − 1 do
10: X[Right − T] := X[Middle − 1 − T] // Shift elements from first half
// of X to its second half
11: for T := 0 to K − 1 do
12: X[Left + T] := Temp[T]

5.1.3 QuickSort: Another Divide and Conquer Algorithm

QuickSort is a popular sorting algorithm that is usually fast in practice. It
is implemented as function “qsort” in the UNIX operating system and in
C++.

Idea of QuickSort Let S be the input set (an ordered set) to be sorted.
(S is usually represented as an array as in MergeSort implementation,
but here we will be a bit more abstract and represent the input as a set.)
Choose the first element of S, say y — called the pivot element. The whole
set is then partitioned based on y into two parts — S₁ and S₂. All elements
of S₁ are less than or equal to y and all elements of S₂ are greater than y.
We then recursively sort S₁ and S₂. The output is the sorted list of S₁, y,
and sorted list of S₂ in that order. The pseudocode is given in Figure 15.

Note that QuickSort is a “Conquer and Divide” algorithm in the sense
that (unlike MergeSort), it first partitions the array around the pivot (the
left part being less than or equal to the pivot and the right part greater than the pivot) and then it recursively sorts the left part and the right part separately. It is easy to implement this partitioning “in place”, i.e., if $S$ is stored as an array, then the partitioning around the pivot can be implemented by exchanging elements in the array itself, without the need for an auxiliary array (as was in the case of MergeSort). This makes QuickSort code save space leading to better running time, although this is not an asymptotic improvement. As is clear from the pseudocode, the partitioning can be done in time linear in the size of the set, i.e., $O(n)$, where $n$ is the number of elements in the set.

Algorithm 15 Algorithm $QS(S)$. **Input:** A set $S$. **Output:** The set $S$ in sorted order.

1: if $|S| \leq 1$ then
2: return $S$
3: else
4: Let $y$ be the first element in $S$.
5: Compare all elements of $S$ to $y$. Let

$$S_1 = \{x \in S - \{y\} \mid x \leq y\}$$

$$S_2 = \{x \in S - \{y\} \mid x > y\}.$$ 

(Elements in $S_1$ and $S_2$ are in the same order as in $S$.)
6: Return the list (in the following order):

$$QS(S_1), y, QS(S_2).$$

**Correctness** The correctness of QuickSort can be shown by mathematical induction and is left as an exercise (Exercise 5.4).

**Time Analysis: Worst-case** Let $T(n)$ be the worst-case running time (number of comparisons) of Algorithm QS on a set $S$ of size $n$. By “worst-case running time” we mean the running time that is the maximum among all possible inputs. Note that worst-case running time is usually the default notion of running time used in the analysis of algorithms — a single “bad” input can substantially increase the running time of an algorithm. However, for some problems and algorithms, it might be the case that all inputs take approximately the same amount of time. In particular, in the case of Quicksort, it is instructive to understand the “worst-case”, “best-case” and “average-case” performance of the algorithm. We
will analyse the worst-case and best-case time below, but defer the analysis of average-case later (this involves use of probability). For convenience of analysis, we assume that all elements are distinct.

**Theorem 5.1.** The worst-case time, \( T(n) = \Theta(n^2) \).

**Proof.** We can write the recurrence for QuickSort as the following:

\[
T(n) \leq \max_{0 \leq k \leq n-1} \{T(k) + T(n - k - 1)\} + n - 1.
\]

The reason is as follows. The pivot partitions the array of size \( n \) into two disjoint parts each of size \( k \) (the size of set \( S_1 \)) and \( n - k - 1 \) (the size of set \( S_2 \)), where \( k \) depends on the pivot chosen. (Note that the pivot itself is excluded from the two parts). For example, if the pivot is the median (i.e., the middle element in the sorted order), then \( k \) is close to \( n/2 \). On the other hand, if the pivot is (say) the largest element, then \( k \) is \( n - 1 \) (i.e., all the elements are smaller than the pivot) and if the pivot is the smallest element, then \( k = 0 \) (i.e., all the elements are larger than the pivot). We take the maximum over all \( k \) since we want an upper bound on \( T(n) \). The cost for partitioning is \( n - 1 \) comparisons (regardless of \( k \)), since the pivot has to be compared with the rest of the \( n - 1 \) elements.

The worst-case is when the partitioning produces one subproblem of size \( n - 1 \) and another of size \( 0 \) (Exercise 5.5 asks you to show this). That is, the partitioning that leads to the maximum imbalance (See Figure 5.4 (top)).

Since partitioning costs \( n - 1 \) comparisons,

\[
T(n) = T(0) + T(n - 1) + n - 1 = T(n - 1) + n - 1,
\]

since \( T(0) = 0 \). Solving the above recurrence gives \( T(n) = \Theta(n^2) \) (this is left as an exercise — See Exercise 5.5).

**Time Analysis: Best-case** Let \( T(n) \) be the best-case running time of Quicksort on a set \( S \) of size \( n \).

**Theorem 5.2.** \( T(n) = O(n \log n) \).

**Proof.** We can write the recurrence for best case performance of QuickSort as follows:

\[
T(n) \geq \min_{0 \leq k \leq n-1} \{T(k) + T(n - k - 1)\} + n - 1.
\]
5.1. PROBLEM: SORTING

Figure 5.4: QuickSort recursion tree: worst-case (top) and best-case (bottom). The worst-case tree is very skewed (to the right). This happens when the input is already sorted; thus the pivot is always the smallest element in each recursive call and all elements are to the right of the pivot; hence one subproblem is empty and all the remaining elements are in the other subproblem. The depth of the tree is \( n \). The best-case is when the pivot is the middle element, i.e., the median, in every recursive call when the two subproblems of of almost equal size. The depth of the tree is at most \( \log_2 n \).
The best case is when the partitioning produces two subproblems each of size about \( \frac{n}{2} \). (Exercise 5.6 asks you to prove that no matter what the partition is, the best-case is \( O(n \log n) \) by showing that the solution of the above recurrence is \( \Omega(n \log n) \).) In other words, when we partition the subproblems into (almost) equal size. The reason is that this is the partition that keeps the tree balanced and leads to a recursion tree of depth \( \log n \) (See Figure 5.4 (bottom)). Note that this is the smallest depth possible for a binary tree of \( n \) nodes. Smaller the depth of the recursion tree, smaller the total number of comparisons, since the total number of comparisons in each level (except at the leaf level) is proportional to the total number of elements in that level (summing across all the nodes). The recurrence is \( T(n) \leq 2T(n/2) + n \), which gives \( T(n) = O(n \log n) \) by the DC Recurrence Theorem.

(Note that the \( O(n \log n) \) bound holds as long as the two subproblem sizes are within a constant factor of each other, e.g., see Example 2.7.)

From the above, it is clear that although the best case performance of QuickSort is \( \Theta(n \log n) \) (this is the best possible for any algorithm due to the existence of a lower bound of \( \Omega(n \log n) \) for any comparison-based sorting algorithm — See Exercise 5.7), its worst-case time is much worse compared to that of MergeSort (which has worst-case running time of \( O(n \log n) \)). Despite this, QuickSort is preferred widely in practice! We will examine this issue more closely when we analyze the average-case performance of QuickSort.

### Average-Case Analysis of QuickSort

In the last section we discussed the best-case and the worst-case analysis of QuickSort. Here we discuss the average-case analysis of QuickSort. But before that, we need to understand what “average-case” analysis of an algorithm means. This literally means: the performance of the algorithm averaged over all possible inputs for the algorithm. First let’s elaborate on this meaning for algorithms with a finite number of inputs which is the case for many algorithms, including QuickSort (as we will see below). For each input, find the running time of the algorithm, sum the running time over all the (finite) inputs and compute the average by dividing the sum by the total number of inputs — this gives the average-case performance of the algorithm.

Let’s explain the above with respect to QuickSort. First, it might appear
5.1. PROBLEM: SORTING

that the number of inputs for QuickSort is not finite. Indeed it is, because the only thing that matters for comparison-based algorithms such as QuickSort is the relative ordering of the elements, not their values. How many possible orderings are there for an input size of \( n \) elements: it is \( n! \), which is the total number of inputs for comparison-based algorithms such as QuickSort. Thus the average-case analysis of QuickSort is the average performance of QuickSort which is the average runtime (as usual, measured by the number of comparisons) over all the \( n! \) different inputs. Mathematically, we can write it as follows. Order the \( n! \) permutations of the input in some order and let \( i = 1, \ldots, n! \) denote the \( i \)th permutation of the input. Let \( T_i \) be the number of comparisons that QuickSort does on the \( i \)th permutation. Then the average run time of QuickSort, i.e., the average-case performance of QuickSort is defined as

\[
T_{avg} = \frac{1}{n!} \sum_{i=1}^{n!} T_i.
\]

How can we compute \( T_{avg} \) of QuickSort. This seems to be a non-trivial task as it seems to involve determining the number of comparisons of QuickSort on each of the \( n! \) permutations which is a huge number. Even if this is possible in principle, it seems difficult to compute this quantity, especially, analytically.

Surprisingly, this can be done by using some facts about probability and averages (see Chapter 8). The first is, computing \( T_{avg} \) is equivalent to choosing a random permutation, i.e., choosing a permutation uniformly at random, and computing the performance of QuickSort on that random permutation. This follows from the definition of “average” value of a set of values which is defined as simply as a weighted sum of the values, weighted by the respective probabilities of choosing the values. That is, let set \( S = \{s_1, s_2, \ldots, s_n\} \) of \( n \) values, and each element \( s_i \) be chosen with probability \( p_i \), then the average value is

\[
Average = \sum_{i=1}^{n} p_i s_i.
\]

Note that the sum of all the \( p_i \)'s should equal 1, i.e., \( \sum_{i=1}^{n} p_i = 1 \), since \( p_i \)'s form a probability distribution.

Coming back to QuickSort, the set \( S \) consists of \( n! \) values, where the

\(^1\text{Note that one can have non-comparison based algorithms for sorting which can make use of the actual values of the elements for sorting.}\)
ith value corresponds to the number of comparisons of QuickSort on the ith permutation. If each value (i.e., permutation) is chosen randomly, i.e., uniformly at random (in other words, each permutation is equally likely), then \( p_i = 1/n! \). Thus

\[
T_{\text{avg}} = \frac{1}{n!} \sum_{i=1}^{n!} T_i = \sum_{i=1}^{n!} \frac{1}{n!} T_i.
\]

Hence, to compute \( T_{\text{avg}} \), it is enough the compute the (average or expected) performance of QuickSort on a random permutation.

Analyzing the performance of QuickSort on a random input permutation (i.e., each permutation is equally likely to be the input), is called as average-case analysis or probabilistic analysis of QuickSort. This is discussed in Section 8.4.1. Here we simply state the result and refer the reader to the proof in Section 8.4.1.

**Theorem 5.3.** The average-case performance of QuickSort (i.e., Algorithm 15) is \( O(n \log n) \).

The above theorem says that on an average input QuickSort takes \( O(n \log n) \) comparisons. This is significantly better than the worst-case performance of \( \Theta(n^2) \). Thus, on average, i.e., for a significant number of inputs, QuickSort takes \( O(n \log n) \) comparisons. In Chapter 8, we show that, in fact, for most inputs QuickSort takes \( O(n \log n) \) time and only for a tiny fraction of inputs, the number of comparisons is much larger. In other words, “bad” inputs for QuickSort (e.g., input in increasing or decreasing order) constitute only a tiny fraction of the total possible number of all \( n! \) inputs. In practice, it turns out that such bad inputs show up much less often. Thus, in the real-world (where most inputs are likely to be good), QuickSort performs well.

### 5.2 Problem: Selection

**Problem 5.2 (Selection).** *Input: An array \( S \) of \( n \) elements (coming from a totally ordered domain, e.g., integers, real numbers, strings). Output: The \( k \)th-smallest element, i.e., the element with “rank” \( k \) in \( S \).*

Selection is also a basic problem. In particular, selecting the “middle” element, i.e., the median arises in many applications. Note that the median is the \( \lfloor n/2 \rfloor \) smallest element (if \( n \) is even, the \( \lfloor n/2 \rfloor \) smallest element is also a median). Clearly, we can find the median, or any ith smallest
element, in $O(n \log n)$ time via sorting. Thus the non-trivial question is whether one can do better, in particular, if one can solve the problem in linear, i.e., $O(n)$ time. Note that, as in sorting, we only use comparisons to infer the order between elements.

### A Linear Time Selection Algorithm

**Main idea behind the algorithm.** Assume that we are looking for the $k$th smallest element. The idea is “Conquer and Divide” and is very similar to the partitioning paradigm of the QuickSort algorithm. The input set $S$ is partitioned into two parts with respect to a pivot element: all elements less than the pivot go to the left (call this set $S_1$) and all elements greater than the pivot go to the right (call this $S_2$). (We assume that all elements in the input set are distinct.) If the size of $S_1$ is equal to $k - 1$, then clearly the pivot is the desired $k$th element. On the other hand, if the size of $S_1$ is bigger than $k - 1$, then clearly the desired element is in the set $S_1$; otherwise it will be in $S_2$, but its rank will be $k - |S_1| - 1$ (Why?). To achieve linear time, what property do we need regarding the size of the subproblem generated? Note that there will be only one subproblem (on set $S_1$ or $S_2$). To achieve linear time, we require subproblem size to go down by a constant factor. This will imply that the number of comparisons in each recursive level goes down by a constant factor leading to a linear time (number of comparisons) overall.

To ensure that the subproblem size goes down by a constant factor, the pivot has to be chosen carefully — it has to be chosen such that its rank is “somewhere in the middle”, i.e., within a constant factor of $n/2$ (the rank of the median). The algorithm uses a clever idea: it finds such a pivot by recursively solving another selection problem as follows. Partition the set $S$ into groups of 5 elements (the last group might have less than 5), e.g., the first 5 elements, the next 5 elements as so on. Find the median of each of these groups — there will be about $n/5$ medians; this is easy to find in linear time, since each group has only 5 elements. We now recursively find the median of these $n/5$ medians. It turns out that this “median of medians” is a good pivot, that partitions the overall set into approximately balanced sets. This can be shown by showing that a constant fraction of the number of the elements in the set are smaller and larger than this pivot (see proof of Theorem 5.5).

Algorithm 16 gives the pseudocode.
Algorithm 16 Select \((S, k)\). Selects the \(k\)-th smallest element in the set \(S\) (assume all elements are distinct).

1: \(n = |S|\)
2: if \(n \leq 1\) then
3:    Return \(S\)
4: Partition array \(S\) into \(\lfloor \frac{n}{5} \rfloor\) groups of 5 elements each, and a leftover group of up to 4 elements.
5: Find the median of each of the groups, let \(R\) be the set of these \(\lceil \frac{n}{5} \rceil\) values.
6: \(y = \text{Select}\left(R, \lceil \frac{|R|}{2} \rceil\right)\)
7: Compare all elements of \(S\) to \(y\). Let
   \[S_1 = \{x \in S \mid x < y\}\]
   \[S_2 = \{x \in S \mid x > y\}\]
8: if \(|S_1| = k - 1\) then
9:    Return \(y\)
10: else if \(|S_1| \geq k\) then
11:    Return \(\text{Select}(S_1, k)\)
12: else
13:    Return \(\text{Select}(S_2, k - |S_1| - 1)\)

Correctness

Theorem 5.4. Algorithm Select returns the correct value.

Proof. The correctness can be shown by induction. We leave it as an exercise for the reader (Exercise 5.8).

Run-time

Theorem 5.5. The run-time of Algorithm Select is \(O(n)\).

Proof. We first analyze the following: How many elements in \(S\) are larger than \(y\), the “median of medians” value computed in step 6 of the algorithm?

Excluding the leftover group, and the group that includes \(y\), in at least half of the remaining groups, there are at least three elements that are \(> y\). Thus, at least

\[3\left(\frac{1}{2}\lfloor \frac{n}{5} \rfloor - 2\right) \geq \frac{3n}{10} - 6\]

in \(S\) are greater than \(y\).
5.3. **Problem: Multiplying Two Numbers**

Similarly, at least $\frac{3n}{10} - 6$ elements in $S$ are $\leq y$. Thus, Select is called in step 6 with at most $\frac{7n}{10} + 6$ elements.

Let $T(n) = \text{run-time on sets of size } n$. Then, assuming that the number of comparisons in the top level of the recursion (i.e., in steps 4, 5, and 7) is bounded by $\alpha n$, for some constant $\alpha > 0$, we have:

$$T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil \right) + T\left(\frac{7n}{10} + 6\right) + \alpha n.$$  

We show that $T(n) \leq cn$ for some constant $c > 0$ by using induction. Assume $T(k) \leq ck$ for all $k < n$. Then, by induction hypothesis:

$$T(n) \leq c(n/5 + 1) + c(7n/10 + 6) + \alpha n$$
$$\leq 9cn/10 + 7c + \alpha n$$
$$\leq cn$$

if $n > 70$ and sufficiently large constant $c$ (compared to $\alpha$). (How much large should $c$ be?) Also, the base case (say, $n = 70$) can be satisfied by choosing a constant $c$ sufficiently large.  

5.3 **Problem: Multiplying two numbers**

We consider a very basic problem: Given two numbers $x$ and $y$ (say, in decimal or in binary), how fast can we multiply them? Let’s assume that $x$ and $y$ both have $n$ digits. We will assume that multiplying or adding two single digit numbers takes constant time. We want to minimize the total time to multiply two $n$-digit numbers. We know an algorithm for this problem that is taught in schools, namely “school multiplication.” This algorithm, multiplies each digit of $y$ with $x$, one by one, and then adds all the respective sums. Clearly this algorithm takes $\Theta(n^2)$ time, since it involves $n^2$ single digit multiplications and about $n^2$ single digit additions. Incidentally, the 9th century mathematician Al-Khwarizmi (the term “algorithms” is coming from his name) who systematically developed algorithms for the four arithmetic operations (in decimal system), had a different algorithm for multiplication which also needed $\Theta(n^2)$ operations. On the other hand, note that, crucially, adding two $n$-digit numbers takes only $O(n)$ time. We exploit this to design an improved algorithm using divide and conquer.
Main Idea of the Algorithm  Let $x$ and $y$ be two $n$-bit numbers. How can we use divide and conquer to multiply? One obvious way is to split the two numbers into two parts — the first (most significant) $n/2$ bits and the next (least significant) $n/2$ bits. As usual, we will assume $n$ to be a power of 2. Then we can write:

$$x = 2^{n/2}x_1 + x_2,$$

and

$$y = 2^{n/2}y_1 + y_2,$$

where $x_1(y_1)$ and $x_2(y_2)$ are the most significant bits and least significant bits of $x(y)$ respectively. Hence,

$$xy = 2^n(x_1y_1) + 2^{n/2}(x_1y_2 + x_2y_1) + x_2y_2.$$

Thus, we have reduced the problem of multiplying two $n$-bit numbers to multiplying four $n/2$-bit numbers ($x_1y_1, x_1y_2, x_2y_1,$ and $x_2y_2$) plus some constant number of additions and shifts (multiplying by power of 2). Note that additions and shifts take only $O(n)$ time. If $T(n)$ represents the number of bit operations for multiplying two $n$-bit numbers, then based on the above formula, we can write the recurrence:

$$T(n) = 4T(n/2) + O(n).$$

As usual, the base case is $T(1) = O(1)$. The solution to the above recurrence via the DC recurrence theorem is $\Theta(n^2)$, which is no better than the school multiplication. How to improve this?

The key idea is to reduce the number of multiplications at the cost of a few more additions and subtractions (costing $O(n)$ time total). The trick is to compute $x_1y_2 + x_2y_1$ using an idea that can be traced to Gauss:

$$(x_1 + x_2)(y_1 + y_2) = x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2.$$

Hence,

$$x_1y_2 + x_2y_1 = (x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2.$$

Thus, we can compute $x_1y_2 + x_2y_1$ using only three multiplications (two of which $x_1y_1$ and $x_2y_2$ are needed anyway!). Now we can write a better recurrence:

$$T(n) = 3T(n/2) + O(n).$$

(Strictly speaking, the above recurrence should be $T(n) \leq 3T(n/2 + 1) + O(n)$, since $(x_1 + x_2)(y_1 + y_2)$ can involve multiplying two $(n/2 + 1)$-bit
5.4. PROBLEM: MATRIX MULTIPLICATION

The solution to the above recurrence via the DC recurrence theorem is $\Theta(n^{\log_3 2}) = O(n^{1.585})$, which is significantly better than $\Theta(n^2)$.

There is an interesting story behind the above algorithm which is due to Karatsuba who was a student with Kolmogorov, one of the prominent mathematicians of the last century. Kolmogorov had conjectured that the school multiplication algorithm that had a complexity of $\Theta(n^2)$ is optimal and had called a seminar to discuss the computational complexity of multiplication among other problems. Soon after this, Karatsuba disproved the conjecture by discovering his algorithm which runs in $O(n^{1.585})$. Kolmogorov was excited by this result and he gave many lectures on this. He also wrote a paper attributing the result to Karatsuba who came to know about the published paper much later!

Karatsuba’s algorithm raises the question whether there are faster algorithms for integer multiplication. Indeed, we will see that one can design substantially faster algorithms using Fast Fourier Transform (FFT) that we will see in Section 5.5. The currently fastest known algorithm (after a long line of research) for multiplying two integers is $O(n \log n)$. One can achieve close to this bound using the FFT technique. It is conjectured that $\Omega(n \log n)$ is a lower bound for the complexity of multiplying two integers.

5.4 Problem: Matrix Multiplication

We consider another fundamental problem, this time involving matrices (which is a fundamental object we will see throughout the course — See Appendix F for basic facts on matrices): Given two $n \times n$ matrices $A$ and $B$ we want to multiply them, i.e., we want to compute $C = A \cdot B$. Note that $C$ is another $n \times n$ matrix whose elements are defined as:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}.$$  

In other words, the $(i,j)$th element of $C$ is computed by elementwise multiplication of the $i$th row of $A$ and the $j$th column of $B$ and adding the result of the $n$ multiplications. Thus $c_{i,j}$ can be computed using $O(n)$ operations (where one operation is either an addition or multiplication of two elements of the matrix), a straightforward algorithm takes $O(n^3)$ time to compute all elements of $C$. Note that, on the other hand, adding two $n \times n$ matrices, takes $O(n^2)$ time. We will exploit this crucially in
designing a faster divide and conquer algorithm.

### 5.4.1 A Divide and Conquer Algorithm

Suppose we are given two $n \times n$ matrices $A$ and $B$. Our approach is similar to the idea we used in multiplying two integers in Section 5.3. There we exploited that adding two integers is cheaper than multiplying; here we do the same. Let’s partition each matrix into two four submatrices of dimension $n/2 \times n/2$ (as usual, we will assume $n$ is a power of 2) as follows.

Let

\[
A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & g \\ f & h \end{pmatrix}
\]

Then the product $C = AB$ can be similarly be partitioned into four $n/2 \times n/2$ submatrices as:

\[
C = AB = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} r & s \\ t & u \end{pmatrix}
\]

where, the submatrices of $C$ can be computed as follows (Why?):

\[
\begin{align*}
    r &= ae + bf \\
    s &= ag + bh \\
    t &= ce + df \\
    u &= cg + dh
\end{align*}
\]

Thus to compute $r, s, t,$ and $u$, we need to perform a total of 8 matrix products each of which is a product of two $n/2 \times n/2$ matrices. Hence, a simple recursive algorithm gives

\[
T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2) = \Theta(n^3),
\]

using the DC recurrence theorem. Can we improve this?

To get a better running time, we use an idea that is very similar in spirit to the idea that was used to improve the performance of multiplying two $n$-bit numbers in Section 5.3. The observation is that adding (or subtracting) two matrices is cheaper than multiplying. Hence if we can reduce the number of multiplications from 8 to something smaller (say 7), at the cost
of a few more (a constant number) of additions/subtractions, then we can get a faster algorithm. This is exactly what Strassen’s algorithm does.

5.4.2 Strassen’s Algorithm

Strassen found a clever (and highly non-trivial) way to compute \( r, s, t \) and \( u \) using 7 matrix multiplications.

First we define 14 \( \frac{n}{2} \times \frac{n}{2} \) matrices \( A_1, ..., A_7 \) and \( B_1, ..., B_7 \) as shown below.

- \( A_1 = a, B_1 = g - h \).
- \( A_2 = (a + b), B_2 = h \).
- \( A_3 = c + d, B_3 = e \).
- \( A_4 = d, B_4 = f - e \).
- \( A_5 = (a + d), B_5 = (e + h) \).
- \( A_6 = (b - d), B_6 = (f + h) \).
- \( A_7 = (a - c), B_7 = (e + g) \).

Next we define the 7 matrix products \( P_i = A_i \cdot B_i \) as follows:

- \( P_1 = A_1 B_1 = a(g - h) = ag - ah \).
- \( P_2 = A_2 B_2 = (a + b)h = ah + bh \).
- \( P_3 = A_3 B_3 = (c + d)e = ce + de \).
- \( P_4 = A_4 B_4 = df - de \).
- \( P_5 = A_5 B_5 = (a + d)(e + h) = ae + ah + de + dh \).
- \( P_6 = A_6 B_6 = (b - d)(f + h) = bf + bh - df - dh \).
- \( P_7 = A_7 B_7 = (a - c)(e + g) = ae + ag - ce - cg \).

We can compute \( r, s, t \) and \( u \) by linear combinations (additions and subtractions) of \( P_1, ..., P_7 \) as follows.

- Computing \( s \): \( P_1 + P_2 = ag - ah + ah + bh = ag + bh = s \).
- Computing \( t \): \( P_3 + P_4 = ce + de + df - de = ce + df = t \).
5. MORE DIVIDE AND CONQUER

- Computing $r$: \[ P_3 + P_4 - P_2 + P_6 = ae + ah + de + dh + df - de - ah - bh + bf + bh - df - dh = ae + bf = r. \]

- Computing $u$: \[ P_5 + P_1 - P_3 - P_7 = ae + ah + de + dh + ag - ah - ce - de - ae - ag + ce + cg = cg + dh = u. \]

Thus, Strassen’s recursive algorithm to compute the product of $2n \times n$ matrices is:

- Compute recursively the product of $7 \frac{n}{2} \times \frac{n}{2}$ matrices $P_1, \ldots, P_7$.

- Perform $O(1)$ matrix additions and subtractions to compute $r, s, t, u$.

This leads to the recurrence:

\[ T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2) = \Theta\left(n^\log_2 7\right) = O(n^{2.81}) \]

using the DC recurrence Theorem.

5.5 Problem: Fourier Transform

We study a problem of great importance in science and engineering, namely, computing the Fourier transform. Fourier transform is named after Joseph Fourier who used it solve a physics problem in early 19th century. Fourier transform is an amazing insight that transforms (rather decomposes) functions in terms of much simpler functions (read sine and cosine) which are much easier to manipulate than the original function — this has tremendous applications in many areas such as signal processing, data compression, partial differential equations, multiplying large polynomials and numbers etc.

To illustrate the Fourier transform, we will focus on the application of multiplying two polynomials (which also arises a lot in signal processing).

**Problem 5.3.** Given two degree $d$ polynomials $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ and $B(x) = b_0 + b_1 x + \cdots + b_d x^d$, compute the product $C(x) = A(x)B(x) = c_0 + c_1 x + \cdots + c_{2d} x^{2d}$, where the coefficients

\[ c_k = a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0 = \sum_{i=0}^k a_i b_{k-i}, \]

(for $i > d$, set $a_i = b_i = 0$).
5.5. **PROBLEM: FOURIER TRANSFORM**

**Coefficient representation** The polynomial $A(x) = \sum_{i=0}^{d} a_i x^i$ is represented by the coefficient vector $a = (a_0, a_1, ..., a_d)$.

**Sum** As in the case of multiplying two integers, summing two polynomials is easier than multiplication; we will leverage this to get a faster way to multiplying two polynomials (as we did for multiplying integers and matrices).

Given two polynomials $A(x) = \sum_{i=0}^{d} a_i x^i$ and $B(x) = \sum_{i=0}^{d} b_i x^i$, their sum is

$$C(x) = A(x) + B(x) = \sum_{i=0}^{d} (a_i + b_i) x^i.$$  

The degree of $C(x)$ is the max degree of $A(x)$ and $B(x)$. The sum of two degree $d$ polynomials, given in a coefficient representation, can thus be computed in $O(d)$ time.

**Product** Given two polynomials $A(x) = \sum_{i=0}^{d} a_i x^i$ and $B(x) = \sum_{i=0}^{d} b_i x^i$, their product is

$$D(x) = A(x)B(x) = \sum_{i=0}^{2d} d_i x^i,$$

where, as mentioned earlier,

$$d_i = \sum_{k=0}^{i} a_k b_{i-k}.$$  

Thus, the coefficient vector $d$ is the convolution of vectors $a$ and $b$. The degree of $D(x)$ is the sum of the degrees of $A(x)$ and $B(x)$. If we compute $d_i$ from the above formula it takes $O(i)$ steps. Thus finding all $2d + 1$ coefficients can take $\Theta(d^2)$ steps. We will show how using the fast Fourier transform (FFT), we can find all the coefficients in $O(d \log d)$ time.

**Point-value representation** To multiply faster, we use a transformation from the “coefficient” representation of a polynomial (the traditional representation) to a “point-value” representation as follows. A degree $d$ polynomial $A(x) = \sum_{i=0}^{d} a_i x^i$ can be represented by a set of $d + 1$ pairs

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_d, y_d)\}$$

such that

- for all $i \neq j$, $x_i \neq x_j$ (i.e., all points are distinct).
• for every \( k \), \( y_k = A(x_k) \);

A key fact (Theorem 5.6) that facilitates the point-value representation is that a degree \( d \) polynomial is uniquely determined (i.e., all its \( d + 1 \) coefficients are fixed uniquely) by its values at any \( d + 1 \) distinct points. Thus we can specify polynomial \( A(x) \) either by its coefficient vector \( a = (a_0, a_1, \ldots, a_d) \) or by its values \( A(x_0), A(x_1), \ldots, A(x_d) \) at \( d + 1 \) distinct points.

Note that a polynomial can be evaluated at any \( x_i \), i.e., value \( A(x_i) \) can be computed in \( O(d) \) time by using the so-called Horner’s rule:

\[
A(x_i) = a_0 + x_i(a_1 + x_i(a_2 + \cdots + x_i(a_{d-1} + x_0a_d)))
\]

**Theorem 5.6.** For any set of \( d \) point value pairs \((x_i, y_i)\) (where \( x_i \)s are all distinct) there is a unique degree \( d-1 \) polynomial \( A(x) \) such that \( A(x_i) = y_i \) for all pairs.

**Proof.** We need to solve

\[
\begin{pmatrix}
1 & x_0 & x_0^2 & \cdots & x_0^{d-1} \\
1 & x_1 & x_1^2 & \cdots & x_1^{d-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_{d-1} & x_{d-1}^2 & \cdots & x_{d-1}^{d-1}
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
\vdots \\
a_{d-1}
\end{pmatrix}
=
\begin{pmatrix}
y_0 \\
y_1 \\
\vdots \\
y_{d-1}
\end{pmatrix}
\]

The \( d \times d \) matrix (call it \( M \)) that appears on the left side of the above equation is called the Vandermonde matrix and has nice properties. In particular, the determinant of the Vandermonde matrix is \( \Pi_{j<k} (x_k - x_j) \) (Exercise 5.13). Thus, if all the \( x_i \)’s are distinct, the matrix is nonsingular (i.e., it is invertible) and the linear system has a unique solution. That is, the (unique) coefficient vector is given by: \( a^T = M^{-1}y^T \), where \( a^T \) and \( y^T \) are the column vectors of the row vectors \((a_0, \ldots, a_{d-1})\) and \((y_0, \ldots, y_{d-1})\) respectively.

**Multiplication in the point-value representation** Given two polynomials in (same) point value representation \( \{(x_0, y_0^1), (x_1, y_1^1), \ldots, (x_d, y_d^1)\} \) and \( \{(x_0, y_0^2), (x_1, y_1^2), \ldots, (x_d, y_d^2)\} \), the sum of two degree \( d \) polynomials in point value representation is computed in \( O(d) \) time:

\[
\{(x_0, y_0^1 + y_0^2), (x_1, y_1^1 + y_1^2), \ldots, (x_d, y_d^1 + y_d^2)\}
\]

To compute the product of two degree \( d \) polynomials we need an “extended” point value representation of \( 2d + 1 \) points (Why?). Given such
5.5. **PROBLEM: FOURIER TRANSFORM**

A representation, the product of two polynomials in point value representation can again be easily computed in $O(d)$ time:

$$\left\{ (x_0, y_0^1 y_0^2), (x_1, y_1^1 y_1^2), \ldots, (x_{2d+1}, y_{2d+1}^1 y_{2d+1}^2) \right\}.$$

**Fast Polynomial Multiplication via FFT** The idea to compute the product of two degree $d$ polynomials in coefficient representation is as follows:

1. Evaluate the polynomials in $2d + 1$ (distinct) points to create an extended $2d + 1$ point value representation of the polynomials.
2. Compute the product of the two polynomials in $O(d)$ time.
3. Convert the point value representation of the product back to coefficient representation.

We show that using the FFT method (1) and (3) can be done in $O(d \log d)$ time. Henceforth we will focus on solving the following problem (called as Evaluation) using FFT:

**Problem 5.4** (Evaluation of a polynomial). *Given a degree $n-1$ polynomial (in coefficient form), evaluate its value at $n$ distinct points.*

Later we will focus on the "opposite" problem (called as Interpolation):

**Problem 5.5** (Interpolation of a polynomial). *Given the values of a degree $n-1$ polynomial at $n$ distinct points, find the coefficient vector of the polynomial.*

**Key idea of FFT** The key idea of FFT is to evaluate a degree $n-1$ polynomial, say,

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_i x^i + \cdots + a_{n-1} x^{n-1},$$

not at some arbitrary set of $n$ distinct points, but rather at a very special set of $n$ distinct points which have a peculiar property. For the following discussion, we will assume throughout that $n$ is a power of 2 (this assumption can be easily removed – how?). To understand the idea, let’s consider evaluating $A(x)$ at two points, say $x_0$ and $-x_0$. We will first split $A(x)$ into even and odd powers:

$$A^0(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}$$

$$A^1(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$$
Then

\[ A(x) = A^0(x^2) + xA^1(x^2) \]

Thus, \( A(x_0) = A^0(x_0^2) + xA^1(x_0^2) \) and \( A(-x_0) = A^0(-x_0^2) - xA^1(-x_0^2) \). Note that the above two values involve a lot of overlapping computations: we need to perform evaluations of two polynomials, namely \( A^0(x') \) and \( A^1(x') \) at the point \( x' = x_0^2 \). This is exactly a divide and conquer type operation, assuming that the \( n \) distinct points come in pairs, i.e., \( \pm x_0, \pm x_1, \ldots, \pm x_{n/2} \).

Note that we need this trend to continue in each level of the recursion, and not just at the topmost level: in other words, we need the \( n/2 \) points in the next level, namely, \( x_0^2, x_1^2, \ldots, x_{n/2}^2 \) to also come in \( \pm \) pairs. What special numbers can guarantee this property: enter complex numbers, in particular, the \( n \) complex square roots of unity! See Figure 5.5.

**Complex roots of unity** A complex number \( w \) is the \( n \)-th root of unity if

\[ w^n = 1 \]

There are \( n \) complex \( n \)-th roots of unity given by

\[ e^{2\pi ik/n} \quad \text{for} \quad k = 0, \ldots n - 1 \]

were \( e^{iu} = \cos(u) + i \sin(u) \) and \( i = \sqrt{-1} \).

The principal \( n \)-th root of unity is

\[ w_n = e^{2\pi i/n} \]

the other roots are powers of \( w_n \), i.e., \( w_n^2, w_n^3, \ldots, w_n^{n-1}, w_n^n = 1 \). Exercise 5.14 asks you to show that that 1, \( w_n, w_n^2, w_n^3, \ldots, w_n^{n-1} \) are the \( n \) distinct complex \( n \)-th roots of unity.

**Operations on the roots of unity** For any \( j \) and \( k \):

\[ w_n^k w_n^j = w_n^{j+k} \]

Since \( w_n^n = 1 \)

\[ w_n^k w_n^j = w_n^{j+k} = w_n^{(j+k) \mod n} \]

and

\[ w_n^{-k} = w_n^{n-k} \]
5.5. PROBLEM: FOURIER TRANSFORM

DFT  The Discrete Fourier Transform (DFT) of a coefficient vector \( a = (a_0, a_1, \ldots, a_{n-1}) \) is a vector \( y = (y_0, y_1, \ldots, y_{n-1}) \) such that

\[
y_k = A(w_n^k) = \sum_{j=0}^{n-1} a_j w_n^{kj}.
\]

and we can write the vector \( y \) as:

\[
y = DFT_n(a).
\]

Using Fast Fourier Transform (FFT) we can compute \( DFT_n(a) \) in \( O(n \log n) \) steps, instead of \( O(n^2) \).

FFT  Assume that \( n \) is a power of 2 (otherwise complete to the nearest power of 2).

Given the polynomial \( A(x) = \sum_{j=0}^{n-1} a_j x^j \) we define two polynomials

\[
A[0](x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}
\]

\[
A[1](x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}
\]

Then

\[
A(x) = A[0](x^2) + x A[1](x^2)
\]

To compute \( DFT_n(a) \) (in \( n \) points) we need to compute the polynomials \( A[0](y) \) and \( A[1](y) \) in \( n/2 \) points

\[
(w_n^0)^2, (w_n^1)^2, \ldots, (w_n^{n-1})^2
\]

**Theorem 5.7.** The set \((w_n^0)^2, (w_n^1)^2, \ldots, (w_n^{n-1})^2\) contains only \( n/2 \) distinct points. In other words, the \( n \) complex \( n \)th roots of unity come in positive-negative pairs. In particular, \( w_n^{k+n/2} = -w_n^k \) and \((w_n^{k+n/2})^2 = (w_n^k)^2 = w_n^{k} \).

**Proof.** We’ll show that the squares of \( n \) complex \( n \)-th roots of unity are the \( n/2 \) complex \( n/2 \)-th roots of unity. Assume that \( k \leq \frac{n}{2} \).

\[
(w_n^k)^2 = (e^{2\pi ik/n})^2 = e^{(2\pi ik)/(n/2)} = w_n^{k/2}
\]
Figure 5.5: Showing the 8th-roots of unity and how the recurrence proceeds on them. The top level represents the 8th-roots of unity, the second level 4th-roots of unity, and the third level the square roots of unity. At each level, the roots are grouped into positive-negative pairs; for each pair only one of them need to be solved in the next level of recursion. From the bottom, each level is obtained by taking successive square roots. At the top level we evaluate the polynomial at the 8 points corresponding to the 8th-roots of unity. This is accomplished by evaluating at the polynomial (recursively) at the 4 points corresponding to the 4th-roots of unity and so on.
Thus, \( w_n^{k+n/2} = -w_n^k \).

Computing the \( \text{DFT}_n(a) \) is reduced to (see Algorithm 17 for pseudocode):

1. Computing two \( \text{DFT}_{n/2} \).

2. Combining the results:

Given \( y_k^0 = A^0(w_n^{k+n/2}) = A^0((w_n^k)^2) \) and \( y_k^1 = A^1(w_n^{k+n/2}) = A^1((w_n^k)^2) \),
for \( k \leq n/2 \). For \( k < n/2 \), by Theorem 5.7, \( w_n^{k+n/2} = -w_n^k \), \( y_k^0 = y_k^1 \), and \( y_k^{k+n/2} = y_k^1 \). Thus,

\[
\begin{align*}
y_k &= y_k^0 + w_n^k y_k^1 \\
y_{k+n/2} &= y_k^0 - w_n^k y_k^1
\end{align*}
\]

**Algorithm 17** Algorithm \( \text{DFT}(a_0, a_1, a_2, \ldots, a_{n-1}, n) \). Input: \( n \) (a power of 2) and a vector \( a = (a_0, \ldots, a_{n-1}) \). Output: a vector \( y = (y_0, \ldots, y_{n-1}) \) such that \( y_k = \sum_{i=0}^{n-1} a_i (w_n^i)^k \).

1: if \( n = 1 \) then
2: Output \( a_0 \)
3: else
4: \( w_n = e^{2\pi i/n} \)
5: \( (b_0, b_1, \ldots, b_{n/2-1}) = \text{DFT}(a_0, a_2, \ldots, a_{n-2}, n/2) \)
6: \( (c_0, c_1, \ldots, c_{n/2-1}) = \text{DFT}(a_1, a_3, \ldots, a_{n-1}, n/2) \)
7: for \( k := 0 \) to \( n/2 - 1 \) do
8: \( y_k = b_k + w_n^k c_k \)
9: \( y_{k+n/2} = b_k - w_n^k c_k \)
10: Output \( y = (y_0, y_1, \ldots, y_{n-1}) \)

Hence, we have the recurrence

\[
T(n) = 2T(n/2) + O(n)
\]
which yeilds the solution $T(n) = O(n \log n)$.

**Theorem 5.8.** A point value representation of a $n$th degree polynomial given in a coefficient representation can be generated in $O(n \log n)$ time.

**From point-value to coefficient representation: Interpolation**  Given the DFT $y = (y_0, \ldots, y_{n-1})$ of a degree $n$ polynomial we want to generate the coefficient representation $a = (a_0, \ldots, a_{n-1})$ of the polynomial.

We need to solve

$$
\begin{pmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & w_n & w_n^2 & \cdots & w_n^{n-1} \\
1 & w_n^2 & w_n^4 & \cdots & w_n^{2(n-1)} \\
1 & w_n^3 & w_n^6 & \cdots & w_n^{3(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & w_n^{n-1} & w_n^{2(n-1)} & \cdots & w_n^{(n-1)(n-1)}
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
\vdots \\
a_{n-1}
\end{pmatrix}
= 
\begin{pmatrix}
y_0 \\
y_1 \\
y_2 \\
\vdots \\
y_{n-1}
\end{pmatrix}
$$

or $y = V_n a$. We solve for $a$: $a = V_n^{-1}y$. It turns out that the entries of matrix $V_n^{-1}$ can be easily calculated as shown by the following theorem.

**Theorem 5.9.** The $(i, j)$ entry in $V_n^{-1}$ is $\frac{w_n^{ij}}{n}$.

**Proof.** We show that $V_n^{-1}V_n = I_n$:

The $(j, j')$ entry of $V_n^{-1}V_n$

$$
[V_n^{-1}V_n]_{j,j'} = \sum_{k=0}^{n-1} \frac{w_n^{-kj}}{n} (w_n^{kj'})
= \frac{1}{n} \sum_{k=0}^{n-1} w_n^{-k(j-j')}
$$

If $j = j'$ the summation is 1.
If \( j \neq j' \)

\[
\sum_{k=0}^{n-1} w_n^{-k(j-j')} = \sum_{k=0}^{n-1} (w_n^{j-j'})^k \\
= \frac{(w_n^{j-j'})^n - 1}{w_n^{j-j} - 1} \\
= \frac{(w_n^{n})^{j-j} - 1}{w_n^{j-j} - 1} \\
= \frac{(1)^{j-j} - 1}{w_n^{j-j} - 1} \\
= 0.
\]

Thus, we need to compute

\[
a_i = \frac{1}{n} \sum_{k=0}^{n-1} y_k w_n^{-ki}.
\]

Note that this is similar to the evaluation algorithm, but instead of using \( w_n = e^{2\pi i/n} \), we use \( w_n^{-1} = e^{-2\pi i/n} \) as the “principal” root of unity. This can again be computed by using the FFT algorithm in \( O(n \log n) \) time. Note that we have to divide by a factor of \( 1/n \) to get the final answer.

**Theorem 5.10.** Given a point value representation of an \( n \) degree polynomial in \( n \)th roots of unity, the coefficient representation of that polynomial can be computed in \( O(n \log n) \) time.

Since we can do both evaluation and interpolation in \( O(n \log n) \) time, we have the following theorem.

**Theorem 5.11.** The product of two \( n \)-degree polynomials can be computed in \( O(n \log n) \) time.

### 5.6 Worked Exercises

**Worked Exercise 5.1.** Given two SORTED (in ascending order) arrays \( A[1..n] \) and \( B[1..n] \), each containing \( n \) numbers, give an \( O(\log n) \)-time algorithm (counting the number of comparisons) to find the median of all the \( 2n \) elements in arrays \( A \) and \( B \).

**Solution:**
We need to find the median of the combined arrays $A$ and $B$, in other words, we want to find the $n$th smallest of the $2n$ combined elements.

The main idea is to compare the median element of $A$ to the median element of $B$: If the former is greater, then the elements in the larger half of $A$ can be thrown away (for being too big — their ranks exceed $n$), and the elements in the smaller half of $B$ can also be thrown away (for being too small — their ranks are surely smaller than $n$); in other words we recurse on the smaller half of $A$ and the larger half of $B$. If it is the median of $B$ that is greater than the median of $A$ then we simply interchange the roles of $A$ and $B$ in the previous sentence. The reason it is correct to recurse in this way is that the element we are searching for has, in the recursive call, “lost” $n/2$ elements smaller than it and $n/2$ elements greater than it (the halves of $A$ and $B$ that were “thrown away”), and therefore its rank in the reduced-size problem is $n - (n/2) = n/2$. A more formal exposition is given below.

We denote the $i$th element in set $A$ as $A[i]$ and the $i$th element in set $B$ as $B[i]$. We will assume that $n$ is a power of 2 (the algorithm can be extended if this is not the case).

We define $A_1 = \{A[1], A[2], ..., A[n/2-1]\}$ and $A_2 = \{A[n/2+1], ..., A[n]\}$ as $A_2$. Notice that $A = A_1 \cup \{A[n/2]\} \cup A_2$. Similarly, we can define $B_1 = \{B[1], B[2], ..., B[n/2-1]\}$ and $B_2 = \{B[n/2 + 1], ..., B[n]\}$.

We first compare the median number of the two sets, i.e., $A[n/2]$ and $B[n/2]$. There are three possible results from the comparison:

1. $A[n/2] = B[n/2]$. In this case, the $n$th smallest element of all $2n$ elements is $A[n/2]$. This is based on the observation that $n - 1$ elements (i.e., any element in $A_1$ or $B_1$, and $B[n/2]$) are no greater than $A[n/2]$ and all other elements (i.e., any element in $A_2$ or $B_2$) are no less than $A[n/2]$.

2. $A[n/2] > B[n/2]$. In this case, since any element in $A_2$ is no less than at least $n$ elements (i.e., any element in $A_1$ or $B_1$, and $B[n/2]$), it cannot be the $n$th smallest element. Similarly, since any element in $B_1$ and $B[n/2]$ is no greater than at least $n+1$ elements (i.e., any element in $A_2$ or $B_2$, and $A[n/2]$), it cannot be the $n$th smallest element. Therefore, the $n$th smallest element in all $2n$ elements should be the $n/2$-th smallest element in $A_1 \cup \{A[n/2]\}$ and $B_2$, both of which are of size $n/2$.

3. $A[n/2] < B[n/2]$. Using similar argument, we can prove that the $n$th smallest element in all $2n$ elements should be the $n/2$-th smallest
element in $A_2$ and $B_1 \cup \{ B[n/2] \}$, both of which are of size $n/2$.

In case 1, the $n$th smallest element can be found in constant time. In either case 2 or case 3, we reduced the problem to the same problem with 1/2 size. There is constant work for the transformation (only one comparison of the median elements of the two sets). Let the time for solving the problem to be $T(n)$. Then, based on the above argument, we have

$$T(n) = T(n/2) + 1$$

Using the recurrence theorem, $T(n)$ is thus $O(\log n)$.

Worked Exercise 5.2. Given $n$ points on the $x$-axis line (in some arbitrary order), find the closest pair of points in $O(n \log n)$ time.

- Show that the problem can be solved using sorting.

- Give a divide and conquer algorithm that does not use sorting. (The advantage of this approach is that it generalizes to points in higher dimensions, e.g., 2 dimensions). (Hint: Partition the points into two subproblems of equal size, recursively solve the subproblems, and efficiently combine them to solve the original problem; exploit the fact that the recursion returns the closest pair in the respective subproblems.)

Solution:

(a) The problem can be solved by sorting the points and by examining the distance between successive pairs of points and taking the pair with the minimum distance. Clearly, the closest pair will occur consecutively in the sorted order.

(b) The main idea is to use the Selection algorithm that we learnt in class to partition the problem into two equal parts. In particular, we call Select algorithm (Algorithm 16) and find the median of the points. The median will lie (approximately) in the middle and we partition the points about the median, i.e., all points smaller than the median will be to the left of the median and the rest of the points to the right. Let $L$ be the set of points to the left of the median and $R$ be the set of points to the right of the median. (Note that the median point is not included in $R$ and $S$). Recursively solve the closest pair on two subproblems of (almost) equal size. When the recursion returns, the respective closest pair of points in both $R$ and $S$ are returned. (The base/termination case is when there are just two points, in which case, we just return them). Compare the distances between
these two pairs, i.e., the closest pair of the left and the closest pair of the
right and take the minimum among them. Of course, this may not be the
closest pair of the entire set, because, we have not compared the median
with the rest. Compare the distance between the median and each of the
other points to find if there is a closer pair. Note that we don’t have to
compare any other pair (i.e., between points in the left and right of median)
— Why?

Note that the above algorithm does not involve sorting.
The recurrence is $T(n) \leq 2T(n/2) + O(n)$ since the total cost of finding
the median, partitioning, and the final checking of all points with the
median is $O(n)$.

The solution to the above recurrence is $O(n \log n)$, by the recurrence
theorem.

It is easy to write a pseudocode based on the above description (left to
you to do).

**Worked Exercise 5.3.** We are given a sequence of $n$ positive numbers $a_1, \ldots, a_n$
and a fixed number $k > 0$. We want to find a pair of numbers $a_i$ and $a_j$ such
that $j > i$ and $a_j - a_i = k$. If there is no such pair we want to output “false”.
This can be solved easily in $\Theta(n^2)$ time by examining all pairs of numbers.
Your task is to give a divide and conquer algorithm that runs in $O(n \log n)$
time. Show the correctness of your algorithm and the analysis of its running
time.

**Solution:**

The main idea is to use the divide and conquer strategy that is used in
MergeSort. Let the time to solve this problem and at the same time sort the
sequence $S$ of numbers to be $T(n)$, where $n$ is the number of numbers in
the sequence. Note that we do the “extra work” of sorting, although this is
not required for the solution; however, sorting is important to get $O(n \log n)$
run time.

Using divide and conquer, we divide the sequence into two sub-sequences
$S_1$ (the first half) and $S_2$ (the second half), each of which is of length $n/2$
(assume, without loss of generality, that $n$ is even).

We solve recursively the two sub-sequences. Note by our assumption,
when the algorithm returns, it also leaves the sequence sorted (besides the
answer to the subproblem). If such a pair exists in either $S_1$ or $S_2$ we return
it. If not, we have to find whether their exists a pair $x, y$, such that $y - x = k$,
and $x \in S_1$ and $y \in S_2$ (note that the index of all elements in $S_2$ will be
greater than $S_1$). We also note that when the recursion returns $S_1$ and $S_2$
are sorted. Using the fact that $S_1$ and $S_2$ are sorted, we can find whether
such a pair \( x, y \) exists in \( O(n) \) time.

Let \( S_1 = x_1, x_2, \ldots, x_{n/2} \) and \( S_2 = y_1, y_2, \ldots, y_{n/2} \). We start by examining the pairs \( x_i \) and \( y_1 \). Let \( x_i \) and \( y_j \) be the currently examined pair.

- If \( (y_j - x_i) = k \), then we return the pair.
- If \( (y_j - x_i) > k \), then we examine the next pair \( x_{i+1} \) and \( y_j \). (If \( i = n/2 \) then return "false").
- If \( (y_j - x_i) < k \), then we examine the next pair \( x_i \) and \( y_{j+1} \). (If \( j = n/2 \) then return "false").

Note that the above process is very similar to the merging process in MergeSort. In fact, at the same time, we also merge \( S_1 \) and \( S_2 \) to get a sorted sequence \( S \) (as in mergesort). Since each time we advance to the next number in at least one sub-sequence and the total length is \( n \), the time to search for the pair is \( O(n) \). Merging takes \( O(n) \) time as well.

Therefore, we have

\[
T(n) = 2T(n/2) + O(n)
\]

Based on the DC recurrence theorem, \( T(n) = O(n \log n) \).

Note that the formal pseudocode is very similar to that of MergeSort as well (please do it!).

### 5.7 Exercises

**Exercise 5.1.** Consider the array \( S \) consisting of 8 elements in the following order: 8, 2, 6, 7, 5, 1, 4, 3. Find the total number of comparisons that each of the following algorithms take on the above input:

- SimpleSort (Algorithm 11)
- MergeSort (Algorithm 12)
- Quicksort (Algorithm 15)

Show the steps of the algorithm when calculating the number of comparisons.

**Exercise 5.2.** Show the correctness of MergeSort using mathematical induction. Assume that the Merge procedure is correct.
Exercise 5.3. Show that the pseudocode Merge correctly implements the Merge procedure. (Hint: Use Induction to argue that each element, starting from the first element, is added in the Temp array in the correct order.)

Exercise 5.4. Show the correctness of Quicksort by using mathematical induction.

Exercise 5.5. Show that QuickSort can take $\Theta(n^2)$ time for sorting a set of $n$ elements and this is the worst possible.

Exercise 5.6. Show that every input for the quicksort algorithm takes $\Omega(n \log n)$ time. (Hint: Show that the solution of the recurrence for the best case of QuickSort given in proof of Theorem 5.2 is $\Omega(n \log n)$ by using induction.)

Exercise 5.7. Show that any comparison-based sorting algorithm (i.e., an algorithm that uses only comparisons between pairs of elements to infer the ordering) needs $\Omega(n \log n)$ comparisons (in the worst-case). (Hint: Represent the execution of a comparison-based sorting algorithm as a decision tree. A decision tree is a binary tree, where each node represents a comparison between two elements, say $x_i$ and $x_j$; the left child is taken by the algorithm if $x_i < x_j$, otherwise the right child is taken. Argue that the number of leaves of this decision tree should be $n!$. Calculate a lower bound on the the depth of this tree (hence the number of comparisons needed).

Exercise 5.8. Show the correctness of Algorithm Select.

Exercise 5.9. Given three SORTED (in ascending order) arrays $A[1..n], B[1..n], \text{ and } C[1..n]$, each containing $n$ numbers, give an $O(\log n)$-time algorithm (again, counting the number of comparisons) to find the $n$th smallest number of all $3n$ elements in arrays $A, B, \text{ and } C$.

Exercise 5.10. Show how to modify QuickSort algorithm so that it runs in worst case $O(n \log n)$ time. Write the pseudocode of the modified algorithm and analyze its worst-case run time. (Hint: The modification is choosing the pivot in a better way. How?)

Exercise 5.11. Write the complete pseudocode for the divide and conquer algorithm discussed above for multiplying two integers (don’t forget the base case!).

Exercise 5.12. Show that the solution to the recurrence $T(n) \leq 3T(n/2 + 1) + O(n)$ (with constant base case) is $T(n) = O(n \log_2 3)$.

Exercise 5.13. Show that the determinant of the Vandermonde matrix as given above is $\prod_{j<k}(x_k - x_j)$. (Hint: Use induction.)
5.7. EXERCISES

Exercise 5.14. Show that $1, w_n, w_n^2, \ldots, w_n^{n-1}$ are the $n$ distinct complex $n$-th roots of unity.

Exercise 5.15. Suppose you are working for a company that enforces piracy laws. You are given a set of $n$ CDs and your task is to find whether at least $n/2$ of the set are the same (this is considered a potential piracy). The only tool you have is a machine that can answer the following type of query: given two CDs, are they the same or not? Your goal is to design a divide and conquer algorithm that takes $O(n \log n)$ queries. (Hint: (1) To have at least $n/2$ identical CDs in the set, at least one of the halves of the set should contain at least $n/4$ of them. (2) When you solve your subproblems you may want to return more than just saying that are “enough CDs of one kind”.)

Exercise 5.16. You are given an $n \times n$ matrix represented as a 2-dimensional array $A[][]$ (there are totally $n^2$ elements). Each row of $A$ is sorted in increasing order and each column of $A$ is sorted in increasing order. (Other than these properties, the elements of $A$ are arbitrary.) Your goal is to find whether some given element $x$ is in $A$ or not. Note that you can do only comparisons between elements.

- Give an algorithm that takes $O(n)$ comparisons.
- Can we solve the problem using asymptotically lesser number of comparisons, i.e., in $o(n)$ comparisons?

Exercise 5.17. We are given an array $A$ containing $n$ numbers, and an integer $k$ that divides $n$. We want to find $k$ numbers $x_1, x_2, \ldots, x_k$ of $A$ such that $x_i$ is the $(\frac{n}{k} i)$-th smallest element of $A$. Develop a divide and conquer algorithm that finds the $x_i$’s that runs in time $O(n \log k)$. Prove its correctness and running time.

Exercise 5.18. Consider the max-sum problem: Given an array $A$ of size $n$ containing positive and negative integers, determine indices $i$ and $j$, $1 \leq i \leq j \leq n$, such that $A[i] + A[i+1] + \ldots + A[j]$ is a maximum.

(a) Consider the array $A = [5, 10, -15, 20, -4, 6, 4, 8, -10, 20]$. Find the indices $i$ and $j$ that give the maximum sum and state the maximum sum.

(b) Give a divide and conquer algorithm that runs in $O(n \log n)$ time. Assume, that adding or comparing two numbers takes constant time.

(Hint: Split the array into two (almost) equal parts and recursively solve the problem on the two parts. The non-trivial part is combining the solutions — note that it is possible that the indices $i$ and $j$ that give the optimal sum might be on opposite parts.)
Exercise 5.19. Recall that the Fibonacci numbers are defined by the following recurrence: \(F_0 = 0, F_1 = 1\) and \(F_n = F_{n-1} + F_{n-2}\) for \(n \geq 2\).

1. Using the above recurrence, give a simple algorithm to compute \(F_n\), given \(n\). The complexity of the algorithm in terms of the number of arithmetic operations should be \(O(n^2)\). Show that the algorithm indeed has this complexity. Recall that an arithmetic operation is the cost of adding or multiplying two single digit numbers. (Hint: How many digits are there in \(F_n\)? Use the fact that \(F_n = \Theta((\frac{1+\sqrt{5}}{2})^n)\).

2. Our goal is to use the following matrix identity to compute the \(n\)th Fibonacci number more efficiently.

\[
\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}
\] (5.1)

(a) Show that

\[
\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}
\] (5.2)

(b) Show that \(O(\log n)\) matrix multiplications are enough to compute \(F_n\).

(c) Using the fast algorithm for multiplying two integers (Karatsuba’s algorithm) show that \(F_n\) can be computed in \(O(n^{\log_2 3} \log n)\) arithmetic operations.

(d) Can you improve your analysis of (c) and show a cost of \(O(n^{\log_2 3})\) arithmetic operations?
6 Dynamic Programming

In the previous chapters, we studied the technique of divide and conquer which gives efficient algorithms for many problems. In this chapter, we focus on another powerful technique for solving problems called dynamic programming (DP).

6.1 Divide and Conquer vs. DP

As we saw in divide and conquer, the basic strategy in dynamic programming is again breaking problems into subproblems and combining their solutions to obtain the solution to the original problem. However, there are differences between the two strategies. While in divide and conquer, the problem is decomposed into independent and disjoint subproblems, in dynamic programming, it is almost always decomposed into independent and overlapping subproblems. Independent means the solution to one subproblem does not affect the solution to another subproblem. Thus the solution to each subproblem can be computed independently of one and another. Disjoint means that each subproblem is different; in many examples, the subproblems operate on different parts of the input and hence are naturally disjoint. For example, in mergesort, a divide and conquer algorithm for sorting (Section 5.1.2), the input array is partitioned into two (almost) equal and disjoint subarrays and the subproblem is solved in each subarray recursively. Since the subarrays are disjoint, the subproblems are disjoint. In other cases, the subproblems may share some input, but they are different subproblems. For example, in the linear-time selection algorithm (Section 5.2), the subproblem of finding the “median of medians” shares some input values with the other two subproblems one of either side of the pivot (note that only one of these will actually
execute); however, the subproblems are different and hence disjoint.

On the other hand, a key ingredient of dynamic programming is that the subproblems are overlapping, i.e., the same subproblem occurs in the solution of other subproblems. A main point of dynamic programming is to solve each subproblem only once (there is no point in solving the same subproblem again and again which is inefficient!). We will illustrate the technique with various examples throughout this chapter.

Dynamic programming (DP) is generally a more difficult technique than divide and conquer. The main reason is that, in many applications of DP, the difficulty is in figuring out how to decompose a given problem into subproblems. Thus the nontrivial work in a DP solution is figuring out this decomposition.

The best way to state the decomposition is by using a recursive formula. The recursive formula gives the solution of the problem in terms of the solutions of its subproblems. It is easy to show the correctness of the recursive formula by using induction. Translating the recursive formula into a pseudocode is also fairly straightforward — the only care that has to be taken is that each (distinct) subproblem has to be solved exactly once since the same subproblem can arise in the solution of other subproblems, i.e., overlapping subproblems. Again, we will see how to do this translation with the help of various examples. The good news is that once we have the correct recursive formulation, an efficient implementation (that solves each subproblem only once) is rather standard and “mechanical” to write. However, coming with a correct recursive formulation (that leads to a correct and efficient algorithm) is usually an “art”. Nevertheless, there are a few “patterns” in coming with this recursive formulation which occurs repeatedly in many problems. Studying these patterns will help in designing DP algorithms. We will see these patterns in this chapter.

### 6.2 Optimization problems

Dynamic programming is especially a powerful technique for solving optimization problems. In optimization problems (very much like in search problems that we saw in Chapter 2) the goal is to find (search for) an optimum (e.g., maximum or minimum) solution to a problem specified on an input domain. Usually the problem comes with some constraints and the goal is to find an optimal solution that satisfies the problem’s constraints. For example, consider the max-sum problem (see Exercise 5.18) stated as follows:
Definition 6.2.1 (Maxsum Problem). Given an array $A$ of size $n$ containing positive and negative integers, the goal is to determine indices $i$ and $j$, $1 \leq i \leq j \leq n$, such that $A[i] + A[i+1] + \ldots + A[j]$ is a maximum. This is an optimization problem since we want to optimize, i.e., maximize, some value (i.e., the sum) over an input domain. The input domain for this is the set of all pair of indices $i$ and $j$, $1 \leq i \leq j \leq n$, with each pair giving rise to a sum. Thus there are $n^2$ such sums. Note that all these sums are “potential” solutions for the problem (the optimum is the one which is the maximum sum). These are called feasible solutions. Feasible solutions are those that satisfy the constraints of the problem. Here the constraint is that the array sum has to be over a contiguous subarray, i.e., it includes all array values between indices $i$ and $j$ (including $A[i]$ and $A[j]$). It is easy to see that the problem can be solved by calculating all the possible $n^2$ array sums and taking the maximum among those. This takes at least $n^2$ operations; actually, even more since to compute a particular sum $A[i] + \ldots + A[j]$ requires $j - i$ addition operations. It is not difficult to calculate that the total number of addition operations over all the $\Theta(n^2)$ array sums is $\Theta(n^3)$. Using divide and conquer (Exercise 5.18), one can design an (somewhat complicated) algorithm that takes $O(n \log n)$ operations. We will see that using DP we can come with a simple and elegant $O(n)$ time algorithm which is also easy to implement. Thus DP generally gives rise to efficient algorithms for problems that otherwise are inefficient to solve.

6.3 Illustrating the DP Technique using the Maxsum Problem

We will illustrate the key steps involved in solving a problem using DP by using the maxsum problem as an example.

Step 1: Understand the problem. As discussed in Section 2.2.1, this is crucial for solving any problem. In this step, we figure out the input and output of the problem and understand the constraints (if any). In the maxsum problem, the input is an array of values and the output is a contiguous subarray, i.e., $A[i], A[i+1], \ldots, A[j]$ of values such that the sum $A[i] + \cdots + A[j]$ is maximized (among all possible such subarrays).
How many such subarrays are there? There is a subarray for every pair of distinct indices \( i \) and \( j \), which can be chosen in \( \binom{n}{2} \) different ways, hence there are so many subproblems. The only constraint is that the subarray is contiguous.

**Step 2: Focus first on the value of the solution.** It is helpful in many cases (especially optimization problems) to focus on the value (maximum or minimum) of the solution rather than the solution itself. This simplifies the next steps. For example, in the maxsum problem, we will first focus on finding the maximum value rather than the subarray that gives that maximum sum. Typically, it is easy to extend the DP solution that gives the maximum sum to also give the solution that has this maximum value.

**Step 3: Formulate subproblems.** As mentioned in Step 2, we will first focus on finding the value of the maximum sum, rather than the subarray which gives that maximum sum. Later, we will discuss how to extend this to finding the subarray as well.

What are the subproblems? This is an important step in DP. Figuring out the right subproblems usually leads to the correct recursive formulation which captures the DP solution. What are the subproblems for the maxsum problem? Of course, a straightforward way is to have a subproblem for each subarray, i.e., for each pair of indices \( i \) and \( j \) (\( 1 \leq i \leq j \leq n \)), we have a subproblem of computing the maximum sum restricted to that subarray. While this is a valid decomposition, this is not efficient, because there are \( n^2 \) subproblems. In a DP solution, the total time is at least proportional to the number of subproblems (since it takes at least constant time to solve a subproblem) and hence the running time with this decomposition is at least \( n^2 \). If we are shooting for an \( O(n) \) time algorithm, the number of subproblems have to be \( O(n) \) as well. Generally, the **smaller the total number of subproblems, the better.** How to come with such a decomposition?

This is the tricky part. One helpful idea in finding subproblems is to introduce **auxiliary variables.** Here is a first attempt keeping in mind that we are shooting for \( O(n) \) subproblems. Let \( M[i] \) be the solution to the max-sum problem considering only the first \( i \) elements of the array \( A \). That is, we introduce the auxiliary variable \( i \) that restricts the array to only the first \( i \) elements. Clearly, the original problem is one of the subproblems, i.e., \( M[n] \). Is this a good decomposition? To answer this, we need to check if we can write a recursive formulation for \( M[i] \) in terms of
6.3. ILLUSTRATING THE DP TECHNIQUE USING THE MAXSUM PROBLEM

subproblems of smaller size, i.e., $M[i - 1]$ and below. We note that $M[i]$ depends on whether the last element, i.e., $A[i]$ is in the solution or not. Hence, to write a recursion for $M[i]$ we need to know whether $A[i]$ is in the solution or not. This motivates the following subproblems.

We consider the following two subproblems based on the above fact:

Let $M^{in}[i]$ be the solution to the max-sum problem considering only the first $i$ elements of the array $A$, i.e., the maximum sum considering the array $A[1, \ldots, i]$, with the condition that the solution includes the $i$th element, i.e., the last element $A[i]$ is in the max-sum.

Similarly, let $M^{out}[i]$ be the solution to the max-sum problem considering the first $i$ elements of the array $A$, i.e., the maximum sum considering the array $A[1, \ldots, i]$, with the condition that the solution does not include the $i$th element, i.e., the last element $A[i]$ is not in the max-sum.

The solution to the max-sum problem in the array $A[1, \ldots, i]$ is

$$M[i] = \max\{M^{in}[i], M^{out}[i]\},$$

i.e., the maximum of the (only) two possibilities — $M^{in}[i]$ and $M^{out}[i]$ — where the first includes $A[i]$ and the second excludes it.

The important point about the subproblems is that it includes the original problem as well which is simply $M[n]$. This is typical of DP, the decomposed subproblems typically include the original problem as one of the subproblems. Although we are not interested in the solution of other subproblems, it turns out their solutions will help in building the solution to the original problem.

The total number of subproblems is $O(n)$, or $2n$ to be precise — $M^{in}[i]$ and $M^{out}[i]$ for $1 \leq i \leq n$.

Step 4: Recursive formulation. We now come to the key step of expressing the solution of the problem recursively in terms of the solutions to its subproblems. This is called recursive formulation. Once we have the correct recursive formulation, it turns out it is easy to write an algorithm that implements the formulation (see next step).

---

1. Note that writing the recursive formulation is the next step in DP. However, it is quite obvious that these two steps go hand in hand; if we cannot easily write a recursive formula for our subproblems, then we try to modify it.

2. A very naive way of formulating subproblems based on the fact that each element is in the solution or not is to look at all possible subsets of the array. There are $2^n$ such subsets. This is exponentially large and does not constitute an efficient decomposition since it ignores the key constraint that a feasible solution subset has to be contiguous. Even if we allow only for contiguous subsets to be contiguous, there are $n^2$ of them which is polynomial but still much bigger than $O(n)$ that we are shooting for.
Recursive formulation is like writing a recurrence. There are one or more base cases which show the solution to base-case subproblems which are usually those that are of small size. Then the recurrence shows how to compute the solution of larger-sized subproblems in terms of smaller-sized subproblems. The advantage of recursive formulation is that its correctness can be quickly established by (as usual) mathematical induction.

Let’s illustrate recursive formulation by using the maxsum problem. As mentioned above, $M[i] = \max\{M^{in}[i], M^{out}[i]\}$, since there are only two possibilities, either the last element ($A[i]$) is in the optimal solution or not. Note that $M[n]$ gives the maxsum value for the whole array (which is the answer we want to find).

The recursive formulation is writing a recurrence to compute $M^{in}[i]$ and $M^{out}[i]$ recursively:

\[
M^{in}[i + 1] = \max\{M^{in}[i] + A[i + 1], A[i + 1]\}.
\]
\[
M^{out}[i + 1] = M[i].
\]

The base cases are $M^{in}[1] = A[1]$ and $M^{out}[1] = -\infty$.

Why is the above recursive formulation correct? Before we formally prove correctness by using mathematical induction, we give the intuition for why the above formulation is correct.

As mentioned earlier, the optimal solution to $M[i]$ will either include or exclude $A[i]$, so naturally there are two cases:

- **Case 1: Exclude $A[i]$**. In this case, the optimal solution for the subarray $A[1 \ldots i]$, i.e., $M[i]$ will simply be the optimal solution to the subarray $A[1 \ldots i - 1]$ which is $M[i - 1]$.

- **Case 2: Include $A[i]$**. In this case, there are only two possibilities, either the optimal solution is $A[i]$ alone or is $M^{in}[i - 1] + A[i]$ (we take the max of the two). Note that the solution determining $M^{in}[i - 1]$ will include element $A[i - 1]$, and so we have contiguity of elements (this is the reason why we needed to decompose like this in the first place!).

**Step 5: Correctness proof of the recursive formulation.** We can show formal correctness of the recursive formulation by using mathematical induction. As usual, the base cases, $M^{in}[1] = A[1]$ and $M^{out}[1] = -\infty$, are trivially true.

\[\text{We use } M^{out}[1] = -\infty, \text{ since we assume that at least one element of } A \text{ should be in the optimal solution. If empty array is a valid solution, then we could have set } M^{out}[1] = 0.\]
6.3. ILLUSTRATING THE DP TECHNIQUE USING THE MAXSUM PROBLEM

To show the induction step, we assume the induction hypothesis that the formulation is correct for \( M^{\text{in}}[j] \) and \( M^{\text{out}}[j] \), for all \( 1 \leq j \leq i \). We show that it is true for \( j = i + 1 \) as well, under the above assumption. From the recurrence, we have \( M^{\text{out}}[i + 1] = M[i] \). This is true, because since we exclude \( A[i + 1] \), and \( M[i] \) is the optimal solution that corresponds to the subarray \( A[1, \ldots, i] \). Note that the optimality of \( M[i] \) is guaranteed by the induction hypothesis. Similarly, from the recurrence, we have \( M^{\text{in}}[i + 1] = \max\{M^{\text{in}}[i] + A[i + 1], A[i + 1]\} \). This is also true, since the optimal value of the subarray \( A[1, \ldots, i + 1] \) that includes \( A[i + 1] \) can either be \( A[i + 1] \) itself or can be a continuation of the optimal solution of subarray \( A[1, \ldots, i] \), which, of course should include \( A[i] \) — such a solution is precisely \( M^{\text{in}}[i] \). This completes the proof.

The proof shows an important property of problems that have DP solutions sometimes referred to as the optimal substructure property which says that the optimal solution of problems contains within itself optimal solution of its subproblems. This is really applying the proof “in reverse.” Suppose \( M^{\text{out}}[i + 1] \) is the optimal solution to the subproblem \( A[1, \ldots, i + 1] \), then this implies, by the correctness of the recurrence, that \( M[i] \) should be the optimal value of the subproblem \( A[1, \ldots, i] \). Suppose not. Then by replacing \( M[i] \) with the optimal value, we get a better optimal value for \( M^{\text{out}}[i] \) which contradicts the optimality of \( M^{\text{out}}[i] \). Similar argument can be made with the recurrence for \( M^{\text{in}}[i + 1] \).

**Step 6: Computing Run Time of DP Problems.** We outline the key points to upper bound the run time of a DP problem.

- Bound the total number of subproblems generated — let it be \( K \).
- Bound the (worst-case) time take to solve each subproblem — let it be \( T \).
- Then the total time of DP is upper bounded by \( O(KT) \) because, there are at most \( K \) subproblems and each takes at most \( T \) time. Sometimes a better bound can be obtained by looking at the time taken by individual subproblems. Let the time taken to solve subproblem \( S_i \) be \( T_i \), \( 1 \leq i \leq K \). Then the overall time is \( \sum_{i=1}^{K} T_i \).

Let’s calculate the running time of the DP algorithm for the maxsum problem. The number of subproblems is \( 2n - M^{\text{in}}[i] \) and \( M^{\text{out}}[i] \) for each \( i \). The time to compute each is just constant, which follows from the recurrence. Hence, the total run time is \( O(n) \).

3: $M^{out}[i] = -\infty$
4: for $i = 2$ to $n$ do
5: \hspace{1em} $M^{out}[i] = M[i - 1]$
6: \hspace{1em} $M^{in}[i] = \max\{M^{in}[i - 1] + A[i], A[i]\}$
7: \hspace{1em} $M[i] = \max\{M^{in}[i], M^{out}[i]\}$
8: Output $M[n]$

Algorithm 19 Algorithm Maxsum-rec($A[1, \ldots, n]$). Input: An array $A[1, \ldots, n]$. Output: Maxsum value. This is the naive recursive implementation of the recursive formulation of the maxsum problem. The recursive functions $M^{in}$ and $M^{out}$ are given in Figures 20 and 21 respectively.

1: return $\max\{M^{in}_{rec}(A[1, \ldots, n]), M^{out}_{rec}(A[1, \ldots, n])\}$

Step 7: Converting the recursive formula into an algorithm. It is relatively straightforward to turn the above recursive formula into a pseudocode — either iterative or recursive. However, care has to be taken, especially in a recursive implementation so that each subproblem is executed exactly once. Otherwise, the running time of the implementation can be exponential. We will next show two different approaches for implementing DP algorithms:

1. Iterative algorithm: Implementing the recursive formulation in an interactive manner is what is known as dynamic programming. This is a bottom-up strategy, where we start from the solution of the base cases and then compute the solutions to larger-sized problems using smaller-sized problems. Let’s illustrate this in the case of the maxsum problem. We first compute $M^{in}[1]$ and $M[1]$ which are both $A[1]$. Note that $M^{in}[2]$ and $M^{out}[2]$ depend only on $M^{in}[1]$ and $M[1]$ and hence can be computed. Similarly $M^{in}[3]$ and $M^{out}[3]$ can be computed and so on. Finally, we compute $M^{in}[n]$ and $M^{out}[n]$. The final answer is $M[n]$ which is equal to the maximum of $M^{in}[n]$ and $M^{out}[n]$. This can be implemented by a for loop. The iterative DP algorithm is given in Figure 6.3. The running time of the iterative algorithm is clearly $O(n)$ since there are are $n - 1$ iterations for loop and each iteration takes constant time. This is clearly an efficient implementation of the DP formulation as each subproblem is com-
6.3. ILLUSTRATING THE DP TECHNIQUE USING THE MAXSUM PROBLEM


1: if $i = 1$ then
2: return $A[1]$
3: else
4: return $\max\{M^{in\, rec}(A[1, \ldots, i-1]) + A[i], A[i]\}$.


1: if $i = 1$ then
2: return $-\infty$
3: else
4: return Maxsum-rec($A[1, \ldots, i-1]$).

uted only once leading to an $O(n)$ run time as postulated in Step 6.

2. Recursive algorithm: A more straightforward way is to implement the recursive formula in a recursive way. There is an almost a direct translation of the recursive formula into a recursive algorithm. See algorithm in Figure 19. However, this translation is bad because it doesn’t take into account overlapping subproblems, i.e., the same subproblems occur in different recursive calls and will be calculated over and over again leading to a very inefficient algorithm.

In the maxsum problem, the subproblem $M^{in}[i]$ occurs in the solution of two different subproblems $M^{in}[i+1]$ and in $M^{out}[i+1] = M[i] = \max\{M^{in}[i], M^{out}[i]\}$.

Since the algorithm in Figure 19 solves the same subproblem more than once, its running time turns out to be significantly more than $O(n)$, in fact it will be $\Theta(n^2)$. Exercise 6.1 asks you to show that the running time of algorithm in Figure 19 is $\Theta(n^2)$.

To get an implementation of the recursive algorithm that runs in $O(n)$ time, we make sure that we compute each subproblem only once. To ensure this, as soon as we compute the solution to a subproblem for the first time, we store its solution (in a suitable data structure like an array); subsequently, if the same subproblem is encountered we simply “look up” this stored solution without the need to recompute the solution from scratch. This is called as the table lookup strategy or memoized strategy. This only requires a simple
Algorithm 22 Algorithm Maxsum-reclookup($A[1,\ldots, n]$). Input: An array $A[1,\ldots, n]$. Output: Maxsum value. This is the efficient recursive implementation of the recursive formulation of the maxsum problem that uses the table lookup strategy. The recursive functions $M^\text{in}$ and $M^\text{out}$ are given in Figures 23 and 24 respectively.

1: for $i = 1$ to $n$ do
2: \hspace{6mm} $M[i] = M^\text{in}[i] = M^\text{out}[i] = -\infty$ // initialization of arrays
3: return max($M^\text{in}_{\text{reclookup}}(A[1,\ldots, n]), M^\text{out}_{\text{reclookup}}(A[1,\ldots, n])$)


1: if $i = 1$ then
2: \hspace{6mm} return $A[1]$
3: else if $M^\text{in}[i] \neq -\infty$ then
4: \hspace{6mm} return $M^\text{in}[i]$ // value already computed
5: else
6: \hspace{6mm} $M^\text{in}[i] = \max\{M^\text{in}_{\text{reclookup}}(A[1,\ldots, i-1]) + A[i], A[i]\}$ // store first
7: \hspace{6mm} return $M^\text{in}[i]$

modification to the inefficient algorithm of Figure 19. The modified algorithm is given in Figure 22. In the modified algorithm we store the values of the computed subproblems — $M^\text{in}[i]$ and $M^\text{out}[i]$ — in an array and then use the stored values to avoid recomputing the solution to a subproblem that was already computed.

Step 8: Computing the solution that gives the optimum value. Typically it is easy to modify the algorithm that computes the optimum value to also output the solution that gives the optimum value. This usually involves doing some additional bookkeeping. Let's illustrate this with the maxsum problem. How do you output the indices that give the maximum sum? We do this by keeping an additional array called $S[i]$ which is either “in” or “out”. If it is “in” then the $A[i]$ is included in the optimal solution of the subproblem with value $M[i]$. $S[i]$ array values are computed as we compute the optimal $M[i]$ values. This can be done by simply checking how $M[i] = \max\{M^\text{in}[i], M^\text{out}[i]\}$ is computed. If $M[i] = M^\text{in}[i]$, then it means that $A[i]$ is part of the optimal solution and hence we set $S[i] = \text{“in”}$; on the other hand if $M[i] = M^\text{out}[i]$, then it means that $A[i]$ is not part of the optimal solution and hence we set $S[i] = \text{“out”}$. Similarly, we also keep an additional array $S^\text{in}[i]$ which records which of the two val-

1: if $i = 1$ then
2: return $-\infty$
3: else if $M^{out}[i] \neq -\infty$ then
4: return $M^{out}[i]$ // value already computed
5: else
6: $M^{out}[i] = \text{Maxsum-reclookup}(A[1,\ldots,i-1])$ // store first
7: return $M^{out}[i]$

ues $M^{in}[i]$ takes, i.e., whether it is $M^{in}[i-1] + A[i]$ or just $A[i]$. In the case of the former, we set $S^{in}[i] = \text{“true”}$, otherwise we set $S^{in}[i] = \text{“false”}$. Figure 25 shows how to modify the DP algorithm of Figure to compute the additional array $A[i]$.

To output the optimal solution we usually trace back the steps that we took to compute the optimum value, which in the case of the maxsum problem is $M[n]$. Note that $M[n]$ is the maximum of $M^{in}[n]$ and $M^{out}[n]$. If the former is the maximum of the two, then $A[n]$ is in the solution, otherwise, it is not. This information is stored in $S[n]$. If $A[n]$ is in the solution, we look at whether $A[n-1]$ is in the solution — this can be inferred by looking at the $S^{in}[n]$. We can then continue whether $A[n-2]$ is in the solution or not and so on. This is easier to do recursively, since we do this in a top-down fashion starting from the optimal value $M[n]$. The algorithm is given in Figure 26.

Exercise 6.1. Write a recurrence for computing the running time of the algorithm of Figure 19. Solve the recurrence and show that it takes $\Theta(n^2)$ time.

We next further illustrate these ideas by looking at various problems.

6.4 Problem: Matrix Chain Multiplication

We are given a chain of $n$ matrices: $< A_1, A_2, \ldots, A_n >$ where matrix $A_i$ has dimension $p_{i-1} \times p_i$, for $i = 1, 2, \ldots, n$. We are interested in finding the “best” way to compute the product of the $n$ matrices:

Definition 6.4.1 (Matrix Chain Multiplication). Find a way to parenthesize the product $A_1 A_2 \ldots A_n$ which minimizes the number of multiplications.
Algorithm 25 Algorithm Maxsum-DP-mod($A[1, \ldots, n]$). Input: An array $A[1, \ldots, n]$. Output: Maxsum value, $S$ and $S^{in}$ arrays. This is the modified DP (iterative) implementation of the recursive formulation of the max-sum problem that also computes the $S$ and $S^{in}$ arrays which are used in outputting the optimal solution.

3: $M^{out}[i] = -\infty$
4: $S[1] = \text{“in”}$
5: for $i = 2$ to $n$ do
6: \hspace{1em} $M^{out}[i] = M[i - 1]$
7: \hspace{1em} $M^{in}[i] = \max\{M^{in}[i - 1] + A[i], A[i]\}$
8: \hspace{1em} $M[i] = \max\{M^{in}[i], M^{out}[i]\}$
9: \hspace{1em} if $M[i] = M^{in}[i]$ then
10: \hspace{2em} $S[i] = \text{“in”}$
11: \hspace{2em} else
12: \hspace{3em} $S[i] = \text{“out”}$
13: \hspace{1em} if $M^{in}[i] = M^{in}[i - 1] + A[i]$ then
14: \hspace{2em} $S^{in}[i] = \text{“true”}$
15: \hspace{2em} else
16: \hspace{3em} $S^{in}[i] = \text{“false”}$
17: Output $M[n]$

To understand the problem, we recall some basic facts about matrix multiplication:

- The number of columns of $A_i$ is equal to the number of rows of $A_{i+1}$ (for $1 \leq i \leq n - 1$). This property is needed for a valid multiplication.

- Given 2 matrices $A, B$, with dimension $m \times n$ and $n \times p$ respectively, the number of element-wise multiplications needed to multiply the 2 matrices is $mnp$. Note that here we assume that multiplications are done in the standard way, i.e., the $i,j$ entry of the product matrix $C = AB$ is computed as: $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$ which takes $n$ multiplications. Since the product matrix $C$ has $mp$ elements, the total number of multiplications is $mnp$.

- Matrix multiplication is associative, i.e., $(AB)C = A(BC)$. Thus the order of multiplying the matrices doesn't matter for the final answer. However, order might matter for minimizing the total number of multiplications (cost).
Algorithm 26 Algorithm Maxsumsol\((A[1 \ldots n], S[1 \ldots n], S^m[1 \ldots n])\).

Input: An array \(A[1 \ldots n], S[1 \ldots n]\), and \(S^m[1 \ldots n]\) (computed in Algorithm 25)

Output: The optimal solution with the elements in the subarray listed backwards.

1: if \(n \geq 1\) then
2: if \(S[n] = \text{“in”}\) then
3: Output \(A[n]\) // \(A[n]\) is included in the optimal solution
4: if \(S^m[n] = \text{“true”}\) then
5: // \(A[n-1]\) is also included in the optimal solution
6: Maxsumsol\((A[1 \ldots n-1], S[1 \ldots n-1], S^m[1 \ldots n-1])\)
7: else
8: // \(S[n] = \text{“out”}\)
9: Maxsumsol\((A[1 \ldots n-1], S[1 \ldots n-1], S^m[1 \ldots n-1])\)

Example  Suppose we are given 3 matrices \(A_1, A_2\) and \(A_3\) with dimensions \(10 \times 1000, 1000 \times 100\) and \(100 \times 200\) respectively. If we multiply according to parenthesization \((A_1A_2)A_3\) then the cost is: \(10 \times 1000 \times 100 + 10 \times 100 \times 200 = 1200000\). If we multiply according to parenthesization \(A_1(A_2A_3)\) then the cost is: \(1000 \times 100 \times 200 + 10 \times 1000 \times 200 = 2200000\).

Number of ways of parenthesizing \(n\) multiplications  Given a chain of \(n\) matrices, the number of different ways to parenthesize is exponential in \(n\). Each parenthesization gives a different way to multiply the chain with different costs. Let \(P(n)\) be the number of different ways of parenthesizing a product of \(n\) matrices. Then we can obtain the following recurrence:

\[
P(n) = \sum_{k=1}^{n-1} P(k)P(n-k), \quad n \geq 2.
\]

\(P(1) = 1\).

We can show that \(P(n)\) is exponential in \(n\), in fact, \(P(n) \geq 2^n\). (See Exercise 6.2) Hence a naive strategy of checking all possible parenthesizations is very inefficient.

Exercise 6.2. Use mathematical induction to prove that \(P(n)\) (defined in the above recurrence) is at least \(2^n\).

6.4.1 DP solution

We first write a recursive formulation for computing the minimum cost parenthesization. For \(1 \leq i \leq j \leq n\), let \(m[i, j]\) be the minimum cost of
paranthesizing the product $A_i A_{i+1} \cdots A_j$. Then we can write a recursive formula to compute $m[i, j]$.

$$m[i, j] = \begin{cases} 
0 & i = j \\
\min_{i \leq k < j} \{ m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j \} & i < j
\end{cases}$$

Writing the above recursive formula is the key step in solving a dynamic programming problem. Using the recursive formula, we can develop an algorithm. As we mentioned earlier there are 2 possible algorithmic strategies: (1) top-down / recursive / table look up / memoization or (2) bottom-up / iterative / DP.

**Key step in the recursive formulation** In many DP problems, the key step in writing a correct recursive formulation is identifying the right subproblem(s). The subproblems may not be obvious. A useful idea is to introduce auxiliary parameters in the problem. For example, though we are interested in finding out the best way to parenthesize the entire matrix chain, by introducing the two auxiliary variables $i$ and $j$, we identify the subproblem(s) of parenthesizing the subchain $A_i \cdots A_j$ which proves useful in solving the desired problem. Note that the original problem is one of the subproblems, i.e., computing $m[1, n]$.

**Correctness of the recursive formulation** To show the correctness of the recursive formulation, as usual we we turn to mathematical induction.

- **Base case ($i = j$):** Trivially true.

- **Recursive step ($i < j$):** The induction hypothesis is that the recursive formulation is correct for all $\ell$ such $j - i < \ell$. Assuming the induction hypothesis, we prove that the formulation is correct for $j - i = \ell$, i.e., $m[i, j]$ is the optimal cost of multiplying $A_i \cdots A_j$.

  Take the minimum of the $j - i = \ell$ possible parenthesizations between $A_i$ and $A_j$ depending on where you split the product, i.e.,

  $$(A_i \cdots A_j) = (A_i \cdots A_k)(A_{k+1} \cdots A_j).$$

  We note that $m[i, k]$ is the minimum cost of multiplying $(A_i \cdots A_k)$ (by induction hypothesis), $m[k + 1, j]$ is the minimum cost of multiplying $A(A_{k+1} \cdots A_j)$ (again by induction hypothesis), and $p_{i-1} p_k p_j$ is the cost of combining the two partial products. Hence, since the split is minimum, the overall cost $m[i, j]$ is also optimal.
Note that the above argument shows the optimal substructure property which is nothing but saying that the optimal solution to a problem consists of optimal solution to its subproblems. Here computing \( m[i, j] \) is the problem and that depends on \( m[i, k] \) and \( m[k + 1, j] \) (for all \( i \leq k < j \)), which are optimal solutions to subproblems. As we saw above, the proof is straightforward by using induction. The optimal substructure property tells us that there is no need to consider suboptimal solutions to subproblems when solving a problem; only optimal solutions to subproblems are relevant. This drastically reduces the number of subproblem solutions to consider — leading to efficient algorithms.

**A Naive Algorithm**  The following is a naive algorithm that directly converts the recursive formula into a recursive algorithm. The exercise asks you to show that the above algorithm takes exponential time.

**cost**\((p, i, j)\)

**Input:** A sequence of matrix dimensions: \( p = <p_0, p_1, \ldots, p_n> \).

**Output:** the optimal cost to multiply \( A_i \ldots A_j \).

1. if \( i = j \) return 0
2. \( m[i, j] = \infty \)
3. for \( k = i \) to \( j - 1 \)
4. \( q = \text{cost}(p, i, k) + \text{cost}(p, k + 1, j) + p_{i-1}p_kp_j \)
5. if \( q < m[i, j] \)
6. \( m[i, j] = q \)
7. return \( m[i, j] \)

**Exercise 6.3.** Show that the naive algorithm cost takes exponential time.

**Bottom-up Algorithm (DP/iterative algorithm)**  Here is the DP algorithm which is an iterative algorithm. Note that the DP algorithm builds the solution bottom up: it first computes 1 length chains (which is 0) and then successively computes chains of increasing length, i.e., lengths 2, 3, \ldots, up to \( n \). Each chain of length \( l \) depends on chains of length \( l - 1 \) or smaller. Hence computing bottom up works.

**Matrix-cost**\((p)\)

**Input:** A sequence of matrix dimensions: \( p = <p_0, p_1, \ldots, p_n> \).

**Output:** the optimal cost to multiply \( A_1 \ldots A_n \).

1. for \( i = 1 \) to \( n \)
2. \( m[i, i] = 0 \)
2 for \( l = 2 \) to \( n \)
3 \hspace{1em} \textbf{for} \hspace{1em} i = 1 \hspace{1em} \text{to} \hspace{1em} n - l + 1
4 \hspace{1em} j = i + l - 1
5 \hspace{1em} m[i, j] = \infty
6 \hspace{1em} \textbf{for} \hspace{1em} k = i \hspace{1em} \text{to} \hspace{1em} j - 1
7 \hspace{1em} \hspace{1em} q = m[i, k] + m[k + 1, j] + p_{l-1} p_k p_j
8 \hspace{1em} \textbf{if} \hspace{1em} q < m[i, j]
9 \hspace{1em} \hspace{1em} m[i, j] = q
10 \textbf{return} \hspace{1em} m[1, n]

\begin{exercise}
Show that the run time of the above algorithm is \( O(n^3) \). Show that it correctly computes the optimal cost.
\end{exercise}

\textbf{Memoization: Top-down Algorithm (recursive/table look up)} While the naive recursive algorithm is bad, the following recursive algorithm that uses table lookup to avoid computation of the same subproblem more than once takes polynomial time.

1 \hspace{1em} \textbf{for} \hspace{1em} i = 1 \hspace{1em} \text{to} \hspace{1em} n
2 \hspace{1em} \hspace{1em} \textbf{for} \hspace{1em} j = i \hspace{1em} \text{to} \hspace{1em} n
3 \hspace{1em} \hspace{1em} \hspace{1em} m[i, j] = \infty
4 \hspace{1em} \textbf{return lookup}(p, 1, n)

\begin{definition}
lookup(p, i, j)
\end{definition}

1 \hspace{1em} \textbf{if} \hspace{1em} m[i, j] < \infty \hspace{1em} \textbf{return} \hspace{1em} m[i, j]
2 \hspace{1em} \hspace{1em} \textbf{if} \hspace{1em} i = j
3 \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} m[i, j] = 0
4 \hspace{1em} \hspace{1em} \textbf{else} \hspace{1em} \textbf{for} \hspace{1em} k = i \hspace{1em} \text{to} \hspace{1em} j - 1
5 \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} q = \text{lookup}(p, i, k) + \text{lookup}(p, k + 1, j) + p_{l-1} p_k p_j
6 \hspace{1em} \hspace{1em} \hspace{1em} \textbf{if} \hspace{1em} q < m[i, j]
7 \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} m[i, j] = q
8 \hspace{1em} \textbf{return} \hspace{1em} m[i, j]

\begin{exercise}
What is the run time of the above algorithm?
\end{exercise}

\textbf{Run time of Matrix Multiplication} The number of subproblems is \( \Theta(n^2) \) since there is a distinct subproblem for every pair of indices \( i \) and \( j \), \( 1 \leq i \leq j \leq n \). Each subproblem can be solved in \( O(n) \) time. Hence total time is \( O(n^3) \).
Getting the actual parenthesization. We only focused on getting the minimum cost. It is easy to modify the code to obtain the actual parenthesization that achieves the minimum cost as well, in the same time bounds (see Exercise 6.6).

Exercise 6.6. Modify the recursive formulation that enables computation of the optimal parenthesization. Modify the DP (iterative) algorithm so as to incorporate this additional computation. Give the algorithm and the pseudocode for the same.

6.5 Problem: Aligning two Sequences

We consider a problem that is fundamental to computational molecular biology: Given two (e.g., DNA or protein) sequences (strings) $s[1 \ldots m]$ and $t[1 \ldots n]$ we want the “best” alignment between the two sequences.

Suppose we are given two DNA strings $s$ and $t$ as follows.

\[
\begin{align*}
s &= \text{GACGGATTAG} \\
t &= \text{GATCGGAATAG}
\end{align*}
\]

The goal is to find the “best” way to “align” the two strings. We define what is meant by alignment next.

Alignment. An alignment is an insertion of spaces in arbitrary locations along the sequences so that they end up with the same size. A space in one sequence should not align with a space in the other. But spaces can be inserted at the beginning or at the end. An alignment is scored by a scoring scheme. We assume a given scoring matrix $score$ where the entry $score[x, y]$ gives the alignment score for characters $x$ and $y$. For simplicity, we will assume that the score of any letter with space is -1. The score for an alignment is the sum of the scores of its aligned characters. A best alignment is one which receives the maximum score called the similarity $sim(s, t)$.

Example. We want to find the best way to align $s = \text{GACGGATTAG}$ and $t = \text{GATCGGAATAG}$. Assume that the score for aligning two characters which are the same is 1, but aligning two different characters has score 0, while aligning a letter with a space has score -1. Then the best alignment
of $s$ and $t$ is:

$$s = GA - CGGATTAG$$
$$t = GATCGGAATAG$$

Note that a space was introduced in $s$ after the first two characters. The score for this alignment is $1 + 1 - 1 + 1 + 1 + 1 + 0 + 1 + 1 + 1 = 8$. This is the maximum possible alignment score and hence this is a best alignment (there can be more than one best alignment).

A Naive solution A naive approach is to look at all possible alignments of the two input strings, compute the score for each alignment and then take the maximum. This is very inefficient as the number of possible alignments is at least exponential in the size of the strings. For example, if one string has $2n$ characters and the other string has $n$ characters, one has to insert at least $n$ gaps to align the two strings; the number of ways that one can insert $n$ gaps among the $n$ characters is at least $\binom{2n}{n} \geq 2^n$$^{4}$

DP leads to a significantly more efficient solution as described below.

Recursive formulation As in the matrix multiplication problem, to compute the optimal alignment score we first write the recursive formulation. Again the key is to identify the right subproblems. Here we will use the subproblem(s) of finding the best alignment of the substrings $s[1 \ldots i]$ and $t[1 \ldots j]$, where $1 \leq i \leq m$ and $1 \leq j \leq n$. Note that we only allow the right end to vary. One could have allowed “both” ends to vary, i.e., consider subproblems of aligning strings $s[i \ldots j]$ and $t[k \ldots \ell]$. However this is not necessary and leads to an increased run time (though still polynomial) since the number of subproblems is more.

$$sim(s[1 \ldots i], t[1 \ldots j]) = \max \begin{cases} 
    sim(s[1 \ldots i - 1], t[1 \ldots j - 1]) + score(s[i], t[j]) \\
    sim(s[1 \ldots i], t[1 \ldots j - 1]) + score(-, t[j]) \\
    sim(s[1 \ldots i - 1], t[1 \ldots j]) + score(s[i], -)
\end{cases}$$

The base cases are:

$$sim(s[1 \ldots i], t[\]] = -i$$
$$sim(s[\], t[1 \ldots j]) = -j$$

$^{4}$See Appendix G for the inequality used to infer this lower bound.
6.6. **PROBLEM: 0/1 KNAPSACK**

\[ \text{sim}(s[], t[]) = 0 \]

The correctness of the recursive formula follows from mathematical induction which is left as an exercise.

**Exercise 6.7.** Show that the above recursive formulation correctly computes the similarity score.

**Dynamic Programming (Bottom-up) Algorithm**  We can convert the recursive formulation into a DP algorithm as follows in a fairly straightforward fashion.

**Input:** sequences \( s[1 \ldots m] \) and \( t[1 \ldots n] \)

**Output:** \( \text{sim}(s, t) \)

\[
\begin{align*}
\text{for } i &= 0 \text{ to } m \\
& \quad a[i, 0] = -i \\
\text{for } j &= 0 \text{ to } n \\
& \quad a[0, j] = -j \\
\text{for } i &= 1 \text{ to } m \\
& \quad \quad \text{for } j = 1 \text{ to } n \\
& \quad \quad \quad a[i, j] = \max\{a[i-1, j-1] + \text{score}(s[i], t[j]), \\
& \quad \quad \quad \quad a[i-1, j] + \text{score}(-, t[j]), \\
& \quad \quad \quad \quad a[i, j-1] + \text{score}(s[i], -)\} \\
\text{return } a[m, n]
\end{align*}
\]

It is easy to see that the run time of the above algorithm is \( O(mn) \).

**Exercise 6.8.** Give a memoized (recursive) algorithm for the alignment problem.

**Exercise 6.9.** Modify the recursive formulation to output the optimal alignment as well.

### 6.6 Problem: 0/1 Knapsack

We consider the integral (or 0/1) knapsack problem which is a modification of the fractional knapsack problem considered in Chapter 7. The problem is defined as follows.

**Problem 6.1 (0/1 Knapsack).** We are given a set \( S = \{a_1, a_2, \ldots, a_n\} \) of \( n \) objects and a knapsack. Object \( a_i \) has an integer weight \( w_i \) and the knapsack has integer capacity \( m \), i.e., the maximum weight of all items
placed in the knapsack cannot exceed \( m \). An object \( a_i \) is either placed into the knapsack or entirely omitted (no fractional part allowed), i.e., \( x_i = 1 \) if object is placed in the knapsack, and 0 otherwise. If the object \( a_i \) is placed, then a profit of \( p_i \) is earned. The objective is to maximize the total profit earned subject to the knapsack constraint.

We want to find the optimal solution using Dynamic Programming. That is we want to find the solution vector \( x_1, x_2, \ldots, x_n \) which is 0-1 vector that maximizes the total profit earned subject to the knapsack constraint.

**A Naive Solution**  As usual, a simple, but naive solution is to consider all possible subsets of objects (since each object can be either in the optimal solution or not) and take the subset that satisfies the knapsack capacity and maximizes the total profit. Since the number of subsets is \( 2^n \), this leads to exponential running time. DP, again, leads to a significantly improved running time as explained below.

**Subproblems**  The key, again, for a correct recursive formulation is identifying the right subproblems. For this, as mentioned earlier, introducing auxiliary variables help. Let’s introduce two new variables \( j, y \) as follows. Let \( KNAP(j, y) \) represent the problem: maximize \( \sum_{1 \leq i \leq j} p_i x_i \); subject to the constraint \( \sum_{1 \leq i \leq j} w_i x_i \leq y \) with the (integral) requirement that \( x_i \in \{0, 1\}, 1 \leq i \leq j \). The above is simply a restatement of the knapsack problem as an explicit maximization problem. In words, it is the subproblem that considers only the first \( j \) items with a knapsack capacity \( y \). The Knapsack problem is simply \( KNAP(n, m) \). Note that we assume all weights and \( m \) are integers.

**Recursive formulation**  Let \( P_j(y) \) be the optimal solution to \( KNAP(j, y) \). Then, we can write:

\[
P_j(y) = \max\{P_{j-1}(y), P_{j-1}(y - w_j) + p_j\}.
\]

The base cases are: \( P_0(y) = 0 \) for all \( y \geq 0 \) and \( P_1(y) = -\infty \) if \( y < 0 \).

**Correctness of the formulation**  The above formulation is correct because to compute \( P_j(y) \) we need to look at only two choices: either the \( a_j \)th object is in the knapsack or not. If it is not included, then \( P_j(y) = P_{j-1}(y) \), since no weight has been added. If it is included, then \( P_j(y) =...
6.7. A FULLY POLYNOMIAL-TIME APPROXIMATION SCHEME FOR KNAPSACK

\[ P_{j-1}(y - w_j) + p_j, \] since we have to subtract the weight \( w_j \) from the knapsack capacity and include its profit. The correctness of the formulation, as usual, can be established by induction.

Again induction implies that the optimal substructure property holds. \( P_j(y) \) should contain within itself optimal solution to its subproblem(s) — which are \( P_{j-1}(y) \) (in the first case) and \( P_{j-1}(y - w_j) \) (in the second case). Of course, we take the maximum value of the two cases.

**DP algorithm** A DP algorithm is straightforward from the above formulation. The desired optimal value is \( P_n(m) \) which we compute in a bottom up manner. That is, we first start from the base cases and compute \( P_1(\cdot), P_2(\cdot), \ldots \) in this order. The time complexity is \( O(nm) \). Note that the running time depends on the number of objects as well as the (magnitude) of the (integer) capacity. Hence this is not a polynomial time algorithm, since the running time is not a polynomial function of the input size which is \( \log m \) (since \( m \) can be represented using \( \log m \) bits), but rather a polynomial function of \( m \). Such a running time is referred to as pseudo-polynomial and the algorithm is called a pseudo-polynomial algorithm. Such algorithms are fast when \( m \) is small; more importantly they can be converted to yield efficient polynomial time algorithms that give an approximately optimal solution. This is discussed in the next section.

**Exercise 6.10.** Give pseudocode of the DP algorithm (iterative) for computing the optimal profit for the knapsack problem based on the above formulation. Give also pseudocode of the recursive algorithm (with table lookup) for computing the optimal profit based on the above formulation. Analyze and show that the running time of both the algorithms are \( O(nm) \). Give an algorithm to output the optimal solution, i.e., the subset of objects that gives the optimal profit.

6.7 A Fully Polynomial-Time Approximation Scheme for Knapsack

We note that Knapsack is an NP-complete problem, i.e., in other words, there is unlikely to be a polynomial time algorithm for Knapsack. It is important to note that by “polynomial” we mean polynomial in the input size, i.e., the number of items \( n \) and the sizes of the integers used in the input (represented in binary). In other words, the it should be a polynomial function of \( n \) and \( \log(m) \) (the capacity), and \( \log(w_i) \) (for all weights \( w_i \).
of all items) and $\log(p_i)$ (for all profits). The DP-based pseudo-polynomial algorithm presented in the previous section is thus not a (truly) polynomial time algorithm, it is a pseudo-polynomial one. As mentioned there, it is fast (i.e., polynomial) when the capacity, weights and profits are small, i.e., polynomial function of $n$. However, one can obtain a truly polynomial approximation algorithm for Knapsack that can achieve as close to the optimal as possible (however, the run time will increase the closer we get to the optimum). In other words, we can achieve something called a fully polynomial-time approximation (FPTAS) scheme, which is essentially the best possible that one can hope for an NP-complete problem.

Before we proceed, we will define formally an approximation algorithm and an FPTAS, which is a special type of approximation algorithm.

### 6.7.1 Approximation Algorithm and FPTAS

An approximation algorithm for an optimization problem is (typically) a polynomial time algorithm that computes a solution whose value is “close” to the optimal value. The closeness is defined as a ratio between optimal and approximate solution values.

**Definition 6.7.1.** An approximation algorithm has a ratio bound $\rho(n)$, if for any input of size $n$, the optimal solution $C^*(n)$ and the algorithm solution $C(n)$ satisfy the relation:

- For minimization problems: $\frac{C(n)}{C^*(n)} \leq \rho(n)$
- For maximization problems: $\frac{C(n)}{C^*(n)} \geq \rho(n)$

To establish an approximation ratio, we need to compare the cost of the solution of the algorithm (ALG) to the cost of an optimal solution (OPT). Typically, this means that we have to find “good” polynomial-time computable bounds for OPT.

**Definition 6.7.2 (FPTAS).** A minimization (resp. maximization) problem has a fully polynomial-time approximation scheme if for any $\varepsilon > 0$ there is an $1 + \varepsilon$ (resp. $1 - \varepsilon$)-approximation algorithm for the problem that runs in time which is polynomial in both the problem (input) size and $\frac{1}{\varepsilon}$.

Note that an FPTAS is actually a family of algorithms, parameterized by $\varepsilon$. Given an $\varepsilon$ the algorithm guarantees that the solution is within a factor of $1 - \varepsilon$ of the optimal solution; the algorithm runs in time polynomial in the input size and $\frac{1}{\varepsilon}$. Thus it gives a tradeoff between accuracy of the
solution and the running time; smaller the $\varepsilon$, closer is the solution to the optimal, but larger is the run time.

Problems which have pseudo-polynomial time algorithms can be typically converted to yield FPTASs as is illustrated next.

### 6.7.2 Alternate DP algorithm for Knapsack

We will use DP to design a pseudo-polynomial time algorithm for knapsack; this DP algorithm is a bit different from one proposed in Section 6.6.

In the earlier formulation, we fixed a bound on the capacity and sought to maximize the profit subject to the capacity bound. In the alternate formulation, we fix a particular profit and seek to minimize the total size that achieves (if possible) the particular profit. In this formulation, the algorithm's run time is a function of the profit values (unlike the earlier formulation which was a function of the maximum capacity). This is readily suitable for designing a FPTAS.

Again, we are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of $n$ objects, with specified weights and profits (all values are positive integers), and a knapsack capacity $m$, a positive integer. The problem is to find a subset of objects whose total weight is bounded by $m$ and the total profit is maximized. (Without loss of generality, we will assume that every object has weight at most $m$.)

Let $P$ be the profit of the most profitable object, i.e., $P = \max_{a \in S} \text{profit}(a)$. Note that $nP$ is a trivial upper bound for any solution.

For each $i \in \{1, \ldots, n\}$ and $p \in \{1, \ldots, nP\}$, let $S_{i,p}$ denote a subset of $\{a_1, \ldots, a_i\}$ whose total profit is exactly $p$ and whose total weight is minimized. Let $W(i, p)$ denote the sum of the weights of the objects in the set $S_{i,p}$. Note that $W(i, p) = \infty$ if no such set exists.

The recursive formulation is as follows. There are two cases:

1. If $\text{profit}(a_{i+1}) \leq p$, then we have two choices: either include object $a_{i+1}$ or not. Thus we have:

   $$W(i + 1, p) = \min\{W(i, p), \text{size}(a_{i+1}) + W(i, p - \text{profit}(a_{i+1}))\}$$

2. If $\text{profit}(a_{i+1}) > p$, then

   $$W(i + 1, p) = W(i, p)$$
The base case $W(1, p)$ can be trivially computed for every $p \in \{1, \ldots, nP\}$. Hence all values $W(i, p)$ can be computed in $O(n^2 P)$ time (see Exercise 6.17).

The maximum profit achievable by objects of total weight bounded by $m$ is $\max\{p | W(n, p) \leq m\}$.

### 6.7.3 FPTAS for Knapsack

The idea used to design an $(1 - \varepsilon)$-approximation algorithm is simple. If $P$ is “small” (say polynomial in $n$), then the DP algorithm gives a polynomial time algorithm. However, $P$ can be large (e.g., it can be exponential in $n$). So the high-level idea behind the approximation algorithm is to scale down the value of $P$. If we are interested in only approximating the optimal, we can round the numerical values by ignoring a certain number of least significant bits of the profit values. The number of bits we will ignore will depend on the error parameter $\varepsilon$. Intuitively, the optimal profit will also be rounded down; however, there will be some loss of accuracy because of rounding.

We will simply run the DP algorithm on the rounded profit values which returns the optimal (exact) solution for the rounded values. Note that the run time of the DP solution is $O(n^2 P)$ where $P$ is the value of the most profitable object. The idea of rounding is simple: we round down all profits by a factor of $K = \frac{\varepsilon P}{n}$, i.e., the rounded profit $p'_i = \lfloor \frac{p_i}{K} \rfloor$. Because of the rounding down, the largest profit value is also rounded down to $n/\varepsilon$ and the running time becomes $O(n^3/\varepsilon)$ which is a polynomial in $n$ and $\varepsilon$. But what about the optimal value? We show next that the optimal value can be obtained within an factor of $(1 - \varepsilon)$; in other words, the effect of rounding can be bounded.

**The algorithm.** Here is the FPTAS algorithm:

1. Given $\varepsilon > 0$, let $K = \frac{\varepsilon P}{n}$.
2. For each object $a_i$ define $\text{profit}'(a_i) = \lfloor \text{profit}(a_i)/K \rfloor$.
3. Run the above dynamic programming algorithm with the new profits and output the most profitable set.

**Analysis** We now formalize the intuition discussed earlier.
Lemma 6.1. Let \( A \) denote the set output by the algorithm. Then
\[
\text{profit}(A) \geq (1 - \varepsilon) \text{OPT}.
\]

Proof. To bound the approximation ratio, we compare the optimal solution with the solution output by the algorithm. Since we cannot exactly compute the optimal solution, we obtain a good, polynomially computable upper bound on the optimal solution and compare it with the solution obtained by DP.

Let \( O \) denote the optimal set. For any object \( a \), because of the rounding down,
\[
\frac{\text{profit}(a)}{K} - \left\lfloor \frac{\text{profit}(a)}{K} \right\rfloor \leq 1.
\]
That is,
\[
\frac{\text{profit}(a)}{K} - \text{profit}'(a) \leq 1.
\]
Multiplying by \( K \), we have
\[
\text{profit}(a) - K \times \text{profit}'(a) \leq K.
\]

The above inequality holds for each element in the optimal set \( O \).
\[
\sum_{a \in O} \text{profit}(a) - K \times \sum_{a \in O} \text{profit}'(a) \leq \sum_{a \in O} K.
\]
Since there are at most \( n \) elements in the optimal set, we have:
\[
\text{profit}(O) - K \times \text{profit}'(O) \leq nK,
\]
, i.e.,
\[
K \times \text{profit}'(O) \geq \text{profit}(O) - nK.
\]

The key observation is that dynamic programming solution must return a set at least as good as \( O \) under the new profits. This is because dynamic programming computes the optimal solution with respect to the rounded profits. Hence,
\[
\text{profit}'(A) \geq \text{profit}'(O).
\]
and thus
\[
\text{profit}(A) \geq K \times \text{profit}'(A) \geq K \times \text{profit}'(O) \geq \text{profit}(O) - nK = \text{OPT} - \varepsilon P \geq (1 - \varepsilon) \text{OPT}.
\]
since $OPT \geq P$ (the optimum is at least the profit of the most profitable object, since we assume all objects have weight bounded by the capacity).

\[ \Box \]

**Theorem 6.1.** The above algorithm is an FPTAS for Knapsack.

**Proof.** The solution is within $(1 - \varepsilon)OPT$.

The running time of the algorithm is

\[
O(n^2 \left\lfloor \frac{P}{K} \right\rfloor) = O(n^2 \left\lfloor \frac{n}{\varepsilon} \right\rfloor) = O(n^3 / \varepsilon)
\]

which is polynomial in $n$ and $1/\varepsilon$.

\[ \Box \]

### 6.8 DP Summary

We summarize the common themes underlying the application of DP to various problems that we studied in this chapter.

**Key ingredients of DP** For DP to apply for a problem — typically, an optimization problem — it should have two key ingredients:

1. **Optimal substructure property**: The optimal solution is computed from optimal solutions to independent subproblems. Hence, instead of looking at all feasible solutions for the subproblems (which can be very inefficient), it is enough to look at only optimal solutions for the subproblems.

2. **Overlapping subproblems**. Overlapping means the same subproblem occurs in the solution of many different subproblems. We compute the solution of each subproblem only once.

**Two algorithmic implementations** As was illustrated by various examples, once we formulate the all-important recursive formulation of the problem, there are two algorithmic ways for solving DP-based problems.

1. **Iterative algorithm**: Based on the recursive formula we write an iterative algorithm that solves the recursion in a bottom-up manner. That is, it first solves the base-case subproblems and then progressively solves subproblems of increasing size based on the solutions to smaller-sized subproblems. Note that recursive formula shows how to compute a subproblem of size $k$ from subproblems of
smaller size. This iterative strategy is called dynamic programming which is a bottom-up strategy. (The term “programming” is actually a misnomer and has been used for historical reasons.)

2. Recursive algorithm: An alternate way is to solve the problem recursively using the recursive formula directly, i.e., by writing a recursive code. However, as we saw in the problems we studied, a naive recursive implementation can solve the same subproblems repeatedly and can be very inefficient. Hence we have to the strategy of table look up, also known as memoization. Compute the solution of each subproblem only once and store the result in a table. When the same subproblem is encountered subsequently, get the value by looking up the table. This strategy is called memoization or table look up — which is a top-down strategy, since it is a recursive strategy.

6.9 Worked Exercises

Worked Exercise 6.1. Suppose you want to dial a telephone number consisting of \( n \) digits on a standard 12-button telephone keypad, using the index fingers of both hands. Assuming both fingers start over the 1 button, devise an \( O(n) \) time algorithm using dynamic programming (in particular, give the recursive formulation) which determines which finger to use for each of the \( n \) digits so as to minimize the sum of the total distance traveled by both fingers. Assume that the distance between two buttons \( i \) and \( j \) is given by the function \( \text{dist}(i, j) \). You should first give the recursive formulation, clearly explaining what the subproblems are. Then devise two efficient algorithms that implements your recursive formulation — a recursive algorithm (using top-down, lookup strategy) and an iterative (DP, bottom-up strategy). Analyze the running time of your algorithms.

Solution:

Let \( S \) be the \( n \)-digit number to be dialled. (\( S[i] \) contains the \( i \)th digit). Let the array \( D_i(k) \) be the minimum distance needed to dial (starting from both fingers resting on button 1) the first \( i \) digits of the phone number such that one of the fingers rests on the button \( S[i] \) (after dialing \( S[i] \) ) and the other rests on the button \( k \). (Without loss of generality we will assume that \( S[i - 1] \neq S[i] \), otherwise \( D_i(k) = D_{i-1}(k) \)).

Note that computing \( D_i(k) \) for different \( i \)s and \( k \)s are the subproblems that we need to solve. The number of subproblems is at most \( 12n \), since
there are \( n \) possible values for \( i \) and there are at most 12 values (buttons) for \( k \).

Then for \( i \geq 1 \) we have the following recursive formulation:

\[
D_i(k) = D_{i-1}(k) + \text{dist}(S[i-1], S[i]) \quad \text{if} \quad k \neq S[i-1] \tag{6.1}
\]

\[
= \min_{1 \leq j \leq 12} \{ D_{i-1}(j) + \text{dist}(j, S[i]) \} \quad \text{if} \quad k = S[i-1] \tag{6.2}
\]

and the base cases are: \( D_0(k) = 0 \) if \( k = 1 \), and \( \infty \) otherwise, and we assume that \( S[0] = 1 \). Note that the \( \text{dist} \) function gives the distance between two buttons. Why is (1) and (2) correct? The first case, i.e., when \( k \neq S[i-1] \) is the case where the other button (i.e., \( k \)) after dialing \( S[i] \) should not be \( S[i-1] \); of course, this can happen only by having the other button at \( k \) after dialing \( S[i-1] \) and then moving the finger on \( S[i-1] \) to dial \( S[i] \). In this case, the cost is \( \text{dist}(S[i-1], S[i]) \) and one should move the finger that was on \( S[i-1] \) to dial \( S[i] \) (leaving the other finger sitting on \( k \)).

The second case is when \( k = S[i-1] \) is the case where the other button (i.e., \( k \)) after dialing \( S[i] \) should be \( S[i-1] \); in this case, there are many possibilities. In particular, we look at possibilities \( D_{i-1}(j) \) where the other button after dialing \( S[i-1] \) is \( j \) and we move this other button to \( k \) (leaving one finger on \( S[i-1] \) as desired in this case).

The original problem that we are looking to solve is \( D_n(k) \) and the minimum cost is the minimum of \( D_n(k) \) over all \( k \).

It is straightforward to transform the above formulation into pseudocode which runs in time \( O(n) \): each entry in \( D_i \) array takes \( O(1) \) time to fill and there are only 12 entries to fill, and there are \( n \) such arrays to construct. It is also easy to recover the finger movements from the above arrays: look at the minimum entry in the \( D_n \) array and trace backwards.

**Worked Exercise 6.2.** Consider the 0/1 Knapsack problem with two knapsacks having capacities \( m_1 \) and \( m_2 \) respectively. We have \( n \) items \( x_1, \ldots, x_n \). Item \( x_i \) has an associated weight \( w_i \) and profit \( p_i \). The goal is to fill as many items as possible in the two knapsacks so that we maximize the total profit. An item can be chosen to be put in either of the two knapsacks (if possible) or not chosen at all. Assume that weights, profits, \( m_1 \), and \( m_2 \) are all positive integers.

(i) Show that the optimal substructure property holds for the problem.

(ii) Let \( f_i(y_1, y_2) \) be the optimal solution for the subproblem restricted to the first \( i \) items \( x_1, \ldots, x_i \) such that the two knapsacks have capacities \( y_1 \) and \( y_2 \) respectively. Give a recursive formulation for computing \( f_i(y_1, y_2) \).

(iii) Give the pseudo-code of the resulting algorithm. You can give either
a DP algorithm or a memoized algorithm. Analyze the running time of your algorithm.

(iv) Describe how to change the algorithm so that it also generates the optimum solution (i.e., which items are chosen and in which knapsack).

Solution:

(i) Let $i$ be an item in an optimal solution $S$ with knapsacks have capacities $m_1$ and $m_2$, respectively. Assume first that $i$ is in knapsack 1. Let $S'$ is the solution derived from $S$ for capacities $m_1 - w_i$ and $m_2$. If $S'$ would not be an optimal solution, we could generate a better solution containing item $i$ and $S$ would not be optimal, a contradiction. A similar argument holds when $i$ is in knapsack 2. Hence, the principle of optimality holds and we can use dynamic programming for generating an optimal solution.

(ii) The recursive formulation is similar to the Knapsack problem in the textbook. However, since we now have two knapsacks, there exist two possibilities on where to place item $i$, in addition to the possibility of not using it at all. Hence:

$$f_i(y_1, y_2) = \max\{f_{i-1}(y_1, y_2), f_{i-1}(y_1 - w_i, y_2) + p_i, f_{i-1}(y_1, y_2 - w_i) + p_i\}$$

Base cases: $f_0(y_1, y_2) = 0$ for all $y_1, y_2 \geq 0$ and for all $i > 0$, $f_i(y_1, y_2) = -\infty$ if $y_1 < 0$ or $y_2 < 0$.

(iii) We use $n$ matrices $T_1, T_2, ..., T_n$, each of size $m_1 \times m_2$. We also use a matrix $m_1 \times m_2$ matrix $T_0$ which has all of its entries initialized to 0. $T_i(r, c)$ contains the value of the optimal solution using elements $1, ..., i$ and filling the knapsacks to $r$ and $c$, respectively. We determine $T_i$ from $T_{i-1}$ by using the recursive formulation to compute an entry in constant time. This means all the entries in matrix $T_i$ is computed in $O(m_1 \times m_2)$ time. The matrices are generated in increasing index order of $i$ and the entries in $T_{i-1}$ are used to compute the entries in $T_i$. As there are $n$ matrices, the total time is $O(n \times m_1 \times m_2)$ time. $T_n(m_1, m_2)$ contains the required optimal solution.

(iv) To generate the items to be included into knapsack 1 and 2, the $T$ matrices need to contain an additional backtrack entry which records whether item $i$ was used to generate $T_i(r, c)$, and if it was used, whether it was placed into knapsack 1 or 2. (A variable with values 0, 1, 2 will handle this). To backtrack and generate the sets, start at $T_n(w_1, w_2)$.

- If the backtrack entry is 0, continue in matrix $T_{n-1}(w_1, w_2)$.
- If the backtrack entry is 1, add item $n$ to knapsack 1 and look at the entry $T_{n-1}(w_1 - p_1, w_2)$.
• If the backtrack entry is 2, add item $n$ to knapsack 2 and look at the entry $T_{n-1}(w_1, w_2 - p_2)$.

The process continues in this fashion, costing $O(1)$ time per step, until the knapsacks are filled or all matrices have been visited. This gives $O(n)$ total time.

6.10 Exercises

Exercise 6.11. Consider the following variant of the max-sum problem.

Given an array $A$ of size $n$ containing positive and negative integers, the goal is to determine indices $i$ and $j$, $1 \leq i \leq j \leq n$, such that $A[i] + A[i+1] + ... + A[j] - A[k]$ is a maximum, where $k$ can be any index between $i$ and $j$, i.e., $i \leq k \leq j$. In other words, we are allowed to exclude up to one element (any one) in the contiguous subarray $A[i], A[j]$, including $A[i]$ or $A[j]$. Note that a solution subarray can be contiguous as well — this is captured by allowing $k$ to be $A[i]$ or $A[j]$, which means that the subarray is contiguous.

Give a dynamic programming (DP) algorithm that computes the optimum value of the modified max-sum problem, i.e., we are interested in just computing the value of the optimum solution.

1. Specify the subproblems clearly explaining the notations you use.

2. Give a recursive formulation that relates how you can solve larger-sized subproblems into smaller-sized subproblems. Also mention the base cases.

3. Implement your formulation by a DP algorithm and a memoized algorithm.

4. Analyze the running time of your algorithms.

Exercise 6.12. Let $x$ and $y$ be strings of symbols from some alphabet set of length $m$ and $n$ respectively ($m$ need not be equal to $n$). Consider the operations of deleting a symbol from $x$, inserting a symbol into $x$ and replacing a symbol in $x$ by another symbol (belonging to the alphabet set). Your goal is to design an efficient algorithm using dynamic programming to find the minimum number of such operations (as well as to find the operations that transform $x$ to $y$ that correspond to the minimum number) needed to transform $x$ into $y$. Your algorithm should run in time $O(mn)$. (Hint: This problem is similar to the string alignment problem.)
1. Suppose the two strings are $x = \text{"algorithmic"}$ and $y = \text{"agonistic"}$ and the alphabet set is the English alphabet, what is the minimum number of operations needed to transform $x$ into $y$.

2. Specify the subproblems for computing the minimum number of operations, clearly explaining the notations you use.

3. Give a recursive formulation that relates how you can solve larger-sized subproblems into smaller-sized subproblems. Also mention the base cases.

4. Implement your formulation by a DP (iterative) algorithm.

5. Analyze the running time of your DP algorithm.

6. Modify your algorithm to output the optimal sequence of operations as well.

Exercise 6.13. You start on a long road trip at hotel $a_0$. Along the way there are $n$ hotels, at mile posts $a_1 < a_2 < \cdots < a_n$, where each $a_i$ is measured from the starting point $a_0$. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance $a_n$) which is your destination.

You’d ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you stop at a hotel during a day, you stay there for the rest of that day and continue your trip the next day. If you travel $x$ miles a day, the penalty for that day is $|200 - x|$ (i.e., the absolute value of the difference between $x$ and 200). You want to plan your trip so as to minimize the total penalty—that is the sum, over all travel days, of the daily penalties. Your goal is to give an efficient dynamic programming algorithm that determines the optimal sequence of hotels at which to stop.

(a) Give a recursive formulation for this problem. Clearly explain your notation, in particular, what the subproblems are. Don’t forget to mention the base case(s). (Hint: This problem has a similar approach to the matrix chain multiplication problem.)

(b) Give an efficient algorithm (give pseudocode) based on your recursive formulation. What is the running time of your algorithm and why? Note that your algorithm should not only output the minimum penalty value, but also the optimal sequence of hotels that gives this minimum penalty.

Exercise 6.14. There are $n$ programs that are to be stored in some order on two computer tapes of length $\ell$. Associated with each program $i$ is a
length \( \ell(i), 1 \leq i \leq n \). Assume that \( \sum_{i=1}^{n} \ell(i) \leq \ell \), where \( \ell \) is the length of each tape. A program can be stored in either of the two tapes. If \( S_1 \) is the set of programs on tape 1, then the worst-case access time for a program is proportional to \( \max\{\sum_{i \in S_1} \ell(i), \sum_{i \notin S_1} \ell(i)\} \). An optimal assignment of programs to tapes minimizes the worst-case access time. Design an efficient dynamic programming algorithm (first give the recursive formulation, clearly explaining your notation) to determine the worst-case access time of an optimal assignment, and analyze its running time. Assume that all the lengths of the programs and the lengths of the tapes are integers.

**Exercise 6.15.** Given a sequence of \( n \) (distinct) numbers, \( a_1, a_2, \ldots, a_n \), our goal is to find a longest increasing subsequence (LIS) of the given sequence. For example, if the given sequence is 10, 15, 2, 13, 20, 25, 5, 7, 9, a LIS is 10, 15, 20 (another LIS is 2, 5, 7, 9) of length 4 (no other increasing subsequence has bigger length).

1. Give a dynamic programming algorithm to find the LIS of given sequence of \( n \) numbers in \( O(n^2) \) time. (Hint: Compare the given sequence with the sequence in sorted order and formulate the DP solution.)

2. Give an \( O(n \log n) \) algorithm for the problem. (Hint: Using the fact that the LIS is increasing, can you efficiently add a new element to the existing LIS of the subsequences?)

**Exercise 6.16.** Let \( G = (V, E) \) be a weighted directed graph. (Weights may be positive or negative.) Assume that the graph \( G \) is input as a weighted adjacency matrix \( W \), i.e., \( W_{i,j} = w_{ij} \) if there is an edge between \( i \) and \( j \) and \( w_{ij} \) is the weight of this edge; \( W_{i,j} = \infty \) if there is no edge between \( i \) and \( j \); and \( W_{i,i} = 0 \) for all \( i \in V \).

Given a path between two vertices in \( G \), the bottleneck (value) of the path is the minimum of the weights of its edges. The maximum bottleneck path between two vertices is a path with the maximum bottleneck value (among all possible paths between the two vertices). The goal of this problem is to find the value of the maximum bottleneck path between all pairs of vertices. The goal is to design an \( O(n^3) \) time dynamic programming algorithm that will output the value of the maximum bottleneck path between all pairs of vertices.

(a) Give a recursive formulation for this problem. Clearly explain your notation, in particular, what the subproblems are. Don’t forget to mention the base case(s).
(b) Give the pseudocode for efficient algorithm based on your recursive formulation. What is the running time of your algorithm and why?

Exercise 6.17. Show that all values \( A(i, p) \) in the Knapsack problem (cf. Section 6.7.2) can be computed in \( O(n^2 P) \) time.
Greedy algorithms (or the greedy technique), like Dynamic Programming (DP), are often used in solving optimization problems. As described in Chapter 6, in an optimization problem, typically, we are given a set of inputs and the goal is to find an output that satisfies some constraints. Any output that satisfies these constraints is called a feasible solution. In an optimization problem, we need to find a feasible solution that maximizes or minimizes a given objective function. A feasible solution that does this is called an optimal solution.

Greedy algorithms or the greedy technique can be considered as a special case of DP in the following sense. Recall that in DP, we decompose a problem into one or more subproblems. Most often the case, there will be more than one subproblem arising from the decomposition and one has to solve all the subproblems (recursively) to find the solution to the original problem. In greedy, the situation is special — typically we get only one subproblem to solve in the decomposition. This makes the algorithm particularly efficient, since in each “step” we solve only one subproblem. The key in greedy is choosing the correct (one) subproblem to solve; the non-trivial part is to show that it is enough to consider just this particular subproblem and ignore the other subproblems since they don’t lead to the (optimal) solution. Unlike DP, problems should have a special structure that allows greedy algorithms to work.

The greedy technique works in stages. At each stage, a decision is made depending on whether it is “best” at this stage (hence the name “greedy”). For example, a simple criterion can be whether the decision made at the current stage will lead to a feasible solution or not. Thus, greedy choice (that looks good currently) is made in the hope that it will lead to a globally optimal solution. And if greedy succeeds, there will be
Some important problems admit greedy solutions that are provably optimal, i.e., the greedy algorithm indeed produces the optimal solution. We will see some examples this chapter and in later chapters as well. While greedy does not help in solving a lot of optimization problems, when they do work they are usually very efficient and simple. In some cases, even if they don’t give the optimal solution, they can give a provably “good” solution (one that has a value close to the optimal).

7.1 Problem: Fractional Knapsack

The knapsack problem is a classical problem in optimization.

**Problem 7.1** (Fractional Knapsack). We are given $n$ objects and a knapsack. Object $i$ has a weight $w_i$ and the knapsack has capacity $m$. If a fraction $x_i$, $0 \leq x_i \leq 1$ of object $i$ is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to maximize the total profit earned subject to the knapsack constraint.

Note that, unlike the 0-1 knapsack problem considered in Chapter 6, in the fractional knapsack problem, we are allowed to choose a fractions of objects. This change allows one to design a simple and efficient algorithm that runs in polynomial time and returns the optimal solution.

Before we outline an algorithm for the problem, we make the following easy observations:

- If the sum of all the weights is $\leq m$, then $x_i = 1$, $1 \leq i \leq n$ is an optimal solution.
- All optimal solutions will fill the knapsack exactly (why?).

7.1.1 Greedy Strategies

Consider the following instance of the knapsack problem: $n = 3$, $m = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$. The following are four different feasible solutions each with with a different value:

1. $(x_1, x_2, x_3) = (1/2, 1/3, 1/4)$, $\sum w_i x_i = 16.5$, $p_i x_i = 24.25$.
2. $(1, 2/15, 0)$, 20, 28.2.
3. $(0, 2/3, 1)$, 20, 31.
4. \((0, 1, 1/2), 20, 31.5\).

Many simple “greedy” strategies are possible:

1. Fill the knapsack by including next the object with largest profit. If an object under consideration doesn’t fit, then a fraction of it is included to fill the knapsack. (solution 2 in the above instance)

2. Fill the knapsack by including objects in nondecreasing order of weights. (solution 3).

3. Consider objects in nonincreasing order of \(p_i/w_i\). (solution 4).

We can see that greedy strategy number 3, i.e., choosing objects in nonincreasing of \(p_i/w_i\) gives the highest value (31.25) among the other the four different solutions in the example above. Is this the best possible, i.e., the optimal solution? We will next prove that it is indeed so. Note that this greedy strategy can be implemented in \(O(n \log n)\) time via sorting the elements in nonincreasing order of \(p_i/w_i\).

Note that in a greedy strategy, we consider only one subproblem at a time. For example, in the greedy strategy number 3, at each step, we choose the object with the highest profit to weight ratio among the items that are not yet chosen. The main point in greedy is to show that it is enough to consider just this ordering to get the optimal solution.

### 7.1.2 Proof of Optimality

**Theorem 7.1.** If objects are included in the nonincreasing order of \(p_i/w_i\) then this gives an optimal solution to the knapsack problem.

Before we prove this theorem, we outline the following general proof technique which is typically useful in proving optimality of greedy algorithms. Suppose we want to show that the solution output by a greedy algorithm is indeed the optimal solution. Assume that greedy outputs the solution \((x_1, x_2, \ldots, x_n)\) and the optimal solution is \((o_1, o_2, \ldots, o_n)\). We compare the greedy solution with the optimal solution. If the two solutions differ, then find the first index where they differ (say \(i\)). Then show how to make \(o_i\) in the optimal solution equal to \(x_i\) (of the greedy solution) without loss of the total value of the optimal solution. Repeated use of this transformation, transforms the optimal solution to greedy without losing optimality and thereby shows that the greedy solution is (also) optimal. This proof technique can be called as an “exchange” or “transform” argument.
Proof of Theorem 7.1. Let \( x = (x_1, \ldots, x_n) \) be the solution generated by the greedy algorithm. Since greedy packs the knapsack fully, we have \( \sum w_i x_i = m \). If \( x_i = 1 \) for all \( i \), then clearly the solution is optimal. Let \( j \) be the first index such that \( x_j \neq 1 \). Then, \( x_i = 1 \) for \( 1 \leq i < j \), \( x_i = 0 \) for \( j < i \leq n \), and \( 0 \leq x_j < 1 \).

Let \( y = (y_1, \ldots, y_n) \) be an optimal solution. Note that crucially, we consider the optimal solution in the greedy order; this is crucial for the analysis below. Since \( y \) is optimal, we can assume that \( \sum w_i y_i = m \). Let \( k \) be the least index such that \( y_k \neq x_k \). Then \( y_k < x_k \). To see this consider three possibilities:

- If \( k < j \), then \( x_k = 1 \). Since \( y_k \neq x_k \), and so \( y_k < x_k \).
- If \( k = j \), then since \( w_j x_j = m \) and \( y_i = x_i \) for all \( 1 \leq i < j \), either \( y_k < x_k \) or \( \sum w_i y_i > m \).
- If \( k > j \), then \( \sum w_i y_i > m \).

Suppose we increase \( y_k \) to \( x_k \) and decrease as many of \( (y_k + 1, \ldots, y_n) \) as necessary. This results in a new solution \((z_1, \ldots, z_n)\) with \( z_i = x_i \) for \( 1 \leq i \leq k \) and \( \sum_{k < i \leq n} w_i (y_i - z_i) = w_k (z_k - y_k) \).

We calculate the total profit for \( z \):

\[
\sum_{1 \leq i \leq n} p_i z_i = \sum_{1 \leq i \leq n} p_i y_i + p_k (z_k - y_k) - \sum_{k < i \leq n} p_i (y_i - z_i)
\]

\[
= \sum_{1 \leq i \leq n} p_i y_i + (p_k/w_k) (z_k - y_k) w_k - \sum_{k < i \leq n} (p_i/w_i) (y_i - z_i) w_i
\]

Since we considered the items in greedy order, we have \( p_k/w_k \geq p_i/w_i \), for all \( i > k \). Hence the above expression is lower bounded by:

\[
\geq \sum_{1 \leq i \leq n} p_i y_i + \left( (z_k - y_k) w_k - \sum_{k < i \leq n} (y_i - z_i) w_i \right) (p_k/w_k)
\]

\[
= \sum_{1 \leq i \leq n} p_i y_i.
\]

If \( \sum_{1 \leq i \leq n} p_i z_i > \sum_{1 \leq i \leq n} p_i y_i \), then \( y \) cannot be optimal. Otherwise, \( z \) is optimal and either \( z = x \) or \( z \neq x \); in the case of the latter, repeat the argument with \( z \). \(\square\)
7.2. Problem: Caching or Demand Paging

We consider another classical problem, namely caching or demand paging that is central to modern computer architectures that use cache memory. Assume that a processor has a fast memory device called cache that can store \( k \) items (pages), and a slower, much larger, secondary memory. If a processor requests an item that is in the cache (a hit) it has no cost. If the processor requests an item that is not in the cache (a miss) it costs one unit of time to fetch it to the cache from the secondary memory. When an item is fetched to the cache, another item is evicted. An item is fetched only when it is requested (hence demand paging). Given a sequence of item requests, the goal is to minimize the number of page misses. This is the optimization problem that we will study.

Possible eviction strategies  We can study several different eviction strategies as given below. These are popular strategies that are used in practice.

- Least Recently Used (LRU): Evict the page that was least recently used (i.e., was requested the farthest in the past in the sequence of requests).
- First-in First-out (FIFO): The page that entered the first also is evicted first.
- Least Frequently Used (LFU): The page that has been least frequently accessed (among the current set of pages in the cache) is evicted.

What is the best eviction policy? Suppose we know the complete sequence of page requests in advance. Then the following greedy strategy called the Longest-Forward-Distance (LFD) strategy is the best: If the requested page is not in the cache then evict the page whose next request is the farthest (in the sequence of requests).

Example  Consider the request sequence \( a, b, c, d, a, d, e, a, d, b, c \) and a cache of size \( k = 3 \). LFD strategy will evict \( c \) on the 4th request and \( b \) on the 7th request. However, evicting \( b \) on the 4th step and \( c \) on the 7th step will also give the same misses.
7.2.1 Proof of Optimality of LFD

**Theorem 7.2.** LFD is an optimal eviction strategy for caching.

*Proof.* We will use the proof technique outlined earlier: take any optimal solution and modify it step by step till it becomes equal to the greedy solution, while at the same time not losing optimality. We will show how any optimal algorithm can be modified to behave like LFD without degrading its performance. The proof depends on the following claim.

**Claim 7.1.** Let ALG be an optimal caching algorithm. Let $\sigma$ be any request sequence. For any $i, i = 1, 2, \ldots, |\sigma|$, it is possible to construct an algorithm $\text{ALG}_i$ that satisfies the following 3 properties:

- $\text{ALG}_i$ processes the first $i - 1$ requests exactly as ALG does.
- If the $i$th request results in a page fault, $\text{ALG}_i$ uses the LFD strategy to evict a page.
- $\text{ALG}_i(\sigma) = \text{ALG}(\sigma)$, i.e., $\text{ALG}_i$ has the same number of page misses as ALG on the entire sequence $\sigma$.

We can prove the Theorem by applying the Claim $n = |\sigma|$ times (starting from $i = 1$).

**Proof of Claim** Given ALG we construct $\text{ALG}_i$. Given a set of pages $X$ and a page $p$, let $X + p$ denote the set $X \cup \{p\}$. Without loss of generality, assume that the $i$th request resulted in a page fault. Assume that after processing the $i$th request, let the caches of ALG and $\text{ALG}_i$ contain the page sets $X + v$ and $X + u$ respectively, where $X$ is some set of $k - 1$ common pages and $v$ and $u$ are any pages. Assume that $u \neq v$, otherwise, $\text{ALG}_i$ simply mimics ALG, i.e., makes the same evictions as ALG.

We consider two cases:

**Case 1:** Till $v$ is requested: $\text{ALG}_i$ mimics ALG to serve subsequent requests. If ALG evicts $v$ then $\text{ALG}_i$ evicts $u$, unless $v$ itself has been requested. The number of common pages become $k$ and $\text{ALG}_i$ mimics ALG from now on. Also, number of page misses of $\text{ALG}_i$ is at most that of ALG.

**Case 2:** If $v$ is requested and assume that ALG and $\text{ALG}_i$ have not yet identified. $\text{ALG}_i$ will evict $u$ and the two algorithms identify. But, $\text{ALG}_i$ will incur a page fault and ALG will not. However, by the time $v$ was requested, there must have been at least one request for $u$ after
the $i$th page fault. The first such request incurs a page fault to $ALG$ but not to $ALG_i$. Thus the total number of page faults for $ALG_i$ after servicing $v$ is equal to that for $ALG$.

### 7.3 Problem: Efficient Data Compression

Data compression is a fundamental problem where the goal is to compress a given data as much as possible while still being able to uniquely recover the original data from the compressed data. To compress data we use codes. We encode each character (sometimes even blocks of characters) using an alphabet set (typically binary). The encoded data should be uniquely decodable so as to get back the original data.

Given a string of characters $\alpha \in C$, a variable length code assigns to each character a (binary) code (string of 0s and 1s). Different characters can have different code lengths. Let $d(\alpha)$ be the length of the code for character $\alpha$. Assume that the frequency of character $\alpha$ in the data is $f(\alpha)$. The cost of the code, called the average code length, is

$$B(C) = \sum_{\alpha \in C} f(\alpha) \cdot d(\alpha)$$

**Example 7.1.** Suppose your data consists of only 6 characters — $a, b, c, d, e, f$ — and they occur with frequencies 0.05, 0.1, 0.2, 0.2, 0.4, 0.05 respectively. Let the encoding be as follows: $e = 0$, $d = 10$, $c = 110$, $b = 1101$, $a = 11110$, $f = 11111$. Why is this code uniquely decodable? It is because no code is a prefix of any other code. So by scanning the encoded text from left to right, one can uniquely decode it. For example, the encoded text 110010 can be uniquely decoded to ced.

The cost of this code is $\sum_{\alpha \in C} f(\alpha) \cdot d(\alpha) = 0.05 \times 5 + 0.1 \times 4 + 0.2 \times 3 + 0.2 \times 2 + 0.4 \times 1 + 0.05 \times 5 = 2.3$. Thus the average length of this encoding is 2.3.

Shannon, in his pathbreaking work, showed that a quantity called entropy is a fundamental lower bound on the average length of the code. In other words,

$$B(C) = \sum_{\alpha \in C} f(\alpha) \cdot d(\alpha) \geq H,$$

where $H$ is the entropy of the data set. The entropy is defined as

$$H = \sum_{\alpha \in C} f(\alpha) \log_2(1/f(\alpha)).$$
Figure 7.1: Binary tree representation of the prefix code of Example 7.1.
Example 7.2. The entropy of the data in the above example is: $0.05 \log(1/0.05) + 0.1 \log(1/0.1) + 0.2 \log(1/0.2) + 0.4 \log(1/0.4) + 0.05 \log(1/0.05) = 0.432 + 0.332 + 0.93 + 0.528 = 2.22$.

Prefix Codes. In a Prefix code no codeword is a prefix of another code word. As mentioned in the above example, this allows for easy encoding and decoding. The prefix code can be conveniently represented as a binary tree where the characters are at the leaves and the path taken from the root to a leaf gives the code for that character. We use the usual interpretation of bit 0 for the left child and 1 for the right child. In an optimal code each non-leaf node has two children. Figure 7.1 gives the binary tree representation of the prefix code of Example 7.1. Note that there can be more than one prefix code for an alphabet set.

Exercise 7.2. We are given a text consisting of $k$ characters and their respective frequencies. Show that every prefix code, with binary code lengths $\ell_1, \ell_2, \ldots, \ell_k$ (corresponding to $k$ characters in the data) should satisfy the following inequality:

$$\sum_{i=1}^{k} \frac{1}{2^{\ell_i}} \leq 1.$$ 

7.3.1 Huffman Coding Algorithm

We now present a simple greedy algorithm — called Huffman coding algorithm — that generates an optimal prefix code. The algorithm is quite simple. It builds the prefix tree bottom up. In the first step, it combines the the least two frequent characters and makes them siblings of an internal node. Then it creates a new character whose frequency is the sum of the frequencies of these two. Then the process is again repeated — least two frequent characters are combined and made siblings of a new internal node and so on. In each step the number of characters is reduced by 1. Thus the algorithm finishes in $n - 1$ steps.

Example 7.3. In Example 7.1, the Huffman code is constructed as follows. First, the least two frequent characters are combined — $a$ and $f$ — to give a new character $af$ which has frequency 0.1. In the next step, $af$ and $b$ are combined to obtain $afb$ with frequency 0.2. In the next step, $c$ and $d$ are combined to obtain $cd$ with frequency 0.4. Then $afb$ and $cd$ are combined to obtain $afbcd$ with frequency 0.6. This is then combined with $e$ to obtain the root of the tree. Thus the Huffman code obtained from this tree is: $e = 0, c = 100, d = 101, a = 1100, f = 1101, b = 111$. See Figure 7.2.
Figure 7.2: Binary tree representation of the Huffman (prefix) code of Example 7.1.
The average code length of this code is 2.3. We will show next that this is the optimal average length attainable.

Implementing Huffman’s algorithm efficiently. To implement Huffman’s algorithm efficiently, the key operation is to efficiently choose two characters with the lowest frequencies in each step. This can be accomplished by storing the characters in a min heap data structure (see Appendix B) with their frequencies as key values. Then the two lowest frequencies can be found by performing two EXTRACT-MIN operations each of which costs $O(\log n)$ time. Adding a character whose frequency is the sum of the frequencies of the two lowest frequencies can be done by an INSERT operation which again takes $O(\log n)$ time. Hence the overall time to perform one step of Huffman’s algorithm is $O(\log n)$ and since there are $n-1$ steps, the overall time is $O(n \log n)$. We leave the detailed implementation as an exercise (Exercise 7.3).

Thus we have the following theorem.

**Theorem 7.3.** The Huffman algorithm encodes $n$ characters in $O(n \log n)$ time.

**Exercise 7.3.** Show how to implement the Huffman algorithm in $O(n \log n)$ time.

### 7.3.2 Optimality of Huffman Coding

**Lemma 7.1.** Let $x$ and $y$ be the two characters in $C$ with the lowest frequencies. There is an optimal prefix code for $C$ for which the codewords of $x$ and $y$ have the same length and differ only in the last bit.

**Proof.** Given a tree $T$ of optimal prefix code of $C$ we generate a new tree $T''$ by moving $x$ and $y$ to be siblings with maximum depth. Let $b$ and $c$ be two characters that are encoded by two sibling leaves of maximum depth. Assume $f(b) \leq f(c)$ and $f(x) \leq f(y)$, which implies $f(x) < f(b)$ and $f(y) < f(c)$. Exchange $x$ and $b$ to generate a tree $T'$ and then $y$ and $c$ to generate tree $T''$. See Figure 7.3. We show that the exchange can only reduce the cost.

We first show that exchanging $x$ and $b$ cannot increase the cost.

\[
B(T) - B(T') = \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c) \\
= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b) \\
= f(x)d_T(x) + f(b)d_T(b) - f(x)d_T(b) - f(b)d_T(x) \\
= (f(b) - f(x))(d_T(b) - d_T(x)) \geq 0.
\]
Figure 7.3: Illustration of proof of Lemma 7.1. $x$ and $y$ are the two lowest frequency characters. They can be made siblings at the lowest depth without increasing the cost of the code.
Similar argument for the move from $T'$ to $T''$. 

**Lemma 7.2.** Let $T$ be a tree representing an optimal prefix code for $C$. Consider two characters $x$ and $y$ that appears as siblings in the tree. Let $z$ be their parent in the tree. Consider $z$ a character with frequency $f(z) = f(x) + f(y)$, the tree $T' = T - \{x, y\}$ represents an optimal prefix code for the alphabet $C' = C - \{x, y\} \cup \{z\}$.

**Proof.** See Figure 7.4.

$$B(T) = \sum_{c \in C - \{x, y\}} f(c)d_T(c) + f(x)d_T(x) + f(y)d_T(y)$$

Since, $c \in C - \{x, y\}$, $d_T(c) = d_{T'}(c)$, we can write:

$$B(T') = \sum_{c \in C - \{x, y\}} f(c)d_T(c) + f(z)d_T(z)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_T(c) + (f(x) + f(y))d_T(z)$$

Since, $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have:

$$= \sum_{c \in C - \{x, y\}} f(c)d_T(c) + f(x)(d_T(x) - 1) + f(y)(d_T(y) - 1)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_T(c) + f(x)d_T(x) + f(y)d_T(y) - f(x) - f(y)$$

$$= B(T) - f(x) - f(y).$$

Hence,

$$B(T) = B(T') + f(x) + f(y).$$

If $T'$ is not optimal, there is a tree $T''$ such that $B(T'') < B(T')$. Replacing $z$ with $x$ and $y$ in $T''$ will give a code with cost

$$B(T'') + f(x) + f(y) < B(T).$$

This contradicts the fact that $T$ was optimal. Hence $T'$ is optimal. 

**Theorem 7.4.** The procedure Huffman generates an optimal prefix code.

**Proof.** Lemmas 7.1 shows that at every step, to get an optimal cost tree you can take the two lowest frequency characters and make them siblings
Figure 7.4: Illustration of proof of Lemma 7.2.
at the lowest depth. Lemma 7.2 shows that you can merge the two lowest frequency characters and continue with the remaining set of characters; the resulting tree will still be optimal with respect to the new characters. Hence applying both lemmas repeatedly in each step, shows that the final tree is optimal.

7.4 Problem: Set Cover

The set cover problem is a fundamental optimization problem that arises in many applications. Many other optimization problems can be reduced to solving set cover. The set cover problem is defined as follows:

**Definition 7.4.1 (Set Cover).** Given a ground set \( X \) and a collection of non-empty subsets of \( X \), the set cover problem is to cover the ground set by using a *minimum* number of sets from the collection. (Note that by "cover" we mean that the union of the chosen sets should equal the ground set.)

Consider the following example.

**Example 7.4.** Let the ground set \( X = \{1, 2, 3, \ldots, 10\} \) and the collection of subsets be \( \{\{1, 2, 3\}, \{1, 2, 3, 4, 5\}, \{3, 4, 5\}, \{6, 7, 8\}, \{8, 9, 10\}\} \). One way to cover the ground set is by using the first, third, fourth and the fifth sets. This takes four sets. Alternatively, one can cover using only three sets in the collection, namely, the second, fourth, and the fifth. This is the optimal solution.

The set cover problem is a classic NP-hard optimization problem, and hence it is unlikely to have a polynomial time algorithm that finds the optimal solution. In fact, it is NP-hard to even *approximate* the solution to within a *logarithmic factor* of the optimal. Recall (from Chapter 6.7.1) that the approximation factor for a minimization problem such as set cover is the ratio of the solution output by the algorithm to the optimal solution. We show that a simple greedy algorithm gives a logarithmic factor approximation, which is the best possible.

7.4.1 A Greedy Algorithm

We will show that the following greedy algorithm gives an \( O(\log \Delta) \) approximation, where \( \Delta \) is the size of the largest set. The algorithm proceeds in iterations. In each iteration, it includes the set (which is chosen from the set collection) that covers the most *uncovered* elements.
Greedy Algorithm:

1. $C = \emptyset$
2. while $C \neq U$ do
   - Pick a set $S$ that maximizes the number of (yet) uncovered elements.
   - $C = C \cup S$
3. Output the picked sets.

### 7.4.2 Performance of Greedy Algorithm

We first show that the greedy (sequential) algorithm gives a $O(\log \Delta)$ approximation to the optimal.

**Theorem 7.5.** The greedy algorithm gives a set cover of size that is within $O(\log \Delta)$ factor of the optimal.

**Proof.** Let $r_i$ be the number of elements that are uncovered at the beginning of iteration $i$ by the greedy algorithm. This means that $r_1 = n$. Let $OPT$ be the optimum set cover and let its size be $|OPT|$. At the start of iteration $i$, since the optimum covers all the elements with $|OPT|$ number of sets, there is at least one set which has at least $r_i/|OPT|$ elements. Hence, greedy will cover at least $r_i/|OPT|$ elements in iteration $i$. Thus the number of elements that remain uncovered at the beginning of iteration $i + 1$ is

$$r_{i+1} \leq r_i - r_i/|OPT| = r_i(1 - 1/|OPT|).$$

From the above recurrence, we have the number of elements uncovered at the beginning of iteration $f$ is

$$r_f = r_1(1 - 1/|OPT|)^f = n(1 - 1/|OPT|)^f.$$

Plugging $f = |OPT| \ln(n/|OPT|)$ (where $\ln$ is the natural logarithm), we have

$$r_f \leq n(1 - 1/|OPT|)^{|OPT|\ln(n/|OPT|)} \leq ne^{-\frac{1}{|OPT|}|OPT|\ln(n/|OPT|)} = |OPT|.$$

Note that we used the inequality, $(1 + x) \leq e^x$ for all $x$, to upper bound $(1 - 1/|OPT|)$ by $e^{-1/|OPT|}$.

Hence after picking $f$ sets, there are only $|OPT|$ elements left that are (still) uncovered; this can be covered by picking at most $|OPT|$ more sets. Hence the number of sets picked by greedy is at most

$$|OPT| \ln(n/|OPT|) + |OPT| = O(|OPT| \log(n/|OPT|)).$$
Since $|OPT| \geq n/\Delta$ (Exercise \textit{7.9}), we have that the approximation ratio of greedy to be $O(\log(n/|OPT|)) = O(\log \Delta)$. 

\section*{7.5 Worked Exercises}

\textbf{Worked Exercise 7.1.} Consider a highway that is $M$ miles long consisting of $L$ lanes. At every mile, there exists a lane that is blocked from mile $i$ to (just before) mile $i+1$. This information is recorded in an array $LC[0, 1, \ldots, M-1]$ of length $M$, where $LC(i) = j$ means lane $j$ is blocked from mile $i$ to $i+1$. For example, $LC = [1, 1, 3, 4, 1, 2, 2, 5, 4, 2, 2]$ describes the blockages on a 5-line highway of length 11. The objective is to find a route making the minimum number of lane changes. Changing from one lane into another one has a cost of 1. For the example given above, a solution with one lane change exists: start in lane 5 and change into lane 1 at mile 6 or later. It is not hard to see that more than one optimal solution can exist.

(i) For $M = 12$ and $L = 4$ give an example of blocked positions that maximizes the number of lane changes. State the bound in terms of $M$ and $L$ and explain your answer.

(ii) Consider now a greedy algorithm that starts in the lane having the farthest occurrence of a blockage and, whenever forced to switch lanes, it always switches into the lane having the farthest occurrence of a blockage. In the above example, greedy will start in lane 5 (since lane 5 has the farthest occurrence of the blockage) and at mile 6 change into lane 1 or lane 3 (these two lanes have no further blockages). Describe this greedy approach in pseudo-code and analyze its time performance in terms of $M$ and $L$. Do not assume that either of these variables is a constant. Make sure to describe what data structures are used to select the next lane after a lane change.

(iii) Prove that this greedy approach always generates a solution minimizing the number of lane changes.

\textbf{Solution:}

i) $M = 12$ and $L = 4$. Example of blocked position: $LC = [1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]$ 

In the above example, every lane is blocked every $L$ miles. Thus, every solution must change lane at least once every $L$ miles, in general. This is the upper bound of the maximum number of lane changes in an optimal solution is $M/L$ time.

ii) The greedy algorithm uses the following idea: The path starts in the lane that has its first obstacles the latest. Then, when a lane encounters an obstacle, change into the lane which has the next obstacle furthest down the
road. This approach can be implemented in $O(M + L)$ time by scanning highway of length $M$ and using an array of size $L$ to assist in determining the next lane to switch into. The switch happens at most every $M/L$ positions.

We give more detailed as customary in a detailed algorithmic description as some had problems with the details. We use an array $\text{lane}[i]$ to store the lanes which are blocked since the last change of lane.

```
Require: L // Total number of lanes
         M // Total number of miles
         LC[i] // Array of blocked lane at mile i
1: count ← 0 // Number of lanes that has been blocked
2: changes ← 0 // Number of lane changes that has been made
3: for j ← 0 to L − 1 do
4:    lane[j] ← false
5: for i ← 0 to M − 1 do
6:    if lane[LC[i]] = false then
7:        lane[LC[i]] = true
8:        count ← count + 1
9:    if count = L then
10:       solution[changes] ← (LC[i], i − 1) // Use lane LC[i] until mile i − 1
11:       changes ← changes + 1
12: for j ← 0 to L − 1 do
13:    lane[j] ← false
14:    lane[LC[i]] ← true // The same lane could not be chosen successively
15:    count ← 1
16: for j ← 0 to L - 1 do
17:    if lane[j] = false then
18:       solution[changes] ← (j, M) // Choose the last lane
19: break
20: return solution
```

Time Complexity Analysis
The for-loops at line 3 and at line 16 will take $O(L)$ time each. The for-loop at line 5 will take $O(M)$ time. However, since the if-statement at line 9 will run at most $O(M/L)$ time by the result of part (i), the inner for-loop at line 12 will run at most $O(M/L) \times O(L) = O(M)$ time. As a result, the total running time is $O(M) + O(L) = O(M + L)$.

iii) Suppose our greedy solution is different from the optimal solution. Let
7.6 Exercises

Exercise 7.4. Consider the Huffman algorithm. Assume that the set of characters and their frequencies are given in sorted order, i.e., in increasing order of frequencies. Then show that the Huffman algorithm can be implemented in $O(n)$ time.

Exercise 7.5. You are given a set of $n$ jobs. Associated with job $i$ is an integer $d_i \geq 0$ and a profit $p_i > 0$. For any job $i$, the profit $p_i$ is earned if and only if the job is completed by its deadline. To complete a job, one has to process the job on a machine for exactly one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset $F$ of the jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution $F$ is $\sum_{i \in F} p_i$. An optimal solution is a feasible solution with maximum value.

Now consider the following greedy algorithm to find an optimal solution $F$ of the above job sequencing problem.
1. Sort the jobs in nonincreasing order of the $p_i$’s.
2. Let $a_1, a_2, \ldots, a_n$ are the sorted list of the jobs (job $a_1$ gives largest profit).
3. $F := \{a_1\}$;
4. for $k := 2$ to $n$ do
   4.1 if $(F \cup a_k)$ is a feasible solution
   4.1.1 $F := (F \cup a_k)$;

   a) Show that the above greedy method obtains an optimal solution to the job sequencing problem.

   b) Give an $O(n \log n)$ time algorithm to check feasibility of a solution.

Exercise 7.6. Assume that there is a set of $n$ distinct points on the $x$-axis: $x_1 < x_2 < \cdots < x_n$ (you can take $x_1 = 0$). You don’t know the coordinates of these $n$ points, but you know all the $\binom{n}{2}$ pairwise distances between them, i.e., you know $|x_i - x_j|$, for all $1 \leq i < j \leq n$. You also know which of these pairwise distances have at least one of the endpoints as either $x_1$ or $x_n$, i.e., you know the distances of the form $|x_1 - x_j|$ and $|x_n - x_j|$, for all $j$. But you don’t know which one of the two (whether the distance is from $x_1$ or $x_n$).

Give an algorithm to output the coordinates of $n$ points that are consistent with all the given pairwise inter-distances. (Hint: First determine all the “complementary” pairs of distances, i.e., pairs of distances $(x_i, y_i)$ such that $x_i + y_i = |x_1 - x_n|$. Each such pair corresponds to a point. Then use these to determine the points in a greedy way.)

Exercise 7.7. The bin packing problem is as follows: we are given a set of $n$ objects, where the size $s_i$ of the $i$-th object satisfies $0 < s_i < 1$, and an unlimited supply of unit-sized bins. We wish to pack all the objects into the minimum number of bins. Each bin can hold any subset of the objects as long as the total size of the objects does not exceed 1.

(a) Consider the following greedy algorithm for assigning objects to bins: Assume you have already placed objects $1, 2, \ldots, k - 1$ into bins $1, 2, \ldots, i$. To place object $k$, consider the bins with increasing indices and place object $k$ into the first bin that can accommodate it. If none of the $i$ bins can, place object $k$ into a new bin. Show that this greedy algorithm does not always return the optimal solution to the bin packing problem. Note that you do not have to give an implementation of the algorithm.

(b) Consider another greedy algorithm for assigning objects to bins: Assume you have already placed objects $1, 2, \ldots, k - 1$. Place object $k$ in a used bin that has the maximum amount of available space. If no such bin exists, put the object in an unused (new) bin.

1. Give an example showing that the above greedy algorithm does not always generate the optimal solution (i.e., it may not necessarily pack all the objects in the smallest number of bins).
2. Describe how to efficiently implement the above algorithm using a heap data structure. Analyze the running time of your algorithm. (For your analysis, you can assume an implementation of the heap operations with the appropriate run time.)

**Exercise 7.8.** There are $n$ programs that are to be stored in some order on a computer tape of length $\ell$. Associated with each program $i$ is a length $\ell(i), 1 \leq i \leq n$. Assume that all programs can be stored in the tape (i.e., $\sum_{i=1}^{n} \ell(i) \leq \ell$). To retrieve a particular program from the tape the time needed is proportional to the distance from the beginning of the tape to the position where the program begins plus the length of the program. That is, if the programs are stored in the order $I = i_1, i_2, \ldots, i_n$, the time $t_j$ needed to retrieve program $i_j$ is proportional to $\sum_{1 \leq k \leq j} \ell(i_k)$.

We assume that all programs are retrieved equally often and hence we want to reduce the mean retrieval time (MRT) which is $\frac{1}{n} (\sum_{1 \leq j \leq n} t_j)$. Our goal is to find a permutation for the $n$ programs so that when they are stored on the tape in this order the MRT is minimized. Note that minimizing MRT is equivalent to minimizing $d(I) = \sum_{j=1}^{n} \sum_{k=1}^{j} \ell(i_k)$.

(a) Consider the following example where we have 3 programs and $(l_1, l_2, l_3) = (5, 10, 3)$. What is the (optimal) ordering of these 3 programs that will minimize the MRT and what is the optimal value of MRT?
(b) Give a greedy algorithm to find the optimal order assuming that we have $n$ programs and length of program $i$ is $\ell_i, 1 \leq i \leq n$. What is the running time of your algorithm?
(c) Prove that your greedy algorithm always gives the optimal ordering.

**Exercise 7.9.** Show that in the set cover problem, $|OPT| \geq n/\Delta$. 

7. GREEDY ALGORITHMS
"It is remarkable that this science, which originated in the consideration of games and chances, should have become the most important object of human knowledge... The most important questions of life are, for the most part, really only problems of probability."

– Pierre-Simon, Marquis de Laplace (1749–1827)

"If somebody would ask me, what in the last 10 years, what was the most important change in the study of algorithms I would have to say that people getting really familiar with randomized algorithms had to be the winner."

– Donald Knuth, Randomization and Religion, 1999

In some very real sense, computation is inherently randomized. It can be argued that the probability that a computer will be destroyed by a meteorite during any given microsecond of its operation is at least $2^{-100}$.

– Christos Papadimitriou, Computational Complexity

The influence of probability theory in algorithm design and analysis has been profound in the last two decades or so. This chapter will focus on the interplay between algorithms and probability, in particular, randomized algorithms and probabilistic analysis of algorithms. Randomized algorithms are typically faster, simpler, and easier to implement, compared to purely deterministic (i.e., algorithms that don't involve any randomness) algorithms. (We will see many examples of this throughout the course..."
in many domains.) This chapter will introduce some basic probabilistic tools and techniques that help us in the design of randomized algorithms and their analysis. We will see throughout the course that randomization and probabilistic analysis are immensely used in various algorithmic areas ranging from fundamental algorithms and data structures (such as sorting, fingerprinting, hashing), to network/graph algorithms, communication networks, distributed/parallel computing, data compression, reliable communication, approximation algorithms etc.

This chapter will use basic concepts in probability. It will be useful to refer to the Appendix A on probability if needed.

8.1 Randomization

Randomization in algorithm design and analysis comes in two flavours:

- **Randomized algorithms:** These are algorithms that perform random steps, i.e., one or more steps of the algorithm will involve random coin tosses. Depending on the outcome of the toss (whether HEADS or TAILS, each of which occurs with equal probability), the algorithm will act differently. Randomized algorithms are typically faster and simpler to implement than their deterministic counterparts. However, the correctness of the algorithm or its performance or both will come with probabilistic guarantees. Note that in a randomized algorithm, usually, the input to the algorithm is arbitrary, i.e., no probability distribution on the inputs is assumed.

- **Probabilistic analysis of algorithms:** In probabilistic analysis, performance of the algorithm (which is typically deterministic, but can be randomized as well) is analyzed assuming that the input to the algorithm is chosen according to some probability distribution on the input domain. Probabilistic analysis is useful in analyzing the performance of an algorithm on a “typical” input; often this is called as average case analysis. Probabilistic analysis is very useful in situations where the worst-case performance is much worse compared to the average-case performance. This can explain why certain problems are easier to solve in practice than their theoretical worst-case guarantees. We will see such examples throughout the course.
8.2 Introducing a Randomized Algorithm

Problem 8.1 (Finding a repeated element). Consider an array of \( n \) numbers (assume \( n \) is even) that has \( n/2 \) distinct elements and \( n/2 \) copies of another element. We want an algorithm to identify the repeated element.

We note that any deterministic algorithm for the above will need at least \( n/2 + 2 \) steps in the worst case. Consider an adversary who has full knowledge of the algorithm. The adversary can make sure that the first \( n/2 + 1 \) elements examined by the algorithm are all distinct.

A simple randomized algorithm Now, consider the following randomized algorithm:

- Randomly pick two array elements and check whether they come from different cells and have the same value.
- If they do, output \( \text{true} \) else output \( \text{false} \).

There are generally two questions that one can ask regarding a randomized algorithm: (1) Does the algorithm always give a correct answer? If not, what is the probability that the answer output is correct? (2) What is the running time of the algorithm? Usually the running time is a random variable, and hence one is interested in the average running time of the algorithm.

In the case of the repeated element problem, the questions are: (1) What is the probability that the algorithm will find the repeated element? (2) What is the running time of the algorithm?

Analysis of the randomized algorithm There are \( n^2 \) possible combinations of picking the two elements. Out of these, \( \frac{n}{2} \left( \frac{n}{2} - 1 \right) \) combinations will result in success (Why?). Since every combination is equally likely, the probability that the algorithm will succeed is \( \frac{n/2(n/2-1)}{n^2} = \frac{1/2 - 1/n}{n} = \frac{1}{4} - \frac{1}{2n} \geq \frac{1}{5} \) for \( n \geq 10 \). Thus the probability that the algorithm fails is at most \( 4/5 \).

Reducing the failure probability How to reduce the failure probability? Repeat the algorithm again, if we don’t succeed. Each time we make independent random choices. If we repeat the algorithm 100 times, the probability of failure is \( (4/5)^{100} < 10^{-9} \). If we repeat the algorithm \( c \log n \) times the probability that the algorithm fails is \( < (4/5)^{c \log n} = \frac{1}{n^c \log(n/4)} = 1/n \).
if $c = 1/\log(5/4)$. Thus with $O(\log n)$ repetitions, the algorithm succeeds **with high probability (whp)**, i.e., with probability at least $1 - 1/n^c$, where $c > 0$ is some constant. Thus the algorithm succeeds in $O(\log n)$ repetitions whp.

Note that the notion of “with high probability” is used often in the analysis of randomized algorithms, where the “high probability” is with respect to $n$, the size of the input.

**Definition 8.2.1.** In probabilistic analysis, “with high probability (whp)” (with respect to a parameter $n$), means with probability at least $1 - 1/n^c$, where $c > 0$ is some fixed constant.

**Analyzing the running time** We augment the standard RAM computation model (Chapter 2.4) with a new operation: **Choose a random number uniformly from the set** $\{a_1, a_2, ..., a_k\}$. We assume that this operation takes 1 step (or, constant time). This is reasonable to assume if the size of the set, i.e., $k$, is not too large. In any case, one can show that if tossing a fair coin takes constant time, then choosing a random element from a set of size $k$ can be done in $O(\log k)$ time. (Exercise 8.1)

By our model, each random choice of picking an element randomly can be implemented in 1 step, thus the algorithm (one repetition) takes only constant time. Thus, in $O(\log n)$ time, we can find the repeated element whp.

### 8.3 Verifying Matrix Multiplication

We give a simple randomized algorithm that checks the correctness of multiplying two matrices. Remarkably, using randomization, we show that verification can be done in $\Theta(n^2)$ time, which is significantly faster than the best known algorithms for multiplying two matrices. Strassen’s algorithm in Chapter 5 was one such algorithm, but faster algorithms are known, but these algorithms take time asymptotically larger than $n^2$, i.e., they take $\Omega(n^{2+\varepsilon})$ for some $\varepsilon > 0$. No deterministic algorithms that match the $\Theta(n^2)$ bound are known. We will see later in Chapter 9 that randomization is also very useful (faster) in verifying other operations such as comparing strings.

The problem is as follows: Given three $n \times n$ integer matrices $A$, $B$, and $C$, verify whether $AB = C$. One can accomplish this by just multiplying $A$ and $B$ and checking whether the product matches $C$ element by element.
8.3. VERIFYING MATRIX MULTIPLICATION

Simple matrix multiplication takes $\Theta(n^3)$ time, while the best known algorithm takes at least $\Omega(n^{2.37})$ time.

8.3.1 A Randomized Algorithm

The randomized algorithm is extremely simple and is as follows.

1. Choose a random (column) vector $\bar{r} = (r_1, r_2, \ldots, r_n)^T \in \{0, 1\}^n$, i.e., each entry is chosen with probability $1/2$.

2. Compute $A(B\bar{r})$ and $C\bar{r}$.

3. If $A(B\bar{r}) \neq C\bar{r}$, then output $AB \neq C$; Else output $AB = C$.

The above algorithm takes $\Theta(n^2)$ time because it just involves multiplying a matrix with a vector.

8.3.2 Bounding the Error Probability

Clearly, the algorithm can output a wrong answer, since the algorithm checks using only one vector, e.g., if the vector chosen is the all-zeros vector $((0, 0, \ldots, 0))$, then the check is true, but $AB$ may not be equal to $C$. We now bound the probability that a bad vector choice is made.

**Theorem 8.1.** If $AB \neq C$, and $\bar{r}$ is chosen uniformly at random from $\{0, 1\}$, then

$$\Pr(AB\bar{r} = C\bar{r}) \leq 1/2$$

**Proof.** Let $D = AB - C \neq 0$. Hence, $D$ has some non-zero entry. Without loss of generality, let it be $d_{11}$. $AB\bar{r} = C\bar{r}$ implies that $(AB - C)\bar{r} = 0$, i.e., $D\bar{r} = 0$. In particular, this means that we should have $\sum_{j=1}^{n} d_{1j} r_j = 0$, i.e.,

$$r_1 = -\frac{\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$$

What is the probability of the above equality to hold? We show that the probability of the above happening is at most $1/2$.

It appears difficult to compute the probability of the above event happening under the assumption that all the $r_i$s are chosen independently and uniformly at random, all at once. We instead use the principle of deferred decisions (See Appendix). Instead of choosing the $r_1, r_2, \ldots, r_n$ all at once, we set $r_n, \ldots, r_2, r_1$ one by one in that order. Once we have set all values from $r_n$ to $r_2$, the right hand of the above equality is fixed (it is some particular value). Now $r_1$, the left hand side, can take two values, depending
on the random choice. The equality holds in at most one choice of \( r \). Hence the probability is at most \( 1/2 \). (It can be zero if the right hand side value is not 0 or 1.)

We can get a high probability algorithm, by repeating the above algorithm a few times, say \( O(\log n) \) times. It is easy to show (as we did in the case of the “Repeated element”) that the error probability is reduced to \( 1/n \) or less.

### 8.4 Randomized Quicksort

In Section 5.1.3 we presented Quicksort. This was a deterministic algorithm (algorithm makes no random choices) and its worst case performance can be \( \Theta(n^2) \). Note that this worst-case performance can happen if the input is sorted (or reverse-sorted) or nearly sorted. How to make Quicksort perform well on all inputs? That is, can we make Quicksort run in \( O(n \log n) \) time (i.e., so many comparisons) for all inputs? Enter Randomization. By making use of randomization, we can modify Quicksort to perform well on all inputs (with high probability). More precisely, we will first show that the “average case” or “expected” performance of Quicksort is \( O(n \log n) \). Then we will show the same bound holds not just on average, but also with high probability, i.e., with probability at least (say) \( 1 - 1/n \).

**Main idea and intuition of the algorithm.** The pseudocode of randomized Quicksort is given in Figure 27. The only change, compared to the (deterministic) Quicksort, is that the pivot is chosen randomly in Step 4 of the algorithm (as opposed to choosing the first element). Thus, the pivot is equally likely to be any one of the elements in \( S \). The main intuition as to why this is better than the deterministic algorithm is that regardless of the input, the algorithm has a good probability of choosing a pivot whose rank is in the “middle”. More precisely, the probability that the pivot belongs to the middle one-third of the elements when considered in sorted order (i.e., the elements whose ranks are in the interval \([|S|/3, 2|S|/3]\)) — see Figure 8.1 — is \( 1/3 \). Why is it good to choose the pivot among the middle one-third of the elements? This is because, in this case, the set \( S \) is partitioned in an approximately balanced fashion, i.e., \( \max\{|S_1|, |S_2|\} \leq \frac{2}{3}|S| \). If the sets are partitioned in this fashion, in every level of the recursion tree, then it is easy to see that the depth of the recurrence tree is \( O(\log n) \); since \( O(n) \) comparisons are done in every level of the recursion tree, the total cost is \( O(n \log n) \). However, we should note
that in randomized Quicksort, this kind of balanced partitioning does not happen in every level; however, it happens with enough frequency, which we will show gives us the same $O(n \log n)$ behaviour.

**Algorithm 27** Algorithm Randomized Quicksort $QS(S)$. **Input:** A set $S$. **Output:** The set $S$ in sorted order.

1: if $|S| \leq 1$ then
2:    return $S$
3: else
4:    Let $y$ be an element chosen uniformly at random from $S$.
5:    Compare all elements of $S$ to $y$. Let
6:        $S_1 = \{x \in S - \{y\} \mid x \leq y\}$
7:        $S_2 = \{x \in S - \{y\} \mid x > y\}$.
8: (Elements in $S_1$ and $S_2$ are in the same order as in $S$.)
9:    Return the list (in the following order):
10:        $QS(S_1), y, QS(S_2)$.

**Analysis of randomized Quicksort** Since Quicksort is a randomized algorithm, the number of comparisons performed by the algorithm is a random variable. This random variable takes values in the range $[cn \log n, c'n^2]$, where $c, c'$ are some constants (Why?). Thus the parameter of most interest is the expected value of this random variable, which is the most basic statistic associated with the random variable (and thus characterizes it by a single value). We will show that the expected number of comparisons is $O(n \log n)$. Later, we will show that this bound holds also with high probability.

**Theorem 8.2.** Let the random variable $T$ denote number of comparisons in a run of QuickSort. Then,

$$E[T] = O(n \log n).$$

**Proof.** As is typical in probabilistic analysis, we will write $T$ as a sum of simpler, i.e., 0-1 random variables; then the expectation of $T$ can be computed using linearity of expectation.

Let $s_1, \ldots, s_n$ be the elements of $S$ in sorted order. For $i = 1, \ldots, n$, and $j > i$, define 0-1 (or indicator) random variable $X_{i,j}$, such that $X_{i,j} = 1$ iff
Figure 8.1: Elements are listed in sorted order. The probability that a randomly chosen pivot lands in the middle one-third of the set is $1/3$.

$s_i$ is compared to $s_j$ in the run of the algorithm. Note that $X_{ij}$ “indicates” occurrence of the event denoting the comparison of elements $s_i$ and $s_j$.

The number of comparisons in running the algorithm is

$$T = \sum_{i=1}^{n} \sum_{j > i} X_{i,j}.$$

We are interested in $E[T]$. Using linearity of expectation, we write

$$E[T] = E[\sum_{i=1}^{n} \sum_{j > i} X_{i,j}] = \sum_{i=1}^{n} \sum_{j > i} E[X_{i,j}] = \sum_{i=1}^{n} \sum_{j > i} \Pr(X_{ij} = 1).$$

What is the probability that $X_{i,j} = 1$? To evaluate this, we use the following crucial argument. $s_i$ is compared to $s_j$ if and only if either $s_i$ or $s_j$ is chosen as the pivot before any of the $j - i - 1$ elements between $s_i$ and $s_j$ are chosen as pivots. First we note, that the choosing elements $s_k$, where $k < i$ or $k > j$ will not have any influence of whether $s_i$ and $s_j$ are compared or not. On the other hand, choosing an element $s_k$, where $i < k < j$, will result in $s_i$ and $s_j$ going to different sets in the next level of the recursion and hence will not be compared again.

A pivot is chosen uniformly at random from the elements in the set
8.4. RANDOMIZED QUICKSORT

\{s_i, s_{i+1}, \ldots, s_j\}. Thus,

\[ Pr(X_{i,j} = 1) = \frac{2}{j - i + 1}. \]

Hence,

\[ E[T] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n} \sum_{k=1}^{n-i+1} \frac{2}{k+1} \leq 2n \sum_{k=1}^{n} \frac{1}{k} \leq 2n H_n = 2n \log n + O(n). \]

In the above, we used the fact that \( \sum_{k=1}^{n} 1/k = H_n \), where \( H_n \) is called the Harmonic sum and it can be shown that \( H_n = \Theta(\log n) \).

8.4.1 Probabilistic Analysis of Deterministic Quicksort

There is an entirely different way to probabilistically analyze Quicksort. Consider the deterministic Quicksort (where the pivot is always the first element), but assume a probability distribution on the input. Note that there are \( n! \) different possible inputs to Quicksort (for a input size \( n \)) — corresponding to \( n! \) different orderings of the input set (note that, since we only do comparisons, only the ordering matters). Some permutations are “bad” for Quicksort, i.e., they can result in \( \Theta(n^2) \) comparisons (e.g., sorted order). On the other hand, there are clearly “good” permutations, i.e., they result in \( O(n \log n) \) comparisons. This raises a basic question: What fraction of the \( n! \) permutations are good? It will turn out that except for a tiny fraction, the vast majority of permutations are good. In particular, we will show that if one chooses a random permutation (i.e., a permutation that is equally likely to be any one of the \( n! \) permutations), then the expected performance of deterministic Quicksort is \( O(n \log n) \). (It is crucial to note that the expectation is with respect to the random choice of the input and not due to the randomness in the algorithm.) This analysis sheds some light on why Quicksort does very well in practice, despite its worst-case performance: on a “typical” input (in fact for most inputs), Quicksort’s performance is similar to the best-case, i.e., \( O(n \log n) \).

**Theorem 8.3.** The expected run time of (deterministic) Quicksort on a random input, uniformly chosen from all possible permutations of \( S \) is \( O(n \log n) \).

**Proof.** The proof is almost identical to that of Theorem 8.2. Set \( X_{i,j} \) as before. If all permutations have equal probability, all permutations of
8. RANDOMIZATION AND PROBABILITY ANALYSIS

$s_i, \ldots, s_j$ have equal probability, thus

$$Pr(X_{i,j}) = \frac{2}{j - i + 1}.$$  

Thus, as before, we obtain:

$$E\left[\sum_{i=1}^{n} \sum_{j>i} X_{i,j}\right] = O(n \log n).$$

---

8.5 Randomized algorithms vs. probabilistic (average-case) analysis

We compare the two flavors below:

**Randomized Algorithms:**

- Analysis is true for **any** input.
- The sample space is the space of random choices made by the algorithm.
- Repeated runs are independent.

**Probabilistic Analysis:**

- The sample space is the space of all possible inputs.
- If the algorithm is **deterministic** repeated runs give the same output.

8.5.1 Randomized Algorithm classification

A **Monte Carlo Algorithm** is a randomized algorithm that may produce an incorrect solution. Thus, in a Monte-Carlo algorithm, one has to bound the probability of error. The algorithm for finding the repeated element (Section 8.2) is a Monte-Carlo randomized algorithm, since the output can be incorrect. The same is the case with the randomized algorithm for verifying the product of two matrices. A **Las Vegas algorithm** is a randomized algorithm that **always** produces the correct output. Randomized Quicksort is a Las Vegas randomized algorithm, since the output is always sorted. In both types of algorithms, the run-time is a random variable.
8.6 Randomized Selection

Just like Randomized Quicksort is the randomized counterpart for the (deterministic) Quicksort algorithm, randomized selection (Algorithm 28 gives the pseudocode) is the randomized counterpart of the linear time selection algorithm of Section 8.6. The randomized variant is surprisingly simple and similar to that of randomized Quicksort: the pivot is chosen randomly among the set of elements. Note that the Random-Select algorithm is significantly simpler compared to the deterministic linear time selection algorithm presented in Section 5.2. We will show that this randomized algorithm, though its worst case performance can be $\Theta(n^2)$, its expected performance is $O(n)$.

Algorithm 28 Random-Select $(S, k)$. Selects the $k$-th smallest element in the set $S$ (assume all elements are distinct).

1: $n = |S|
2: \text{if } n \leq 1 \text{ then}
3: \quad \text{Return } S
4: \text{Let } y \text{ be an element chosen uniformly at random from } S.
5: \text{Compare all elements of } S \text{ to } y. \text{ Let}
6: \quad S_1 = \{ x \in S \mid x < y \}
7: \quad S_2 = \{ x \in S \mid x > y \}
8: \text{if } |S_1| = k - 1 \text{ then}
9: \quad \text{Return } y
10: \text{else if } |S_1| \geq k \text{ then}
11: \quad \text{Return Random-Select}(S_1, k)
12: \text{else}
13: \quad \text{Return Random-Select}(S_2, k - |S_1| - 1)

Correctness and Worst-case runtime

Theorem 8.4. The algorithm returns a singleton with the correct value.

The above proof is left as an exercise.

Run time

Theorem 8.5. The worst-case run-time of the algorithm is $\Theta(n^2)$.

The above proof is left as an exercise (Exercise 8.2).
Expected run-time analysis  As explained in randomized Quicksort, intuitively, choosing a random pivot and partitioning around it, creates an approximately balanced partition (see Figure 8.1). There is a constant probability of creating a partition where both of two partitions have at most, say $\frac{2}{3}$ fraction of elements. The probability of getting the above partition is exactly the probability of choosing a pivot in the middle one-third of the elements (considered in sorted order).

**Theorem 8.6.** The expected run-time of the Random-Select algorithm is $O(n)$, where $n$ is the size of the input set.

Let random variable $T(n)$ be the run time of the algorithm on an input size of $n$ elements.

Consider the elements of the set in sorted order (just for analysis sake).

If the pivot is one of the elements belonging to the middle one-third, then both the partitions will have size at most $\frac{2}{3}n$. (Why?) The probability that this happens is $\frac{n}{3} = \frac{1}{3}$.

We can write a recurrence relation for $T(n)$:

$$T(n) \leqslant \left(\frac{1}{3}\right)T\left(\frac{2n}{3}\right) + \left(\frac{2}{3}\right)T(n) + n.$$  

Explanation for the above recurrence:

- With probability $1/3$ (when pivot is chosen in the middle part): The set is partitioned into two parts. Assuming the maximum imbalance, one is of size at most $\frac{2n}{3}$ and the other is of size at least $\frac{n}{3}$. Since $T(n)$ is increasing function of $n$ (Why?), we will assume the worst-situation where the $k$th smallest element (the desired element) is in the larger of the two partitions, i.e., the partition of size $\frac{2n}{3}$.

- With probability $2/3$ (when pivot is chosen in the first or the third part): We assume there is no reduction in the size of the subproblem (again the worst situation).

We take expectations on both sides of the above recurrence:

$$E[T(n)] \leqslant E\left[\left(\frac{1}{3}\right)T\left(\frac{2n}{3}\right) + \left(\frac{2}{3}\right)T(n) + n\right].$$

---

1Strictly speaking, this inequality is a stochastic dominance inequality, since it relates random variables. Note that, unlike the deterministic setting, $T(n)$ is not a value, but rather a random variable. Hence, as such one cannot write inequalities (or equalities) with respect to random variables. In this case, the interpretation is that the random variable on the left is stochastically dominated by the random variable on the right. In other words, if $X$ and $Y$ are two random variables, and we write $X \leqslant Y$, we mean that for any value $a$, $\Pr(X \geqslant a) \leqslant \Pr(Y \geqslant a)$. 
By linearity of expectation, we have:

\[ E[T(n)] \leq (1/3) E[T(2n/3)] + (2/3) E[T(n)] + n. \]

Simplifying the above recurrence, we have:

\[ E[T(n)] \leq E[T(2n/3)] + 3n. \]

We can use the DC recurrence theorem (or induction) to show that the solution to the above recurrence is: \( E[T(n)] = O(n) \).

**8.7 High Probability Bounds: Showing Concentration near Expected Value**

While expectation of a random variable is a very useful parameter, in many applications, it is important to understand how the random variable takes values away from its expected value. In particular, a key question is: What is the probability that the random variable takes a value that is far away from the expectation? If the above probability is large (say a constant), then although the expected value is a desired value, it means that there is a large probability that it will deviate from its expected value. Thus a fundamental task in probabilistic analysis is to compute a probability bound on the deviation from expectation. In particular, in many applications, we would like to show that the random variable is concentrated near the expectation, i.e., the probability of the random variable taking values significantly away from the expected value is small. We will see examples of this paradigm in many applications.

**Probabilistic tools** We will use three major probabilistic tools to bound deviation from expectation: Markov’s inequality, Chebyshev’s inequality, and Chernoff’s bound. We refer to Appendix A (Section C.3) for the statement of these inequalities. Here we will focus on applying these inequalities in algorithmic applications.

**8.7.1 Load Balancing: Balls into Bins Paradigm**

A basic paradigm in probabilistic analysis is the “balls into bins” paradigm. We are given \( m \) balls and \( n \) bins. Assume that each ball is thrown randomly among the bins, i.e., the location of each ball is independently and uniformly chosen at random from the \( n \) possibilities. A common question
that pops up in many applications is to bound the maximum number of balls in any bin. We will use this setting to illustrate the application of Chernoff bound in a direct way.

**Theorem 8.7.** Assume that \( m \) balls are thrown independently and uniformly at random into \( n \) bins. Let \( L \) be the maximum number of balls in any bin.

1. If \( m \geq n \log n \), then \( L = O(m/n) \) whp.

2. If \( m = n \), then \( L = O(\log n) \) whp.

**Proof.** Define the indicator random variable \( X_k^i \) as: 1 if the \( i \)th ball falls into the \( k \)th bin and 0 otherwise. Let \( L_k \) be the number of balls that end up in bin \( k \). Then, \( L_k = \sum_{i=1}^{m} X_k^i \). As usual, we first calculate the expectation of \( L_k \):

\[
E[L_k] = E[\sum_{i=1}^{m} X_k^i] = \sum_{i=1}^{m} E[X_k^i] = \sum_{i=1}^{m} \Pr(X_k^i = 1).
\]

Since \( \Pr(X_k^i = 1) = 1/n \), we have \( E[L_k] = m/n \).

We now apply the Chernoff bound (Theorem C.5, part 1c) to the random variable \( L_k \):

1. If \( m \geq n \log n \), then \( \mu = m/n \geq \log n \) and

\[
\Pr(L_k \geq 6\mu) = \Pr(L_k \geq 6 \log n) \leq 2^{-6 \log n} = 1/n^6.
\]

2. If \( m = n \), then \( \mu = m/n = 1 \) and

\[
\Pr(L_k \geq 6 \log n \mu) \leq 2^{-6 \log n} = 1/n^6.
\]

Note that to get a high probability bound, we need to use \( \Theta(\log n) \) factor deviation.

Let random variable \( L \) denote the maximum number of balls in any bin. Then \( L = \max_{k=1}^{n} L_k \). Note that the random variables \( L_k \) are not independent. Also the \( \max \) function is not linear, so we cannot even compute expectation of \( L \) via linearity of expectation. However, we can appeal to the union bound (Statement C.2), to bound the probability that there are more than \( 6 \log n \) balls in any bin as follows. Let the event \( B_k \) denote the (bad) event that more than \( 6 \log n \) balls landed in bin \( k \). We just showed that this probability is at most \( 1/n^6 \). By union bound,

\[
\Pr(\bigcup_{1 \leq k \leq n} B_k) \leq \sum_{k=1}^{n} \Pr(B_k) \leq n/n^6 = 1/n^5.
\]
Thus the probability that no bin receives more than $6 \log n$ balls is at most $1/n^5$. Note that this bound is $O(m/n)$, when $m \geq n \log n$. The theorem follows.

8.8 High Probability Analysis of Randomized Quick-sort

We showed in Section 5.1.3 that randomized Quicksort makes $O(n \log n)$ comparisons on average. Here, we would like to show much stronger result: The number of comparisons made by Quicksort is $O(n \log n)$ with high probability (whp). Thus, in other words, we are showing that the performance of randomized Quicksort is “concentrated” near its average, almost always.

One way to show the high probability bound is to use the approach of the proof of Theorem 8.2. As in Theorem 8.2, let $T$ be the random variable that denotes the number of comparisons in a run of randomized Quicksort. Then, as in the proof of 8.2, we write

$$T = \sum_{i=1}^{n} \sum_{j>i} X_{i,j}.$$ 

Unfortunately, one cannot use Chernoff bound directly to show concentration of $T$, since it is the sum of dependent random variables. We use a different approach.

We can view an execution of the randomized Quicksort algorithm using the following recursion tree:

- The root of the recursion tree is the (initial) problem of sorting a given set of $n$ distinct numbers; and a leaf is a subproblem of sorting a singleton set.

- An internal node of this tree is a subproblem of sorting a set $S$ (of size greater than 1).

- Its left child (if any) is the subproblem of sorting a set $S_1 \subset S$ consisting of elements smaller than (or equal to) the pivot.

- Its right child (if any) is the subproblem of sorting a set $S_2 \subset S$ consisting of elements larger than the pivot.

One can analyze the number of comparisons made by QuickSort using the above recursion tree. The number of comparisons in each level of
the recursion is bounded by $O(n)$. Thus the total number of comparisons in $(nh)$, where $h$ is the depth of the recursion tree. To show a $O(n \log n)$ high probability bound on the total number of comparisons, it is enough to show that the height of the tree is bounded by $O(\log n)$ with high probability.

**High Probability Analysis**

**Theorem 8.8.** Randomized Quicksort runs in $O(n \log n)$ time with high probability, i.e., with probability at least $1 - 1/n^b$, for some constant $b > 0$.

**Proof.** Suppose the size of the set to be sorted at a particular node is $S$. A node in the execution tree is labeled **good** if the pivot element divides the set into two parts, each of size not exceeding $2|S|/3$. Otherwise the node is called **bad**.

We now have:

1. The probability of a node being labeled good is $1/3$. Why? As explained in Section 8.4, if the pivot is selected among the middle third of the elements, then the size of each of the two subproblems does not exceed $2|S|/3$.

2. The number of good nodes in any root to leaf path is bounded by $\log_{3/2} n < c \log n$ for some (fixed) constant $c$. This is because each good node reduces the sizes of both of the subproblems by a factor of $2/3$. Hence, this can happen for only $\log_{3/2} n$ times — and thus only so many good nodes are possible.

The main idea of the proof is to first bound the length of a root to leaf path in the recursion tree. Let’s fix one root to leaf — call it $P$. Each node in this path is labelled either good or bad. There can be at most $\log_{3/2} n$ good nodes in the path. We now bound the probability that the path cannot be very long. More precisely, it cannot be much longer than the average path length of $3c \log n$ — the reason being that since each node has a probability $1/3$ of being good, on average we expect a path length of $3c \log n$.

We next bound the probability that a path of length $ac \log n$ (for some constant $a > 1$) will have at most $c \log n$ good nodes. The mean $\mu = \frac{1}{3}(ac \log n)$. Using the Chernoff bound (lower tail, Theorem C.5, part 1c)

$$\Pr(X < c \log n) = \Pr(X < (1 - (1 - \frac{3}{a}))\mu) \leq e^{-\mu(1-3/a)^2(1/2)} \leq 1/n^2,$$

for a suitably large (but fixed) constant $a$. 
We have thus shown that a root to leaf path cannot be longer than $ac \log n$ with probability at least $1 - 1/n^2$. To bound the height of the recursion tee, we have to show that all root to leaf paths are not longer than $ac \log n$ with high probability. Let $\ell$ be the number of root to leaf paths ($\ell \leq n$ there are at most only $n$ elements). For $1 \leq i \leq \ell$, let $B_i$ be the (bad) event denoting the probability that path $i$ has length longer than $ac \log n$. By union bound,

$$\Pr(\bigcup_{1 \leq i \leq \ell} B_i) \leq \sum_{1 \leq i \leq \ell} \Pr(B_i) \leq n/n^2 = 1/n.$$  

Thus with probability at most $1/n$ no root to leaf path has length more than $ac \log n$. Since the total work done at each level of the tree is $O(n)$, the running time is bounded by $O(n \log n)$ with high probability. \hfill \Box

### 8.9 A Randomized Data Structure: Skip Lists

Data structures are needed for efficient implementation of algorithms. Choosing the right data structure can also help in simplifying the implementation of an algorithm. Thus an important aim in algorithm design is to organize data in a suitable way to efficiently support certain kinds of operations on the data stored.

Consider the problem of storing and searching a database. For example, an university that wants to store and retrieve student records identified by their IDs — typically called keys. We want a data structure that can INSERT, SEARCH and DELETE, given any key. What is the best way to organize the keys?

**Dictionary Data Structure** A dictionary data structure supports INSERT, DELETE, and SEARCH operations:

- **INSERT($D, k$)** — Insert key $k$ (along with its record) into the dictionary $D$.
- **DELETE($D, k$)** — Remove key $k$ (and its record) from the dictionary $D$.
- **SEARCH($D, k$)** — Return (the record associated with) key $k$ if it is present in $D$. 


Implementing a Dictionary  How to implement a dictionary data structure? One simple way is using an array: store the keys in an array. There are two ways to store the keys in an array — sorted or unsorted which determines the time needed for the 3 operations. Assume that there are \( n \) keys currently in the array \( A = A[1], A[2], \ldots, A[n] \).

We now perform an INSERT, DELETE, or SEARCH operation. The costs are:

- **Unsorted array:** INSERT \( (O(1)) \), DELETE \( (O(n)) \), SEARCH \( (O(n)) \) — Why?
- **Sorted array:** INSERT \( (O(n)) \), DELETE \( (O(n)) \), SEARCH \( (O(\log n)) \) — Why?

Static vs. Dynamic Dictionary  If the dictionary is static — i.e., all the insertions take place in the beginning at once, and there no changes after that — then a sorted array is a good option — SEARCH takes logarithmic time. If the dictionary is dynamic — insertions and/or deletions happen all the time — then array performs poorly. At least one operation takes linear time. We need a better way to implement a dynamic dictionary.

Skip List Data Structure  Skip list is a randomized data structure that efficiently supports the DICTIONARY operations of INSERTION, DELETION, and SEARCH of an element. It is a simpler alternative to deterministic data structures such as AVL trees and red-black trees which have a \( O(\log n) \) worst-case time for all the dictionary operations but are complicated to specify and implement. Skip lists, on the other hand, are conceptually much simpler and also easier to implement. However, since it is randomized, the \( O(\log n) \) time for all dictionary operations is guaranteed with high probability.

In a skip list, many linked lists are joined together. Each linked list is associated with a level as follows. We order these elements (in increasing order) and arrange them in a linked list. This is level 0 and call this list \( L_0 \). Assume that the first element in the list is always \(-\infty\). Recursively, we define level \( i \) of the list as follows. An element in level \( i \) is obtained by selecting each element in level \( i - 1 \) with probability \( 1/2 \) independently of other elements. Copies of the same element in different levels are joined by a doubly linked list (so one can go up and down levels easily). See Figure 8.2.
Figure 8.2: A skip list data structure. The path taken to search for element 11 is shown in red.
Search, Insertion, and Deletion  To search for an element \( e \) we start from the left most element (\(-\infty\)) of the top list and keep going right till we hit the largest element of the list that is smaller or equal to \( e \). We then go to that element and go one level down and do the same, till we reach level 0. (We will assume that we always go to level 0 even if we find the element in a higher level — in practice, anyway, the record associated with the key might be stored only on level 0). Search takes time proportional to the number of levels and the total number of right links traversed.

Insertion of an element \( e \) is done by searching first and going down to level 0. Suppose we reach an element \( f \) in level 0. We insert \( e \) between \( f \) and its successor. We toss a fair coin. If we get Heads, we insert \( e \) one level up (above the copy in level 0), otherwise we stop. We keep inserting \( e \) in higher levels, till we get Tails. Deletion involves search and deleting all occurrences of the element. Insertion and Deletion takes time proportional to Search plus the height of the element.

We next show that both (Search time and height) are bounded by \( O(\log n) \) whp.

8.9.1 Analysis

Height  Let \( H_i \) denote the height of element \( x_i \) — the highest level to which it belongs. Define the height \( H \) of the Skip list to be the maximum of over all the elements’ heights.

Lemma 8.1. \( \Pr(H \geq 3 \log n) \leq 1/n^2. \)

Proof. Each \( H_i \) is geometrically distributed (see Section C.7.2) with parameter 1/2. In other words, the probability for \( H_i \) to be at least \( 3 \log n \), there at least the first \( 3 \log n \) tosses should be Heads and the last should be Tails. The probability of this happening is:

\[
\Pr(H_i \geq 3 \log n) = \sum_{t \geq 3 \log n} \Pr(H_i = t) = \sum_{t \geq 3 \log n} (1/2)^{t+1}
\]

\[
= (1/2)^{3 \log n + 1} \sum_{t \geq 0} (1/2)^t = (1/2)^{3 \log n} = 1/n^3.
\]

Since there are \( n \) elements in total, by union bound, \( \Pr(H \geq 3 \log n) \leq n/n^3 = 1/n^2. \)

Search  The cost of a search is equal to the number of “down” traversals plus number of “right” traversals. The number of down traversals is equal
to the height of the Skip list (note: we assume that we always go to level 0). We start from $-\infty$ at the top list and in every step, we either go down or right. The important thing to note is that each down traversal is due to “Heads” outcome of the corresponding element, i.e., the element that we are traversing as we go down (note that, if we are going down on the $-\infty$ element, the corresponding element is the one that is to the right of $-\infty$). Each right traversal is due to a “Tails” outcome of the corresponding element, i.e., the element that was reached by going right in the current list. This is easy to see if we reverse the search path from the bottom level ($L_0$) — starting from where the search ended — to the $-\infty$ element at the top level. Then, each time we go up, it is because of “Heads” outcome at this element; and each time we go left it is because of a “Tails” outcome of this element (hence this element terminates at this level). Thus the total number of traversals is the number of times needed to toss a fair coin in order to obtain $H$ heads, where $H$ is the height of the Skip list.

Lemma 8.2. Let $T$ be the random variable denoting the total number of traversals for a single search. Then $T = O(\log n)$ w.h.p.

Proof. Since $H < 3 \log n$ w.h.p (by Lemma 8.1), we condition as,

$$
\Pr(T > m) = \Pr(T > m | H < 3 \log n) \Pr(H < 3 \log n) + \Pr(T > m | H \geq 3 \log n) \Pr(H \geq 3 \log n)
$$

$$
\leq \Pr(T > m | H < 3 \log n) + \Pr(H \geq 3 \log n).
$$

For the above we used the identity: For any two events $A$ and $B$:

$$
\Pr(A) = \Pr(A | B) \Pr(B) + \Pr(A | \overline{B}) \Pr(\overline{B}) \leq \Pr(A | B) + \Pr(\overline{B}),
$$

where $\overline{B}$ is the complement of the event $B$.

Fix $m = 72 \log n$. To bound the first term, we bound the probability that there will be less than $3 \log n$ heads when tossing a fair coin $72 \log n$ times. We expect $36 \log n$ Heads. Let $Z$ denote the number of heads. By Chernoff bound (lower tail),

$$
\Pr(Z < 3 \log n) = \Pr(Z \leq (1 - 11/12)(36 \log n)) \leq e^{-3 \log n (11/12)^2 (1/2)} \leq 1/n^2.
$$

Hence, $\Pr(T > m) \leq n^{-2} + n^{-2} \leq 2n^{-2}$.

By union bound, the probability that any search (i.e., of any of the $n$ elements) takes more than $O(\log n)$ time is at most $2/n$. 

8. RANDOMIZATION AND PROBABILISTIC ANALYSIS

8.10 Probably Approximately Correct (PAC) Learning

The notion of probably approximately correct (PAC) learning was introduced by Valiant in the early 1980s which started the field of computational learning theory. The goal of PAC learning is to provide a framework for learning algorithms and this is done by using probability theory. In particular, Chernoff bounds play an important role in the analysis of PAC learning.

Learning algorithms vs. conventional algorithms Learning algorithms are, in some ways, quite different from other algorithms in this book, and in fact, different from the traditional notion of algorithms. The algorithms we have seen so far conform to the traditional notion, i.e., the algorithm — designed for a particular problem — takes an input instance of the problem and outputs a solution to the particular input. For example, for the problem of sorting, the MergeSort algorithm, takes a set of elements and outputs them in sorted order. Learning algorithms are different. They can be considered as “meta” algorithms, since their output is an “algorithm.” Learning algorithms typically (not always) take as input “training” data (i.e., data with known solutions) and output an algorithm (or rule) that performs well on “test” data (i.e., new data or data with unknown solutions).

Classification problem To illustrate the above concepts we will consider a fundamental problem underlying many machine learning applications, namely classification. Here the goal is to efficiently “learn” a good and simple classifying algorithm (or rule) from a given set of training data. The concepts of “learn”, “efficient”, “good” and “simple” have to be formalized. To do this we introduce a very general paradigm of learning called probably approximately correct (PAC) learning. We will then discuss applications to the classification problem.

Suppose we have to classify emails into two categories: spam or not spam. This is a real-world classification problem used everyday in email filters. How to design an algorithm for achieving this? This is actually a difficult problem for a traditional algorithmic application, as the input email can be pretty much anything. However, this set up is ideal for a learning algorithm. A learning algorithm will first be fed a certain number of “training” emails with known classification. If the number of training emails are large, then the learning algorithm may “learn” enough about the setting, i.e., able to figure out a general rule which can be applied to
classify new emails. (This is how a spam filter actually works; it gets better as it sees more email.)

**PAC learning concepts**  PAC learning is a simple but robust framework to address the above problem and gives a formal way to define learning and its complexity. To formalize the concept of learning, Valiant introduced a simple but key assumption, called the *stationarity assumption* which is as follows. We will assume that the training and test instances of a learning problem are coming from the same domain, and more importantly is drawn under the *same* probability distribution.

The problem domain is typically called the *instance space*, denoted by $U$. For example, in the case of emails, the instance space consists of all email messages. The goal is to classify each email into two categories — spam or not spam, i.e., *binary classification*. To be concrete, we will assume our instance space to be points in $d$ dimensions (i.e., $\mathbb{R}^d$) or in $\{0, 1\}^d$. This corresponds to data that is described by $d$ real-valued or Boolean “features.” Boolean features for email messages could be the presence or absence of certain words, for example.

Thus the learning algorithm is given a set of labeled training examples, which are points in instance space $U$ along with their correct classification. This training data could be a collection of email messages, each labeled as spam or not spam. The input to the learning algorithm is the set of training examples and its output will be a classification algorithm that will perform well on new data. Note a trivial learning algorithm simply stores the training data and replies the correct answer on only the training data and replies arbitrarily (say randomly) on new data. This algorithm is useless for learning. A key feature of learning algorithm is *generalization*, i.e., able to come up with a “simple” rule that performs well not only on the training data but on the test data as well. The notion of “simple” can also be formalized.

### 8.10.1 Learning Framework

We assume that data — both training and test data — is drawn from the instance space $U$ according to a fixed (but unknown) probability distribution $D$. This is how we formalize the notion that training and test data are “similar.” Of course, if test and training data have very different classifications (which is possible if they are drawn from different distributions),

---

2We will henceforth deal with binary classification, but the discussion can be generalized to multi-valued classification as well.
then it is not possible for a learning algorithm to predict correctly test data based on training data.

Let’s assume that the instance space is finite throughout this discussion, e.g., as mentioned earlier they can be points in \( \{0, 1\}^d \). A classification function (rule) maps each point (example) of the instance to yes (positive) or no (negative). For example, if the instance space consists of set of all emails (defined by presence or absence of \( d \) features), then a classification function maps each instance to “spam” or “not spam.” Let \( F_c \) be the correct classification rule, i.e., the rule that correctly classifies all the instances. The learning algorithm should try to output a classification rule \( F_a \) such that under the distribution \( D \), the error probability, defined as

\[
\text{err}(F_a) = \Pr(F_c \neq F_a)
\]

is minimized. This error probability called as true error is the probability that an example is not correctly classified by rule output by the learning algorithm assuming that the examples are drawn randomly according to probability distribution \( D \). Thus, we can define true error as follows.

**Definition 8.10.1** (True error). The true error of a classification rule \( F_a \) is simply the probability that it errs on a randomly drawn example from the instance space under distribution \( D \). In other words, it is

\[
\text{err}(F_a) = \Pr_{x \sim D}(F_c(x) \neq F_a(x)),
\]

where \( F_c \) is the correct classification rule.

On the other hand, the training error (also called empirical error) of \( F_a \) is the fraction of training examples in which the function \( F_a \) is incorrect. In other words, we have the following definition.

**Definition 8.10.2** (Training error). Let \( Tr \subseteq U \) be the set of training examples, and \( E = \{u \in Tr | F_a(u) \neq F_c(u)\} \) be the set of training examples which are classified incorrectly by a classification rule \( F_a \). Then the training error of \( F_a \) is defined as

\[
\text{trerr}(F_a) = \frac{|E|}{|Tr|}.
\]

The above notion of errors allow us to define a notion of learning a “general” classification rule (later as we will see it will also allow us to characterize ‘simple’ rules), i.e., a rule that learns something general about the underlying instance space, rather than just “memorizing” the training examples. Indeed, a simple classification rule that simply stores the training examples will return the correct answer on the training examples, but can answer wrongly on new test examples. This is not really “learning”; it is called ”overfitting” the training data. On the other hand, we want the learning algorithm to output a classification rule that has small true error probability (over all examples).
Hypothesis and hypothesis class

To formally handle the above issues such as overfitting, generalization, simplicity etc., a convenient notion is that of a hypothesis and a hypothesis class. A hypothesis is simply a subset of the instance space $U$. There is a one-to-one correspondence between a hypothesis and a classification rule. Given a classification rule $f$, the examples that are classified as positive belong to the hypothesis and those that are classified as negative do not. A hypothesis class over the instance space $U$ is a collection of subsets of $U$. In other words, it is a set of classification rules. We will limit our learning only from a given hypothesis class, i.e., our goal is to learn the best classification rule from the hypothesis class that fits the training data. By "best", we mean a rule that has small true error compared to $F_c$, the correct classification rule.

An example of an hypothesis class for the email classification problem might be set of all disjunctions (i.e., clauses) over $d$ features represented by boolean variables $x_1, \ldots, x_d$ which take values 1 or 0 depending on whether the feature is present or absent. For example, a classification rule for spam can be the disjunction $x_2 \lor \overline{x}_3 \lor x_7$, consisting of presence of one of the features $x_2$ or $x_7$ or absence of feature $x_3$. Other features are irrelevant. That is an email is classified as “spam” if it has $x_2$ or $x_7$ or does not have $x_3$. The number of all such disjunctions is $3^d$ (see Worked Exercise 8.3).

In this way, by choosing a suitable hypothesis class, say by choosing a class where all classifications can be expressed as “simple” functions, one can capture simplicity. Generality is also captured since we are interested in learning only such functions, not arbitrary functions.

The main goal in PAC learning is to obtain bounds on the training set size (i.e., how large should $T_r$ be) for a given hypothesis class, so that one can find an hypothesis (i.e., a classification rule) so that the true error is low. The main idea is that if an hypothesis has low training error, then it also has low true error. Hence any hypothesis in the hypothesis class that has low training error can be chosen.

8.10.2 PAC Learning Theorems

We will show two main theorems of PAC learning. Both relate the size of the training set to the size of the hypothesis class. For these results, we will assume that the hypothesis class is of finite size.

The first result gives a lower bound on the number of training examples needed so that if there is an hypothesis in the hypothesis class that has zero training error, then it will have a small true error, i.e., the hypoth-
esis is probably approximately correct.

**Theorem 8.9 (Zero training error).** Let $HC$ be an hypothesis class and let $0 < \varepsilon \leq 1$ and $0 < \delta \leq 1$ be fixed constants. Assume that the training set examples and the test set examples are drawn independently and randomly from the instance space $U$ under (the same) distribution $D$. If the training set $Tr$ is of size $n \geq \frac{1}{\varepsilon}(\ln |HC| + \ln(1/\delta))$, then with probability at least $1 - \delta$ every hypothesis $h \in HC$ that has training error zero has true error at most $\varepsilon$.

**Proof.** We will prove that with probability at least $1 - \delta$, every hypothesis in $HC$ that has true error more than $\varepsilon$ will have training error greater than zero. This will prove the theorem.

Consider any hypothesis $h \in HC$ that has true error at least $\varepsilon$. We will upper bound the probability that $h$ will have a training error 0. The probability that $h$ will have zero training error is the probability that on all the $n$ training examples, $h$ is correct. Since the training examples are drawn independently and randomly according to $D$, and since $h$ has a true error of at least $\varepsilon$, the probability that $h$ will not err on any of the $n$ training examples is at most

$$(1 - \varepsilon)^n.$$ 

The above bound is for one hypothesis $h$. By using union bound, the probability that any hypothesis in $HC$ will err on any of the $n \geq \frac{1}{\varepsilon}(\ln |HC| + \ln(1/\delta))$ training examples, is at most

$$|HC|(1 - \varepsilon)^n \leq |HC|e^{-\varepsilon n} \leq |HC|e^{-\ln |HC| + \ln(1/\delta)} = \delta.$$

Hence with probability at least $1 - \delta$, the training error will be non-zero.

The above bound on the number of training samples needed for PAC learning is called the sample complexity of the hypothesis class. If $\ln |HC|$ is small then the class is called efficiently PAC-learnable.

We note that the previous theorem is applicable if there is an hypothesis in the hypothesis class that has zero training error. The next theorem can be considered as a generalization of the above result and is applicable in a more general setting where there is no such hypothesis. In particular, suppose the best hypothesis in the hypothesis class has small, but non-zero, training error. Still, we would like to say that for such an hypothesis that has small training error, it has small true error. This is called as the
agnostic setting, i.e., a setting where there is no hypothesis in the class that is “perfect” (with respect to the training set). More precisely, we want to show that every hypothesis in the class that has a small training error will have a small true error as well. This is called uniform convergence, i.e., the training errors and true errors are close uniformly over all hypotheses in the class.

**Theorem 8.10** (Agnostic setting). Let $HC$ be an hypothesis class and let $0 < \varepsilon \leq 1$ and $0 < \delta \leq 1$ be fixed constants. Assume that the training set examples and the test set examples are drawn independently and randomly from the instance space $U$ under (the same) distribution $D$. If the training set $Tr$ is of size $n \geq \frac{1}{2\varepsilon^2} (\ln |HC| + \ln(2/\delta))$, then with probability at least $1 - \delta$ every hypothesis $h \in HC$ satisfies $\|err(h) - trerr(h)\| \leq \varepsilon$, where $err(h)$ and $trerr(h)$ are the true error and training error of $h$ respectively.

**Proof.** Fix an hypothesis $h \in HC$. Let $X_i$ be the indicator random variable that $h$ errs on the $i$th training example, $1 \leq i \leq n$, where $n$ is the number of training examples. Since the examples are drawn randomly and independently from distribution $D$, the $X_i$s are independent. Furthermore, the true error of $h$, $err(h)$, is, by definition, $Pr(X_i = 1) = E[X_i]$. On the other hand, the training error of $h$, $trerr(h)$, is the fraction of training examples that $h$ errs, i.e., $\sum_{i=1}^{n} X_i$.

Let $err(h) = p = Pr(X_i = 1) = E[X_i]$. Let $trerr(h) = X = \frac{\sum_{i=1}^{n} X_i}{n}$. By linearity of expectation, $E[X] = E[\sum_{i=1}^{n} X_i] = np/n = p$. That is, the probability that the training error $X$ (i.e., the error on a random sample) will deviate from its expected (true) error $E[X]$ can be bounded by using a Chernoff bound. We use a convenient variant of the Chernoff bound called the Chernoff-Hoeffding bounds (see Theorem C.6 in Appendix C) to bound the probability that $|err(h) - trerr(h)| > \varepsilon$: $Pr(|X - E[X]| > \varepsilon) \leq 2e^{-2n\varepsilon^2}$.

The above bound is for one hypothesis $h$. Applying the union bound over all hypothesis in $HC$, we have the probability that there is any hypothesis $h \in HC$ such that $|err(h) - trerr(h)| > \varepsilon$ is bounded above by $|HC|2e^{-2n\varepsilon^2}$.

Plugging $n = \frac{1}{2\varepsilon^2} (\ln |HC| + \ln(2/\delta))$ in the above bound, shows that the above probability is at most $\delta$ as required.

\[\Box\]
8.10.3 Examples

Consider the email classification where the instance space is \( \{0, 1\}^d \) consisting of \( d \) boolean features. Consider an hypothesis class \( HC_1 \) that consists of all possible classification rules, i.e., all possible boolean functions over the \( d \) features represented by boolean variables \( x_1, \ldots, x_d \) which take values 1 or 0 depending on whether the feature is present or absent. The number of different boolean functions on \( d \) boolean variables is \( 2^d \). Hence by the PAC learning theorems, the sample complexity of learning an (arbitrary) boolean function is \( O(\ln |HC_1|) = O(2^d) \). This means that one has to see an exponential many examples (exponential in the number of features) to learn this hypothesis class, i.e., one has to see essentially all the possible examples! Hence this hypothesis class is not efficiently learnable.

Consider an hypothesis class \( HC_2 \) that is the set of all disjunctions (i.e., clauses) over \( d \) features represented by boolean variables \( x_1, \ldots, x_d \) which take values 1 or 0 depending on whether the feature is present or absent. The number of all such disjunctions is \( 3^d \) (see Exercise 8.3). Hence by the PAC learning theorems, the sample complexity of learning \( HC_2 \) is \( O(\ln |HC_2|) = O(d) \). This is efficiently learnable, since the number of examples needed is proportional to the number of features.

Worked exercise 8.3 shows how one can learn a disjunction that is consistent with a given set of training examples, assuming such a consistent disjunction exists.

8.11 Worked Exercises

Worked Exercise 8.1. Consider the following insertion sort algorithm to sort a given array of \( n \) numbers. The first and second numbers are compared; if they are out of order, they are swapped so that they are in sorted order. Then the third number is compared with the second; if it is not in the proper order, it is swapped with the second and then compared with the first. Iteratively, the \( k \)th number is handled by swapping it downward until the first \( k \) numbers are in sorted order. Determine the expected (i.e., average) number of swaps that need to be made with insertion sort when the input is a random permutation of \( n \) distinct numbers, i.e., the input is equally likely to be any of the \( n! \) permutations.

Solution:

Let \( s_1, s_2, \ldots, s_n \) be the sequence of \( n \) numbers in sorted order. Let \( X \) be the random variable denoting the total number of swaps performed in
8.11. WORKED EXERCISES

the insertion sort algorithm. Let $X_{ij}$ be the indicator random variable for the event that the two elements $s_i$ and $s_j$ are swapped, that is,

$$X_{ij} = \begin{cases} 
1 & \text{if } s_i \text{ and } s_j \text{ are swapped} \\
0 & \text{otherwise} 
\end{cases}$$

Then we have $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$.

We observe that $E[X_{ij}] = \frac{1}{2}$ since $s_j$ will appear before $s_i$ in exactly half of random permutations. Therefore,

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1$$

$$= \frac{1}{2} \cdot (n-1)n = \frac{n^2 - n}{4}.$$ 

**Worked Exercise 8.2.** In randomized QuickSort (Algorithm 27), given an element $x$ in the input array:

1. What is the maximum number of comparisons that $x$ is involved (i.e., the number of times $x$ is compared with other elements) during the course of the algorithm?

2. What is the minimum number of comparisons that $x$ is involved during the course of the algorithm?

3. What is the average number of comparisons that $x$ is involved during the course of the algorithm?

*Justify your answers.*

Solution: Note that in QuickSort (whether deterministic or randomized), only the pivot is compared with some other element.

1. The maximum number of comparisons any given element is involved is at most $n - 1$, if it chosen as a pivot in the very first iteration; after that it is never involved in any other comparison.

2. The minimum number is 1. This can happen, say, if $x$ is the smallest element in the array and in the very first partition, the pivot is chosen as the second smallest element. The smallest element is compared once with this pivot and then never compared with any other element again.
3. The average number of comparisons for a given element $x$ is a bit tricky to compute directly. But we can use the following reasoning. The total average number of comparisons of randomized QuickSort is $O(n \log n)$ — this we showed in class (also see book), in fact it is $\Theta(n \log n)$. Since there are $n$ elements the average number of comparisons per element is $\Theta(\log n)$. Strictly speaking this argument is not correct, since this is the average over all elements, while we are fixing an element $x$ and asking the average number of comparisons it is involved in (note that in both cases, the average is with respect to the random choices of the QuickSort algorithm). However, we will now argue that the bound of $\Theta(\log n)$ is the correct answer for the latter average as well by a different (correct) argument.

Fix an element $x$ and let its rank be $i$ (i.e., it is the $i$th smallest element). Fix any other element $y$ and let its rank be $j$. Assume without loss of generality $i < j$ (otherwise, the argument below is identical with roles of $i$ and $j$ interchanged). We compute the probability that $x$ and $y$ are compared. Note that this will happen only if one of the two is chosen as a pivot. Furthermore it will happen only if $x$ or $y$ is chosen as a pivot before any of the elements whose ranks are in-between $i$ and $j$ (i.e., ranks $i + 1, i + 2, \ldots, j - 1, j$) are chosen as pivots. Why? Because, if a pivot is chosen from the in-between elements first, then $x$ and $y$ will go to different subproblems and will never be compared. Hence the probability that $x$ and $y$ are compared is the probability of choosing $x$ or $y$ from the set of $j - i + 1$ elements ($x$ and $y$ and all the in-between elements) which is $\frac{2}{j - i + 1}$.

Hence the average number of comparisons that $x$ is involved is simply the weighted sum of values, where the values are either 0 or 1 (0 means no comparison, and 1 means no comparison) and the weights are the probabilities. We don't have to worry about 0 values, since they contribute 0. Thus the average is simply the sum of the probabilities of $x$ being compared with another element, summed over all other elements:

$$\sum_{j \neq i} \frac{2}{j - i + 1} = 2 \sum_{k=1}^{n-1} \frac{1}{k} = 2 \sum_{k=1}^{n-1} \frac{1}{k} = 2(1/2+1/3+\cdots+1/n) \leq 2(1/2+1/3+\cdots+1/n) = 2H_n = \Theta(\log n).$$

**Worked Exercise 8.3.** (1) Show that the number of different disjunctions on a set of $d$ boolean variables is $3^d$.

(2) Show how to learn a disjunction that is consistent with a given set of training examples, assuming a disjunction exists that agrees with all the training examples. State and prove your learning algorithm.
8.12. EXERCISES

Solution:

(1) Each disjunction can either have the variable, or its complement, or not have the variable at all. Thus there are three possibilities for each variable. Hence the total number of different disjunctions is $3^d$.

(2) Start with the disjunction $F = x_1 \lor \neg x_1 \lor x_2 \lor \neg x_2 \cdots \lor x_d \lor \neg x_d$ (this evaluates to 1). Consider every negative example (if any). Any feature that appears in a negative example, cannot appear in disjunction. Hence remove the feature variable that appears in any negative example from $F$. For example, if feature $x_2$ appears in a negative example, then remove $x_2$ from $F$ (but leave $\neg x_2$ in $F$). Clearly, such a disjunction satisfies every positive example (if any). Also every negative example, will not be satisfied, since every feature that is present in the negative example only its complement is included. Hence the resultant disjunction satisfies all training examples.

8.12 Exercises

Exercise 8.1. Show that if tossing a fair coin takes constant time (i.e., bit 0 or 1 can be chosen randomly, each with probability $1/2$), then choosing a random element from a set of size $k$ can be done in $O(\log k)$ time.

Exercise 8.2. Prove the correctness and worst-case run time of Random-Select (i.e., in the worst case, it runs in $\Theta(n^2)$ time).

Exercise 8.3. There are three boxes. A magician places a prize in a box chosen uniformly at random. You are asked to pick a box (of course you don’t know which box contains the prize). Then the magician opens an empty box (this is not the one you have picked). He asks you whether you would like to change your pick (to the other unopened box) or keep your pick. What is the best strategy? Justify with correct probabilities.

Exercise 8.4. Consider the following game. We start with one blue ball and one red ball in a bin. We repeatedly do the following: choose one ball from the bin uniformly at random, and then put the ball back in the bin with another ball of the same color. We repeat until there are $n$ balls in the bin. Show that the number of red balls is equally likely to be any number between 1 and $n - 1$.

Exercise 8.5. Give an example of three random events $A$, $B$, and $C$, such that any two of these events are pairwise independent, but all three together are not independent (i.e., $\Pr(A \cap B \cap C) \neq \Pr(A) \Pr(B) \Pr(C)$).
Exercise 8.6. Given an \( n \)-element set \( S \), let \( X \) and \( Y \) be two random subsets of \( S \) (each chosen independently and uniformly at random from all possible subsets of \( S \)). Determine the probability of the following events:

1. \( \Pr(X \supseteq Y) \).
2. \( \Pr(X \cup Y = S) \).

Exercise 8.7. The following program finds the maximum value in a nonempty set of numbers \( S \) of size \( n \).

1. \( \max = -\infty \)
2. for \( i = 1 \) to \( n \) do 
   2.1 choose an element \( x \in S \) uniformly at random 
   2.2 \( S = S - \{ x \} \) (remove \( x \) from \( S \)) 
   2.3 if \( x > \max \) then 
      2.3.1 \( \max = x \)
3. output \( \max \)

Show that the expected number of times the assignment statement in line 2.3.1 is executed is \( O(\lg n) \).

Exercise 8.8. We are given a set of \( n \) points in 2 dimensions. Let \( p[0] \) and \( p[1] \) be the first and second coordinates (respectively) of point \( p \). Each coordinate of each point is an integer. For any two distinct points \( p_i \) and \( p_i' \), we say that \( p_i' \) is worse than \( p_i \) if, for \( j \in \{0, 1\} \), \( p_i[j] \geq p_i'[j] \), and there exists some \( j' \in \{0, 1\} \) such that \( p_i[j'] > p_i'[j'] \). We say that a point \( p_i \) is interesting if it is not worse than any other point.

Assume that each coordinate of each point is chosen independently and uniformly at random from the set \([1, \ldots, n^2]\), i.e., each component is a random integer chosen from the first \( n^2 \) integers (all choices are made independently).

What is the expected number of interesting points among the \( n \) points?

(Hint: Use principle of deferred decisions. See also Problem C.2 in Appendix.)

Exercise 8.9. A permutation \( \pi : [1, n] \to [1, n] \) can be represented as a set of cycles as follows. Let there be one vertex for each number \( i, i = 1, \ldots, n \). If the permutation maps the number \( i \) to the number \( \pi(i) \), then a directed arc is drawn from vertex \( i \) to vertex \( \pi(i) \). This gives a directed graph that is a set of disjoint cycles. Notice that some of the cycles could be self-loops. What is the expected number of cycles in a random permutation of \( n \) numbers?
8.12. EXERCISES

Exercise 8.10. Give an alternate analysis to the randomized select algorithm using an approach that is similar to the one we used to analyze randomized quicksort in Section 8.4. That is let $X$ be a random variable denoting the total number of comparisons performed during an execution of the randomized select algorithm. Show by a direct analysis (a la Quicksort) that $E[X] \leq cn$ for some constant $c$ (of course, $n$ is the number of elements in the input). You should not assume anything about the rank of the element that you are trying to find, i.e., $k$ in $\text{Random-Select}(S,k)$ can be arbitrary.

Exercise 8.11. Assume $m$ balls are thrown randomly in $n$ bins.

1. Assume $m = n$. What is the expected number of empty bins (bins that don’t have any ball)?

2. What should be the minimum $m$ if every bin should contain at least one ball with high probability, i.e., with probability at least $1 - 1/n$?

Exercise 8.12. For any two events $A$ and $B$, show that

$$\Pr(A) = \Pr(A|B) \Pr(B) + \Pr(A|\overline{B}) \Pr(\overline{B}),$$

where $\overline{B}$ is the complement of the event $B$.

Exercise 8.13. There is some wastage of space in a Skip list — elements can be repeated and stored multiple times. Show that, if there are $n$ distinct elements, the total space used over all elements is $O(n)$ in expectation and with high probability. (Hint: For the high probability bound, bound the number of elements, level by level.)

Exercise 8.14. You need a new staff assistant, and you have $n$ people to interview. You want to hire the best candidate for the position. When you interview a candidate, you can given them a score, with the highest score being the best and no ties being possible (in other words, there is a strict ordering among the candidates based on the quality from best to worst). You interview the candidates one by one. Because of your company’s hiring practices, after you interview the $k$th candidate, you either offer the candidate the job before the next interview or you forever, lose the chance to hire that candidate. We suppose candidates are interviewed in a random order, chosen uniformly at random from all $n!$ possible orderings.

We consider the following strategy. First interview $m$ candidates and reject them all; these candidates give you an idea of how strong the field is. After the $m$th candidate, hire the first candidate you interview who is better
than all of the previous candidates you have interviewed (note that, you may end up not hiring anyone under this strategy).

(a) Let $E$ be the event that we hire the best assistant, and let $E_i$ be the event that the $i$th candidate is the best and we hire him. Determine $\Pr(E_i)$, and show that

$$\Pr(E) = \frac{m}{n} \sum_{j=m+1}^{n} \frac{1}{j-1}. $$

(b) Bound $\sum_{j=m+1}^{n} \frac{1}{j-1}$ to obtain

$$\frac{m}{n} (\ln n - \ln m) \leq \Pr(E) \leq \frac{m}{n} (\ln(n-1) - \ln(m-1)).$$

(c) Show that $\frac{m}{n} (\ln n - \ln m)$ is maximized when $m = n/e$, and explain why this means $\Pr(E) \geq 1/e$ for this choice of $m$.

(Note that $\ln n$ is the natural logarithm of $n$.)

**Exercise 8.15.** Suppose we have a Monte Carlo algorithm for a problem that runs in time $T(n)$ for any input size of size $n$ and produces the correct answer (assume that the output is of size at most $O(n)$) with probability $p(n)$. We are also given a verification algorithm for the problem, that takes an input and its solution and verifies whether the given solution is indeed a correct solution; it takes time $V(n)$ to verify $(n$, as given above is the input size of the problem). Using the Monte Carlo algorithm and the verification algorithm, show how to obtain a Las Vegas algorithm that always returns the correct answer. What is the expected running time of the Las Vegas Algorithm?

**Exercise 8.16.** We are given two unsorted sets of numbers $S_1$ and $S_2$, each of size $n$ (the numbers in each set are distinct). For each number in $S_1$, there exists (exactly) one corresponding equivalent number in $S_2$. (For example, think of $S_1$ and $S_2$ as consisting of numbers that denote weights; weights in $S_1$ are in kilograms, whereas weights in $S_2$ are in pounds, and for every weight in $S_1$, there is an equivalent weight in $S_2.$). The goal is to group the equivalent numbers in $S_1$ and $S_2$ in pairs. The only operation allowed is comparing any number (say $a$) in $S_1$ with any number (say $b$) in $S_2$; the comparison will tell whether $a$ is equivalent to $b$, or whether $a < b$ or $a > b$. Two numbers belonging to the same set cannot be compared. Give a randomized algorithm for the problem that takes $O(n \log n)$ comparisons in expectation and with high probability.

**Exercise 8.17.** We showed that the Random-Select algorithm of Section 8.6 runs in $O(n)$ time in expectation. Can we show a high probability analysis
for Random Select as we did for randomized Quicksort? That is, can we show that Random-Select runs in $O(n)$ time with high probability (i.e., probability at least $1 - 1/n$)?

Exercise 8.18. Consider the following algorithm called Good-Select for finding the $k$th smallest element which is a modification of the Algorithm Random-Select. We showed that algorithm Random-Select had $O(n)$ expected number of comparisons. Our motivation for Good-Select is to design an algorithm that has $O(n)$ comparisons, not just in expectation, but with high probability.

The main change in Good-Select is how the pivot is chosen (Step 4 of Algorithm 12). Instead of choosing a random pivot, we do the following: we choose $4 \log n$ elements independently and uniformly at random from the set (the elements are sampled with replacement — so the same element can be selected more than once). Then we sort these $4 \log n$ elements (say, using mergesort) and take the middle element of this sorted set, i.e., the median. This median element is taken as the pivot. The rest of the algorithm proceeds as usual — see full pseudocode in Algorithm 29.

Algorithm 29 Good-Select $(S, k)$. Selects the $k$-th smallest element in the set $S$ (assume all elements are distinct). $n$ is the number of elements in the input set.

1: if $|S| \leq 4 \log n$ then
2: Sort $S$ (using MergeSort).
3: Return the $k$-th smallest element of $S$.
4: Choose $4 \log n$ elements independently and uniformly at random (with replacement) from $S$.
5: Sort these $4 \log n$ elements (using MergeSort) and find their median; let $y$ be this median.
6: Compare all elements of $S$ to $y$. Let

\[ S_1 = \{x \in S \mid x < y\} \]
\[ S_2 = \{x \in S \mid x > y\} \]

7: if $|S_1| = k - 1$ then
8: Return $y$
9: else if $|S_1| \geq k$ then
10: Return Good-Select($S_1, k$)
11: else
12: Return Good-Select($S_2, k - |S_1| - 1$)
1. Show that, with high probability, $y$ is a “good” pivot — i.e., with high probability, the sets $S_1$ and $S_2$ are approximately balanced.

2. Show that the running time (i.e., number of comparisons) of Good-Select is $O(n)$ with high probability.

Exercise 8.19. In a casino, there is a machine where you can play the following game: if you put $1$, you will get either $2$ with probability $1/10$, or $4$ with probability $1/20$, or $100$ with probability $1/1000$ or nothing with the remaining probability. Each game is independent of the others. The casino lost $1000$ in the first 10000 games. Obtain a Chernoff bound for the probability of this happening.

Hint: Use the following Chernoff bound that works even for non-Bernoulli random variables, i.e., random variables that can take values other than 0 or 1.

Let $X_1, \ldots, X_n$ be independent random variables such that $X_i \in [a, b]$ for $1 \leq i \leq n$, i.e., the random variables take values in the range $[a, b]$, where $a, b$ ($a < b$) are some numbers (can be positive or negative). Let $X = \sum_{i=1}^{n} X_i$. Then for any $t > 0$,

\[
\Pr(\left| X - \mathbb{E}[X] \right| \geq t) \leq 2e^{-\frac{2t^2}{(b-a)^2}}.
\]

Exercise 8.20. Consider the instance space $\{0, 1\}^d$ for the classification problem (consisting of $d$ boolean features) and let the hypothesis class $HC$ be the set of all 3-CNF formulas. That is, each hypothesis can be described by a conjunction (AND) of clauses and each clause is a disjunction (OR) of exactly three literals. An example $h \in HC$ is $(x_1 \lor x_2 \lor \hat{x}_3) \land (x_9 \lor \hat{x}_1 \lor \hat{x}_2)$. Note that there is no limit on the number of clauses in a 3-CNF formula. Assume that the correct classification rule is given by some (unknown) 3-CNF formula $F$, and we want to PAC-learn it.

1. Give a lower bound on the number of training examples needed so that with probability at least $1 - \delta$, all 3-CNF formulas consistent with all the training examples have true error at most $\varepsilon$. Note that $0 < \delta, \varepsilon < 1$ are fixed constants. As assumed in the PAC setting, the training and test examples are drawn independently and randomly from the same distribution $D$ (unknown).

2. Give a polynomial time algorithm (polynomial in $d$) to output a 3-CNF formula that is consistent with all the training examples.
Exercise 8.21. Assume that you throw \( n \) balls independently and randomly in \( n \) bins (i.e., each ball has a probability of \( 1/n \) of landing in any bin and the probability is independent of any other ball). Let random variable \( X \) denote the number of bins that are empty (did not receive any ball).

1. Compute \( E[X] \).

2. Show that \( X \) does not exceed \( E[X] + O(\sqrt{n \log n}) \) with high probability.

Hint: Use the fact that the usual Chernoff bound (upper tail) for sum of independent indicator random variables holds even under the condition given below.

Let \( X_1, X_2, \ldots, X_l \in \{0, 1\} \) be random variables such that for all \( i \), and for any \( S \subseteq \{X_1, \ldots, X_{i-1}\} \), \( \Pr[X_i = 1 | \bigwedge_{j \in S} X_j = 1] \leq \Pr[X_i = 1] \). (Such random variables, can be called negatively correlated, since the probability that \( X_i = 1 \) given that \( X_j = 1 \) (\( j < i \)) decreases compared to the unconditioned probability that \( X_i = 1 \).) Then for any \( 0 < \delta < 1 \), \( \Pr[\sum_i X_i \geq (1 + \delta)\mu] \leq e^{-\mu\delta^2/3} \), where \( \mu = \sum_i E[X_i] \).
Hashing and Fingerprinting

Hashing and fingerprinting are related (and essentially similar) techniques that map numbers and objects from a large domain to a much smaller set of values (typically, positive integers). Both are very efficient and used in lots of applications.

9.1 Hashing

Hashing is a simple but powerful technique with a lot of algorithmic applications. It also can be used to support Dynamic Dictionary operations: INSERT, SEARCH, and DELETE (see Section 8.9).

Here is a simple way to motivate the idea of hashing. Assume keys are positive integers: \( k_1, k_2, \ldots \). Assume that you have an array \( A \) whose size is at least as large as the largest key. A fast way to insert/delete/search is to store key \( k_i \) as a boolean value at \( A[k_i] \), i.e., \( A[k_i] = 1, k_i \) is inserted, otherwise 0. This simple strategy gives a \( O(1) \) (i.e., constant) time for INSERT, DELETE, and SEARCH operations. What is the (big) drawback of this strategy? The drawback is that the array’s size depends on the key values and not on the number of keys. Ideally, if we have \( m \) keys, we want to store it using an array of size \( O(m) \), regardless of the values of the keys.

Hashing is a clever way to implement the above strategy without the above drawback. Assume that the keys are drawn from a Universe set \( U \). \( U \) is typically large, say \( |U| = 10^9 \) — e.g., the set of all social security numbers. Also, the number of keys — \( n \) — is usually much smaller than \( |U| \), e.g., consider \( |U| \) to be the population of USA, whereas \( n \) is the number of students in a University.
Figure 9.1: A hash function $h$ maps elements from a large set $U$ to a smaller set $\{0, \ldots, m - 1\}$.

**Hash function** Let the array — called as hash table — be of size $m$ — used to store the keys. (Usually $m$ will be somewhat larger than $n$.) A hash function $h$ is simply a function from $U$ to $\{0, \ldots, m - 1\}$. A hash function maps keys from a Universe (typically a large set) to a much smaller set of numbers $\{0, \ldots, m - 1\}$. See figure 9.1. Note that $m$ is the size of the hash table.

**Example** The figure 9.2 gives a simple example of hashing values to a hash table.

**Hash function properties** What properties should a hash function satisfy? We would like it have the following properties.

- Should be a deterministic function, i.e., $h(k)$ is a fixed hash value for a fixed input $k$. (Why?)

- Maps values from a large universe $U$ to a much smaller valued set $\{1, \ldots, m\}$, where $|U| >> m$.

- Easy to compute.

- Distribute the keys more or less evenly among the $m$ slots — balanced key distribution. (Why?) That is, there should not be too many collisions: A collision occurs when $h(x) = h(y)$, for two keys $x \neq y$. 
Figure 9.2: Two elements are hashed to the same value (2), causing collision.
Choosing an Hash Function  The following are some examples of hash functions. Note that they all satisfy that the range of the function is the set \{0, \ldots, m - 1\} and hence, the result of the hash provides an index to a hash table with size \(m\). This is the reason why these functions have a “mod \(m\)” at the end.

- \(h(k) = k \mod m\), where \(k\) is the key and \(m\) is the size of the hash table. Typically \(m\) is chosen to be a prime (Why?). In practice, it has been observed that a prime not close to a power of 2 is generally a good choice for \(m\).

- \(h(k) = (ak + b) \mod m\), where \(a > 0\) and \(b \geq 0\) are some constants. If \(m\) not a prime, let \(p > m\) be a prime: \(h(k) = ((ak + b) \mod p) \mod m\). Typically, \(p\) is chosen larger than the size of the largest key in the universe.

Note that it is hard to avoid collisions. Indeed, since we are mapping \(|U|\) elements to \(m\) values, for every hash function, there is a subset \(B\) of size at least \(|U|/m\) elements (“bad” set) that map to the same value (because of pigeon hole principle). Thus whatever hash function we use, if an “adversary” chooses a subset of elements from the bad set, then all of them hash to the same value! We will use randomization to get around this seemingly impossible task.

**Exercise 9.1.** A hash function \(h : U \rightarrow M\) is perfect for a set \(S\) if it causes no collisions for pairs in \(S\).

- For any given set \(S\) such that \(|S| \leq m\), show that there is always a perfect hash function for \(S\).

- For any \(S\) such that \(|S| > m\), show that there is no perfect hash function.

- If \(|U| > m\), show that there is no perfect hash function for all \(S \subset U\), such that \(1 < |S| \leq m\).

### 9.1.1 Hashing by Chaining

Even with the “best” choice of an hash function, there will be some collisions. We store all the elements hashed to the same slot as a list, e.g., a linked list or container (vector). Given a hash table \(T[1..m]\), \(T[k]\) points to a container of all the elements hashed to \(T[k]\). See figure 9.3.
Figure 9.3: Hashing with chaining. The elements that hash to the same value (2) are stored in a linked list.
9. HASHING AND FINGERPRINTING

**INSERT, SEARCH, and DELETE** All the above operations are now easy under chaining.

- **INSERT** \( k \): add \( k \) to the container \( T[h(k)] \).
- **SEARCH** \( k \): search in \( T[h(k)] \).
- **DELETE** \( k \): search in \( T[h(k)] \) and delete the element.

The time for each of the above operations is proportional to the length of the respective lists.

**Analysis of Hashing with Chaining** Let \( n \) be the number of keys stored in the table and \( m \) be the number of slots. The worst case insert time is clearly, \( O(1) \) whereas the worst case search/delete time is \( O(n) \). (Why?)

What is the search/Delete Time for a “good” hash function (defined below). Define load factor \( \alpha = \frac{n}{m} \). Assume that the hash function is good, i.e., it distributes keys (approximately) evenly among the slots. (We will later show how such a function can be constructed that has such a property on average.) That is, the number of keys hashed to a particular slot is \( O(\alpha) \). Assume that \( h(.) \) can be computed in time \( O(1) \). Then the overall time for search (and deletion) is \( O(1) + O(\alpha) = O(\alpha) \). Thus, if we choose \( m = \Theta(n) \), then search/delete takes \( O(1) \), i.e., constant time.

### 9.2 Universal Hashing*

Universal hashing mitigates the problem of designing a hash function that works well for all subsets of the domain \( U \). Although (as mentioned earlier), no one hash function is good for all subsets of \( U \), choosing a random hash function from a family of hash functions allows us to avoid the worst case situation of having too many collisions. Thus randomizing the choice of hash function allows us to foil the adversary — there is no “worst-case” instance for a random function from the family. Of course, one should note that once an hash function is chosen, it is not changed henceforth. The point is that the adversary is oblivious to the random choice of the hash function and hence it is hard for it to come up with a “bad” set without this knowledge.

What property should the family of hash functions satisfy? This is captured in the following definition.
9.2. UNIVERSAL HASHING

**Definition 9.2.1** (Universal family of hash functions). A family $H$ of hash functions from $U$ to $M$ is **universal** if for all $x, y \in U$, such that $x \neq y$, and for a randomly chosen function $h$ from $H$

$$\Pr(h(x) = h(y)) \leq \frac{1}{m}.$$ 

There are two questions that arise: (1) Why is the above property useful (or what does it buy us)? and (2) Can one design a family of functions that satisfy the above property which says that the collision probability between any two distinct elements is small, i.e., at most $1/m$. We answer both these questions next. The next theorem answers the first question.

**Theorem 9.1.** Assume that we hash $n$ keys to a table of size $m$, $n \leq m$, using a hash function $h$ chosen at random from a universal family of hash functions, and we resolve collisions by chaining. Then searching for a key takes expected time $O(\alpha)$. Similarly, INSERT and DELETE take expected time $O(\alpha)$.

**Proof.** For every pair of distinct keys $k$ and $l$, define the indicator random variable (r.v.) $X_{kl} = 1$ if and only if $h(k) = h(l)$, else $X_{kl} = 0$. Then $E[X_{kl}] = \Pr(X_{kl} = 1) \leq 1/m$.

For each key $k$, define the r.v. $Y_k$ that equals the number of keys other than $k$ that hash to the same slot as $k$ in the table $T$: $Y_k = \sum_{l \in T, l \neq k} X_{kl}$. Thus

$$E[Y_k] = \sum_{l \in T, l \neq k} E[X_{kl}] \leq \sum_{l \in T, l \neq k} 1/m.$$ 

We have two cases:

1. If key $k \notin T$: $E[Y_k] \leq n/m = \alpha$.
2. If key $k \in T$: $E[Y_k] \leq (n - 1)/m < \alpha$.

Thus search takes $O(\alpha)$ time. \hfill \qed

**Corollary 9.1.** Using universal hashing and collision resolution by chaining in a table with $m$ slots, it takes expected time $O(s)$ to handle any sequence of $s$ INSERT, SEARCH, and DELETE operations containing $O(m)$ INSERT operations.

**Proof.** There are at most $O(m)$ insert operations, and hence at most so many elements. Thus the load factor $\alpha = O(1)$. Thus each operation takes $O(1)$ expected time. Using linearity of expectation, $s$ operations take $O(s)$ time. \hfill \qed
9.2.1 Constructing universal hash functions

It is clear that universal families exists because we can show that the set of all functions from $U$ to $M$ is universal (this is left as an exercise). Let $|U| = u$ and $|M| = m$. Then, there are $m^u$ functions from $U$ to $M$ — requires $u \log m$ bits to choose, represent and store as a table — which is as large as the size of the universe.

We next show how to construct a universal family of hash functions that is of small size.

Choose a prime number $p$ such that $0 \leq k \leq p - 1$, i.e., $p$ is larger than the largest key in $U$. Define the following. Let $\mathbb{Z}_p = \{0, 1, \ldots, p - 1\}$ and $\mathbb{Z}^*_p = \{1, 2, \ldots, p - 1\}$.

Note that since $p$ is a prime number $\mathbb{Z}_p$ and $\mathbb{Z}^*_p$ are groups — the first being the additive group modulo $p$ and the second is the multiplicative group modulo $p$ (See Appendix D). These two groups simply mean that in the former the group operation is addition and the latter it is multiplication. Since these are groups we can essentially do arithmetic like that we are used to when doing arithmetic using integers. The only difference being that this group is finite — arithmetic operations are done modulo $p$. That is, we can add and subtract (in $\mathbb{Z}_p$) and multiply and divide (in $\mathbb{Z}^*_p$). The fact that these are groups, in $\mathbb{Z}_p$ every element has an additive inverse (and hence subtraction can be done) and in $\mathbb{Z}^*_p$ every element has a multiplicative inverse (and hence division which is equivalent to multiplying by the inverse is possible).

Exercise 9.2. Show that the set of all functions from $U$ to $M$ is a universal hash family.

A universal family of hash functions

**Definition 9.2.2 ($H_{p,m}$ family of hash functions).** Let

$$h_{a,b}(k) = ((ak + b) \mod p) \mod m$$

be a hash function for any $a \in \mathbb{Z}^*_p$ and any $b \in \mathbb{Z}_p$. Then we define

$$H_{p,m} = \{h_{a,b} : a \in \mathbb{Z}^*_p, b \in \mathbb{Z}_p\}.$$

We will show next that the above family is universal. We note that the above family can be efficiently stored, evaluated, and chosen in $O(\log p)$ space and time. Indeed, choosing a random hash function is easy from this set: simply choose $a$ and $b$ uniformly at random from their respective
9.2. **UNIVERSAL HASHING**

ranges. Note that $O(\log p) = O(\log u)$ if $p$ is chosen such that $u < p < 2u$ (A prime always exists between any number $n$ and $2n$ — this is known as *Bertrand’s postulate*).

**Theorem 9.2.** $\mathcal{H}_{p,m}$ is universal.

**Proof.** We show that $\mathcal{H}_{p,m}$ satisfies the definition of universal hash function. That is given any two distinct elements $x$ and $y$, the probability that they collide on randomly chosen hash function from the family is at most $1/m$.

Consider two distinct keys $k$ and $\ell$ from $\mathbb{Z}_p$. For a given hash function $h_{a,b}$ we let

\[
\begin{align*}
  r &= (ak + b) \mod p. \\
  s &= (a\ell + b) \mod p.
\end{align*}
\]

$r \neq s$ because

\[
r - s = a(k - \ell)(\mod p)
\]

and both $a$ and $k - \ell$ are nonzero (mod $p$). Thus there is no collision at the “mod $p$” level if the two keys are distinct.

Moreover, each distinct pair $(a, b)$ with $a \neq 0$ yields a distinct pair $(r, s)$ with $r \neq s$ because we can solve for $a$ and $b$ uniquely given $r$ and $s$:

\[
a = ((r - s)((k - \ell)^{-1} \mod p)) \mod p
\]

and

\[
b = (r - ak) \mod p.
\]

Thus, for given pair of distinct keys $k$ and $\ell$, if we pick $(a, b)$ randomly then $(r, s)$ is also picked randomly. Thus the probability that keys $k$ and $\ell$ collide is equal to the probability that $r = s(\mod m)$ with $r \neq s$. For a given $r$, the number of values of $s$ such that $r \neq s$ and $r = s(\mod m)$ is at most $\lceil p/m \rceil - 1$. The reason for this is that there are at most $\lceil p/m \rceil$ values between 0 and $p - 1$ that leave the same remainder when divided by $m$. We subtract 1 since we are looking for such values that does not include $r$ itself. Now we have,

\[
\lceil p/m \rceil - 1 \leq (p + m - 1)/m - 1 = (p - 1)/m.
\]

(We use the fact that $\lceil a/b \rceil \leq (a + b - 1)/b$.)

Thus probability that $s$ collides with $r$ is $\leq 1/m$. \qed
9.2.2 An Application of Universal Hashing: Perfect Hashing

By perfect hashing we mean that the worst case time to search is $O(1)$. We will design a hashing scheme that gives a worst-case search time of $O(1)$ and takes linear space, assuming that the set of keys are static. That is, the keys are inserted once into the hash table in such a way that the time to search for any key is constant.

A two-level hashing scheme  Clearly, any one hash function will cause collision. We will use the following two-level hashing scheme. We will use hashing with chaining. For elements that hash to the same slot in the first level, we will use a secondary hash table that will further hash these elements to another table. To search for an element, use the first level hash function to find the slot in the primary hash table and then use another hash function (corresponding to this slot) to hash again to find the slot in the secondary hash table.

1. Use universal hashing (from the class $\mathcal{H}_{p,m}$) to hash a given set of $n$ keys to a table of $m = n$ slots.
2. For $0 \leq j \leq m - 1$:
   - let $n_j$ be the number of elements hashed to slot $j$;
   - use universal hashing again (from the class $\mathcal{H}_{p,n_j^2}$) to hash these in a secondary table of size $n_j^2$.

We show the following theorem.

**Theorem 9.3.** There exists a two-level hashing scheme that uses $O(n)$ space (for a set of $n$ keys) and guarantees a worst case search time of $O(1)$ assuming that the set of keys is static.

To prove the theorem we need the following lemma which shows the main idea behind the scheme, namely, if we use $n^2$ slots for $n$ keys, then there exists a hash function which does not cause any collisions.

**Lemma 9.1.** If we store $k$ keys in a hash table of size $k^2$ using universal hashing (from the class $\mathcal{H}_{p,k^2}$), then the probability of there being any collision is $< 1/2$. Thus, there exists an hash function that will not produce any collision. Moreover, such a function can be found in expected constant number of trials.

**Proof.** Let random variable $X$ denote the number of collisions. Then we can write $X = \sum_{i \neq j} X_{ij}$, where $X_{ij}$ is the indicator r.v. for the event that
two distinct keys $i$ and $j$ collide. In other $X_{ij} = 1$ if they collide, otherwise $0$. By the property of universal hashing, $E[X_{ij}] = \Pr(X_{ij} = 1) \leq 1/k^2$, since the number of slots in the hash table $m = k^2$. Then, by linearity of expectation:

$$E[X] = E[\sum_{i \neq j} X_{ij}] = \sum_{i \neq j} E[X_{ij}] = \left(\frac{k}{2}\right)1/k^2 < 1/2.$$ 

Using Markov’s inequality (see Appendix A), we have

$$\Pr(X \geq 1) \leq E[X] < 1/2.$$ 

Thus if we choose a random hash function from the family, with probability at least $1/2$, there will be no collision. We hash all the elements and check and see if there is any collision. If there is a collision, we try again with another random hash function choice. Note that the probability of “success” (i.e., getting no collision) is at least $1/2$. Thus we have a geometric distribution (see Appendix A) with probability of success at least $1/2$. Thus the expected number of times we repeat before we get success is at most 2.

While the above lemma shows that if we use $k^2$ slots for $k$ keys then there is choice of a hash function that causes no collision. However, $k^2$ is quadratic in the number of keys; we desire $O(k)$ space. This is the reason for using a secondary level hash function. The first level hash reduces the number of collision per slot and then we use the “quadratic” blow up on the number of elements hashing to the same slot to eliminate collisions. Let $n_j$ be the number of elements hashing to slot $j$ in the primary table. Then the amount of space in the secondary table is $n_j^2$. We bound the total space as follows.

**Lemma 9.2.** In the two-level hashing scheme the expected amount of storage required for all the secondary hash tables is (assuming $m = n$):

$$E \left[ \sum_{j=0}^{m-1} n_j^2 \right] < 2n.$$ 

**Proof.** Using the identity, $a^2 = a + 2\binom{a}{2}$:

$$E \left[ \sum_{j=0}^{m-1} n_j^2 \right] = E \left[ \sum_{j=0}^{m-1} (n_j + 2\binom{n_j}{2}) \right].$$
\[
\Pr[n_j \geq 1] \leq n + 2 \left( \binom{n}{2} \right) \frac{1}{m} = n + 2n(n - 1)/2n < 2n.
\]

Note that \(\sum_{j=0}^{m-1} n_j\) is simply the sum of all elements which is \(n\) and \(\sum_{j=0}^{m-1} \binom{n_j}{2}\) is the same as the total number of (pairwise) collisions. Hence, \(E[\sum_{j=0}^{m-1} \binom{n_j}{2}] = \binom{n}{2} \frac{1}{m}\), by the property of universal hashing and linearity of expectation.

Thus, if we choose universal hashing in the secondary level (i.e., for a random choice from the class \(\mathcal{H}_{p,n^2}\), for each primary slot \(j\)), we expect to use no more than \(2n\) space. Using Markov’s inequality gives the following corollary.

**Corollary 9.2.** In the two-level hashing scheme the probability that the total storage used for secondary hash tables exceeds \(4n\) is \(< 1/2\). Hence the expected number of trials needed to have at most \(4n\) secondary space is constant.

Thus to get perfect hashing, we first try a few (constant on average) choices of the primary hash function to get at most \(4n\) secondary table space. Then, we hash all the elements that hash to the same primary slot using the quadratic blow up idea — we need to repeat only a constant number of times till we find a secondary hash function that causes no collision.

**9.3 Hashing Strings and Fingerprinting**

In many applications, we want to hash strings, e.g., storing words in an English dictionary or hashing a name to a database. The idea is simple: convert the string into a number and then hash the number. Let’s associate a number with each character, e.g.: \(a \rightarrow 1, b \rightarrow 2, c \rightarrow 3, \ldots, z \rightarrow 26\).

A more convenient way is to associate the ASCII value of the character. Characters are represented using the American Standard Code for Information Interchange (ASCII). Each character is represented using a number between 0 and 127 (8 bits). Figure 9.4 shows the ASCII table.
### 9.3. HASHING STRINGS AND FINGERPRINTING

#### Figure 9.4: ASCII code for characters.

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Null</td>
<td>32</td>
<td>20</td>
<td>Space</td>
<td>64</td>
<td>40</td>
<td>$</td>
<td>96</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>Start of heading</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>Start of text</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>End of text</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>End of transmit</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>Enquiry</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>Acknowledge</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>Audiobell</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>Backspace</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>Horizontal tab</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>Line feed</td>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>J</td>
<td>106</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>Vertical tab</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>Form feed</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>Carriage return</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>Shift out</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>Shift in</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>Data link escape</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
<td>112</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>Device control 1</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
<td>113</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>Device control 2</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
<td>114</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>Device control 3</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
<td>115</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>Device control 4</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
<td>116</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>Neg. acknowledge</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
<td>117</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>Synchronous idle</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
<td>118</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>End trans. block</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
<td>119</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>Cancel</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
<td>120</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>End of medium</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
<td>121</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>Substitution</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
<td>122</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>Escape</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>`</td>
<td>123</td>
<td>7B</td>
<td>(</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>File separator</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
<td>124</td>
<td>7C</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>Group separator</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td></td>
<td></td>
<td>125</td>
<td>7D</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>Record separator</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>Unit separator</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>\</td>
<td>127</td>
<td>7F</td>
<td>\</td>
</tr>
</tbody>
</table>
Hashing a string  Given a string $x = x_0, ..., x_{n-1}$ consisting of the characters $x_0 \ldots, x_{n-1}$, define a hash function of $x$:

$$h_p(x) = \left( \sum_{i=0}^{n-1} a(x_i)r^{n-i-1} \right) \mod p.$$ 

where $a(x_i)$ is the ascii value of character $x_i$, $r > 1$ is a radix, and $p$ is a (suitably chosen) prime. We will call $h_p(x)$ as the “fingerprint” of $x$.

Since there are at most 128 characters, we can choose $r = 128$ — then we can interpret the number $\sum_{i=0}^{n} a(x_i)r^{n-i-1}$ as equivalent to the string represented in base 128. But as we will see later, we can choose $r$ to be other values as well, e.g., we can choose $r = 2$ as well. $p$ should be chosen larger than 128 and also somewhat larger than $n$ — say, $100n$, where $n$ is the number of characters in the string. We will see shortly why such a $p$ is a good choice.

**Example**  Let’s hash the string $x = "ALGO"$. In ASCII, $A = 65$, $L = 76$, $G = 71$, $O = 79$. Let’s choose $r = 128$ and $p = 131$.

$$h(x) = (65(128)^3 + 76(128)^2 + 71(128) + 79) \mod 131 = 137569231 \mod 131 = 105.$$  

Computing the above sum involves heavy calculation, as the sum becomes large. Can we do this better?

**Mod properties**  We use the following properties of mod function in conjunction with Horner’s rule (see below) to reduce the sizes of the intermediate sums.

- **sum rule**: $(a + b) \mod p = (a \mod p + b \mod p) \mod p$.
- **product rule**: $(a \times b) \mod p = ((a \mod p) \times (b \mod p)) \mod p$.
- **exponentiation rule**: $(a^p \mod p) = ((a \mod p)^p) \mod p$. The exponentiation rule is very useful in avoiding large integers.

**Horner’s rule**  Suppose we want to calculate:

$$h(x) = (65(128)^3 + 76(128)^2 + 71(128) + 79) \mod 131.$$  

Rewrite the sum as follows — called Horner’s rule:

$$(65(128)^3 + 76(128)^2 + 71(128) + 79) = 128(128(65 + 76) + 71) + 79.$$
9.3. HASHING STRINGS AND FINGERPRINTING

We compute the above sum by repeated modding — mod using the product and sum rule repeatedly.

In the above example, we proceed as:

\[(128 \times 65) \mod 131 = 67.\]

\[(67 + 76) \mod 131 = 12.\]

\[128(12) \mod 131 = 95.\]

\[95 + 71 \mod 131 = 35.\]

\[128(35) \mod 131 = 26.\]

\[26 + 79 = 105 \mod 131 = 105.\]

Thus we get the final answer is 105, but more importantly, all intermediate sums in the computation are less than \((128)(130) = 16640\).

The pseudocode for computing hash function using Horner’s rule is shown in Figure 30. From the pseudocode, it is clear that the running time is \(O(n)\), where \(n\) is the length of the string.

**Algorithm 30** Fingerprint\((x)\). Input: A string \(x = x_0, \ldots, x_{n-1}\) and output is \(h_p(x) = \left(\sum_{i=0}^{n-1} a(x_i) r^{n-i-1}\right) \mod p\). \(a(x_i)\) is the ascii value of character \(x_i\), \(r\) is a radix, \(p\) is a (suitably chosen) prime.

1: \(sum = 0\)

2: \(\textbf{for } i := 0 \textbf{ to } n - 1 \textbf{ do}\)

3: \(\text{sum } = (((r \times sum) \mod p) + a(x_i)) \mod p\)

4: \(\text{Output sum}\)

9.3.1 Problem: Comparing Strings

**Problem 9.1.** Given two strings \(a = a_0, \ldots, a_{n-1}\) and \(b = b_0, \ldots, b_{n-1}\) we want to check for equality, i.e., whether the two strings are the same or not.

The straightforward way to solve the above problem is to just compare the two strings character by character, i.e., if \(a_i = b_i\), for all \(0 \leq i \leq n - 1\). This takes \(O(n)\) character comparisons. A better way is to use algorithm Fingerprint to solve this. We compare the fingerprint of the two strings:

- If they are not the same, then output “No”.
- If they are the same, then output “Yes”.

Why is this better? Is this always correct? We answer these two questions next.
9. HASHING AND FINGERPRINTING

Why is this better? Fingerprint is much smaller than the string itself. Suppose we are trying to check the equality of two long strings, e.g., DNA sequences (can have billions of characters). Assume that one sequence is in Singapore and the other in USA. The naive way is to send all the \( n \) characters (\( n \) is the length) from Singapore to USA. The fingerprint way is to send the fingerprint of the string in Singapore to USA; compare it with the fingerprint of the string in USA. For this only \( O(\log p) \) bits need to be communicated, since the fingerprint is less than \( p \) and hence can be represented in \( O(\log p) \) bits. \( p \) is chosen to be \( \Theta(n) \), and hence only \( O(\log n) \) bits need to be communicated! Thus equality of two \( n \) length strings can be checked by comparing only \( O(\log n) \) bits.

Correctness There is a chance that fingerprint gives the wrong answer. Compare the fingerprint of the two strings:

- If they are not the same, then output “No”. No error here.
- If they are the same, then output “Yes”. Error is possible — false positive.

The false positive case occurs when there is a collision: \( h_p(x) = h_p(y) \), for \( x \neq y \). We will show that the probability that this happens is small, if \( p \) and \( r \) are chosen appropriately in the fingerprint function.

9.3.2 Fingerprinting Theorem

**Theorem 9.4.** Let the length of the string be \( n \). Let \( p \) be a prime number that is larger than 127 (the largest ascii value) and also larger than \( \lambda n \), where \( \lambda \) is some parameter (which determines the probability of error). Let \( r \) be the radix chosen uniformly at random in \([0, p-1]\). Then the probability of having a collision, i.e., for any two different strings \( x \) and \( y \) (each of length \( n \)),

\[
Pr(h_p(x) = h_p(y)) = \frac{n}{p} \leq \frac{1}{\lambda}.
\]

**Proof.** Assume that \( x \neq y \), but \( h_p(x) = h_p(y) \), i.e.,

\[
\left( \sum_{i=0}^{n-1} a(x_i)r^i \right) \mod p = \left( \sum_{i=0}^{n-1} a(y_i)r^i \right) \mod p.
\]

That is, in terms of congruence:

\[
\sum_{i=0}^{n-1} a(x_i)r^i \equiv \sum_{i=0}^{n-1} a(y_i)r^i \pmod{p}.
\]
9.3. HASHING STRINGS AND FINGERPRINTING

Thus we can write a congruence equation (consider \(r\) as the unknown variable):

\[
\sum_{i=0}^{n-1} (a(x_i)r^i) - \sum_{i=0}^{n-1} a(y_i)r^i = \sum_{i=0}^{n-1} (a(x_i) - a(y_i))r^i \equiv 0 \pmod{p}.
\]

Note that \(r\) and the coefficients of \(r\) are in \([0, p-1]\) (and not all coefficients are zero, since \(x \neq y\)). Thus the above is a polynomial congruence (mod \(p\)) equation of degree (at most) \(n-1\). Lagrange’s Theorem says that it has at most \(n-1\) solutions, i.e., only so many choices of \(r\) for which the equation is satisfied. If we choose \(r\) randomly in \([0, p-1]\), then the probability that one of the above choices is chosen is at most \(n/p\).

\[\square\]

9.3.3 Application to Pattern Matching

**Problem 9.2.** Problem: We are given a long text string \(X = x_1, \ldots, x_n\) and a shorter pattern string \(Y = y_1, \ldots, y_m\). The **pattern matching** problem is to find an occurrence of \(Y\) in \(X\) or return NONE if no such occurrence exists.

Pattern matching is a fundamental problem with lots of applications. For example, when you search for a word in a webpage or try to do a “find” in your editor, you are solving the pattern matching problem. There is an easy algorithm that takes \(O(mn)\) time: simply compare the pattern string with the text string starting at every position \(i\), \(1 \leq i \leq n - m\). We design a simple algorithm using fingerprinting that takes \(O(m + n)\) time with high probability.

**An \(O(m + n)\) time algorithm** We are given as Input: \(X = X = x_1, \ldots, x_n\) and \(Y = y_1, \ldots, y_m\). The goal is to Output: a position in \(X\) where \(Y\) occurs, or output “NONE”.

We denote \(X(i) = x_i, \ldots, x_{i+m-1}\), i.e., the substring of length \(m\) starting at position \(i\).

The idea is as follows:

- Compare the fingerprint of the substrings \(X(i)\) and \(Y\) at every \(i\), \(1 \leq i \leq m\).

- Fingerprint of \(X(i + 1)\) can be computed from fingerprint of \(X(i)\) in constant time as follows. The fingerprint of \(X(i)\) is:

\[
h_p(X(i)) = (a(x_i)r^{m-1} + a(x_{i+1})r^{m-2} + \cdots + a(x_{i+m-1})) \pmod{p}.
\]
The fingerprint go \( X(i + 1) \) can be computed from \( X(i) \) as follows:

\[
h_p(X(i + 1)) = (a(x_i + 1)r^{m - 1} + \cdots + a(x_{i+m-1})r + a(x_{i+m})) \mod p.
\]

Thus,

\[
h_p(X(i + 1)) = ((h_p(X(i)) - a(x_i)r^{m-1})r + a(x_{i+m})) \mod p.
\]

We can compute \( h_p(X(i + 1)) \) in an “incremental” fashion using the value of \( h_p(X(i)) \) without the need to compute from scratch.

The pseudocode of Algorithm Match is given in Figure 31. This algorithm is called Rabin-Karp’s algorithm. Algorithm Match can work even when the whole of \( X \) does not fit in memory or if \( X \) is given in an “online” fashion. This is because, we need to see only \( m \) consecutive positions of \( X \) at any given time. It is best to compute \( r^{m-1} \) using the exponentiation rule and repeated squaring to avoid large integers.

**Algorithm 31 Algorithm Match((X, Y)).** Outputs an occurrence of \( Y \) in \( X \).

1: Choose \( p \) to be a prime number > \( mn^2 \)
2: \( i = 1 \)
3: while \( i \leq n - m \) do
4: if \( h_p(Y) \neq h_p(X(i)) \) then
5: \( h_p(X(i + 1)) = \)
6: \( ((h_p(X(i)) - a(x_{i+1})r^{m-1})r + a(x_{i+m})) \mod p \)
7: else if \( Y = X(i) \) then
8: Return \( i \)
9: \( i := i + 1 \)
10: Return NONE

**Analysis of Algorithm Match**

**Theorem 9.5.** Algorithm Match runs in \( O(m + n) \) time and outputs the correct answer with high probability.

**Proof.** We first note that assuming no false positives, the algorithm terminates in \( O(m + n) \) time. We next upper bound the probability of getting a false positive.

For a given \( i \), assuming \( Y \neq X(i) \):

\[
\Pr\{h_p(Y) = h_p(X(i))\} \leq \frac{m}{mn^2} = \frac{1}{n^2}
\]
by our choice of \( p \) and the fingerprinting theorem. By the union bound (Appendix A), for every \( 1 \leq i \leq n - m + 1 \), assuming that \( Y \neq X(i) \),

\[
\Pr\{\cup_i (h_p(Y) = h_p(X(i)))\} \leq \Pr(h_p(Y) = h_p(X_1)) + \cdots + \Pr(h_p(Y) = h_p(X_{n-m+1}))
\]

\[
\leq \frac{1}{n^2} + \cdots + \frac{1}{n^2} = (n - m + 1) \frac{1}{n^2} \leq 1/n.
\]

Thus, if there is a false positive, then that happens with probability \( \leq 1/n \).

Thus the algorithm \textbf{Match} runs in time \( O(m + n) \) with probability at least \( 1 - 1/n \) (high probability).

\[
\square
\]

### 9.4 Worked Exercises

**Worked Exercise 9.1.** Assume that have a hash function that hashes keys to uniformly random locations. That is each key is equally like to hash to any particular slot in the hash table. Assume that the hash can be computed in \( O(1) \) time. In the hash table collisions are resolved by chaining. There are \( n \) keys and the size of the hash table is \( m \).

After hashing all the keys into the table, we search for the keys. Assuming that each of the \( n \) keys are equally likely to be searched, show that any search takes \( \Theta(1 + \alpha) \) expected time, where \( \alpha = n/m \) is the load factor.

**Solution:**

As mentioned, we assume that all keys are equally likely to be searched and compute the average time to search a key.

Assume that a key is inserted at the head of the link list.

Let \( k_i \) denote the \( i \)th key inserted into the table. For keys \( k_i \) and \( k_j \), we define the indicator r.v. \( X_{ij} \) to indicate whether \( h(k_i) = h(k_j) \).

\[
E[X_{ij}] = 1/m.
\]

The expected number of elements examined is:

\[
E\left[ \frac{1}{n} \sum_{i=1}^{n} \left( 1 + \sum_{j=i+1}^{n} X_{ij} \right) \right]
\]

The reason for the above is that the term \( 1 + \sum_{j=i+1}^{n} X_{ij} \) computes the number of comparisons needed to find \( k_i \), \( i \)th inserted element: it is equal to 1 plus the number of elements that are inserted subsequently in the same slot. We then take the sum of over all \( n \) keys and divide by \( 1/n \) since the probability of searching for any key is uniform.
9. HASHING AND FINGERPRINTING

\[
= \frac{1}{n} \sum_{i=1}^{n} \left( 1 + \sum_{j=i+1}^{n} E[X_{ij}] \right)
\]

\[
= \frac{1}{n} \sum_{i=1}^{n} \left( 1 + \sum_{j=i+1}^{n} \frac{1}{m} \right)
\]

\[
= 1 + \frac{1}{nm} \sum_{i=1}^{n} (n - i)
\]

\[
= 1 + \frac{1}{nm} (n^2 - n(n + 1)/2)
\]

\[
= 1 + \frac{n - 1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}
\]

Thus total time needed is \(\Theta(1 + \alpha)\).

9.5 Exercises

Exercise 9.3. Suppose we want to output all occurrences of pattern \(Y\) in text \(X\). Consider the Algorithm Match of Figure 31 and instead of stopping at the first occurrence, we continue and output all occurrences. Suppose the pattern \(Y\) occurs in the text \(X\) in many locations, say in \(\Theta(n)\) locations (e.g., the pattern can occur in every location). In this case, the Algorithm Match takes \(\Theta(mn)\) time. Can you modify Algorithm Match so that in all cases, it outputs correctly (with high probability) all occurrences of the pattern in \(O(m + n)\) time.

Exercise 9.4. We have a hash table of size \(m\) to store a set of \(n\) keys (from a universe \(U\)), where \(n \leq m/2\). Let \(h\) be a hash function from \(U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}\). We use the following technique called open addressing for inserting an element: given a key we successively examine a probe sequence \(< h(k, 0), h(k, 1), \ldots, h(k, m - 1) >\) and insert the key at the first empty slot.

For the purpose of analysis we will make the assumption of uniform hashing i.e., each key is equally likely to have any of the \(m!\) permutations of \(< 0, 1, \ldots, m - 1 >\) as its probe sequence.

1. Show that for \(i = 1, 2, \ldots, n\) the probability that the \(i\)th insertion requires strictly more than \(k\) probes is at most \(1/2^k\).
2. Let the random variable $X_i$ denote the number of probes required by the $i$th insertion. Show that $\Pr\{X_i > 2\lg n\} \leq 1/n^2$.

3. Let $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the $n$ insertions. Show that $E[X] = O(\lg n)$.

**Exercise 9.5.** A “good” hash function satisfies the property of uniform hashing i.e., if keys are drawn uniformly at random from the set $\{0, 1, \ldots, n-1\}$ the probability of hashing into any particular slot is $1/m$. (Assume that $n(>0)$ is a multiple of $m$, the number of slots.) Is the hash function $h(k) = k^2 \mod m$ good? Justify your answer.

**Exercise 9.6.** We will implement an approximate and efficient membership tester called Bloom filter as follows. We have a set $S$ of $n$ elements (numbers). We store it in the Bloom filter which is simply a bit array of size $m$ (each slot in the array can store one bit) as follows. (Initially all bits are set to 0.) Choose $k$ hash functions — $h_1, h_2, \ldots, h_k$ — each hash function is chosen randomly and independently from the universal hash family $\mathcal{H}_{p,m}$, where $m$ is the number of slots in the bit array and $p$ is a prime number larger than the largest element. To insert a number $x$ in the Bloom filter, simply compute $h_1(x), h_2(x), \ldots, h_k(x)$ and set the corresponding indices in the bit array to be 1 (if they are not already set to 1). To check for membership, given an element $x$, we similarly, compute the $k$ hashes and check and see whether all of them are 1; if so we say that $x$ is in the filter, otherwise not.

- Argue that the membership query using a Bloom filter can only give a false positive, but never a false negative.

- Given $n, m = 2n$, and $k$, what should be $k$ (in terms of $n$ and $m$), so that the probability of getting a false positive is at most $1/n$.

**Exercise 9.7.** We are given two unsorted sets of numbers $S_1$ and $S_2$, each of size $n$ (the numbers in each set are distinct). The two sets are equal, i.e., $S_1 = S_2$; however, the numbers are ordered differently in the two sets. The goal is to group the equivalent numbers in $S_1$ and $S_2$ in pairs. Two numbers belonging to the same set cannot be compared. Using universal hashing, give a randomized algorithm for the problem that takes $O(n)$ time in expectation and $O(n)$ space. (Note that this problem, is similar to the problem of 8.16 but there we required $O(n \log n)$ time.)
10

Algorithms for Numbers

In this chapter, we will study two important algorithms of tremendous real-world importance that involve numbers and extensively make use of number-theoretic concepts. The two applications are: primality testing and public key cryptography. Both are crucial ingredients in today’s digital economy.

This chapter needs basic number-theoretic concepts for which we refer to Appendix D.

10.1 Problem: Primality Testing

We will study a randomized algorithm for testing whether an integer is prime or not. As mentioned earlier, primality testing is one of the most fundamental problems in mathematics and an efficient algorithm for the same has been sought for centuries. Till about 2002 or so, the most efficient algorithm known, i.e., that works in polynomial time, was a randomized algorithm. Indeed, it was unclear for a long time whether there was a deterministic polynomial time algorithm for primality testing. Today, thanks to the result of Agrawal, Kayal, and Saxena, we know that primality testing can be accomplished in deterministic polynomial time. However, the randomized algorithm that we are going to see today is simpler and faster than the deterministic algorithm and is typically used in practice. The downside of the randomized algorithm is that it can err with a small probability, but this probability can be made as small as desired.

The randomization idea used in the primality testing is quite simple and can be explained as follows. Using number theory, it is first established that if a number, say $n$, is not prime (i.e., composite) then there are a lot of “witnesses” to this fact. Witnesses are numbers that are less than
A witness satisfies a simple test (will be discussed below) that shows conclusively that $n$ is not prime. The crucial point established is that the number of such witnesses are abundant: at least a constant fraction of $n$. This suggests a simple randomization idea that is very similar to the one that we saw in Section 8.2: Choose a random number between 1 and $n$ and check whether it is a witness or not. If it is a witness, then we immediately know that $n$ is not prime. If $n$ is indeed not prime, then the probability of choosing a witness is at least a constant (because of the abundance of witnesses). Repeating the process a few times (say $O(\log n)$ times) — each time, independently choosing a random number between 1 and $n$ — the probability that we fail to find a witness is very small.

10.1.1 An Almost Correct Algorithm: Pseudoprimality Testing

Our randomized algorithm (and many other primality testing algorithms) is based on a simple theorem due to Fermat, called as Fermat’s Little Theorem which is proved in Appendix D.

**Theorem 10.1** (Fermat’s Little Theorem (FLT)). If $n$ is a prime, then for every $a$ ($1 \leq a < n$),

$$a^{n-1} \equiv 1 \pmod{n}.$$  

Using Fermat’s Little Theorem, we can design a simple primality testing algorithm called Pseudo-prime test (Figure 32) which is “almost” correct, in a sense which will be clear below. FLT tells us that if a number is prime it will pass this test for all values of $a$ belonging to the set $\{1, \ldots, n-1\}$. On the other hand, FLT is not an “if and only if” theorem. That is, it does not say the converse of the theorem: if $n$ is composite it will fail the test for at least one value of $a$. However, it turns out that the converse is true for essentially all composite numbers, except for a very tiny fraction of composite numbers called as Carmichael numbers. Carmichael numbers are those numbers that “pass” the Fermat test (i.e., satisfies the FLT) for all values of $a$ in the set $\{1, \ldots, n-1\}$. Thus one can consider Carmichael numbers as “Pseudo-primes”, i.e., like prime numbers they satisfy the FLT. Carmichael numbers are very rare, e.g., there are only about 646 numbers in the first billion integers, i.e., less that 0.0001%. We now show that for non-Carmichael composite numbers the test fails with probability at least $1/2$. Hence by repeating the Pseudo-prime test (each random choice is independent of the rest) for a few times (say $O(\log n)$ times), with high probability, i.e., with probability at least $1 - 1/n$ we can detect whether a number is pseudo-prime or not.
We will see that this already gives a good way to choose a large *random* prime number.

**Algorithm 32** Algorithm Pseudo-prime Test. **Input:** A positive integer \(n\). **Output:** yes (Pseudoprime) or no (not Pseudoprime)

1. Choose an integer randomly from the set \(\{2, \ldots, n-1\}\).
2. If \(a^{n-1} \equiv 1 \pmod{n}\) then
   3. Return yes
   4. Else
   5. Return no

The following is the crucial lemma.

**Lemma 10.1.** Fix a number \(n\). If \(n\) fails to satisfy the FLT for some positive integer \(a\) in \(\{1, \ldots, n-1\}\), then it fails to satisfy the FLT for at least half of the values in the set \(\{1, \ldots, n-1\}\).

**Proof.** Consider the set \(A\) of values in \(Z_n^* = \{1, \ldots, n-1\}\) that satisfy the FLT, i.e., \(A = \{a \in Z_n^* | a^{n-1} \equiv 1 \pmod{n}\}\). Clearly, \(A\) is nonempty, since 1 belongs to \(A\). Also \(A\) is closed under multiplication modulo \(n\): let \(a_1, a_2 \in A\), then \(a_1^{n-1}a_2^{n-1} \equiv 1 \pmod{n}\); hence \((a_1a_2)^{n-1} \equiv 1 \pmod{n}\) as well. Hence \(A\) is a subgroup of \(Z_n^*\) by the closure property of subgroup (Lemma D.2). By the lemma’s premise, there is an \(a \in Z_n^*\), that fails to satisfy FLT. Hence \(a \notin A\). Hence \(A\) is a proper subgroup and hence by Lagrange’s theorem, \(|A| \leq |Z_n^*|/2 = (n-1)/2\). Hence at most half of the values in the set \(\{1, \ldots, n-1\}\) satisfy the FLT, implying the lemma. \(\square\)

The above lemma, implies the following theorem.

**Theorem 10.2.** Algorithm Pseudoprime always returns “yes” if the input number is prime or a Carmichael number. Otherwise, with probability at least \(1/2\) it returns “no”.

It is important to note that in the FLT test, we need to compute \(a^{n-1}(\mod{n})\). If done naively, this will take \(O(n)\) multiplications which is not polynomial in the input size. The following exercise (Exercise 10.1) asks you to show that \(a^{n-1}(\mod{n})\) can be computed in \(O(\log{n})\) multiplications (See also Chapter 9 Section 9.3).

**Exercise 10.1.** Show how to compute \(a^{n-1}(\mod{n})\) by using only \(O(\log{n})\) multiplications (each multiplication involves only numbers of size \(O(\log{n})\) bits). (Hint: You can use divide and conquer and the exponentiation rule for mod: \((a^y \mod{p}) = ((a \mod{p})^y) \mod{p}\). See Chapter 9 Section 9.3.)
Run time analysis  We analyze the running time of Algorithm Pseudo-prime Test. Choosing a random integer from the set \{2, \ldots, n - 1\} can be accomplished in \(O(\log n)\) time (Exercise 8.1). Computing \(a^{n-1} \equiv 1 \pmod{n}\) can be done in \(O(\log n)\) multiplications (Exercise 10.1). If we use the standard way to multiply two \(b\)-bit numbers in time \(\Theta(b^2)\), then the modular exponentiation can be done in \(O(\log^3 n)\) time. Hence the overall time is \(O(\log^3 n)\) (i.e., cubic in the input size).

In practical applications, e.g., in RSA cryptography (Section 10.2), one needs to choose a large prime number. A fast way of accomplishing this is as follows. Choose a large random number and apply the Pseudo-prime test a few times (e.g., \(2 \log n\) times, where \(n\) is the number chosen). If the test succeeds in all times, then we output the number as prime, otherwise not. One can argue that the number output is prime with high probability. The reason is that there are far more primes than Carmichael numbers. There is an important theorem in number theory called the Prime number theorem that says that the number of primes less than \(n\) is approximately \(n/\ln n\). Hence, finding a prime near \(n\) (say within a factor of 2) by randomly choosing a number near \(n\) will succeed with probability about \(\Theta(1/\ln n)\). On the other hand, the probability of choosing a Carmichael number near \(n\) is asymptotically much smaller.

### 10.1.2 A Correct Primality Testing Algorithm

Somewhat surprisingly, it turns out that we can modify the Algorithm Pseudoprime only a bit to get a primality testing algorithm that with high probability returns “no” when the input number is composite, even if it is a Carmichael number. (See Algorithm Prime Test — Figure 33) If it is a prime, then it always will return “yes”. The main modification is that given that \(a^{n-1} \equiv 1 \pmod{n} \) (where \(a\) is randomly chosen as before), we compute the square root of \(a^{n-1}\) modulo \(n\), i.e., the number \(\pm a^{(n-1)/2}\) (modulo \(n\)). A result in number theory is the following (which we state without proof):

**Theorem 10.3.** If \(n\) is a prime number then the square root of 1 (modulo \(n\)) is always trivial, i.e., it is either \(1 \pmod{n}\) = 1 or \(-1 \pmod{n}\) = \(n - 1\).

The above theorem says that if there exists a non-trivial square root of 1 modulo \(n\), i.e., some number other than \(\pm 1 \pmod{n}\), then \(n\) is composite. That is, if the algorithm finds a non-trivial square root of 1 modulo \(n\) (i.e., \(x_i = 1\) and \(x_{i-1} \neq \pm 1 \pmod{n}\)) or if \(a^{n-1} \not\equiv 1 \pmod{n}\), then \(n\) is composite — due to Theorem 10.3 or FLT respectively. On the other hand,
if the algorithm does not find a non-trivial square root of 1 in the for loop and if \( n \) passes the FLT (i.e., \( n \) is a Carmichael number), then we show the following theorem (which we state without proof).

**Theorem 10.4.** If \( n \) is a Carmichael number, then for at least half the values of \( a \), i.e., at least for half the values in the set \( \{1, \ldots, n - 1\} \), the algorithm finds a non-trivial square root of 1 modulo \( n \). Hence if \( a \) is chosen randomly, then the probability that the algorithm returns "no" (i.e., composite) is at least \( 1/2 \).

From the above theorem, it follows that if we repeat the Algorithm Primetest a few times (say \( O(\log n) \) times) with high probability, we will be able to detect a Carmichael number. Hence, this test “takes care” of Carmichael numbers as well.

**Algorithm 33** Algorithm Prime Test. **Input:** A positive integer \( n \). **Output:** yes (prime) or no (not prime)

1. Let \( n - 1 = 2^k b \) where \( b \) is an odd number.
2. Choose an integer \( a \) randomly from the set \( \{2, \ldots, n - 1\} \).
3. \( x_0 = a^b \mod n \).
4. for \( i = 1 \) to \( k \) do
   5. \( x_i = (x_{i-1})^2 \mod n \)
6. if \( x_i = 1 \) and \( x_{i-1} \neq 1 \) and \( x_{i-1} \neq n - 1 \) then
   7. Return “no”.
8. if \( x_k \neq 1 \) then
   9. Return “no”.
10. Return “yes”.

10.2 Public Key Cryptography

In cryptography two parties want to send a message securely, i.e., Alice wants to send a message to Bob and wants to be sure that no eavesdropper can learn anything about the message being sent. One way to do this is for Alice to encode the message using a key and then send the encoded message. Bob can decode the message using the same key. This is pretty secure. However, the catch is both Alice and Bob need to know the key a priori. Thus the key itself has to be shared in a secure way.

Public-key cryptography is an ingenious method around the above problem. In this system, each party has a public and a private key. The
private key is known only to the party, but the public key is advertised to everybody. Suppose Alice wants to send a message to Bob. She will first encode the message using Bob’s public key and send it. Bob can then decode the message using his private key. Can we construct such a cryptosystem?

### 10.2.1 RSA Cryptosystem

RSA is exactly such a system and is widely used today for secure transactions. The RSA system is as follows. Suppose Bob wants to create a RSA-based public-key crypto system. He will first choose two (large) random prime numbers — $p$ and $q$ — and compute their product $n = pq$. (In today’s applications, $p$ and $q$ will be (about) 1000-bit numbers.) Let $d$ be an integer that is relatively prime to $(p−1)(q−1)$, i.e., $gcd(d, (p−1)(q−1)) = 1$. Let $e$ be the inverse of $d$ modulo $(p−1)(q−1)$ (since $d$ is relatively prime to $(p−1)(q−1)$, the inverse exists via Corollary [D.1]). Bob’s private key is $d$ and his public key is $e$. He will keep $d$ private and announce the numbers $e$ and $n$ to the world.

Suppose Alice wants to send a message $x \in \{1, \ldots, n − 1\}$ to Bob. She will then use Bob’s public key to encrypt message $x$ using the following encryption function:

$$E(x, n, e) = x^e \pmod{n} = y.$$ 

She then sends $y$ to Bob.

Bob uses his private key to decrypt the message using the decryption function:

$$D(y, n, d) = y^d \pmod{n}.$$ 

The following theorem shows that the decryption function used by Bob correctly decrypts the message, yielding the original message sent by Alice.

**Theorem 10.5.** If $y = E(x, n, e) = x^e \pmod{n}$, then $D(y, n, d) = y^d \pmod{n} = x$.

**Proof.** We have:

$$y^d \pmod{n} = x^{ed} \pmod{n}.$$ 

Since $ed \equiv 1 \pmod{(p−1)(q−1)}$, we can write $ed = 1 + c(p−1)(q−1)$, where $c$ is some integer. Hence, we have

$$x^{ed} \pmod{n} = x^{1+c(p−1)(q−1)} \pmod{n} = x x^{(p−1)(q−1)c} \pmod{n}.$$
We want to show that \( x^{(p-1)(q-1)c} = x \pmod{n} \), i.e., \( n \) divides \( x^{(p-1)(q-1)c} - 1 \). By applying FLT for the primes \( p \) and \( q \), and using the modular exponentiation rule (Section 9.3), we have: 

\[
x^{(p-1)(q-1)c} - 1 \equiv 0 \pmod{p}
\]

and

\[
x^{(p-1)(q-1)c} - 1 \equiv 0 \pmod{q}
\]

Hence \( p \) divides \( x^{(p-1)(q-1)c} - 1 \) and \( q \) divides \( x^{(p-1)(q-1)c} - 1 \) and since \( n = pq \) (product of two primes) divides \( x^{(p-1)(q-1)c} - 1 \).

### 10.2.2 Why is RSA hard to break?

The reason RSA is hard to break is because of two hard computational problems that RSA uses. The first is factoring large prime numbers. If the eavesdropper can factorize \( pq \) into primes \( p \) and \( q \), then it is easy for him to decode the message, since if he knows \( p \) and \( q \) he can compute \( (p-1)(q-1) \mod p \) and similarly, \( x^{(p-1)(q-1)c} - 1 \equiv 0 \pmod{p} \) and similarly, \( x^{(p-1)(q-1)c} - 1 \equiv 0 \pmod{q} \). Hence \( p \) divides \( x^{(p-1)(q-1)c} - 1 \) and \( q \) divides \( x^{(p-1)(q-1)c} - 1 \) and since \( n = pq \) (product of two primes) divides \( x^{(p-1)(q-1)c} - 1 \).

The second is inverting an exponentiation modulo a prime, i.e., given \( r^x \mod p \), where \( p \) is a prime and \( r \) is a primitive root (i.e., \( r \) is a generator of the group \( \mathbb{Z}_p^* \) — see D.2.3) and \( x < p \) is some integer, find \( x \) (you know \( p \) and \( r \)). This is like taking logarithm to the base \( r \). This is considered difficult to do in polynomial time. Currently no polynomial time algorithms are known for both problems. Hence the security of RSA depends on the presumed hardness of the above two number-theoretic problems. It is now known that the above problems can be solved efficiently (in polynomial time) using a quantum computer.

### 10.2.3 Randomized RSA

Note that RSA is a purely deterministic protocol (no randomness involved). This can be a weakness, as it leads to attacks by seeing repetitions of messages, or by using known messages and encoding them.

Another glaring problem is encoding small messages, e.g., a message of just 1 bit. Since \( x^e \mod n = x \) if \( x \) is a single bit message, there is no encoding at all.

The way to avoid these problems is by using randomization. Suppose you want to encode a single bit. Choose a random number \( x < n/2 \) and then transmit to Bob \( y = (2x + b)^e \mod n \). Bob receives \( y \) and uses his private key to recover the bit \( b \) from the message \( 2x + b \) (i.e., \( b \) will be the last bit in the message). This method can be shown to be as secure as the original RSA, but in addition due to randomization it is immune to attacks such as detecting repetitions etc. In fact, note that every message
can be sent like this if we can encode each bit individually; however, this will blow up the size of the messages.

10.3 Exercises

Exercise 10.2. The lcm (least common multiple) of two integers $a$ and $b$ is the smallest number divisible by both $a$ and $b$. Given two $n$-bit numbers $a$ and $b$, give an efficient algorithm to compute the lcm of $a$ and $b$. Analyze the running time of your algorithm.

Exercise 10.3. Show that if $a$ is a non-trivial square root of 1 (modulo $n$), i.e., $x^2 \equiv 1 \pmod{n}$, but $x \not\equiv \pm 1 \pmod{n}$, then $n$ is composite.

Exercise 10.4. Instead of using $N = pq$ (where $p$ and $q$ are primes) in the RSA, suppose we use just $p$ as the modulus. Similar to RSA, we have public and private keys $e$ and $d$ respectively (chosen as before), and a message $m \in \{1, \ldots, p-1\}$ is encrypted as $y = m^e \pmod{p}$. Prove that this cryptosystem is not secure, by showing that an adversary can efficiently recover $m$ given $p, e, \text{ and } y$. Analyze the running time of the recovery algorithm.

Exercise 10.5. Show that $(n-1)! \equiv -1 \pmod{n}$ if and only if $n$ is a prime. Why can/can’t we use this as a test for checking primality?
Graphs or networks are fundamental objects that can be used to represent various kinds of relationships in the real world. A graph (also called as a network) consists of nodes/vertices and edges/links where nodes are elements from some set and edges represent connections (relationships) between pairs of nodes. Edges can be directed or undirected. There can be weights on the edges which can denote the “strength” of the connections. A graph is usually denoted by the notation $G = (V, E)$, where $V$ is the set of nodes and $E$ is the set of edges. $V$ is typically a set such as $\{1, 2, \ldots, n\}$ denoting $n$ vertices labelled $1, 2, \ldots, n$ and $E$ consists of elements (2-tuples) of the form $(u, v)$ where $(u, v)$ represents an edge connecting node $u$ with node $v$. We usually denote the number of nodes by $n$ and the number of edges by $m$.

We refer to Appendix E for basic graph concepts and examples.

Graphs naturally model many real-world data, especially data with relationships. Some prominent examples of real-world graphs are:

- A communication network (e.g., the Internet) can be represented as a graph where nodes are end-hosts (computers, routers etc.) and edges are communication links between end-hosts.

- Nodes can represent persons and two persons are connected if they are mutually friends (Social network).

- Nodes can represent webpages and there is a (directed) link between a webpage $A$ and a webpage $B$ if there is a hyperlink from page $A$ pointing to page $B$ (Web graph).
Nodes can represent points (places) in a map and edges can represent roads between the points. Edges can have weights on them, which can represent the road distance. (Road network).

In this chapter, we will focus on two fundamental graph search algorithms, namely depth-first search and breadth-first search and their applications. Both search algorithms are useful in numerous other graph algorithms and applications. Both algorithms can be implemented in linear time, i.e., \( O(m + n) \) time, where, \( m \) denotes the number of edges and \( n \) the number of vertices. Thus linear means “linear in the size of the graph”, which is \( m + n \). These linear time algorithms are the basis of fast algorithms for several other fundamental graph problems.

11.1 Representing a Graph

Before we turn to graph algorithms, we first discuss how to represent a graph in a computer, namely the data structures needed to store a graph. There are typically two ways to represent a graph \( G = (V, E) \), where \( V \) is the set of vertices (nodes) and \( E \) is the set of edges (links). Throughout, we will use \(|V| = n\) (denoting the number of vertices) and \(|E| = m\) (denoting the number of edges). We will also assume that the vertices are numbered from 1 to \( n \), unless otherwise stated.

11.1.1 Adjacency List (Adj) Representation

The adjacency list representation consists of an array of vectors (or linked lists), i.e., \( A[i] \) is a vector (linked-list) corresponding to vertex \( i \), \( 1 \leq i \leq n \). There is a vector (or a linked list) for each vertex. The vector contains a list of all neighbors of the vertex. If it is a weighted graph, the weight is also stored along with the neighbour. If \((i, j)\) is an (undirected) edge in \( G \), then \( i \) appears in \( j \)'s list and vice-versa. If the edge \((i, j)\) has a weight \( w(i, j) \) then it is stored alongside \( i \) in \( j \)'s list and vice-versa. In an undirected graph, the size of the adjacency list of \( v \) is \( deg(v) \).

Figure [11.1] shows a graph and its representation.

**Space and Time for Adjacency List** The total space needed for an adjacency list is \( O(m + n) \). It gives a sequential way to access (or check) a particular neighbor of a vertex, as one has to go through the adjacency list of the vertex. This takes time that is linear in the size of the adjacency list.
Figure 11.1: A graph and its representation. The left shows the adjacency list representation and the right the adjacency matrix representation.
11.1.2 Adjacency Matrix Representation

Alternatively, one can use an adjacency matrix representation, where the graph is stored as a 2-dimensional $|V| \times |V|$ array, where $A[i,j] = 1$ iff $(i,j)$ is an edge in the graph, otherwise $A[i,j] = 0$. For a weighted graph, $A[i,j] = w(i,j)$.

**Space and Time for Adjacency Matrix**  
Total space needed for an adjacency matrix is $\Theta(n^2)$. The advantage of matrix representation is that it provides random access to any edge — takes $O(1)$ time.

Note that the two representations have different space and access time bounds. For sparse graphs, i.e., graphs with few edges (say $O(n)$ edges), it might be a good idea to store them as an adjacency list rather than as an adjacency matrix, since it saves space. On the other hand, if the graph is dense, say, has $\Theta(n^2)$ edges, then adjacency matrix representation is more suitable, since it gives random access to an edge with essentially optimal space.

We will next study two fundamental algorithms to search (explore) a graph: **Depth-first search (DFS)** and **Breadth-first search (BFS)**. While DFS explores graph by always exploring the most recently visited vertex, BFS explores nodes in increasing distances from a given source node. We first focus on DFS which can be naturally implemented as a recursive algorithm.

11.2 Depth-first Search (DFS)

In DFS, the search starts from a vertex $s$, and edges are explored out of the most recently visited vertex. DFS yields a spanning tree called depth-first spanning (DFS) tree (see Appendix E for definition of spanning tree and its properties). DFS can be implemented in $O(|V| + |E|)$ (i.e., linear) time.

Figure 11.2 shows a graph and its DFS tree.

We will next explore in detail DFS and its algorithmic applications.

11.2.1 Depth-first Search (DFS) Algorithm in an Undirected graph

We will first explore DFS in an undirected graph. We are given an undirected and connected graph $G = (V,E)$ represented by its adjacency list; and a vertex $s \in V$. The goal is to output a DFS tree $T$. As a byproduct, it will also be useful to output the DFS numbering of vertices, i.e., the order
Figure 11.2: An undirected graph and its DFS tree rooted at vertex \( a \) where the DFS is started. Thick lines show the tree edges and the dotted lines show the back edges. The \textit{num} values of the vertices — which shows the order in which the vertices are visited — are also shown.
in which the vertices are visited in the graph during the DFS search. As we will see, this numbering is very useful for various applications.

DFS starts at a given source vertex \( s \) and it explores edges in a depth-first manner, i.e., an edge is explored from the currently (most recently) visited vertex — call it \( v \) (initially \( v \) is \( s \)). If the explored edge leads to an already visited vertex, then another edge out of \( v \) is explored. If all edges out of \( v \) leads to vertices that were already visited (i.e., all neighbors of \( v \) have already been visited), then the search backtracks to the vertex that was visited prior to \( v \) (i.e., the vertex from which \( v \) was visited). The algorithm that we have described can be written in an iterative (non-recursive) way; to implement backtracking the most natural data structure is a stack. This implementation is a bit cumbersome, as we have to explicitly maintain the stack.

Alternatively, it is easiest to write DFS as a recursive algorithm. The recursive algorithm does not need an explicit data structure and is compact and elegant. The pseudocode is given in Algorithm 34. The pseudocode has two auxiliary data structures:

1. An array \( visited[] \) that is initialized to false for every vertex \( v \in V \), i.e., \( visited[v] = false \). This array maintains whether the vertex has already been visited or not.

2. An array \( num \) — \( num[v] \) gives the DFS number of \( v \), i.e., the order in which vertex \( v \) was visited in the graph. This DFS numbering is very useful in understanding the structure of the graph.

3. A tree \( T \) (which is maintained as a list of edges and is initially empty) which stores the DFS tree edges. Note that \( T \) will have exactly \( n - 1 \) edges at the end of the algorithm since it is a spanning tree of \( G \) and \( G \) is assumed to be connected.

Let’s describe the recursive algorithm which is quite simple and short: when a vertex \( v \) is visited, the DFS is called recursively on all unvisited neighbors of \( v \) one by one. \( visited[v] \) is updated to “true” when DFS was invoked on \( v \). Note that it is invoked only if \( visited[v] = false \) prior to that — this is checked in the for loop before DFS is called. When a vertex is visited for the first time, that means the edge that is used to visit it belongs to the DFS tree; this is added to \( T \) (Line 6). Also, when a vertex is visited for the first time, its DFS numbering is set (Line 5).

In this algorithm, we maintain the DFS tree edges as a list (or an array). However, the algorithm can be easily modified to maintain the tree structure, by maintaining parent and child relationships for every vertex.
11.2. DEPTH-FIRST SEARCH (DFS)

It is clear that if an edge \((a, b)\) is added to the tree \([6]\), then \(a\) is the parent of \(b\) and \(b\) is the child of \(a\) in the DFS tree.

**Algorithm 34** Algorithm DFS\((G, s)\). Input: An undirected graph \(G\) and a source vertex \(s\). Output: DFS tree edges \(T\) rooted at \(s\) and \(num\) array which gives the order of visiting each vertex. \(T\) is initially empty. \(count\) is initialized to 1.

1: \(visited[s] = true\)
2: \(num[s] = count\)
3: \(count = count + 1\)
4: for vertex \(v \in Adj(s)\) do
5: \(if\ \ visited[v] = false\ then\)
6: Add edge \((s, v)\) to \(T\)
7: \(DFS(G, v)\)

**Running time of DFS** The run time of DFS can be upper bounded by the number of recursive calls made plus the number of times the if statement is executed. First, we bound the number of recursive calls. A recursive call is made each time a vertex is visited for the first time, i.e., visited value for that vertex is false. Note that once a vertex is visited its visited value will be set to true and hence won't be visited again. Thus each vertex is visited exactly once. Thus the number of recursive calls made is \(O(n)\). Next, we bound the number of times the if statement is executed. Fix a vertex \(v\). The if statement is executed as many times the for loop is run. The for loop runs \(deg(v)\) times, since that is the size of the adjacency list. Thus the total number of iterations overall is: \(\sum_{v \in V} deg(v) = 2m\). Hence overall run time of DFS is \(O(n + m)\), i.e., linear in the size of the graph.

11.2.2 A key property of DFS

DFS is very useful in understanding the structure of the graph. An important byproduct of DFS is that it allows the edges of graph \(G\) to be classified into two types based on the output DFS tree \(T\).

1. **tree edges**: edges that belong to \(T\).
2. **back (non-tree) edges**: edges that don't belong to \(T\).

The next lemma shows an important property of back edges.

**Lemma 11.1** (Property of back edges). If \((v, w)\) is a back edge, then in the DFS spanning tree \(T\), \(v\) is an ancestor of \(w\) or vice versa.
11. FUNDAMENTAL GRAPH ALGORITHMS

Proof. Assume that \( v \) is visited before \( w \). Then we will show that \( v \) will be an ancestor of \( w \). \( DFS(v) \) is called before \( DFS(w) \). Thus, when \( v \) is visited \( w \) is still not visited. On the other hand, when \( DFS(v) \) has finished, \( DFS(w) \) would have been finished, since all recursive calls made by neighbors of \( v \) will have to finish before the call \( DFS(v) \) finishes. Thus all previously unvisited vertices visited by \( DFS(v) \) will become descendants of \( v \) in \( T \). This includes \( w \) as well, since \( w \) is on the Adjacency list of \( v \). Hence \((v, w)\) is a back edge.

Back edges are useful in many ways. For example, a back edge indicates the presence of a cycle. It is also useful in identifying cut vertices and edges as described in Section \( \text{[11.2.4]} \). Before that we first describe a fundamental application of DFS, namely, checking connectivity of a graph.

11.2.3 DFS Application: Finding connected components of a graph

We are given a graph \( G = (V, E) \) (possibly disconnected) and we want the check if the graph is connected or not. More generally, in many applications, we want to output the connected components of the graph, if the graph is disconnected. DFS can be used to find all the connected components of the graph in linear time. The pseudocode is given in Algorithm 35. This pseudocode is just simply running DFS starting from every node in the graph, thus it calls \( DFS(G, v) \) for every node \( v \). Each call will return a connected component of \( G \). If the graph is connected, then calling DFS from any node will explore all nodes in the graph and thus all nodes will be visited in the first DFS call itself. Hence only one component (the whole graph) is returned. If the graph is disconnected, each call will return the component containing the vertex where the DFS is called.

Algorithm 35 Algorithm DFS-graph(\( G \)). Input: An undirected graph \( G = (V, E) \) represented by its adjacency list. Output: The induced spanning tree of the connected components of \( G \).

1: for all \( v \in V \) do
2: \( visited[v] = false \)
3: for each \( v \in V \) do
4: \( if visited[v] = false \) then
5: \( DFS(G, v) \) //outputs spanning tree of \( v \)'s connected component
 DFS Application: Finding cut vertices and cut edges of a graph

Let $G = (V, E)$ be a connected undirected graph. A vertex $x$ is said to be the cut-vertex of $G$ if removing $x$ from $G$ disconnects the graph. Equivalently, $x$ is a cut vertex if there exists two other vertices $u$ and $v$ such that every path between $u$ and $v$ contains the vertex $x$. Note that that above property means that $x$, $u$, and $v$ cannot lie in a common cycle. (Otherwise, removing $x$ will not disconnect $u$ and $v$, since there is an alternate path via the cycle.) A graph is said to be biconnected if it has no cut vertices (See Worked Exercise [11.1]. See Figure [11.3].

Similarly, an edge $e \in E$ is a cut-edge if removing $e$ disconnects the graph. It is easy to see that a cut-edge cannot lie in any cycle of the graph.

Using DFS to find all cut-vertices A straightforward algorithm for finding a cut-vertex based on its definition can take $O(mn)$ time (see Exercise [11.3]). Using DFS we can give an elegant algorithm that runs in $O(n + m)$ time. We use the back edge property of the DFS spanning tree to help us characterize whether a vertex is cut vertex or not. The following lemma shows exactly that.

Lemma 11.2. Let $T$ be a DFS spanning tree of $G$. A vertex $x$ is a cut vertex of $G$ if and only if:

1. $x$ is the root of $T$ and $x$ has more than one child.
2. $x$ is not the root, and for some child of $y$ of $x$, there is no back edge between any descendant of $y$ (including $y$ itself) and a (proper) ancestor of $x$.

Proof. We first show the “if” part. If condition (1) is satisfied then clearly removing the root disconnects the graph. If condition (2) is satisfied, then removing $x$ will disconnect $y$ from the root. This is because, since there is no back edge from $y$ to any proper ancestor of $x$ (i.e., any ancestor of $x$ other than $x$ itself), removing $x$ disconnects $y$ and any proper ancestor of $x$ (say the root). Thus, we have shown that $x$ is a cut vertex under both conditions.

Next we show the “only if” part. That is we have to show that if the two conditions are false, then $x$ is not a cut-vertex (this is contrapositive to showing that $x$ is a cut vertex only if one of the two conditions hold). If $x$ is a root and condition (1) does not hold, then $x$ has only one child, and, clearly, removing $x$ does not disconnect the graph (it does not disconnect
Figure 11.3: A graph $G$ (top) and its DFS tree (bottom). The $\text{num}$ values of the vertices are shown in black and the $\text{low}$ values are shown in red. The cut vertices of $G$ are $b$, $d$ and $g$. Note that each of their $\text{num}$ values is less than or equal to the $\text{low}$ values of at least one their respective children. For example, $d$ is a cut vertex because one of its children, $e$, has its $\text{low}$ value greater than or equal to $d$'s $\text{num}$ value. The DFS tree of $G$ can be used to find all its cut vertices. Note that a cut vertex will not have a back edge that goes from its descendent to its ancestor. For example, there is a back edge from $d$'s descendent $f$, but it does not go to $d$'s ancestor, just $d$. Hence $d$ is a cut-vertex, since it is the only vertex that connects its descendant $f$ to its ancestors.
any two nodes \( u \) and \( v \) since they are all descendants of a single child). If \( x \) is not a root, then there are two cases. If \( x \) is a leaf (i.e., has no children), then clearly it cannot be a cut vertex. On the other hand, if \( x \) has at least one child, and every child of \( x \) has a back edge to a proper ancestor of \( x \), then removing \( x \) will not disconnect any child, since they are part of a cycle created by the back edge.

To design an efficient algorithm we have to check the two conditions of the above lemma efficiently. It is easy to check condition 1. To get an efficient algorithm we have to check condition 2 efficiently.

**Checking condition 2 in DFS** We can check for condition 2 while doing DFS itself by slightly modifying the DFS algorithm to compute an array called \( low[] \) defined as follows. Note that the DFS number of each vertex \( v \) is given by the \( num \) array. \( low[v] \) is the highest ancestor (i.e., the one with the lowest DFS number) \( v \) can reach (including \( v \)) either by a back edge from \( v \) or by a back edge from one of the descendants of \( v \).

**Defining \( low \) value** For a vertex \( v \), we first define a quantity called \( \text{back-edge-num}[v] \) as follows. In words, \( \text{back-edge-num}[v] \) is simply the highest ancestor (i.e., the one with the lowest DFS number) that any descendant of \( v \) (including \( v \)) can reach via a back edge.

\[
\text{back-edge-num}[v] = \min\{\text{num}[w] \mid \exists \text{ a back edge } (x, w) \text{ such that } x \text{ is a descendant of } v \text{ (including } v)\}
\]

Then we define \( low[v] \) as:

\[
low[v] = \min\{\text{num}[v], \text{back-edge-num}[v]\}.
\]

In other words, \( low[v] \) is the minimum of \( \text{num}[v] \) and \( \text{back-edge-num}[v] \). Note that \( low \) value of a node depends only on its \( \text{num} \) value and the \( low \) values of its descendants, in particular the \( low \) values of its children.

We note that Condition 2 is satisfied for vertex \( x \) if it has a child \( y \) such that \( low[y] \geq \text{num}[x] \). This is because \( low[y] \) is highest ancestor that \( y \) can reach via back edge and that is not a proper ancestor of \( x \), since \( low[y] \geq \text{num}[x] \). Hence, we can use \( low \) values to check for Condition 2 efficiently.

**Computing \( low \) values** We would like to compute \( low \) values during DFS. To do this we need to compute \( low \) values recursively. This can be done,
11. FUNDAMENTAL GRAPH ALGORITHMS

since the low value of a vertex \( x \) depends on its num value, the low value of its children and whether \( x \) has any back edge.

We can compute the low values recursively:

\[
\text{low}[x] = \min\{\text{num}[x], \{\text{low}[y] \mid y \text{ is a child of } x\}, \{\text{num}[w] \mid (x, w) \text{ is a back edge}\}\}.
\]

When the DFS call finishes at vertex \( x \) it has already finished exploring all the children of \( x \). Hence \( x \) has all the information needed to compute its low value.

**DFS-based algorithm for finding cut vertices**  The algorithm takes as an input an undirected and connected graph \( G = (V, E) \) represented by its adjacency list; and a vertex \( s \in V \) and it outputs the cut vertices of \( G \) (if any). The algorithm's pseudocode is presented in Algorithm 36. The algorithm invokes DFS on a source vertex \( s \) (any arbitrary vertex of the graph). It uses the following auxiliary data structures: num array, visited array, low array, and parent array. parent\([v]\) stores the parent of \( v \) in the tree \( T \), it is not defined for root of the tree. Initially, visited\([v]\) = false, and parent\([v]\) = null for all \( v \). Initially \( T = \emptyset \) and count = 1.

The algorithm implements the recursive way of computing low value as defined earlier. When DFS\((v)\) finishes and returns, its low value has already been computed, since they depend on itself and its descendants (which are already finished). Hence if low\([v]\) \( \geq \) num\([s]\) (Line 10), Condition 2 is satisfied and we can output \( s \) as a cut vertex. Note that the algorithm is invoked at vertex \( s \) and hence \( s \) is the root. It will be a cut vertex if it has more than one child — this is checked at the end, when the call at \( s \) finishes. We will leave the formal proof of correctness (using induction) as an exercise.

### 11.2.5 Depth-first search in a directed graph

DFS in a directed graph is similar to that of an undirected graph. The pseudocode is the same as that of DFS in an undirected graph; however, note that a vertex \( v \) is adjacent to \( u \) only if there is a directed edge from \( u \) to \( v \). Figure 11.4 shows a directed graph and a DFS tree of the graph. In an undirected graph, DFS classifies the edges into two types: tree and back edges. In the directed case, it classifies into four types described below.

**Classification of edges in a directed graph**  DFS in a directed graph \( G \) classifies the edges of \( G \) into 4 types:
Algorithm 36 Algorithm DFS-CV(s). Input: An undirected and connected graph \( G = (V, E) \) represented by its adjacency list; and a vertex \( s \in V \). Output: the cut vertices of \( G \) (if any). Note that the status of \( s \), which is the root is output separately, by checking whether it has more than one child.

1: \( visited[s] = true \)
2: \( num[s] = count \)
3: \( count = count + 1 \)
4: \( low[s] = num[s] \)
5: for Each vertex \( v \in Adj(s) \) do
6: \( \text{if } visited[v] = false \text{ then} \)
7: Add edge \((s, v)\) to \( T \)
8: \( parent[v] = s \)
9: \( DFS-CV(v) \)
10: \( \text{if } low[v] \geq num[s] \text{ then} \)
11: Output \( s \) is a cut-vertex
12: \( low[s] = \min\{low[s], low[v]\} \)
13: else if \( v \neq parent[s] \text{ then} \)
14: //checks whether \((s, v)\) is a back edge
15: \( low[s] = \min\{low[s], num[v]\} \)

1. Tree edges: Edges that belong to the DFS forest/tree. An edge \((u, v)\) is a tree edge if \( v \) is visited for the first time when exploring edge \((u, v)\).

2. Back edges: Edges that connecting a vertex to its ancestor in the DFS tree.

3. Forward edges: Edges that connect a vertex to its descendent in the DFS tree.

4. Cross edges: All other edges of \( G \). They can go between vertices in the same DFS tree or between vertices in different trees.

The above classification is useful in many applications involving DFS in a directed graph. For example, they are useful in understanding the connectivity structure in directed graphs. Directed graphs have a different notion of connectivity compared to undirected graphs. A directed graph is said to be strongly connected if for every pair of nodes \( u \) and \( v \), there is a directed path from \( u \) to \( v \) and from \( v \) to \( u \). It is called weakly connected if the graph is connected by ignoring the directions (i.e., the underlying undirected graph is connected). The edge classification of DFS helps in finding strongly connected components in directed graphs. Worked
Figure 11.4: Example: A directed graph and its DFS tree.

Exercise 11.2 solves the problem of checking whether a directed graph is strongly connected or not. This approach can be modified to find all the strongly connected components of a directed graph as well (see Exercise 11.8).

11.2.6 Topological Sort

We will consider an important application of DFS in a directed graph, called topological (topo) sort. Topo sort can be used to find a linear ordering of the vertices of a directed acyclic graph (DAG). A DAG is a directed graph with no (directed) cycle. See Figure 11.5 for an example.

DAGs are used in real-world applications to model precedences among events. For example, you have to complete a set of $n$ operations (e.g., these might be operations to assemble a car in a factory). There are precedences among the operations which are represented as directed edges: $(u, v)$ denotes that operation $u$ has to be performed before operation $v$. The above edges form a DAG, if it is valid set of operations. Topological sorting of a DAG gives a valid ordering of performing operations.
Figure 11.5: Example: A DAG. The top shows a topological ordering of the vertices of the DAG so that all edges are directed from left to right. The bottom shows the DFS tree of the DAG. Note that it has no back edges.
such that all the precedence constraints are obeyed. Note that there may be more than one valid ordering — topo sort returns one such ordering.

Topological ordering can also be viewed as an ordering of the vertices of the DAG along a horizontal line so that all directed edges go from left to right. Note that if the graph is not a DAG, then there is no topological ordering possible (we leave it to the reader to prove this).

**Using DFS for Topo sort**  We need to modify DFS code a little for doing topo sort as follows. The modified DFS code computes finishing order for vertices (defined below) as well. Recall that $\text{num}[v]$ represents the order in which the vertices were visited, i.e., the time at which the vertex was first visited. Let $\text{fin}[v]$ represent the order in which $v$ was finished processing, i.e., the time at which the recursive call $DFS(v)$ returns. Algorithm 37 gives the DFS algorithm that also outputs the finishing times of vertices.

**Algorithm 37 Algorithm DFS-graph($G$).** Input: A directed graph. Output: finishing times of vertices.

1: for All $v \in V$ do
2: \hspace{1em} visited[$v$] = false
3: \hspace{1em} $time = 0$
4: for All $v \in V$ do
5: \hspace{2em} if visited[$v$] = false then
6: \hspace{3em} DFS($G$, $v$)

**Algorithm 38 Algorithm DFS($G$, $s$).** Input: A directed graph. Output: finishing times of vertices.

1: \hspace{1em} visited[$s$] = true
2: \hspace{1em} $time = time + 1$
3: \hspace{1em} $num[s] = time$
4: for Each vertex $v \in Adj(s)$ do
5: \hspace{2em} if visited[$v$] = false then
6: \hspace{3em} DFS($v$)
7: \hspace{1em} $time = time + 1$
8: \hspace{1em} fin[$s$] = time

**Algorithm Topo Sort**  The Topo Sort algorithm takes as input a DAG $G = (V, E)$ represented by its adjacency list. It outputs a topological ordering of vertices of $G$. The algorithm’s pseudocode is given in Algorithm 39. The algorithm simply calls DFS on $G$ to compute the finish times and list the
11.2. Depth-First Search (DFS)

vertices in decreasing order of their finish times. We will show that this order is a valid topological ordering.

Algorithm 39 Algorithm Topo-sort(G). Input: A DAG G. Output: Topological ordering of the vertices of G.

1: Call DFS(G) to compute fin[v] for each vertex v.
2: List vertices in decreasing order of their finish times.

One way of implementing the Step 2 of the Topo Sort algorithm is to store vertices as they finish in a vector (or array) and then returning the vertices in reverse order.

It is clear that the running time of Topo Sort is the same as DFS, i.e., $O(|V| + |E|)$.

Correctness of Topo Sort The correctness of Topo Sort uses the following key property of a DAG.

Lemma 11.3. In a DAG G, DFS yields no back edges.

Proof. Suppose DFS on G yields a back edge (u, v). Then v is an ancestor of u in the DFS (directed) tree. Thus there is a path from v to u in G (using the tree edges), and the back edge completes a directed cycle. This contradicts the definition of a DAG.

Theorem 11.1. Algorithm Topo-sort correctly produces a topological ordering of a DAG.

Proof. It is enough to show that if there is a directed edge (u, v) then $fin[v] < fin[u]$. DFS will explore edge (u, v) when it is exploring vertex u. There are 2 cases:

1. v is not yet visited: Then v will be visited for the first time and $DFS(v)$ will be called which will finish first before $DFS(u)$ finishes. Hence $fin[v] < fin[u]$.

2. v is already visited: Note that (u, v) cannot be a back edge, i.e., v cannot be an ancestor of u. The only possibility is that v has already finished exploring. Since u is not yet finished, $fin[u] > fin[v]$.
11. FUNDAMENTAL GRAPH ALGORITHMS

11.3 Breadth-first search (BFS)

In BFS, the search starts from a vertex $s$, and visits vertices in increasing order of distance from $s$. BFS yields a spanning tree called breadth-first spanning (BFS) tree in an undirected connected graph. An important application of breadth-first search is finding the (shortest path) distance (as well as shortest path) from $s$ to all other vertices. Recall that the distance between two vertices is the number of edges in a shortest path between them. BFS can be implemented in $O(|V| + |E|)$ time.

Figure 11.3 shows an undirected graph and its BFS tree.

11.3.1 BFS Algorithm

The pseudocode for the BFS algorithm is given in Algorithm 40. It takes as input an undirected and connected graph $G = (V, E)$ represented by its adjacency list; and a source vertex $s \in V$. It outputs the following:

1. BFS tree $T$ rooted at $s$.

2. $dist$ array — $dist[v]$ gives the distance of $s$ to $v$. (the $dist$ array is initialized to $\infty$ for all nodes.)

3. $parent$ array — $parent[v]$ stores the parent of $v$ in the tree $T$.
11.3. BREADTH-FIRST SEARCH (BFS)

The algorithm uses the following auxiliary data structures:

1. A queue \( Q \) that stores the list of vertices that will be explored.

2. An array \( \text{visited}[] \) that is initialized to false for every vertex \( v \in V \), i.e., \( \text{visited}[v] = \text{false} \). This array maintains whether the vertex has already been visited or not.

The algorithm simply explores all vertices starting from \( s \) in a “layered” fashion. That is, first all neighbors of \( s \) (i.e., nodes at distance 1 from \( s \)) are explored; then all nodes at distance 2 from \( s \) are explored, and so on. Thus BFS tree is a layered graph with levels: \( s \) is at level 0 (distance 0 from itself), neighbors of \( s \) are at level 1, nodes at distance 2 from \( s \) are at level 2 and so. In other words, nodes are explored in increasing order of distances from \( s \), with a higher distance node will be explored only when all lower distance nodes have been finished exploring. Note that as in DFS, a node is explored only once (enforced by the \( \text{visited} \) array). The most natural way to keep track of the order of the nodes is a queue data structure (See Appendix B), where the neighbors of a node that is currently explored is added to the back of the queue. These neighbors include nodes at the next level and they will explored when all nodes in the current level are finished.

Algorithm 40 Algorithm BFS\((G, s)\). Input: An undirected and connected graph \( G = (V, E) \) represented by its adjacency list; and a source vertex \( s \in V \). Output: BFS tree \( T \) rooted at \( s \); dist array — \( \text{dist}[v] \) gives the distance of \( s \) to \( v \); parent array — \( \text{parent}[v] \) stores the parent of \( v \) in the tree \( T \).

1: \( \text{dist}[s] = 0 \)
2: \( \text{visited}[s] = \text{true} \)
3: \( Q = \emptyset \)
4: \( \text{ENQUEUE}(Q, s) \)
5: while \( Q \neq \emptyset \) do
6: \( u = \text{DEQUEUE}(Q) \)
7: for Each vertex \( v \in \text{Adj}(u) \) do
8: \( \text{if } \text{visited}[v] = \text{false} \) then
9: \( \text{visited}[v] = \text{true} \)
10: \( \text{dist}[v] = \text{dist}[u] + 1 \)
11: \( \text{parent}[v] = u \)
12: Add edge \((u, v)\) to \( T \)
13: \( \text{ENQUEUE}(Q, v) \)
Run Time of BFS  The time for number of operations before the WHILE loop is $O(n)$. We now focus on bounding the time taken to finish the WHILE loop. The number of iterations of the WHILE loop is $O(n)$ since each vertex is inserted into the queue exactly once and each iteration dequeues one vertex. (Note that each ENQUEUE and DEQUEUE operation can be implemented in $O(1)$ time.)

The FOR loop (inside the WHILE loop) takes time $O(\deg(v))$. Thus total time spent in the FOR loop is $O(\sum_{v \in V} \deg(v)) = O(m)$. Hence overall running time of BFS is $O(n + m)$.

Correctness of BFS

**Theorem 11.2.** BFS correctly computes the (shortest path) distance from $s$ to every node $v \in V$. When the algorithm ends, $\text{dist}[v]$ has the correct distance from $s$ to $v$. Moreover, the path from $s$ to $v$ in the BFS tree $T$ gives a shortest path from $s$ to $v$.

**Proof.** Let the (correct) distance from $s$ to a node $v$ be denoted by $d[v]$. We show that BFS correctly computes $d[v]$ for all $v$.

Let $L[v]$ denote the level of node $v$ in BFS tree $T$. $L[v]$ is defined as the number of tree edges from $s$ to $v$. $L[s] = 0$.

We show the theorem by induction on the level of $v$ (i.e., distance from $s$).

**Induction hypothesis:** BFS correctly computes $d[v]$ for all nodes $v$, such that $0 \leq L[v] \leq k$. Moreover, $d[v] = \text{dist}[v] = L[v]$ and the path from $s$ to $v$ in tree $T$ is a shortest path from $s$ to $v$.

**Base case:** Holds trivially for level 0 (i.e., vertex $s$), since $d[s] = \text{dist}[s] = L[s] = 0$ and the shortest path has no edges.

**Induction step:** Assume that the hypothesis holds for all nodes up to level $k$. We will show that it holds for level $k + 1$ as well.

Consider a node $v$ at level $k + 1$, i.e., $L[v] = k + 1$. Note that $d[v] \geq k + 1$, since otherwise if $d[v] \leq k$ the induction hypothesis for $v$ will be contradicted.

Consider the tree path $P = s, \ldots, u, v$. Since $u$ is the parent of $v$ in the tree, $u$ belongs to level $k$ and by induction hypothesis: $d[u] = \text{dist}[u] = L[u] = k$. Hence there is a path $(P)$ from $s$ to $v$ having length $k + 1$. This should be a shortest path between $s$ and $v$. Note that $\text{dist}[v]$ is set to $\text{dist}[u] + 1 = k + 1$. Hence, $d[v] = \text{dist}[v] = L[v] = k + 1$. 

\qed
11.4 Worked Exercises

Worked Exercise 11.1. A graph is called biconnected if it does not contain any cut-vertex. The biconnected components of a graph \( G = (V, E) \) is a partition of the edges into sets such that the (sub)graph formed by each set of edges is biconnected.

(a) Modify the algorithm that finds cut vertices to find biconnected components also. Show how you will modify it so that it will output the biconnected components also. Note that your algorithm should run in \( O(|V| + |E|) \) time. Give a clear description of your algorithm and the pseudocode and argue the running time bound.

(b) Run your algorithm on the undirected graph given in Figure 11.3. Show the main steps and output the cut vertices and the biconnected components of the graph.

Solution:

(a) Note that the algorithm for finding cut-vertices is recursive and outputs cut-vertices in a bottom-up manner. Refer to Algorithm 36. The main observation is when we make the recursive call find-cut(\( w \)) (assume that \( v \) is the parent of \( w \) in the dfs tree, i.e, \( w \) is a neighbor of \( v \) and \( w \) is not visited yet) then \((v, w)\) together with all edges (both tree and back edges) encountered during this find-cut(\( w \)) call (except for edges in other biconnected components contained in subtree \( w \)) forms a biconnected component. When we find a cut-vertex \( v \) (say, after returning from one of its children, \( w \)) then all the edges that we explored from \((v, w)\) onwards form a biconnected component (except those in other biconnected components contained in the subtree of \( w \) — notice the recursive nature of this statement). Thus we get to know the edges of a biconnected component only when after we discover a cut-vertex. Hence to keep track of edges that will be explored we use a stack. We keep pushing edges on to the stack until a cut vertex is found, then we pop edges of the stack (until \((v,w)\)). These set of edges will form a biconnected component.

The modified algorithm is given below:

(Initially, time = 0 and visited[\( v \)] = false for all \( v \). Num is also initialized to zero for all \( v \). Initially Stack S is empty.)

\[ \text{Findcut}(v) \]

\[ \text{visited}[v] = \text{true}; \text{time}++; \text{Low}[v] = \text{Num}[v] = \text{time}; \]

for each neighbor \( w \) of \( v \)

if parent[\( v \)] \( \neq w \) and \( \text{Num}[w] < \text{Num}[v] \) then

push \((v,w)\) to top of stack \( S \); // \((v,w)\) is not on stack before
if (not visited[w])
    parent[w] = v;
Findcut[w];
if (Low[w] ⩾ Num[v]) // cut vertex
    output v is a cut vertex;
    pop edges from stack up to (v,w) and output these edges as a biconnected component;
endif
    Low[v] = min(Low[v],Low[w]);
endif
else if parent [v] ≠ w) // (v,w) is a back edge
    Low[v] = min(Low[v], Num[w]);
endif
endfor

Note that the above algorithm will output the biconnected components correctly regardless whether the root has one or more than one children (why ?). However to output root as a cutvertex or not we have to test separately (see Section 11.2.4). In line 3 of the algorithm we are checking whether the edge (v,w) is not already in the stack before we push it. For this, we check whether w is not a parent of v and Num[w] < Num[v]. (If w is a parent of v then we pushed (w,v) into the stack when we visited w.) If both these are true then (v,w) is either a back edge or a tree edge (that we are just going to explore — Num[w] will be zero). So this edge has not been explored before and we push it to the Stack.

(b) The cut vertices of the graph in Figure 11.3 are b, g, d. The 4 biconnected components are the edges induced by the following sets of vertices: 
\{a, b, c, d\}, \{g, h, i\}, \{d, e, f\}, \{b, g\}.

**Worked Exercise 11.2.** Using two applications of DFS in a directed graph, give an algorithm to check if a given directed graph is strongly connected or not. Argue the correctness of your algorithm.

**Solution:**

As defined in Section 11.2.5, a directed graph is strongly connected, if for every pair of vertices u and v, there is a directed path from u to v and a directed path from v to u in the graph. The algorithm for testing whether a directed graph G is strongly connected is as follows.

1. Run DFS(G, v) (from any source vertex v). Check if all vertices of G are in the DFS spanning tree. If there are less than n vertices in the spanning tree, then output “G is not strongly connected.”
2. Reverse the directions of edges in $G$. That is $(u, v)$ is an edge, then delete it and add $(v, u)$ as an edge (note that if both $(u, v)$ and $(v, u)$ are in $G$, then nothing is really added.) Call the reversed graph $G'$. Run DFS($G', v$). Check if all vertices of $G$ are in the DFS spanning tree. If there are less than $n$ vertices in the DFS spanning tree, then output "$G$ is not strongly connected."

3. Output "$G$ is strongly connected."

The running time of the above algorithm is clearly $O(n + m)$, since DFS runs in linear time. Reversing edges, can be done in linear time, by simply visiting each edge in the adjacency list.

**Correctness** We show that the algorithm outputs that the graph is strongly connected if and only if the graph is indeed so. If, in Step 1, all vertices are not in the spanning tree, then clearly some vertex is not reachable from $v$ in $G$ and hence not visited when doing DFS from $v$. Thus, $G$ is not strongly connected. If, in Step 2, all vertices are not in the DFS spanning tree, that means that some vertex (say $u$) is not reachable from $v$ in $G'$. Since edges in $G'$ are reversed, this means that there is no directed path from $u$ to $v$ in $G$. Thus the graph $G$ is not strongly connected.

On the other hand, if the spanning trees in both steps include all the vertices, that means all vertices in $G$ are reachable from $v$ and all other vertices can reach $v$. Hence this means every pair of vertex can reach other — through $v$. For example, consider vertices $x$ and $y$. There is a path from $v$ to $x$ and $v$ to $y$. Also there is a path from $x$ to $v$ and $y$ to $v$. Hence there is a path from $x$ to $y$ and from $y$ to $x$. Hence the graph $G$ is strongly connected.

### 11.5 Exercises

**Exercise 11.1.** Consider the graph in Figure 11.1. It has two connected components, one containing the vertex $g$ by itself and the rest of the vertices. We focus only on the latter connected component with 6 vertices — $a, b, c, d, e, f$ — call it $H$.

(a) Draw the DFS tree of $H$ starting from vertex $a$. What are its tree edges and back edges? Give also the DFS numbering of the vertices.

(b) List the cut vertices of $H$ (if any). Find the *low* values of all its vertices.

(c) List the biconnected components of $H$.

(d) List the cut edges of $H$ (if any).
Exercise 11.2. You are given a connected undirected graph. Starting from some vertex \( s \), the goal is to go through each edge of the graph \textbf{exactly once} and come back to the starting vertex \( s \). This is not possible for all graphs, e.g., in the Figure 11.5 it is not possible in graph \( G_1 \) for any starting vertex. However it is possible in graph \( G_2 \).

(a) Prove a condition that if satisfied allows one, starting from some vertex \( s \), the goal is to go through each edge of the graph \textbf{exactly once} and come back to the starting vertex \( s \).

(b) Give an algorithm to accomplish the above.

Exercise 11.3. Give a simple \( O(mn) \) time algorithm to output all the cut vertices of a graph.

Exercise 11.4. (a) A cut-edge of a graph \( G = (V, E) \) is an edge whose removal partitions the graph into two or more connected components (note that some of these components can be single vertex). Give a \( O(|V| + |E|) \) time algorithm to find all the cut-edges of \( G \). Give a clear description of your algorithm and the pseudocode and argue the running time bound. (Hint: Again this is a modification of the algorithm to find cut-vertices. What is the condition for an edge to be a cut-edge? Can it lie on a simple cycle of \( G \)?)

(b) Run your algorithm on the undirected graph given in Figure 11.3 of the textbook. Show the main steps and output the cut edges of the graph.

Exercise 11.5. Given an undirected connected graph \( G = (V, E) \) with \( |V| = n \) nodes, let \( u \) and \( v \) be two nodes in \( G \) such that their distance is greater than \( n/2 \), i.e., the shortest path between \( u \) and \( v \) has at least \( \frac{n}{2} + 1 \) edges.
11.5. EXERCISES

(a) Show that there is at least one vertex \( x \) in the graph whose removal disconnects \( u \) and \( v \). (Hint: Can there be a cycle in \( G \) that includes \( u \) and \( v \)?)

(b) Give an \( O(|V| + |E|) \)-time (i.e., linear time) algorithm to find such a vertex \( x \).

Exercise 11.6. Let \( G = (V, E) \) be a directed graph. We call \( G \) to be mildly-connected if for all pairs of vertices \( u, v \in V \), there is a path either from \( u \) to \( v \) or from \( v \) to \( u \) (or possibly both). Give an \( O(|V| + |E|) \) (i.e., linear) time algorithm to determine if a given directed graph is mildly-connected.

(Hint: First find the component graph \( G' = (V', E') \) of \( G \) (i.e., the vertices of \( G' \) correspond to the strongly connected components and there is a directed edge between two such vertices \( u', v' \in V' \) if there a directed edge between some vertex of the component (corresponding to) \( u' \) and some vertex of component (corresponding to) \( v' \)). (You should show how to construct the component graph in linear time.) Note that the component graph is a DAG (directed acyclic graph) (why?). What property should this DAG have for \( G \) to be mildly connected? Topological sort will prove useful now.)

Exercise 11.7. The diameter of a graph \( G = (V, E) \) is the maximum distance among all pairs of vertices. That is if \( d(u, v) \) is the distance between vertices \( u \) and \( v \) in \( G \), \( \text{diam}(G) = \max_{u,v \in V} d(u,v) \). The goal is to design an efficient algorithm to compute the diameter of a tree (i.e., the graph is a tree) and analyze the running time of the algorithm. Assume that you are given a rooted tree with parent-child relationships, i.e., each node, except the root, has a pointer to its parent and also pointers to its children (if any). Assume that the tree is rooted at a node with degree at least two (i.e., the root has at least two children). Note that, one does not need to assume this, since given any tree, you can choose a node with degree at least two, and do a BFS from that node and create a tree with parent-child relationships.

(i) Show that the distance determining the diameter will always be between two leaves.

(ii) Describe and analyze an \( O(n) \) time algorithm to compute the diameter of tree \( T \).

Exercise 11.8. A strongly connected component of a directed graph \( G = (V, E) \) is a maximal subset of vertices \( S \subseteq V \) such that for every pair of vertices \( u \) and \( v \) in \( S \), there is a directed path from \( u \) to \( v \) as well as a directed path from \( v \) to \( u \), i.e., \( u \) and \( v \) are mutually reachable from each other. Note that “maximal” means that the set \( S \) is maximal, i.e., no more vertices can be added to \( S \) and still guarantee the mutual reachability property.
(1) Find the strongly connected components of the directed graph in Figure 11.4.

(2) Let $S_1, S_2, \ldots$ be the strongly connected components of $G$. Construct the "component graph" $G'$ of the directed graph $G$ as follows: Each node of the component graph $G'$ represents a strongly connected component of $G$ (i.e., there are as many nodes in $G'$ as there are strongly connected components in $G$) and there is a directed edge between two nodes (representing, say, strongly connected components $S_i$ and $S_j$) if a vertex in component $S_i$ has a directed edge to a vertex in component $S_j$.

Draw the component graph of the directed graph in Figure 11.4.

(3) Show that the component graph $G'$ of any directed graph $G$ is always a DAG. (Hint: What if there is a directed cycle in the component graph?)

(4) Consider the following algorithm for finding the strongly connected components of a given directed graph $G$:

1. Run the Algorithm DFS-graph (Algorithm 37) on $G$ and compute the finish times $f_{in}[v]$ of every vertex.

2. Compute the reverse graph $G^r$ by reversing the directions of edges in $G$.

3. Run the Algorithm DFS-graph (Algorithm 37) on $G^r$, but in the for loop consider the vertices in the decreasing order of $f_{in}[v]$ computed in Step 1.

4. Output the vertices of each tree in the depth-first forest formed in Step 3 as a separate strongly connected component (i.e., each time $DFS(G, v)$ returns, all the vertices it explores form a strongly connected component).

Run the above algorithm on the the directed graph in Figure 11.4 and show that it indeed outputs the strongly connected components. Show the steps 1-4 of the algorithm on this graph.

Exercise 11.9. Implement DFS as an iterative algorithm using the stack data structure. (Hint: This is a modification of the BFS algorithm using the stack (instead of a queue), but with a crucial difference on what to push and pop).

Exercise 11.10. An undirected graph $G = (V, E)$ is called bipartite if it contains no odd cycle, i.e., no cycle has an odd number of edges. Note that a graph with no cycles is also bipartite. We are going to show an
important property of bipartite graphs: its vertex set can be partitioned into
two disjoint sets $A$ and $B$, i.e., $A \cup B = V$ and $A \cap B = \phi$, such that all the
edges are between vertices in $A$ and vertices in $B$ — in other words, there
are no edges between any two vertices in $A$ and any two vertices in $B$.

The goal of this problem is that, given a connected bipartite graph
$G = (V,E)$, to find a partition of the vertex set into $A$ and $B$ such that
all edges are between some vertex in $A$ and some vertex in $B$.

(a) Do Breadth-First Search (BFS) on $G$ from some starting vertex, say $a$, and compute the distance (i.e., the level in the BFS tree) of every vertex $v$ to $a$. Show that if $G$ is bipartite then there are no edges between vertices in the same level. (Hint: Use the odd cycle property of bipartite graph).

(b) Show that you can output the sets $A$ and $B$ in linear time, i.e., $O(|V| + |E|)$ time.
12

Graph Algorithms: MST and Shortest Paths

In this chapter, we study two fundamental graph optimization problems, namely the minimum spanning tree (MST) and the shortest paths problem. Both problems have numerous real-world applications and also arise as subroutines in solving other problems (including non-graph problems that can be cast as graph problems). This chapter uses techniques from both greedy algorithms (Chapter 7) and dynamic programming (Chapter 6). We use greedy technique to solve MST efficiently, while for shortest paths, both greedy and dynamic programming are used. It will be helpful to review the techniques of both greedy and dynamic programming while reading this chapter. MST and shortest paths are two further examples of applying the above algorithmic techniques in the context of graph problems.

12.1 Minimum Spanning Tree (MST) Problem

The MST problem is defined as follows. Given a weighted undirected connected graph \( G = (V, E, w) \) \((w \) is the weight function assigning weights to edges), the goal is to find a spanning tree of \( G \) (see Appendix E.7) with minimum total weight (i.e., sum of the weights of its edges is minimum). MST is a fundamental graph optimization problem with applications in many settings. For example, consider a communication network modelled as a weighted graph. Edge weight may represent the delay of sending a message through the edge. Suppose we want to repeatedly “broadcast” messages from one node to all other nodes in the network so that the total delay is minimized. A MST serves as a backbone for efficient broadcasting.
in a communication network. Broadcasting via a MST minimizes the total delay.

In this section, we will design two greedy algorithms for this problem. Figure 12.1 shows a weighted graph and its MST.

### 12.1.1 Kruskal’s (Greedy) Algorithm

We first consider a classical algorithm called Kruskal’s algorithm which is a simple greedy algorithm. The idea of the algorithm is as follows. Given a weighted graph, consider its edges in order of non-decreasing weight; this makes sense since we want to add edges with as small weight as possible. We add edges one at a time to the MST. Note that we need to add exactly \( n - 1 \) edges (since any spanning tree should have exactly those many edges — See Theorem E.1 in Appendix E). Initially the set of tree edges is empty.

The algorithm simply adds edges one by one to the tree in the above order if it does not close a cycle with edges already added to the tree. Figure 12.1 shows a weighted graph and the order in which edges are added by the Kruskal’s algorithm.

The above algorithm is greedy because, once an edge is added to the tree it is never removed (no backtracking). We will show next that the above algorithm indeed produces a MST, i.e., the added edges induces a MST of the input graph. The algorithm’s pseudocode is given below.

**Pseudocode for Kruskal’s Algorithm**

**Input:** A weighted, undirected, connected graph \( G = (V, E) \), \( n = |V| \), \( m = |E| \).

**Output:** A MST \( T \) of \( G \) (\( T \) is the set of MST edges).

**Algorithm** Kruskal\((G)\)

1. Start with no edges in \( T \), i.e., \( T = \emptyset \).
2. Sort edge set \( E \) in increasing order of weight. Let the sorted order be \( e_1, e_2, \ldots, e_m \).
3. \( i = 1 \).
4. while \( |T| < n \) do
   4.1 Add \( e_i \) to \( T \) if it does not close a cycle with edges already in \( T \).
   4.2 \( i = i + 1 \).
5. Output \( T \).

**Correctness of Kruskal’s algorithm**

It is clear that Kruskal’s algorithm produces a spanning tree (why?). We next show that it produces a spanning tree whose weight is minimum.
12.1. MINIMUM SPANNING TREE (MST) PROBLEM

Figure 12.1: A connected weighted graph (left) and its MST (right). Note that MST is a spanning tree and hence will have exactly $n - 1$ edges on a connected graph of $n$ nodes. (Top) Kruskal’s algorithm, starting from an empty tree, adds edges in the order of increasing weight, starting from the smallest weighted edge, unless the added edge closes a cycle with edges already added. In this example, Kruskal adds edges in the following order (shown in red): $(a, b), (b, g), (b, d), (a, c), (d, e), (d, f)$. The cut that validates the addition of edge $(a, c)$ is the partition (shown by red line) — $(\{a, b, d, g\}, \{c, e, f\})$, because $(a, c)$ is the lightest edge crossing this cut. (Bottom) Prim’s algorithm, starting from a tree with one vertex, adds edges so that the tree is extended by one additional vertex at a time. The edge that connects a non-tree vertex to a vertex in the tree via the least-weighted edge is added. In this example, Prim adds edges in the following order (shown in red) starting from a tree with the singleton vertex $f$: $(f, d), (b, d), (a, b), (b, g), (a, c), (d, e)$. The cut that validates the addition of edge $(a, c)$ is the partition (shown by red line) — $(\{f, a, b, d, g\}, \{c, e\})$ because $(a, c)$ is the lightest edge crossing this cut.
Recall the following technique which was used earlier (see Chapter 7) in proving optimality of greedy algorithms. Compare the greedy solution with the optimal. If the two solutions differ, then find the first \( x_i \) where they differ. Then show how to make \( x_i \) in the optimal solution equal to that of the greedy solution without losing optimality. Repeated use of this transformation shows that the greedy solution is optimal. This proof idea can be called as an “exchange” or “transform” argument. We will use an “exchange” argument to show that the greedy algorithm produces the optimal.

**Correctness Proof of Kruskal’s Algorithm**

**Theorem 12.1.** Kruskal’s algorithm produces a MST.

We need the following definition for the proof. A cut \((S, V - S)\) of an undirected graph \(G = (V, E)\) is a partition of \(V\). An edge crosses the cut if one of its endvertices is in \(S\) and the other is in \(V - S\). We will show that the tree \(T\) output has minimum weight among all spanning trees in \(G\).
Proof. Let $T'$ be a MST of $G$. We will show either $T' = T$ or we can transform $T'$ to $T$ without losing optimality. Sort edges of $T'$ in increasing order of weight. Let $e = (u, v)$ be the first edge added by Kruskal that is not in $T'$. Let $U$ be the set of edges in $T$ just before $e$ is added. Just before $e$ is added consider the cut $(S, V - S)$, where $S$ is the set of all vertices that are connected to vertex $u$ using (only) edges of $U$. Since $e$ is added by the algorithm, this means that $v \in V - S$ (otherwise, if $v \in S$, this means by our construction of $S$, $v$ will close a cycle with $u$). That is, $e$ crosses the cut $(S, V - S)$. See Figure 12.2.

The optimal tree $T'$ also should have at least one tree edge crossing the cut. (If no edges of $T'$ cross the cut, then the tree is not connected.) Let $e' \in T'$ cross the cut. We note that $w(e') \geq w(e)$. This is because, $e'$ cannot be $e$ or any of the other edges added before $e$, since all such edges do not cross the cut (by construction of the cut $(S, V - S)$). Remove $e'$ from $T'$ and add $e$ to get a spanning tree $T''$. Why is $T''$ a spanning tree? Removing $e'$ from $T'$ disconnects $T'$ into two connected components — one containing $u$ and the other containing $v$. Adding $e = (u, v)$ connects them back.

Now: $w(T'') = w(T') - w(e') + w(e) \leq w(T')$, since $w(e') \geq w(e)$. Thus $T''$ is also a MST, but it has one more edge ($e$) which is in common with $T$. Repeat the argument by comparing $T$ and $T''$. 

Implementing Kruskal’s Algorithm Efficiently  
Assume that we take the graph as an adjacency list. Since we process the edges in increasing order of weight, we can first sort all the edges. This takes $O(|E| \log |E|) = O(|E| \log |V|)$ time. The main non-trivial operation is the condition in line 4.1 — checking for a cycle.

We can implement line 4.1 of the pseudocode in $O(\log |V|)$ time using a suitable data structure as explained below. Hence the whole algorithm can be implemented in $O(|E| \log |V|)$ time.

Note that Kruskal’s algorithm can be viewed as follows. It maintains a dynamic collection of connected components (each component is a tree) at every step. Initially, every vertex is a tree by itself (i.e., singleton nodes). In each step, Kruskal’s algorithm adds an edge that does not close a cycle with the edges added, i.e., it adds an edge between two different trees and joins them into one tree. Each edge reduces the number of trees by one — thus starting with $n$ trees, after adding $n - 1$ edges we are left with one tree which is the MST.
Algorithm 41 Algorithm Kruskal(G). Input: A weighted, undirected, connected graph \( G = (V, E) \), \( n = |V|, m = |E| \); Output: A MST \( T \) of \( G \) (\( T \) is the set of MST edges).

1: \( T = \emptyset \).
2: for Each vertex \( v \in V \) do
3:  Make-Set(\( v \)) // create a singleton set for each vertex in \( G \)
4: Sort edge set \( E \) in increasing order of weight. Let the sorted order be \( e_1, e_2, \ldots, e_m \).
5: \( i = 1 \).
6: while \( |T| < n \) do
7:  Let \( e_i = (u_i, v_i) \)
8:  \( \ell_1 = \text{find}(u_i) \)
9:  \( \ell_2 = \text{find}(v_i) \)
10: if \( \ell_1 \neq \ell_2 \) then
11:  Since the representatives of \( u_i \) and \( v_i \) are different, they belong to different trees and does not close a cycle with edges already in \( T \)
12:  \( T = T \cup e_i \)
13: union(\( \ell_1, \ell_2 \)) // combine the two trees
14: \( i = i + 1 \).
15: Output \( T \).

To efficiently check whether an edge \((u, v)\) closes a cycle in each step, we need to efficiently check if \( u \) and \( v \) belong to the same tree or not — in the former case, it closes a cycle, and in the latter case it does not. Further, in the latter case, it joins two trees and makes them as one tree and thus the number of trees is reduced by one. We need a data structure to maintain the different trees dynamically as they change so that we can efficiently check whether an edge \((u, v)\) connects two different trees or not. We accomplish this using the following general data structure which is known as a union-find data structure (see Appendix B.4), since it implements two operations in a collection of disjoint sets (here, each set is a tree):

- **find**: Given an element \( u \), \( \text{find}(u) \) finds the set, i.e., the representative of the set, where \( u \) belongs to. Note that each set has a unique representative (or leader).

- **union**: Given two elements \( u \) and \( v \) belonging to two different sets, \( \text{union}(u, v) \) joins the two sets into one set.

A union-find data structure also supports a third operation (for initializing the sets) called Make-Set(\( v \)) which creates a singleton set with ele-
ment \( v \). We initially call Make-Set on each vertex \( v \) to create \( n \) singleton sets — each representing a tree by itself.

Appendix B.4 shows a way to implement the union-find operations so that any find or union operation takes \( O(\log n) \) time, where \( n = |V| \) is the total number of elements in the set. In the case of Kruskal, \( n \) is the number of nodes.

Note that we can implement Kruskal’s algorithm efficiently by using a union-find data structure to maintain the disjoint trees in every step. To check whether an edge \((u, v)\) closes a cycle, we check whether \( \text{find}(u) = \text{find}(v) \), i.e., whether \( u \) and \( v \) belong to the same set (i.e., same tree), by checking whether their representatives are the same. If they are different, then \( u \) and \( v \) belong to different trees and hence doesn’t close a cycle. In that case, we can do union\((u, v)\) to join the two trees into one tree. Both operations take \( O(\log |V|) \) time with the implementation given in Appendix B.4. Hence the whole algorithm can be implemented in \( O(\log |V|) \) time. Algorithm 41 gives a pseudocode for Kruskal’s algorithm using the union-find data structure.

### 12.1.2 Prim’s Algorithm

Prim’s algorithm (which is essentially the same as Dijkstra’s algorithm for shortest paths — see 12.2.1) builds the MST one edge at a time starting from some vertex which will be the root of the tree. Let \( G \) be a connected (undirected) graph and let \( r \) be some vertex in \( G \). The algorithm builds a MST rooted at \( r \). Unlike Kruskal, it will maintain only one tree and in every step will add one new vertex to the tree — in other words, an edge that connects a vertex in the tree to a vertex outside the tree. The edge that is chosen to be added will be the minimum weight edge connecting the tree \( A \) to a vertex not in the tree. Worked Exercise 12.1 shows that it is safe to add this edge to the MST. Figure 12.1 shows a weighted graph and the order in which edges are added by the Prim’s algorithm.

During the algorithm, all vertices that are not in the tree are kept in a min-heap data structure \( Q \) based on a key defined as: \( \text{key}[v] \) is the minimum weight of any edge connecting \( v \) to a vertex in the tree. If there is no such edge then \( \text{key}[v] = \infty \). \( \pi[v] \) names the parent of \( v \) in the tree.

Note that \( \text{key}[v] \) has to be updated after every iteration. Suppose \( u \) is the vertex added in the current iteration. Then clearly, the only key values that need to be updated are the neighbors of \( u \). That’s what Prim’s algorithm does.
The tree is kept implicitly as
\[ A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}. \]

When the algorithm terminates \( Q \) is empty and the MST is given by
\[ A = \{(v, \pi[v]) : v \in V - \{r\}\}. \]

At every step of the Prim’s algorithm, one new vertex is added to the tree rooted at \( r \). The new vertex \( u \) that is added has the smallest key value that is defined as above. The key values are kept in a heap data structure and hence the smallest key value can be obtained by doing an EXTRACT-MIN operation on the heap data structure. Note that once a vertex \( u \) is added to the tree, the key values of its neighbors may have to be updated (i.e., decreased), since their key values can decrease because \( u \) is now part of the tree. For a neighbor \( v \) of \( u \), if \( w(u, v) \) is smaller than its current key value, i.e., if \( w(u, v) < key[v] \), then \( key[v] \) is set to \( w(u, v) \), since \( v \) is connected to a tree vertex, i.e., \( u \), via smaller weight. The decrease of key value of \( v \) which is in the heap can be accomplished using a DECREASE-KEY operation, which decreases the key value of a give key to a new value (See Appendix B.1).

**Pseudocode for Prim’s Algorithm**

MST-Prim\((G, r)\)

1. **for** each \( u \in V[G] \)
2. \( key[u] = \infty \)
3. \( \pi[u] = NIL \)
4. \( key[r] = 0 \)
5. \( Q = V(G) \)
6. **while** \( Q \) is not empty
7. \( u = \text{EXTRACT-MIN}(Q) \)
8. **for** each \( v \in \text{Adj}[u] \)
9. **if** \( v \in Q \) and \( w(u, v) < key[v] \)
10. \( \text{DECREASE-KEY}(Q, key[v], w(u, v)) \)
11. \( \pi[v] = u \)
12. **return** \( \{(v, \pi[v]) : v \in V - \{r\}\} \)

**Running time**

The running time of Prim’s algorithm depends on the heap implementation. We observe that the algorithm uses \( O(n) \) EXTRACT-MIN operations and \( O(m) \) DECREASE-KEY operations. Since each node is extracted exactly once (once a node is removed from the heap, it is never
inserted again in the heap), there are \( n \) EXTRACT-MIN operations. For each node \( u \), there are \( \text{deg}(u) \) DECREASE-KEY operations, where \( \text{deg}(u) \) is the number of neighbors of \( u \). This is because, when a node \( u \) is extracted from the heap, all of its neighbors are examined exactly once in the for loop (Line 8 of the Prim’s algorithm). Hence the total number of DECREASE-KEY operations is \( \sum_{u \in V} \text{deg}(u) = O(m) \).

If we use a binary heap implementation for the heap data structure (Appendix B.1), then each EXTRACT-MIN and DECREASE-KEY operation takes \( O(\log n) \) time. Thus overall, this yields a \( O(n \log n + m \log n) = O(m \log n) \) time (since \( m \geq n - 1 \), as the graph is connected). Note that we assume that we are given the location of the key in the heap to implement the heap operations. This can be accomplished in constant time by maintaining an index (or pointer) to every vertex in the binary heap. Thus locating a key in the heap takes only \( O(1) \) time.

### 12.2 Shortest Paths

The shortest paths problem is one of the most important graph problems. In this problem, we are given a directed graph \( G = (V, E) \) and a weight function \( w : E \to \mathbb{Q} \). The weight of a path \( p = v_0, v_1, v_2, \ldots, v_k \) is defined as the sum of the weights of the edges in the path:

\[
w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).
\]

The weight of the shortest path from \( u \) to \( v \), \( \delta(u, v) \) is the minimum of \( w(p) \) for all \( p \) connecting \( u \) to \( v \), and \( \infty \) if there is no such path in \( G \).

#### Various Shortest Path Problems

There are many types of shortest path problems:

- **Single Source Shortest Paths (SSSP)**: Compute shortest paths from a given source to all vertices in the graph.
- **Single Destination Shortest Paths (SDSP)**: Compute shortest paths to a given destination from all vertices in the graph.
- **Single Pair Shortest Path (SPSP)**: Compute shortest path for a given pair of vertices.
- **All Pairs Shortest Paths (APSP)**: Compute the shortest paths for all pairs of vertices in the graph.
Negative edge weights are allowed, but the problem is not well defined when a **negative weight cycle** is present. A negative weight cycle is one which is a cycle in the graph and whose sum of edges of the cycle is less than zero. If the negative cycle is reachable from the source vertex, then by using the negative cycle path many times, one can reduce the shortest path distance to $-\infty$. Hence we assume that there is no negative weight cycle, or at least it is not reachable from the source vertex from which we need to find shortest paths, as in the SSSP or APSP problems.

We present various important algorithms for all the above problems. For the SSSP problem, we present two classical algorithms, namely Dijkstra’s and Bellman-Ford’s algorithms. Note that the SDSP problem can be solved by reversing the directions of the edges and solving the SSSP problem with the given destination as the source.

For the SPSP problem, where the goal is to find a shortest path from a given source to a given destination, we present an algorithm called $A^*$ that is a variant of Dijkstra’s algorithm. Although $A^*$ performance may not be better than Dijkstra’s algorithm in the worst-case, in many instances, especially in real-world problems, it can perform significantly better than Dijkstra’s.

One can solve the APSP problem, by solving $n$ instances of the SSSP problem, one from each node as a source. While this can be faster in some cases, it can be much slower in others. We present a well-known dynamic programming algorithm called **Floyd-Warshall** algorithm for this problem.

**An important property of shortest path** Shortest path has the following important property which shows that a shortest path has the following **optimal substructure property** that we studied in Dynamic Programming (Chapter 6). The property says that a shortest path contains within it other shortest paths. This property enables both greedy and DP algorithms for shortest paths.

**Lemma 12.1** (Optimal Substructure Property). If $p = v_0, v_1, ..., v_j, ..., v_k$ is a shortest path from $v_0$ to $v_k$, then $p' = v_0, v_1, ..., v_j$ is a shortest path from $v_0$ to $v_j$.

**Proof.** Assume that $p''$ is a shorter path from $v_0$ to $v_j$, then $p''$ followed by $v_{j+1}, ..., v_k$ would be a shorter path from $v_0$ to $v_k$ which contradicts the optimality of $p$. \qed
12.2. SHORTEST PATHS

12.2.1 Single Source Shortest Paths Problem

The single Source Shortest Path Problem (SSSP) is a classical and important problem. The problem is to compute shortest paths from a given source vertex \( s \) to all vertices in the graph. This problem has many applications, e.g., finding the shortest route in a map, or finding routes in the Internet. Dijkstra’s algorithm is used in the Internet to find short paths for routing.

Figure 12.3 shows an example of a weighted graph and its shortest path tree (SPT).

Dijkstra’s Algorithm

The algorithm constructs a tree of shortest paths (rooted at \( s \)) in a greedy manner. This is called a shortest paths tree (SPT). The root of the tree is \( s \). A path on the tree corresponds to shortest paths to \( s \) for all vertices on that path. The shortest paths tree is constructed in \(|V| \) iterations.
Dijkstra’s algorithm computes the distances from source $s$ to other vertices in increasing order of their distances from $s$. Starting from $S = \emptyset$ and extending $S$ by one vertex in each iteration, the algorithm computes a shortest paths tree restricted to vertices only from $S$. When $S = V$ we get the shortest paths tree for the graph.

As mentioned earlier, Dijkstra’s algorithm has the same structure as Prim’s algorithm. The main difference is in the interpretation of the key values — here we call them $d$ (distance) values, since at the end of the algorithm they hold the correct shortest path distances to the source. (In fact, by changing the interpretation of key values, one can modify Dijkstra to compute solutions of other problems as well, e.g., see Worked Exercise 12.4.) In Dijkstra’s algorithm, the distance value of a non-tree node (i.e., belonging to $V - S$) is the shortest distance to the source vertex using only vertices of $S$ (i.e., tree vertices). It is easy to see that at the start of the algorithm the above interpretation of distance values holds since the $d$ values of all nodes is $\infty$, except the source which is 0. Dijkstra’s algorithm efficiently updates the distance values as a vertex is added (one at a time); if a vertex is added to the tree, only the distance values of its neighbors (outside the tree) are affected and these are updated. Dijsktra makes the greedy choice of adding the vertex outside the tree with the smallest distance value (i.e., one that has the shortest distance to the source using only the tree vertices) is the vertex that has to be added next. We show that this greedy choice is correct by showing a proof using mathematical induction.

### Data structures used in Dijkstra’s Algorithm
This is similar to the Prim’s algorithm but, of course, with a different interpretation of key values.

For all $v \in V$, and in each iteration:

- $d[v]$ — the distance from $s$ to $v$ by a path that uses only vertices of $S$.
- $\pi[v]$ — the predecessor of $v$ on the tree.
- $Q$ — min heap that maintains a list of vertices not in $S$.
- $\text{Extract}\_\text{Min}(Q)$ — the vertex with smallest $d[v]$ among the vertices in $Q$.

### Pseudocode of Dijkstra’s Algorithm
**Algorithm** Dijkstra $(G, w, s)$

1. For all $v \in V$ do
12.2. SHORTEST PATHS

(a) \( d[v] = \infty; // \text{the “key” value of } v \text{ is } d[v] \)

(b) \( \pi[v] = NIL; \)

2. \( d[s] = 0; \)

3. \( S = \emptyset; \)

4. \( Q = V; \)

5. While \( Q \neq \emptyset \) do
   (a) \( u = \text{EXTRACT-MIN}(Q); \)
   (b) \( S = S \cup \{u\}; \)
   (c) For all \( v \in \text{Adj}[u] \) do
      i. If \( d[v] > d[u] + w(u, v) \) then
         A. \( \text{DECREASE-KEY}(Q, d[v], d[u] + w(u, v)) // \text{set } d[v] = d[u] + w(u, v); \)
         B. \( \pi[v] = u; \)

Correctness of Dijkstra’s Algorithm

Theorem 12.2. The algorithm computes a correct shortest distance tree when applied to a graph \( G \) with no negative weight edges.

Proof. We’ll show by induction on the size of \( S \) that for every \( 1 \leq k \leq n \), at the end of the while loop with \( |S| = k \), the functions \( \pi[] \) and \( d[] \) satisfy:

1. If \( v \in S \) then \( d[v] \) is the shortest path distance of \( v \) from \( s \), and \( \pi[v] \) encodes the last edge of that path.

2. If \( v \notin S \) then \( d[v] \) is the shortest path of \( s \) from \( v \) using only vertices of \( S \), and \( \pi[v] \) encodes the last edge of that path.

The induction hypothesis holds for \( |S| = 1 \) since in that case \( S = \{s\} \), and for all \( v \) either \( d[v] = \infty \) or is the weight of the edge \( (s, v) \).

Assume that the induction hypothesis holds for \( |S| = j - 1 \). Consider the \( j \)-th iteration of the while loop:

Lemma 12.2. The shortest path from \( s \) to \( u \) uses only vertices of \( S \). Note that \( u \) is the vertex selected in line 5(a) of the algorithm.

Proof. Assume that a shorter path from \( s \) to \( u \) contains a vertex in \( Q \). Let \( v \) be the first such vertex, then \( d[v] < d[u] \) (since all weights are non-negative). Thus, \( v \) would have been selected instead of \( u \) (in line 5(a)) of the \( j \)-th iteration which is a contradiction.
The above lemma, in conjunction with the second part of the induction hypothesis of the $j-1$th iteration implies the truth of the part 1 of the induction hypothesis of the $j$th iteration. This is because, by the hypothesis, at the end of the $j-1$th iteration, $u$ has the correct $d[u]$ distance value to $s$ considering only vertices in $S$. Lemma 1 says that just considering the vertices of $S$ is enough.

Next we prove part 2 of the induction hypothesis.

**Lemma 12.3.** If vertex $v \in Q$ had a correct value $d[v]$ at the beginning of the while iteration (which is guaranteed by the induction hypothesis), it has a correct value at the end of the iteration.

**Proof.** By the lemma’s assumption we need only to check paths from $s$ to $v$ that contain $u$. The algorithm checks only paths in which $v$ is adjacent to $u$. See Figure [12.2.1]. How about the remaining paths?

If there is a shorter path

$$P = s, v_1, v_2, ..., u, v_k, ..., v$$

with $u$ not adjacent to $v$, then since $v_k$ joined $S$ before $u$, there must be a path from $s$ to $v_k$ that does not use $u$ and is not longer. Thus, $d[v]$ has the correct value (by the premise of the Lemma) without considering paths that use $u$.

Thus, part 2 of the induction hypothesis is satisfied.

**Running time**  The running time of Dijkstra is exactly the same as that of Prim.

Hence Dijkstra’s algorithm can be implemented in $O(m \log n + n \log n) = O(m \log n)$ time using a binary heap or $O(m + n \log n)$ using a Fibonacci heap.

**$A^*$ algorithm**

The $A^*$ algorithm is a heuristic variant of the Dijkstra’s algorithm that is especially well-suited for the Single Pair Shortest Paths (SPSP) problem where the goal is, given a weighted directed graph, and a pair of vertices $s$ and $t$, to find a shortest path between $s$ and $t$. This, in principle, can be solved faster than the SSSP problem since instead of finding shortest paths from a source $s$ to all vertices, the goal is to find a shortest path to a single (target) vertex $t$. The $A^*$ algorithm is a good candidate for this
12.2. **SHORTEST PATHS**

![Diagram of shortest path algorithm]

Figure 12.4: Illustration of proof of Dijkstra’s algorithm. \( u \) is the node that has the smallest \( d \) value. Hence it is the one that is extracted from the heap and added to the tree (i.e., made part of \( S \)). The algorithm updates the \( d \) values of only the non-tree neighbors of \( u \) (i.e., part of \( V - S \)).

The shortest path problem and has many real-world applications, for example, finding the shortest route between two places in a map or finding the shortest route for a robot from a given source to a destination. In fact, the \( A^* \) algorithm was invented for robot motion planning.

The only modification of the \( A^* \) algorithm compared to the Dijkstra’s algorithm is simply changing the key value of a (non-tree) node \( v \) to incorporate an estimate of the shortest path distance from \( v \) to the target \( t \). For the \( A^* \) algorithm to work (i.e., find the shortest path from source to target), the estimate should be a lower bound on the true distance from \( v \) to \( t \). Let \( h(v) \) be such an (heuristic) estimate of the true distance from \( v \) to \( t \). This estimate may depend on the underlying graph and other factors. For example, in road networks, such an estimate can be obtained by using the Euclidean distance between two points which will be always be a lower bound on the shortest path distance between the two points.

The key value in the \( A^* \) algorithm is the sum \( d[v] + h[v] \), where \( d[v] \) is the same as in the Dijkstra’s algorithm, the shortest distance from source \( s \) to \( v \) using only vertices in the tree. The rest of the algorithm is the same as before. At every step, the node with the smallest key value is chosen.
and added to the tree. The algorithm ends when the target is added to
the tree. It can be shown that if the function \( h(v) \) is a lower bound on
the true distance between a node to the target (such a function is called
admissible), then the correct shortest path distance to the target is found.
For example, \( h[v] = 0 \) is a trivial lower bound which is simply the Dijkstra's
algorithm.

It is important to note that unlike Dijsktra's algorithm, when a node
(other than \( t \)) is added to the tree, its key value may not be the true estimate
of the shortest distance of the source \( s \) to the node. However, if \( h[v] \)
is admissible, i.e., never overestimates the true distance of \( v \) to target \( t \), it
is guaranteed that when \( t \) is included in the tree, its key value will be the
correct shortest path distance and the path from \( s \) to \( t \) is the shortest path
from \( s \) to \( t \) (Exercise 12.12 asks you to show this). The advantage of \( A^* \) is
that, depending on how good \( h[v] \) estimates the true distance to the target,
substantially less nodes may need to be explored before a shortest path to
\( t \) is found. Dijkstra, on the other hand, may explore more nodes before
it finds the shortest path to \( t \). Recall that, Dijkstra find the shortest paths
in increasing order of distances from \( s \), without considering distance to
target. Since \( A^* \) makes use of additional information in the form of \( h[v] \)
it can find the shortest path to \( t \) quicker than Dijkstra (by exploring less
number of nodes).

**Bellman-Ford Algorithm**

Bellman-Ford algorithm is a dynamic programming based algorithm for
single-source shortest paths. It computes single source shortest paths
even when some edges have negative weight. The algorithm detects if
there are negative cycles reachable from \( s \). If there are no such negative
cycles, it returns the shortest paths. Like Dijkstra, the Bellman-Ford algo-
hythm is also used in the Internet to find short paths.

**DP formulation** As is typical in DP algorithms (see Chapter 6), we first
need to identify the subproblems and then write the recursive formulation.

Let \( w(u, v) \) denote the weight of edge \((u, v)\). Let \( d^k[v] \) to be the weight
of the shortest path between source vertex \( s \) and vertex \( v \) using paths of
up to \( k \) edges. The subproblems are computing \( d^k[v] \) for \( 1 \leq m \leq n - 1 \)
and for all \( v \in V \). If there are no negative weight cycles then no shortest
path has more than \( n - 1 \) edges, i.e., \( k \) can be at most \( n - 1 \). When \( k = 0 \),
we have
\[ d^{(0)}[v] = \begin{cases} 
0 & : v = s \\
\infty & : v \neq s 
\end{cases} \]

Recursive formula:
\[ d^{(k)}[v] = \min_{u \in \text{Neighbor}(v)} \left\{ d^{(k-1)}[u] + w(u, v) \right\} \]

where \( \text{Neighbor}(v) \) is the set of all neighboring vertices of \( v \).

The correctness of the above recursive formula can be shown by induction — we refer to Theorem [12.4]

**The Algorithm** The algorithm has two parts:

- **Part 1:** Computing Shortest Paths Tree: \(|V| - 1\) iterations, iteration \( i \) computes the shortest path from \( s \) using paths of up to \( i \) edges.

- **Part 2:** Checking for Negative Cycles.

**Pseudocode of the Bellman-Ford Algorithm**

**Algorithm** Bellman-Ford
\((G, w, s)\)

1. For all \( v \in V \) do
   (a) \( d[v] = \infty \);
   (b) \( \pi[v] = NIL \);
2. \( d[s] = 0 \);
3. For \( i = 1 \) to \(|V| - 1\) do
   (a) For all \((u, v) \in E\) do
      i. If \( d[v] > d[u] + w(u, v) \) then
         A. \( d[v] = d[u] + w(u, v) \);
         B. \( \pi[v] = u \);
4. For all \((u, v) \in E\) do
   (a) If \( d[v] > d[u] + w(u, v) \) then return FALSE;
5. return TRUE
Figure 12.5: Illustration of proof of Bellman-Ford algorithm. (Top) $P = v_0, v_1, \ldots, v_k$, where $v_0 = s$ and $v_k = u$ is a shortest path from $s$ to $u$. (Bottom) $v_0, \ldots, v_k$ is a negative weight cycle, i.e., a cycle whose sum of weights is negative.

Run Time

**Theorem 12.3.** The run time of the algorithm is $O(mn)$.

**Proof.** The initialization (1) takes $O(n)$. The path creation (3) takes $O(mn)$. The negative cycle detection (4) takes $O(m)$. Hence, overall the run time is $O(mn)$.

Correctness of Bellman-Ford Algorithm

**Theorem 12.4.** Assuming that $G$ contains no negative cycles reachable from $s$, the algorithm computes the shortest paths for all vertices of $G$.

**Proof.** Fix a vertex $u \in V$, we first prove that the algorithm computes the correct (shortest path) distance from $s$ to $u$. The correctness of the shortest path that achieves this distance follows easily.

Let $P = v_0, v_1, \ldots, v_k$, where $v_0 = s$ and $v_k = u$ be a shortest path from $s$ to $u$. See Figure 12.5. Since there are no negative cycles $P$ is a simple path, $k \leq |V| - 1$.

We prove by induction on $i$ that after the $i$-th iteration of the (3) loop, the algorithm computes the shortest path for $v_i$. 
The hypothesis holds for \( v_0 = s \). Assume that it holds for \( j \leq i - 1 \). After the \( i \)-th iteration, we have:

\[
d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)
\]

since \( P \) is a shortest path from \( s \) to \( v_i \), and the RHS is the distance between \( s \) to \( v_i \) on that path, \( d[v_i] \) is the shortest path distance from \( s \) to \( v_i \).

\[\square\]

**Theorem 12.5.** The algorithm returns TRUE if there are no negative cycles reachable from \( s \), otherwise it returns FALSE.

**Proof.** Assume that there are no negative cycles reachable from \( s \), then by the previous theorem, the algorithm returns a shortest path tree, and \( d[v] \) is the weight of the shortest path to \( s \). Thus, all inequalities in 4(a) don't hold.

Assume that there is a negative weight cycle \( v_0, \ldots, v_k \) reachable from \( s \) \((v_0 = v_k)\). See Figure 12.5

Since the path is reachable from \( s \) the values \( d[v_i] \) are defined. \( \sum_{i=1}^{k} d[v_{i-1}] = \sum_{i=1}^{k} d[v_i] \) and

\[
\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0.
\]

Thus,

\[
\sum_{i=1}^{k} d[v_i] > \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)
\]

So there must be an \( i \) such that

\[
d[v_i] > d[v_{i-1}] + w(v_{i-1}, v_i)
\]

and the algorithm returns FALSE. \[\square\]

### 12.2.2 All-Pairs Shortest Paths

The All-Pairs Shortest Paths (APSP) problem is as follows: Given a graph \( G = (V, E) \), \(|V| = n\), and a weight function \( w \) on the edges, compute the shortest paths between all pairs of vertices.

As mentioned earlier in the context of SSSP as well, the problem is not well defined in the presence of a **negative weight cycle** in the graph.
We can solve an all-pairs shortest-paths problem by running a single source shortest-paths algorithm $n$ times, once for each vertex as a source. Then the running time is:

- $O(n|E| \log n)$ if we use the Dijkstra algorithm (assuming no negative edge weights);
- $O(n^2|E|)$ if we use Bellman-Ford algorithm - i.e., $O(n^4)$ if the graph is dense.

Instead of using SSSP, we will give a direct solution to the APSP problem using a dynamic programming-based algorithm known as Floyd-Warshall algorithm that runs in $O(n^3)$ time. Note that using $n$ applications of Dijkstra’s algorithm is more efficient than Floyd-Warshall if the graph is sparse, i.e., if $|E|$ is significantly less than $\Theta(n^2)$, say, for example, when $|E| = \Theta(n)$. However, Dijkstra’s algorithm will apply only if there are no negative edge weights.

**Input and Output**

Input: an adjacency matrix $W$ where $w_{i,j}$ is the weight of the edge $(i, j)$:

$$w_{i,j} = \begin{cases} 
0 & \text{if } i = j \\
\text{weight of } (i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \\
\infty & \text{if } (i,j) \notin E
\end{cases}$$

Output:
A matrix $D$ where $d_{i,j}$ is the shortest path from $i$ to $j$.

**The Floyd-Warshall algorithm**

The Floyd-Warshall algorithm gives a dynamic programming algorithm for computing all-pairs shortest paths in a given directed weighted graph. For a path $P = v_1, v_2, \ldots, v_k$, the vertices $v_2, \ldots, v_{k-1}$ are intermediate vertices. Let the vertices be numbered from 1 to $n$, i.e., $V = \{1, \ldots, n\}$. The main idea of the Floyd-Warshall algorithm is that the algorithm, in iteration $k$, computes all pairs shortest paths with intermediate vertices in $\{1, \ldots, k\}$, i.e., the intermediate vertices are restricted to belong to the set $\{1, \ldots, k\}$. This uses a different recursive formulation compared to the Bellman-Ford algorithm. (Recall that in the Bellman-Ford algorithm, the algorithm, in iteration $k$, computes the shortest paths that use up to $k$ edges).
Let \( d_{i,j}^{(k)} \) be the distance of a shortest path from \( i \) to \( j \) using only intermediate vertices from the set \( \{1, \ldots, k\} \), i.e., using intermediate vertices only from the first \( k \) (numbered) vertices. Thus the subproblems are computing \( d_{i,j}^{(k)} \) for \( 1 \leq k \leq n \) and for every pair of vertices \( i \) and \( j \). Note that the required shortest path distance is \( d_{i,j}^{(n)} \), i.e., all intermediate vertices are allowed without any restriction.

We can write a recursive formulation for \( d_{i,j}^{(k)} \) as follows:

**Lemma 12.4.**

\[
d_{i,j}^{(k)} = \begin{cases} 
    w_{i,j} & \text{if } k = 0 \\
    \min \left[ d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \right] & \text{if } k \geq 1
\end{cases}
\]

**Proof.** Let \( P \) be a shortest path from \( i \) to \( j \) using vertices in \( \{1, \ldots, k\} \). For any two vertices \( i \) and \( j \), if \( k = 0 \), then clearly, \( d_{i,j}^{(0)} = w_{i,j} \) (note that \( w_{i,j} = \infty \), if \( i \) and \( j \) are not connected by an edge). That is, unless vertices \( i \) and \( j \) are connected directly by an edge, there can't be any path which does not use any intermediate vertex. This shows the base case.

We now argue the recursive case, i.e., \( k \geq 1 \), i.e., there is at least one intermediate vertex. There are two cases. If \( P \) does not use \( k \) then \( d_{i,j}^{(k)} = d_{i,j}^{(k-1)} \). Otherwise, \( P \) uses \( k \), i.e., \( P \) consists of a path \( P_1 \) from \( i \) to \( k \), followed by a path \( P_2 \) from \( k \) to \( j \). See Figure 12.6. By the optimal substructure property, since \( P \) is a shortest path from \( i \) to \( j \), \( P_1 \) is a shortest path from \( i \) to \( k \) using only intermediate vertices in \( \{1, \ldots, k-1\} \) and \( P_2 \) is a shortest path from \( k \) to \( j \) using only intermediate vertices in \( \{1, \ldots, k-1\} \). Note that vertex \( k \) cannot occur (as an intermediate vertex) in \( P_1 \) and \( P_2 \); otherwise \( k \) will occur twice in path \( P \), which makes \( P \) not optimal (there is a cycle in \( P \) if \( k \) occurs twice). The distance of \( P_1 \) is \( d_{i,k}^{(k-1)} \) and the distance of \( P_2 \) is \( d_{k,j}^{(k-1)} \) by definition. Hence, in this case, \( d_{i,j}^{(k)} = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \). We take the minimum of the two cases. \( \square \)

**Run time and Implementation** Floyd-Warshall algorithm can be implemented in \( O(n^3) \) time. This follows from the dynamic programming formulation. The total number of subproblems is \( O(n^3) \) since, for for every pair of vertices \( i \) and \( j \), there are \( n \) subproblems, \( 1 \leq k \leq n \). The time needed to solve a subproblem is constant, since it involves taking minimum of two values. Hence overall time is \( O(n^3) \).

As usual, the algorithm can be implemented recursively using memoization or iteratively using dynamic programming. The latter is usually
Figure 12.6: Illustration of proof of Floyd-Warshall algorithm. $P$ is a shortest path from $i$ to $j$ going through vertex $k$. It consists of a path $P_1$ from $i$ to $k$, followed by a path $P_2$ from $k$ to $j$. Since $P$ is a shortest path from $i$ to $j$, $P_1$ is a shortest path from $i$ to $k$ using only intermediate vertices in $\{1, ..., k-1\}$ and $P_2$ is a shortest path from $k$ to $j$ using only intermediate vertices in $\{1, ..., k-1\}$. 
the choice, since it avoids the overhead of recursion. The pseudocode for
the dynamic programming algorithm is given in Algorithm 42. It simply
uses three for loops to compute \( d_{i,j}^{(k)} \) for all \( 1 \leq k, i, j \leq n \). As mentioned
earlier, the pseudocode assumes that the graph is input as an adjacency
matrix which is convenient for the algorithm. Then adjacency matrix \( W \)
represents matrix \( D^{(0)} \) which is the matrix of distances in the base case,
i.e., \( d_{i,j}^{(0)} = w_{i,j} \). Subsequent iterations compute \( D^{(k)} \) which represents the
matrix of distances in iteration \( k \), i.e., \( d_{i,j}^{(k)} \). The pseudocode computes
only the shortest distance, but not the paths; the algorithm can be easily
modified to do that as well.

The pseudocode given in Algorithm 42 uses \( n \) matrices and hence uses
\( O(n^3) \) space. However, since the matrix \( D^{(k)} \) depends only on matrix \( D^{(k-1)} \)
(see the recursive formulation), we can easily modify the pseudocode to
use only two matrices (see Exercise 12.13). This saves space and thus
the total space needed is only the space required for 2 matrices which is
\( O(n^2) \).

Algorithm 42 Algorithm Floyd-Warshall(\( W \)). Input: A directed weighted
graph \( G = (V,E) \) represented by its adjacency matrix \( W \). Output: The
all-pairs shortest distances represented as matrix \( D^{(n)} \).

1: \( D^{(0)} = W \)
2: for \( k = 1 \) to \( n \) do
3: \hspace{1em} Matrix \( D^{(k)} \) represents the distances \( d_{i,j}^{(k)} \)
4: for \( i = 1 \) to \( n \) do
5: \hspace{2em} for \( j = 1 \) to \( n \) do
6: \hspace{3em} \( d_{i,j}^{(k)} = \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}) \)
7: Output \( D^{(n)} \)

12.3. WORKED EXERCISES

Worked Exercise 12.1. A cut \((S, V - S)\) of an undirected graph \( G = (V, E) \)
is a partition of \( V \). An edge crosses the cut if one of its endpoints is in \( S \) and
the other is in \( V - S \).

The cut property of MST can be stated as follows: For any proper nonempty
subset \( X \) of the vertices, the lightest edge with exactly one endpoint in \( X \) be-
longs to a minimum spanning tree. In other words, we have the following
theorem.

Theorem 12.6 (cut property). Let \( G = (V, E, w) \) be a connected undirected
graph with weight function \( w \) defined on \( E \), i.e., \( w(e) \) gives the weight of
edge e. Let \((S, V - S)\) be any cut of \(G\) and let \((u, v)\) be a minimum weight edge crossing the cut. Then \((u, v)\) belongs to a minimum spanning tree of \(G\).

Prove the cut property.
Solution:
Proof of the cut property:
Let \(T\) be a MST of \(G\). If \(T\) contains \((u, v)\), we are done. Otherwise, we will construct another MST \(T'\) that includes edge \((u, v)\) which will prove the theorem.

Since \((u, v)\) does not belong to \(T\), adding edge \((u, v)\) to \(T\) completes a cycle with the edges on the (unique) path (call it \(p\)) from \(u\) to \(v\) in \(T\).

Since \(T\) is a spanning tree, there is at least one edge \((x, y)\) in \(p\) that crosses the cut \((S, V - S)\), since \(u\) and \(v\) are on different sides of the cut. Note that since \(w(u, v)\) is the lightest edge crossing the cut, \(w(u, v) \leq w(x, y)\).

Remove \((x, y)\) from \(T\) and add \((u, v)\) to \(T\) to get a new spanning tree \(T'\). The weight of \(T'\) is:

\[w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)\]

Thus \(T'\) is also a MST. Thus there is a MST \(T'\) that contains edge \((u, v)\).

In a MST algorithm, the cut property validates the addition of the edge \((u, v)\) and the cut that validates is the cut \((S, V - S)\) that is mentioned in the theorem.

Worked Exercise 12.2. The algorithm of this problem is called Boruvka’s algorithm, the first algorithm proposed for MST (in 1926).

We are given a connected undirected graph \(G = (V, E)\) with positive edge weights. Assume that all the edge weights are distinct. The operation of contracting an edge \(e = (u, v)\) in \(G\) involves the following operations: (1) remove the vertices \(u\) and \(v\) and introduce a new vertex \(x\); (2) for each edge \(e'\) having an endpoint in either \(u\) or \(v\) do: replace the endpoint \(u\) or \(v\) by \(x\), i.e., if \(e' = (w, u)\) (or \(e' = (w, v)\)) it is replaced by the edge \(e' = (w, x)\). (For this problem, we can assume that each edge comes with a distinct label. When we contract edges, we use the same labels on the edges, as they were before contraction, for easy identification. For example, in the above example, the edge label for \(e'\) remains the same after contraction as well, despite the endpoints changing.)

Consider the following algorithm for finding a minimum spanning tree:

1. **repeat** until \(G\) has one vertex
2. **for each vertex** \(v \in G\) do
3. Select the minimum-weight edge incident on $v$.
4. Contract all the selected edges chosen in Step 3.
5. Eliminate self-loops.
6. Eliminate all but the lowest-weight edge among each set of multiple edges (i.e., edges that go across the same pair of vertices).
7. endrepeat
8. Output all the edges selected in Step 3 as the MST edges.

a) Show that the minimum spanning tree of $G$ is unique if all the edge weights are distinct. (Hint: One way to show this is proof by contradiction: assume two distinct minimum spanning trees and get a contradiction.)

b) Argue that the above algorithm correctly outputs the edges of the minimum spanning tree. (Hint: Use the cut property of Exercise 12.7)

c) Give a tight (up to a constant factor) upper bound on the number of iterations (one iteration is execution of Steps 2-6) of the algorithm.

d) Show how to implement one iteration of the algorithm in $O(|V| + |E|)$ time. (Hint: Use depth-first search.) Using your answer to part (d), what is the (overall) running time of the algorithm?

e) Suppose the edge weights are not distinct. Will the algorithm still work? If your answer is "no", then show how to fix it.

Solution:

(a) Assume that there are two minimum spanning trees (MSTs), $T$ and $T'$. Let $e$ be the minimum weight edge that is included in only one MST. WLOG, let $e \in T$ and $e \notin T'$. Then $T' \cup \{e\}$ has a cycle.

The cycle has an edge $e' \notin T$ because if not, $T$ has a cycle. And the weight of $e'$ is greater than that of $e$ because all weights are distinct and by the choice of $e$.

Thus $T' \cup \{e\} - \{e'\}$ is a spanning tree with weight smaller than $T'$, that leads to a contradiction.

(b) Let in some iteration $i$, we are selecting the minimum-weight edge, say $e$, incident on $w$. $w$ can be a composite ("super") node (meaning a collection of nodes due to contraction). Let $l(w)$ be the set of nodes in $G$ that are collected into $w$. $e$ is the minimum-weight edge across the cut $(l(w), V - l(w))$ of $G$. Thus, by the cut property, $e$ must be an edge of some
MST. Since the MST is unique, the algorithm only picks the edges of the same MST.

Contraction of an edge reduces the number of nodes by 1. Thus there will be exactly a total of \( n - 1 \) contractions, since finally we are left with 1 vertex. Each contraction adds an edge to the MST. Thus by the end of the last iteration, exactly \( n - 1 \) edges are selected.

(c) In each iteration, at least half of the vertices are removed. This is because, if there are \( k \) vertices at the beginning of the iteration, then at least \( k/2 \) edges are selected (it can happen that vertex \( u \) selects edge \((u, v)\) and \( v \) selects \((v, u)\)). Each selected edge leads to removal of one vertex. After the first iteration, for example, the number of vertices is at most \( |V|/2 \). Thus, after \( k \)-th iteration, the number of vertices is at most \( |V|/2^k \). And the iterations are executed until \( G \) has one vertex. If \( t \) is the maximum number of iterations, \( 1 \leq \frac{|V|}{2^t} \), i.e., \( t \leq \log_2 |V| \).

(d) We will next show how to implement one iteration in \( O(|E|) \) time, thus giving a total time of \( O(|E| \log |V|) \).

Note that since the given graph is connected \( O(|V| + |E|) = O(|E|) \).

Implement the graph \( G \) using adjacency (adj) list. Represent each vertex by a unique number.

Search the adjacency list to find minimum-weight edge incident on each node (or “super” node) \( v \). This takes \( 2|E| \) time for all nodes.

Using BFS find the connected components of the graph defined by the current nodes and the edges selected in this iteration. Call each component a fragment — this will be treated as a “super” vertex for the next iteration. Let the root of the BFS tree of a fragment be numbered \( r \). Simply propagate this value \( r \) to the all nodes in the fragment (can be done during BFS itself — takes \( O(|E|) \) time). Thus each node in the fragment will be renumbered with the value \( r \). This will be done to every fragment. Thus at the end of this operation each node will know the fragment number of its fragment. Finally, the adjacency list of the fragment will be updated: if there is an edge connecting a node in this fragment to a different fragment then this will be added to the adjacency list of the fragment. This will take totally \( O(|E|) \) time, since only so many edges will be examined. Note that if there are two edges connecting the same two fragments then this can be eliminated (taking care of multiple edges).
The algorithm will not work if the edge weights are not distinct. For example, imagine a triangle of 3 nodes with 3 edges of weight 1 each. It is possible that the algorithm will choose all 3 edges to contract in the first step.

To fix the problem, we create a total ordering of the edge weights by breaking ties between same weight edges based on the labels of the incident nodes (note that each node has a unique label or number). If a node has more than one minimum-weight edge incident on it, select the edge that has the lowest numbered vertex at the other end.

**Worked Exercise 12.3.** Let \( G = (V, E, w) \) be a directed, weighted graph, where \( w(e) \) denotes the weight of edge \( e \). The weights can be negative, but \( G \) contains no negative cycles. Let \( s \) be a distinguished vertex in \( V \). The goal is to compute the shortest paths from \( s \) to all vertices.

(i) Given \( G \) as input, show that Dijkstra’s algorithm does not always generate the correct solution.

(ii) Consider the following algorithm that modifies the weights of \( G \) before running Dijkstra.

Dijkstra-new(\( G \)):

1. From \( G \), generate a new graph \( G' = (V, E, w') \) having the same node and edge set and weights are set as follows. Let \( W = \max_{e \in E} \{|w(e)|\} \), i.e., \( W \) is the largest weight in \( G \) (in absolute value). Set \( w'(e) = w(e) + W \) for every edge in \( E \).

2. Run Dijkstra’s algorithm on \( G' \).

3. Output the shortest paths obtained from \( s \) to every other vertex as the corresponding shortest paths for \( G \).

Does the above algorithm always generate the shortest paths for \( G \)? Note we want to generate the paths, not the value of the shortest paths. Justify your answer.

**Solution:**

Dijkstra’s algorithm for negative edge graph

(i) Consider the graph in figure 12.7(a). The Dijkstra’s algorithm solution and correct solution with the source vertex \( a \) would be:

It can be seen that Dijkstra’s algorithm fails to generate the correct solution in this example.
(ii) Consider the same graph in part (i), the graph \( G' \) generated is shown in figure 12.7(b). Then the solution of Dijkstra's algorithm on graph \( G' \) would be

\[
\begin{align*}
  &b \rightarrow (a, b) \\
  &c \rightarrow (a, d) \\
  &d \rightarrow (a, d)
\end{align*}
\]

It could be observed that this solution is not the correct solution for graph \( G \) provided in part (i). Thus, Dijkstra-new does not generate the shortest paths for \( G \).

**Worked Exercise 12.4.** A communications provider has a number of computers that are connected using a network. Assume that the network is an undirected graph \( G = (V, E) \) where the vertices represent the computers and the edges represent the links between them. Each edge \((u, v) \in E\) has an associated weight \( b(u, v) \) (a positive number) which represents the bandwidth capacity of that edge. Given a path between two nodes, the bottleneck value of the path is the minimum of the weights of the edges in the path. The maximum bottleneck path between two nodes is the path that has the maximum bottleneck value among all paths between them.

(a) In the graph of Figure 12.2, compute the respective maximum bottleneck paths between node \( a \) and all other nodes.

(b) Develop an algorithm that given two nodes \( a \) and \( b \) computes the maximum bottleneck value path from \( a \) to \( b \). What is the running time of
your algorithm. (Hint: Modify Dijkstra’s algorithm to solve this problem.)

Solution:
(a) The following are the maximum bottleneck paths between a and all other nodes:
1. $a \rightarrow b$: $a \rightarrow d \rightarrow b$
2. $a \rightarrow c$: $a \rightarrow d \rightarrow c$
3. $a \rightarrow d$: $a \rightarrow d$
4. $a \rightarrow e$: $a \rightarrow d \rightarrow f \rightarrow e$
5. $a \rightarrow f$: $a \rightarrow d \rightarrow f$
6. $a \rightarrow g$: $a \rightarrow b \rightarrow g$

(b) Although this problem appears to be very different from the shortest paths problem, there is a uniform way of looking at both the problems. In the shortest path problem, the “value” of a path is the sum of weights in the path, and we want to find (among all possible paths) the one with minimum value. In the maximum bottleneck problem, the “value” of a path is the minimum weight edge of that path and we want to find the path with the maximum value. Thus, the definition of “value” differs, and instead of finding a path of minimum value (as in shortest paths), we are finding a path with maximum value.

A modification of Dijkstra’s algorithm can be used to compute the maximum bottleneck value path between $a$ (the source) and all other vertices. In particular, this will yield the maximum bottleneck value path from $a$ to $b$.

Let
\[ b(u, v) \] — the weight (bandwidth) of edge $(u, v) \in E$.
\[ key[v] \] — the value of the maximum bottleneck path from $s$ to $v$.
\[ parent[v] \] — the predecessor of $v$ in the path.

**Algorithm Dijkstra(G, w, s)**

1. For all $v \in V$ do
   (a) $key[v] = 0$
   (b) $parent[v] = NIL$
2. $key[s] = \infty$
3. $S = \emptyset$
4. $Q = V$
5. While $Q \neq \emptyset$ do
(a) \( u = \text{Extract Max}(Q) \)
(b) \( S = S \cup \{u\} \)
(c) For all \( v \in \text{Adj}[u] \) do
   i. if \( \text{key}[v] < \text{Min}(\text{key}[u], b(u, v)) \) then
      A. \( \text{key}[v] = \text{Min}(\text{key}[u], b(u, v)) \) // increase-key of \( v \)
      B. \( \text{parent}[v] = u \)

Note that we can use a binary max-heap to implement the heap operations — extract-max and increase-key.

**Worked Exercise 12.5.** A vertex-cover of a tree \( T \) is a subset \( V' \) of its vertices such that, for each edge in \( T \), at least one of its endpoints belongs to \( V' \). For example, the set containing all the vertices of \( T \) is a vertex-cover. The minimum vertex-cover of \( T \) is a vertex-cover that contains the fewest vertices.

We are given a rooted tree \( T \) of \( n \) vertices. (If you want you can assume the tree to be binary, i.e., each node has at most two children.) Design a dynamic programming algorithm to output the size of the minimum vertex-cover of \( T \), and analyze its running time. Note that you should first give a recursive formulation of the dynamic programming solution, clearly explaining your notation. Then give the pseudocode and analyze its running time. (Hint: Each vertex can either be in the vertex cover or not; this will be useful in setting up your subproblems.)

Solution:

Let \( v \) be any node in the tree \( T \). We will recursively compute the size of the minimum vertex cover of the subtree rooted at \( v \) — call this \( \text{VC}(v) \). If root of \( T \) is \( r \), then \( \text{VC}(r) \) is the answer.

To compute \( \text{VC}(v) \) we define two subproblems: \( \text{VCin}(v) \) and \( \text{VCout}(v) \) — these denote the size of the minimum vertex cover of the subtree rooted at \( v \) that includes \( v \) and the size of the minimum vertex cover of the subtree rooted at \( v \) that excludes \( v \), respectively (there are only two possibilities). Note that for each edge, at least one of its endpoints should be in the vertex cover to cover the edge.

Note that \( \text{VC}(v) = \min\{\text{VCin}(v), \text{VCout}(v)\} \).

Now we can write the recursive formulation to compute \( \text{VCin}(v) \) and \( \text{VCout}(v) \).

Trees (as opposed to general graphs) naturally admit a recursive formulation: to compute a quantity for a tree rooted at a node \( v \), we recursively compute the quantity at each of its subtrees (rooted at its respective
children) and then combine the solution. Since the subtrees don’t have any edges between them (since it is a tree), this usually works.

If \( v \) is a leaf (i.e., it has no children), then:
\[
V_{C_{in}}(v) = 1 \text{ and } V_{C_{out}}(v) = 0.
\]

If \( v \) has \( k \) children (say, \( v_1, v_2, \ldots, v_k \)) then:
\[
V_{C_{in}}(v) = 1 + V_{C}(v_1) + V_{C}(v_2) + \cdots + V_{C}(v_k).
\]

The above says that if \( v \) is included in the vertex cover, then each of its children may or may not be included.

\[
V_{C_{out}}(v) = V_{C_{in}}(v_1) + V_{C_{in}}(v_2) + \cdots + V_{C_{in}}(v_k).
\]

The above says that if \( v \) is excluded in the vertex cover, then each of its children has to be included to cover the edge between \( v \) and its children.

It is straightforward to write the pseudocode from the above formulation. For example, the bottom-up solution will compute \( V_{C}(v) \) starting from the leaves (the base case) and going up the tree.

The number of subproblems is \( 2^n \) — two corresponding to each vertex of the tree. The total time to compute a subproblem \( V_{C}(v) \) is proportional to the number of children of \( v \) which is bounded by the degree of \( v \). Thus, the total time is proportional to the total degree of all nodes which is \( O(n) \) (since it is a tree, there are only \( n-1 \) edges). Hence, the running time is \( O(n) \).

### 12.4 Exercises

**Exercise 12.1.** Consider the weighted graph in Figure 12.1 (right) in Appendix 2. Run the following MST algorithms on this graph. Show the edges added step by step in each algorithm. Also, mention the cut that validates the addition of each edge for Prim’s algorithm.

(a) Kruskal’s algorithm

(b) Prim’s algorithm

(c) Boruvka’s algorithm (see Worked Exercise 12.2).

**Exercise 12.2.** Give an example of a complete graph on five nodes — a, b, c, d, e — (i.e., all pairs of edges between the 5 nodes are present) with suitable positive edge weights (all unique) so that:

1. the MST of the graph is the same as the SPT (shortest path tree) rooted at a.
(2) the MST of the graph is different from the SPT rooted at a.

Show how Prim’s algorithm and Dijkstra’s algorithm compute the MST and SPT respectively on the two examples in parts (1) and (2).

Exercise 12.3. Give an efficient algorithm to find the length (number of edges) of a minimum length negative-weight cycle in a graph.

Exercise 12.4. Let \( G = (V, E) \) be a weighted, directed graph with positive integer weights (all weights greater than 0); and the maximum weight is \( W \).

(a) Modify Dijkstra’s algorithm to compute the shortest paths from a given source vertex \( s \) in \( O(W |V| + |E|) \) time.

(b) Give an \( O(|E| |\lg W|) \) time algorithm for the problem by improving your algorithm of part (a).

Exercise 12.5. Give an \( O(|V||E|) \)-time algorithm for computing the transitive closure of a directed graph \( G = (V, E) \).

Exercise 12.6. You are given an undirected weighted graph \( G = (V, E) \), where each node \( v \in V \) is assigned an unique number called rank. Ranks are given as part of the input and are integers chosen from the interval \([1, n]\), where \( n \) is the number of nodes in the graph (thus, the rank of a node can be simply taken to be its label, which is an integer from \([1, n]\)). Note that the minimum ranked node in the graph has rank 1 and the maximum ranked node has rank \( n \). All weights are positive. For each node \( v \), we would like to compute the minimum ranked node within every distance \( d \). We want to represent this information as a list of pairs called as \( L(v) \) for each node \( v \), defined as follows: \( (a_v(i), u_v(i)) \) \((1 \leq i \leq |L(v)|)\), where \( |L(v)| \) is the number of pairs in the list \( L(v) \), where \( a_v(i) \) is a rank and \( u_v(i) \) is a distance, such that:

(a) the list of ranks is in decreasing order, i.e., \( a_v(1) > \cdots > a_v(|L(v)|) = 1 \); and the list of corresponding distances is in increasing order, i.e., \( u_v(1) < \cdots < u_v(|L(v)|) = \infty \).

(b) for all \( 1 \leq i \leq |L(v)| \), \( a_v(i) \) is the minimum ranked node strictly within distance \( u_v(i) \) of node \( v \) (i.e., the set of all nodes within a distance \( d \) of \( v \), for every \( d < u_v(i) \)).

It follows from the above definition that, for all \( v \), the first rank in the list \( L(v) \) will be \( a_v(1) = \text{rank}(v) \) and the last rank will be 1.

(i) Consider a line graph on \( n \) nodes, i.e., the graph is just a path of length \( n - 1 \). Let all edge weights be 1. Let the nodes be numbered from 1 to \( n \), left to right. Let rank of node \( k \) be \( k \). Give \( L(v) \) for all \( v \).
(ii) In an arbitrary graph \( G \), give an efficient algorithm to compute \( L(v) \) for all \( v \). Analyze the running time. (Hint: Modify Dijkstra).

**Exercise 12.7.** Given a weighted (connected) undirected graph \( G = (V, E, w) \), where \( w(u, v) \) is the weight of edge \((u, v)\). Assume that all weights are positive. The goal is to partition the vertex set \( V \) into \( k \) disjoint non-empty subsets \( C_1, C_2, \ldots, C_k \) (i.e., for all \( 1 \leq i < j \leq k \), \( C_i \cap C_j = \phi \), and \( C_1 \cup C_2 \cdots \cup C_k = V \)) such that the separation between the subsets is maximized. The separation is defined as the minimum distance between any pair of vertices lying in different subsets. (The distance \( d_{ij} \) between two subsets \( C_i \) and \( C_j \) is defined as \( d_{ij} = \min_{u \in C_i, v \in C_j} w(u, v) \) and separation \( d = \min_{1 \leq i < j \leq k} d_{ij} \).) Give an efficient algorithm that outputs the \( k \) disjoint non-empty subsets. Prove the correctness of your algorithm and analyze its running time. (Hint: An MST algorithm is useful).

**Exercise 12.8.** Given a weighted (connected) undirected graph \( G = (V, E, w) \), with \( m = n + O(1) \) edges (i.e., the number of edges is at most the number of vertices plus some constant). Give an \( O(n) \)-time algorithm to compute the minimum spanning tree of \( G \). Prove the correctness and the running time of your algorithm. (Hint: Think of how one can eliminate an edge from being in the MST.)

**Exercise 12.9.** Dijkstra’s single-source shortest-paths (SSSP) algorithm on a directed graph uses a set \( S \) that initially contains only the source \( s \) and that eventually includes all the vertices of the graph. Vertices are added to \( S \) one at a time. Let \( N(v) \) denote the number of times that the \( d[v] \) value of a vertex \( v \) in \( V − S \) changes due to an update (line 5(c)A of the Dijkstra’s algorithm pseudocode). Answer each of the following questions (provide a brief justification for each of your answers).

- Can the \( d[v] \) value of a vertex \( v \) in \( V − S \) ever get smaller than the cost of a shortest \( s \)-to-\( v \) path in the graph?
- Can \( N(v) \) exceed the in-degree of \( v \)? (Recall that the in-degree of a vertex is the number of edges going into it.)
- Can \( N(v) \) be less than the in-degree of \( v \)?
- If \((v, w)\) is the lowest weighted edge in the graph, is it true that \( v \) must be added to \( S \) before \( w \) is added to \( S \)?
- Let the vertex \( v \) have the \( i \)th longest shortest-path from \( s \) to it, and assume that there is no tie for that \( i \)th rank (i.e., no vertex other than
has that particular length of a shortest path from \( s \) to it. Suppose you are told that, at a given point during the execution of Dijkstra's algorithm, the size of \( S \) is \( j \). Is this information sufficient to determine whether, at that point, \( v \) is in \( S \) or in \( V - S \)? Why?

**Exercise 12.10.** We are given a directed graph \( G = (V, E) \) with \( n \) vertices, labelled \( 1, 2, \ldots, n \). All (directed) edges in \( G \) are of the form \((i, j)\) where \( i < j \) (i.e., an edge is always directed from a lower labelled vertex to a higher labelled vertex). Also, every vertex except the vertex labelled \( n \) has at least one edge leaving it, i.e., for every vertex \( i \in \{1, \ldots, n-1\} \), there is at least one edge of the form \((i, j)\) (\( j > i \)). The length of a path is the number of edges in it. Give a dynamic programming algorithm to find the longest (and its length) that begins at vertex labelled \( 1 \) and ends at vertex labelled \( n \). Prove the correctness and analyze your running time. (As usual give the recursive formulation, clearly stating your notation).

**Exercise 12.11.** Give an \( O(mn) \) time algorithm to output the vertices of a negative weight cycle (if one exists) of a given graph \( G(V, E) \).

**Exercise 12.12.** Show that in the \( A^* \) algorithm, if the heuristic function \( h \) that estimates the distance of a node to the target is admissible (i.e., it never overestimates the true shortest path distance), then \( A^* \) is guaranteed to find the shortest path from the source to the target.

**Exercise 12.13.** Give an implementation of Floyd-Warshall algorithm that uses \( O(n^2) \) space and runs in \( O(n^3) \) time.
Graph Algorithms: Maximum Flow and Minimum Cut

This chapter focuses on network flow problems which are fundamental graph optimization problems with a lot of applications. Network flow is also related to a number of other fundamental graph problems such as minimum cut and matching. In a network flow problem, we are given a graph with weighted edges where the weights stand for capacities of the edges. For example, one can think of a network of pipes with each pipe having a capacity. The goal is to route flow (say water or oil) from one or more source nodes to one or more destination nodes (called sinks) obeying the capacity constraints. Depending on the setting, we have different types of flow problems. Such problems arise in a wide variety of applications such as routing, transportation, reliability etc.

13.1 Flow Network

A flow network is a directed graph $G = (V, E)$, and a capacity function

$$c : V \times V \rightarrow R$$

such that:

1. For each $(u, v) \in E$ the capacity $c(u, v) \geq 0$.

2. If $(u, v) \not\in E$ then $c(u, v) = 0$.

3. There is a source $s$ and a sink $t$.

4. Each vertex is on a directed path from $s$ to $t$. 
We will assume without loss of generality, that in a flow network, if \((u, v) \in E\), then \((v, u) \notin E\). Exercise 13.1 asks you to show why this assumption is without loss of generality.

**Flow** A flow in a flow network \(G\) with capacity function \(c\) is a function \(f : V \times V \rightarrow \mathbb{R}\) such that:

1. For all \((u, v) \in E\) (capacity constraint)
   \[ f(u, v) \leq c(u, v) \]

2. For all \(u, v \in V\) (symmetry)
   \[ f(u, v) = -f(v, u) \]

3. For all \(u \in V - \{s, t\}\) (flow conservation)
   \[ \sum_{v \in V} f(u, v) = 0 \]

The **value** of the flow is
\[ |f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t) \]

**Flow Problems** There are many fundamental problems associated with network flow:

- **Maximum flow**: Given a flow network \(G\) with source \(s\) and sink \(t\), find the maximum flow.

- **Minimum-cost flow**: Ship \(d\) units from \(s\) to \(t\) such that the total cost is minimized (each edge charges a certain amount for a unit of flow through it).

- **Multi-commodity flow**: ship \(k\) commodities, each with its own sink, source, and flow value (demand).

In this chapter, we will focus on the maximum flow problem, perhaps the most fundamental of the flow problems.

**Exercise 13.1.** Show we can assume, without loss of generality, that in a flow network, if \((u, v) \in E\), then \((v, u) \notin E\). In particular, show that this assumption does not affect the solvability of the max flow problem in a flow network (which may have edges such as \((u, v)\) and \((v, u)\)).
13.2 Maximum Flow: The Ford-Fulkerson Method

Given a flow network, our goal is to design an efficient algorithm that finds the maximum flow. A very natural algorithm for such a flow is due to Ford and Fulkerson, which is essentially a greedy algorithm. The algorithm consists of phases. In every phase, it pushes more flow, until no more flow is possible, in which case the algorithm terminates. Ford-Fulkerson showed that when the algorithm terminates, the total flow pushed across all the phases constitutes a maximum flow.

Ford Fulkerson Approach: Pseudocode

Ford Fulkerson Method\( (G, c, s, t) \)

1. Initialize flow to 0;

2. While “possible”

   (a) Improve flow (push more flow from \( s \) to \( t \));

Residual Network

Given a flow \( f \) on a flow network \( G \) the residual capacity of an edge \( (u, v) \), \( c_f(u, v) \), is the additional amount of flow that can be send from \( u \) to \( v \) before exceeding the capacity \( c(u, v) \), i.e.,

\[
c_f(u, v) = c(u, v) - f(u, v).
\]

We note that flow is defined on pairs of vertices and not edges. We will assume that edges with zero residual capacity will be removed from the network (since no flow can be pushed through these edges).

Given a flow \( f \) on a flow network \( G \) the residual network \( G_f \) of \( G \) induced by \( f \) is a flow graph on \( G \) with flow capacities \( c_f(u, v) \).

It is important to note that the residual graph might have edges that are not in \( G \), but if \( (u, v) \) is an edge either \( (u, v) \) or \( (v, u) \) is an edge of \( G \).

The main idea of having the “reverse” direction edges is to allow the algorithm to push flow “back” — this has the effect of cancelling the flow that was pushed in the forward direction. Thus if there is a flow of 10 from \( u \) to \( v \), pushing a flow of 3 from \( v \) to \( u \) is equivalent to pushing a (net) flow of 7 from \( u \) to \( v \). As we will see, this is very useful in designing algorithms for network flow.

Figure 13.1 shows a flow network and a residual graph.
Figure 13.1: Figure A shows a flow network. Figure B shows the residual network after routing a flow of 1: $s \rightarrow a \rightarrow b \rightarrow t$. Note how new edges $(a, s), (b, a), (t, b)$ appear in the residual graph. For example, edge $(b, a)$ appears with a capacity of 1, because originally $(b, a)$ had capacity 0 and we push a flow of -1 from $b$ to $a$ (i.e., a flow of 1 from $a$ to $b$); hence the residual capacity of this edge is $0 - (-1) = 1$. Figure C shows the residual network after routing a flow of 1: $s \rightarrow b \rightarrow a \rightarrow t$. Figures D and E show residual graphs after routing a flow of 99 respectively. There is no directed (augmenting) path from $s$ to $t$ in the network of Figure E. Hence no more flow can be pushed. The total flow pushed overall is 200 which is the maximum flow.
13.2. MAXIMUM FLOW: THE FORD-FULKERSON METHOD

Improving Flow

We show that if one can push additional flow in a residual graph, then this flow can be combined with an already existing flow.

Let $f$ be a flow function on $G$. Let $f'$ be a flow function on the residual graph of $G_f$ induced by $f$. Define the “flow” function $f + f'$ as

$$(f + f')(u, v) = f(u, v) + f'(u, v).$$

**Lemma 13.1.** The function $f + f'$ is a flow function in $G$, with flow value $|f + f'| = |f| + |f'|$.

**Proof.**

1. For all $(u, v) \in E$,

$$(f + f')(u, v) = f(u, v) + f'(u, v) \leq f(u, v) + (c(u, v) - f(u, v)) = c(u, v)$$

2. For all $u, v \in V$,

$$(f + f')(u, v) = f(u, v) + f'(u, v) = -f(v, u) - f'(v, u) = -(f + f')(v, u)$$

3. For all $u \in V - \{s, t\}$,

$$\sum_{v \in V} (f + f')(u, v) = \sum_{v \in V} (f(u, v) + f'(u, v)) = \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) = 0 + 0$$

$$|f + f'| = \sum_{v \in V} (f + f')(s, v) = \sum_{v \in V} (f(s, v) + f'(s, v)) = \sum_{v \in V} f(s, v) + \sum_{v \in V} f'(s, v) = |f| + |f'|$$
Augmenting Path  Given a graph \( G = (V, E) \) and a flow \( f \) in \( G \), an augmenting path \( p \) is a simple (directed) path connecting \( s \) to \( t \) in the residual graph \( G_f \). Let 
\[
    c_f(p) = \min\{c_f(u, v) \mid (u, v) \in P\}.
\]

Lemma 13.2. Let 
\[
    f_p(u, v) = \begin{cases} 
        c_f(p) & (u, v) \in p \\
        -c_f(p) & (v, u) \in p \\
        0 & \text{otherwise.}
    \end{cases}
\]

Then \( f_p \) is a flow function in \( G_f \) with value \( |f_p| = c_f(p) > 0 \).

Theorem 13.1. If there is an augmenting path \( p \) in \( G_f \) then then there is a flow \( |f + f_p| > |f| \) in \( G \).

Now we are ready the state the Ford-Fulkerson algorithm in detail.

Ford-Fulkerson Method \((G, c, s, t)\)

1. Initialize \( f(u, v) = 0 \) for every edge in \( E \);
2. While there exists an augmenting path \( p \) in \( G_f \)
   
   (a) Augment the flow \( f \) with \( f_p \);

Figure [13.1] shows applying the Ford-Fulkerson algorithm for four iterations (Figures B, C, D, and E), each iteration increasing the flow pushed till we reach a residual graph (Figure E) where is no more augmenting path.

The proof of correctness of Ford-Fulkerson follows from a fundamental duality theorem called the Max-Flow Min-Cut Theorem.

13.2.1 The Max-Flow Min-Cut Theorem

We need the following definitions to state the theorem.

An \((S, T)\)-cut in a flow network \( G = (V, E) \) is a partition of \( V \) to \( S \) and \( T = V - S \) such that \( s \in S \) and \( t \in T \). Given a flow \( f \) in \( G \), the net flow across a cut \((S, T)\) is
\[
    f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v).
\]
The capacity of the cut is
\[ c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v). \]

**Theorem 13.2** (Max-flow Min-Cut Theorem). Maximum \(s-t\) flow in \(G\) equals the minimum \(c(S, T)\) over all \((S, T)\) cuts.

We first prove the following lemma which will imply that any flow cannot be larger than the minimum cut capacity.

**Lemma 13.3.** Let \(f\) be a flow in \(G\). For any \((S, T)\) cut
\[ f(S, T) = |f|. \]

**Proof.**
\[
\begin{align*}
  f(S, T) & = f(S, V) - f(S, S) & \text{(13.1)} \\
  & = f(S, V) & \text{(13.2)} \\
  & = f(s, V) + f(S - s, V) & \text{(13.3)} \\
  & = f(s, V) & \text{(13.4)} \\
  & = |f|. & \text{(13.5)}
\end{align*}
\]

The first equality follows from the fact that \(T = V - S\) and from flow conservation. Note that \(f(S, S) = 0\). Similarly we can write \(S = s \cup S - s\) and use flow conservation.

**Corollary 13.1.** The flow in \(G\) is bounded by the minimum \((S, T)\)-cut capacity in \(G\).

To prove the Max-flow Min-cut theorem we prove an intermediate result (statement 2 below).

**Theorem 13.3** (Max-flow Min-cut Theorem). Let \(f\) be a flow from \(s\) to \(t\) in \(G\). The following conditions are equivalent:

1. \(f\) is a maximum flow.
2. The residual graph \(G_f\) allows no augmenting paths.
3. \(|f| = c(S, T)\) for some \((S, T)\)-cut.
298 13. GRAPH ALGORITHMS: MAXIMUM FLOW AND MINIMUM CUT

Proof. (1) \(\Rightarrow\) (2):

If \(G_f\) has an augmenting path the flow \(f\) can be improved.

(2) \(\Rightarrow\) (3):

We construct an \((S, T)\) cut as follows: Let

\[ S = \{ v \in V \mid \text{there is a path from } s \text{ to } v \text{ in } G_f \} \]

and let \(T = V - S\).

This partition defines an \((S, T)\)-cut where \(s \in S\) and \(t \in T\); otherwise there is an augmenting path in \(G_f\).

For each \(u \in S\) and \(v \in T\) we have \(f(u, v) = c(u, v)\), otherwise \((u, v) \in E_f\) and \(v \in S\). In other words, all edges of the form \((u, v)\) with \(u \in S\) and \(v \in T\) are saturated, i.e., the flow is equal to the capacity.

Thus \(f(S, T) = c(S, T)\) and we proved that \(|f| = f(S, T)\). Hence

\[ |f| = f(S, T) = c(S, T). \]

(3) \(\Rightarrow\) (1):

We proved that for any \((S, T)\) cut, \(|f| \leq c(S, T)\) (Corollary 13.1), thus if \(|f| = c(S, T)\) for some cut, it is a maximum flow.

13.2.2 Analysis of the Ford-Fulkerson Algorithm

Theorem 13.4. Assume that all the capacities are integral then the runtime of the Ford-Fulkerson algorithm is \(O(|E| \cdot |f|)\) where \(|f|\) is the maximum flow value.

Proof. Steps 1, 2, and 2a each take \(O(|E|)\) time. The while loop is executed \(O(|f|)\) times, since each iteration improves the flow by at least 1. An augmenting path in the residual graph can be found by breadth-first or depth-first search, thus each iteration of the while loop takes \(O(|E|)\) time.

We note that Ford-Fulkerson is a pseudo-polynomial time algorithm, since its running time is proportional to the flow value which can be quite large compared to \(n\) and \(m\), the graph parameters. A (strongly) polynomial algorithm’s running time will depend only on \(n\) and \(m\).

13.2.3 Edmonds-Karp Algorithm: An \(O(m^2 n)\) Algorithm

We present an algorithm due to Edmonds and Karp, which is a simple modification of Ford-Fulkerson, that is strongly polynomial.
Theorem 13.5. If the augmenting path in the Ford-Fulkerson algorithm is found by a breadth-first search (i.e. the algorithm uses a path of minimum number of edges), the run time of the algorithm is $O(m^2n)$, where $m$ is the number of edges and $n$ is the number of vertices.

To prove this theorem, we use the concept of a level graph which is a useful tool for analyzing other maximum flow algorithms as well.

**Definition 13.2.1 (Level Graph).** The level graph $L_G$ of $G$ is the directed breadth-first graph of $G$ (this is considered as an unweighted graph, where the capacities are ignored) with root $s$ with sideways and back edges deleted. The level of a vertex $u$ is the length of the shortest path from $s$ to $u$ in $G$.

In Figure 13.1, the level graph of the flow network of Figure A is the same graph with edge $(a,b)$ removed (see Figure 13.2). The level graph can be done by doing a BFS starting from $s$ on the flow graph (ignoring weights).

**Observations:** The following observations are easy to show:

1. The level graph has no edges from level $i$ to level $j$ for $j \geq i + 2$.
2. Any shortest path from $s$ to any other vertex is a path in the level graph.
3. Any path with either a back or a sideways edge of the level graph will be strictly longer.

Any path from $s$ to $t$ in the level graph will be a shortest path. We use such a path to augment the flow.

The following is a key lemma.

**Lemma 13.4.** (a) Let $p$ be an augmenting path of minimum length in $G$. Let $G'$ be the residual graph obtained by augmenting along $p$ and let $q$ be an augmenting path of minimum length in $G'$. Then $|q| \geq |p|$. Thus the length of the shortest augmenting path cannot decrease by applying the above heuristic.

(b) We can augment along shortest paths of the same length at most $m = |E|$ times before the length of the shortest augmenting path must increase strictly.

**Proof.** (a) $p$ is some path in the level graph $L_G$. After augmenting along $p$, at least one edge will disappear (due to saturation) and at most $|p|$ new
Figure 13.2: Level graph of the flow graph of Figure A of Figure 13.1. Note that edge \((a, b)\) is deleted in the level graph.
edges will appear in the residual graph $G'$. These new edges are back edges and cannot contribute to a shortest path from $s$ to $t$ as long as $t$ is reachable from $s$ in the level graph. Every time we augment using a path in $L_G$ we lose an edge in $L_G$. Thus we can augment at most $m$ times, before $t$ is no longer reachable in $L_G$. Then any augmenting path must use a back or side edge and must be strictly longer.

(b) Since we lose at least one edge (due to saturation) every time we augment, we can augment only $O(m)$ times.

**Edmonds-Karp Algorithm** We are now ready to describe an implementation of Edmonds-Karp algorithm in terms of the level graph framework.

1. Find the level graph $L_G$.
2. Repeatedly augment along paths in $L_G$, updating residual capacities and deleting saturated edges (zero capacity edges) until $t$ is no longer reachable from $s$.
3. Calculate a new level graph from the residual graph at that point and repeat.
4. Continue as long as $t$ is reachable from $s$ (in the residual graph).

The correctness of the algorithm is the same as that of Ford-Fulkerson.

**Running time** After each level graph calculation, the distance between $s$ and $t$ increases by 1. Thus, there are at most $n$ level graph calculations. We need $O(m)$ time to calculate a level graph (using breadth-first search which takes $O(m)$ time). The total number of augmentations in one level graph is at most $O(m)$ and each augmentation takes $O(m)$. Thus total time is $O(m^2n)$.

### 13.2.4 Dinitz Algorithm: An $O(mn^2)$ Algorithm

Dinitz max flow algorithm is similar to Edmonds-Karp, but we find augmenting paths in the level graph $L_G$ more carefully. It is similar to Edmonds-Karp as we compute the level graph many times (at most $n$ times), but differs in the way augmenting paths are found in a level graph.

Define a **blocking flow** as the flow when there is no path from $s$ to $t$ in $L_G$. That is, this is the total flow that has been pushed till there is no more possible using the current $L_G$. We need to recompute a new level graph at this point.
Instead of constructing a blocking flow path by path (as done in Edmonds-Karp), we construct a blocking flow all at once by finding a maximal set of minimum-length augmenting paths. Each such construction is a phase.

In each phase we will construct augmenting paths in \( L_G \) in a depth-first fashion.

**Dinitz Algorithm: One Phase**

1. **Initialize:** Construct a new level graph \( L_G \). Let \( u = s \) and \( p = s \). (\( u \) will denote the vertex currently being visited and \( p \) is a path from \( s \) to \( u \).) Go to **Advance**.

2. **Advance:** If there is no edge out of \( u \), go to **Retreat**. Otherwise, let \( (u, v) \) be such an edge. Set \( p = p.[v] \) and \( u = v \). If \( v \neq t \) then go to **Advance** else go to **Augment**.

3. **Retreat:** If \( u = s \) then **Stop**. Otherwise, delete \( u \) and all adjacent edges from \( L_G \) and remove \( u \) from the end of \( p \). Set \( u = \) last vertex on \( p \). Go to **Advance**.

4. **Augment:** Let \( \Delta \) be the bottleneck capacity along \( p \). Augment path flow along \( p \) and adjust residual capacities along \( p \). Delete newly saturated edges. Set \( u = \) the last vertex on the path \( p \) reachable from \( s \) along unsaturated edges of \( p \). Set \( p = \) the portion of \( p \) up to and including \( u \). Go to **Advance**.

**Runtime** We analyze the time taken for each for the different types of operations in one phase.

1. **Initialize:** Takes \( O(m) \) time, since this can be done by a breadth-first search which takes \( O(m + n) \) time.

2. **Advance:** The time spent in all Advance steps is \( O(mn) \) since there are at most \( 2mn \) advances in each phase. Why? Because there at most \( n \) advances before an augment or retreat and there are at most \( m \) augments (since at least one edge is deleted in each augment) and \( n \) retreats (at least one vertex is deleted in each retreat).

3. **Retreat:** Takes \( O(m + n) \) time. Because there are most \( n \) retreats in each phase and each retreat takes \( O(1) \) time plus time to delete edges which is \( O(m) \).

4. **Augment:** The time spent in all augment steps is \( O(mn) \) time. This is because there are at most \( m \) augments in each phase and each augment takes \( O(n) \) time.
Thus each phase requires $O(mn)$ time and there are at most $n$ phases. Hence total running time of Dinitz algorithm is $O(mn^2)$.

### 13.2.5 MPM Algorithm: An $O(n^3)$ Algorithm

MPM is similar to Edmonds-Karp and Dinitz (i.e., the algorithm consists of phases, and each phase finds a blocking flow using the current level graph), but blocking flows are found for level graphs in $O(n^2)$ time. We consider the capacity of a vertex as opposed to the capacity of an edge.

**Definition 13.2.2.** The capacity $c(v)$ of a vertex $v$ in a flow network is the minimum of the total capacity of its incoming edges and the total capacity of its outgoing edges:

$$c(v) = \min \{ \sum_{u \in V} c(u,v), \sum_{u \in V} c(v,u) \}.$$

The main idea of the MPM algorithm is to push a flow equal to the minimum capacity among all vertices. Note that this is always possible, since every vertex can support this much amount of flow going through it.

The MPM algorithm proceeds in phases. In each phase, we compute the residual graph for the current flow and also compute the level graph. If $t$ does not appear we are done. Otherwise, all vertices not on a path from $s$ to $t$ in the level graph are deleted. We now find a blocking flow as follows.

**MPM Algorithm: One Phase**

**Step 1:** Find a vertex $v$ of minimum capacity $d$. If $d = 0$ goto step 2 else Go to **Step 3**.

**Step 2:** Delete $v$ and all its incident edges and update the capacities of the neighboring vertices. Go to **Step 1**.

**Step 3:** **Push** $d$ units of flow from $v$ to sink and **pull** $d$ units of flow from the source to $v$ to increase the flow through $v$ by $d$:

**Push to sink:** Saturate the outgoing edges of $v$ in order, leaving at most one partially saturated edge. (In other words, send flow in a greedy fashion, saturating each edge fully, with possibly only one edge that might be partially saturated.) Repeat this process on each vertex (all
the way up to $t$) that received flow during the saturation of the edges out of $v$. Delete all saturated edges. Update capacities of all nodes.

**Pull from source:** Saturate the incoming edges of $v$ in order, leaving at most one partially saturated edge. Repeat this process on each vertex (all the way back to $s$) from which flow was taken during the saturation of the edges out of $v$. Delete all saturated edges. Delete $v$ and all its remaining incident edges from the level graph. Update the capacities of all the nodes. Go to **Step 1**.

**Running time** We compute the time needed for one phase.

- $O(m)$ time is needed to compute the residual graph and level graph using BFS.
- $O(n \log n)$ time to find and delete a vertex of minimum capacity using Fibonacci (or binary) heaps.
- $O(n)$ time to delete all the (fully) saturated edges.
- $O(n^2)$ time to do partial saturations, because it is done at most once in Step 3 at each vertex for each choice of $v$ in step 1.
- $O(n^2)$ time to decrement capacities of incident vertices using Fibonacci heaps. (If binary heaps are used, then this will take $O(n^2 \log n)$ time, since each Decrease-Key operation takes $O(\log n)$ time in a binary heap.)

Thus, one phase takes $O(n^2)$ time and there are at most $n$ phases, thus the overall running time of MPM algorithm is $O(n^3)$.

### 13.3 Matching

Matching is another fundamental graph problem. A matching is defined as follows.
Definition 13.3.1. Given a graph \( G = (V, E) \) a set of edges \( M \) is a matching in \( G \) if

1. \( M \subseteq E \);
2. no two edges of \( M \) share the same node;

A maximum matching in (unweighted graph) \( G \) is a matching with maximal cardinality, i.e., the maximum number of edges.

A graph has a perfect matching if it has a matching of size \(|V|/2\) (i.e. every vertex is covered by the matching).

In this section, we show an algorithm to solve the matching problem in bipartite graphs. This is done by giving a simple reduction to the maximum flow problem.

13.3.1 Bipartite Matching Through Maximum Flow

A graph \( G = (V, E) \) is bipartite if the set of vertices \( V \) can be partitioned to two sets \( A \) and \( B \), \((A \cup B = V \text{ and } A \cap B = \emptyset)\), such that every edge in \( E \) has one adjacent vertex in \( A \) and one in \( B \) (i.e., there are no edges between two vertices in \( A \) or two vertices in \( B \)).

Given a bi-partite graph \( G = (A, B, E) \) we can use a maximum flow algorithm to find maximum matching in \( G \) as follows (see Figure 13.3).

Define a directed graph \( G' = (V', E') \):

- \( V' = A \cup B \cup \{s, t\} \)
- Connect \( s \) to all vertices in \( A \). Direct all edges in \( E \) from \( A \) to \( B \). Connect all vertices in \( B \) to \( t \).
- All edges have capacity 1.

A flow is integer value if the flow through each edge is an integer value.

Theorem 13.6. The Ford-Fulkerson algorithm generates an integer value flow in \( G' \).

Proof. Note that each augmentation can be used to push a flow of 1.

Theorem 13.7. The cardinality of the maximum matching in \( G \) equals the value of the maximum flow in \( G' \).
Figure 13.3: A bipartite graph $G$ (top) and the directed graph $G'$ (bottom) which is a flow graph obtained from $G$. All capacities in $G'$ are 1.
13.4. MINIMUM CUT

Proof. 1. We show that if $f$ is the size of the maximum matching in $G$, then we can push a flow of $f$ in $G'$. This is clear, since the matching edges don’t share a common endpoint, one can push a flow of 1 through each edge from $s$ to $t$.

2. We show that if one can push an integer flow of $f$ in $G'$, then there is matching of size $f$ in $G$. Since each node in $A$ and $B$ can be adjacent to only one vertex with integer flow $>0$, a flow $f$ in $G'$ corresponds to a matching $M$ in $G$ with $|f| = |M|$.

Running time The running time of the maximum flow, using the Ford-Fulkerson algorithm, is $O(|V| |E|)$, since the Ford-Fulkerson runs in $O(|E||f|)$ time, where $|f|$ is the maximum flow, which is $O(|V|)$ here.

13.4 Minimum Cut

We study the minimum (min) cut problem, another fundamental graph optimization problem. Recall, that given a cut $(S, T = V - S)$, the size of the cut is the number of edges crossing the cut. (In the weighted case, it is the sum of the weights of the edges crossing the cut.) The min cut problem is to find the minimum sized cut in the graph. There are two variants of the problem: (1) $s-t$ min cut which finds a min cut where $s$ and $t$ are on opposite sides of the cut and (2) global min cut, which is the min cut over all cuts.

Another way to define the global min cut is the minimum set of edges needed to disconnect the graph into two or more pieces. Thus clearly, the global min cut cannot be larger than the minimum degree of the graph (why?)

One obvious way to find the $s-t$ min cut is to use the max flow-min cut duality theorem that we showed. Find the maximum flow between $s$ and $t$ also gives the size of the min cut. (One can also recover the edges also from the max flow. How?) To find the global min cut, one can find the $s-t$ min cut over all pairs of vertices and take the minimum. However, it can be shown that the global min cut can be computed using one max flow computation. Even this takes $O(mn)$ time, since the best max flow algorithm takes that much time (here we saw an $O(n^3)$ algorithm). All these are deterministic algorithms.

In this section, we will describe a randomized min cut algorithm that finds the global min cut in time $O(n^2 \log^3 n)$ algorithm which is significantly faster than the best known deterministic algorithm of $O(mn)$ in
dense graphs. The randomized algorithm is conceptually quite simple and does not use any flow computation.

### 13.4.1 A Randomized Min Cut Algorithm

The algorithm (due to Karger) is as follows. Starting from a $n$-node graph, it repeatedly chooses a random edge and contracts the edge. A contraction operation is defined as follows: contacting an edge $(u, v)$ is equivalent to coalescing $u$ and $v$ into a single vertex (say $z$) and making the neighbors of $u$ and $v$ as neighbors of $z$; any self loops created are deleted, while multi-edges are retained. See Figure 13.4 for an example. Note that each contraction operation reduces the number of vertices by one. The algorithm stops when the number of vertices are reduced to 2; in that situation the edge between the two vertices are output. The pseudocode is as follows.

**Karger's algorithm (one iteration)**

**Input:** A $n$ node graph

**Output:** A minimum set of edges that disconnects the graph.

1. repeat $n - 2$ times:
   (a) Pick an edge uniformly at random.
   (b) Contract the edge.

end repeat

2. output the set of edges connecting the two remaining vertices.

**Analysis of the Algorithm** The following is the key theorem that lower bounds the success probability of the algorithm.

**Theorem 13.8.** The algorithm outputs a min-cut set of edges with probability $\geq \frac{2}{n(n-1)}$.

We need the following crucial lemma.

**Lemma 13.5.** Contraction operation (step 1(b)) does not reduce the size of the min-cut set.

**Proof.** Every cut set in the new graph is a cut set in the original graph. This is because deleting the cut set in the contracted graph also disconnects the original graph (before contractions). \(\square\)

Assume that the graph has a min-cut set of $k$ edges. We compute the probability of finding one such set $C$. The proof of the following lemma is easy and left as an exercise (see Exercise 13.11).
Figure 13.4: Figure shows a graph $G$ (top) and the contracted graph $G'$ (bottom) obtained by contracting edge $(a, d)$ (labelled 4). Note that self-loop (shown in dotted line) is not present in the contracted graph, but multi-edges are retained.
Lemma 13.6. Let $C$ be a min-cut set of $G$. If the run of the algorithm did not contract any edge of $C$, it also did not eliminate any edge of $C$.

Now we are ready to prove Theorem 13.8.

Proof. Since the minimum cut-set has $k$ edges, all vertices have degree $\geq k$, and the graph has $\geq nk/2$ edges.

Let the event $E_i$ denote the event that "the edge contracted in iteration $i$ is not in $C$".

We want

$$\Pr(\cap_{i=1}^{n-2} E_i) = \Pr(E_1) \Pr(E_2|E_1) \Pr(E_3|E_2 \cap E_1) \ldots \Pr(E_{n-2}|\cap_{i=1}^{n-3} E_i).$$

Thus,

$$\Pr(\cap_{i=1}^{n-2} E_i) \geq \Pi_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) = \Pi_{i=1}^{n-2} \left(\frac{n-i-1}{n-i+1}\right) = \frac{2}{n(n-1)}.$$

Exercise 13.2. Show that if Karger’s algorithm was done for $n - t$ steps, i.e., till $t$ vertices remained (instead of 2), then the probability that a min cut will survive in the $t$-vertex graph is at least $\frac{t(t-1)}{n(n-1)}$.

A High Probability Algorithm  

The success probability of Karger’s algorithm is very small, i.e., $\frac{2}{n(n-1)}$. One can easily boost up the success probability to be close to 1 by repeating the algorithm about $n^2 \ln n$ times.

Algorithm High-Probability-Karger

Input: A $n$ node graph

Output: A minimal set of edges that disconnects the graph.

1. for $i = 1$ to $n^2 \ln n$
   1.1. repeat $n - 2$ times:
      (a) Pick an edge uniformly at random.
      (b) Contract the edge.

endrepeat
13.4. MINIMUM CUT

1.2 Let \( C_i \) be the set of edges connecting the two remaining vertices.

\textbf{endfor}

2. output the set with the minimum size among the \( C_i \)’s.

\textbf{Theorem 13.9.} The high probability algorithm outputs a min-cut set with probability at least \( 1 - 1/n^2 \).

\textbf{Proof.} The probability that the algorithm fails to output the min cut in one iteration of the for loop is \( 1 - \frac{2}{n(n-1)} \leq 1 - 1/n^2 \). Since all the iterations are independent of each other (in terms of the random choices), the probability that the min cut is not output in all the \( n^2 \ln n \) iterations is at most

\[
(1 - 1/n^2)^{n^2 \ln n} \leq e^{-n^2 \ln n/n^2} \leq e^{-\ln n} = 1/n.
\]

(Note that we used the inequality that \( 1 - x \leq e^{-x} \), for all \( |x| \leq 1 \).)

\textbf{Running time} The running time of the algorithm depends on the time needed to implement one iteration of Karger’s algorithm. Exercise\[13.3\] asks you to show that one step of the iteration — choosing a random edge and contracting it — can be implemented in \( O(n) \) time; hence Karger’s algorithm (one iteration) can be implemented in \( O(n^2) \) time.

Thus the high probability algorithm, which requires \( O(n^2 \ln n) \) iterations of the Karger’s algorithm, requires \( O(n^4 \ln n) \) time which is significantly more than the best known \( O(mn) \) deterministic algorithm. How can we improve Karger’s algorithm?

\textbf{Exercise 13.3.} Show that one step of Karger’s algorithm — choosing a random edge and contracting it — can be implemented in \( O(n) \) time.

\textbf{13.4.2 Improving Karger’s Algorithm: An} \( O(n^2 \text{ polylog}(n)) \) \textit{algorithm}\*

We notice that the main weakness of the Karger’s algorithm is towards the end, when there are only a few vertices remaining, when the probability of contracting an edge belonging to the min cut increases. In particular in the last step (when there are 3 vertices), the probability of contracting such an edge becomes \( 2/3 \) which is large. Note that, on the other hand, when there are a large number of vertices, the probability of contracting an edge belonging to the min cut is small. This suggests an idea. Use Karger’s algorithm to contract the graph till \( t \) vertices are remaining (for a suitable \( t \)) and then use a slower deterministic algorithm to find the min cut in the remaining graph which always succeeds. Exercise\[13.8\] asks you
to show that this idea does give you a better algorithm with a running time of $O(n^{2+\epsilon})$ for some constant $\epsilon$, but it does not give a $O(n^2 \text{polylog } n)$ algorithm.

The key idea of the improved algorithm is to perform Karger’s iteration two times (each independently) on $G$ till the number of vertices is reduced by a constant factor. This gives two contracted graphs $G_1$ and $G_2$. Find the min cut in $G_1$ and $G_2$ and output the minimum of the two cuts. To find the min cut in $G_1$ and $G_2$, we apply the algorithm recursively. The pseudocode is as follows:

**Algorithm ImprovedCut($G$)**

**Input:** A $n$ node graph.

**Output:** A min cut of $G$.

1. If $n \leq 10$, find and return the min cut (using brute force search).

2. Let $t = \frac{n}{\sqrt{2}}$.

3. Apply Karger’s algorithm $n-t$ times to $G$ to get a contracted graph $G_1$ of $t$ vertices.

4. Apply Karger’s algorithm (again independently) $n-t$ times to $G$ to get another contracted graph $G_2$ of $t$ vertices.

5. Let $C_1 = \text{ImprovedCut}(G_1)$ and $C_2 = \text{ImprovedCut}(G_2)$.

6. Return $\min(C_1, C_2)$.

The main intuition of the choosing the value of $t$ as in the algorithm is as follows. First, consider the recursion tree of the ImprovedCut algorithm. It is a binary tree with $G$ as the root which has two subtrees — roots of the subtrees are $G_1$ and $G_2$ respectively. The size of $G$ goes down by a factor of $\sqrt{2}$. So the depth of the recursion tree is about $2 \log n$ (Why ?). So the number of leaves is about $2^{2\log n} = n^2$. Now consider Karger’s algorithm, which has a failure probability of $\Theta(1/n^2)$. We repeated Karger’s algorithm about $n^2$ times (actually $n^2 \ln n$ times to be precise) to boost the failure probability (Algorithm High-Probability-Karger). The High-Probability-Karger can be represented as a tree of depth 1, with $G$ as the root, and the $n^2$ repetitions forming the $n^2$ leaves. Thus both trees have about the same number of leaves (which is required since the $1/n^2$ failure probability has to be boosted up by so many repetitions), but the ImprovedCut Algorithm shares a lot of work in the initial steps (where the success probability is large), leading to an improved algorithm, unlike the High-Probability-Karger, which repeats all the iterations from scratch.
Run time analysis. We can analyze the run time by writing a recurrence (which is natural, since we are dealing with a recursive algorithm).

\[ T(n) = 2T\left(\frac{n}{\sqrt{2}}\right) + O(n^2). \]

The \(O(n^2)\) term comes from the fact that we need the run \(n - n/\sqrt{2}\) steps of the Karger algorithm which takes \(O(n^2)\) time (actually the whole Karger algorithm — running for \(n - 2\) steps — takes \(O(n^2)\) time). The base case is \(T(n) = O(1)\) for \(n \leq 10\). Using the DC recurrence theorem (Theorem 2.2, Chapter 2), we can show that \(T(n) = O(n^2 \ln n)\).

Success probability analysis. The crucial thing is to show that the success probability, i.e., the probability of outputting a min cut, of the ImprovedCut algorithm is large. Again, we write a recurrence and solve. Let \(P(n)\) denote the probability of the Improved Cut algorithm to output a min cut on an input graph of \(n\) vertices. Then we can write

\[ P(n) \geq 1 - (1 - 1/3(P(n/\sqrt{2})))^2. \]

The reason for the above is as follows. Consider the ImprovedCut algorithm operating on \(G\) with \(n\) nodes. The probability that the min cut will survive \(n - n/\sqrt{2}\) contractions is at least \(1/3\) (by applying result of Exercise 13.2): \(\frac{(n/\sqrt{2})(n/\sqrt{2} - 1)}{n(n-1)} \geq 1/3\), for \(n \geq 10\). The algorithm will output a min cut if the min cut survives the contraction and the min cut is found successfully in the contracted graph of \(n/\sqrt{2}\) nodes. Since we do two independent contractions, the algorithm will succeed if min cut is found in one of the two cases. The base case of the recurrence is \(P(n) = 1\), for \(n \leq 10\).

Exercise 13.12 asks you to show that the solution of the above recurrence is \(\Omega(1/\log n)\). Hence ImprovedCut algorithm finds a min cut with probability at least \(\Omega(1/\log n)\) (note that, this is significantly better than Karger’s algorithm (one iteration) \(\Theta(1/n^2)\) success probability).

Thus, by repeating the ImprovedCut algorithm \(\Theta(\log^2 n)\) times, we get a high probability algorithm.

13.5 Worked Exercises

Worked Exercise 13.1. A path flow in \(G\) is a flow that takes nonzero values only on some simple path from \(s\) to \(t\). In other words, there exist a number
13. GRAPH ALGORITHMS: MAXIMUM FLOW AND MINIMUM CUT

Let $f(u_i, u_{i+1}) = d, 0 \leq i \leq k - 1$
$f(u_{i+1}, u_i) = -d, 0 \leq i \leq k - 1$
$f(u, v) = 0 \text{ for all other } (u, v), u, v \in V.$

Recall that the smallest (residual) capacity along an (augmenting) path is called the bottleneck capacity of that path.

(a) Show that any flow in $G$ can be expressed as a sum of at most $m$ path flows in $G$ and a flow of value 0, where $m$ is the number of edges of $G$. (Hint: Let $f$ be a flow in $G$ and let $|f| > 0$. Consider $G'$, a graph with the new capacity function $c'(e) = \max\{f(e), 0\}$. Consider the flow with value 0 in $G'$.)

(b) Suppose we use the Ford-Fulkerson algorithm with the following heuristic for finding an augmenting path: always choose an augmenting path of maximum bottleneck capacity.

Show that if the edge capacities are integers, then the above heuristic results in a maximum flow $f^*$ in at most $O(m \log |f^*|)$ augmenting steps, where $m$ is the number of edges. (Hint: How can you bound the flow pushed after each augmenting step? Use the result of part (a). You might find the following estimate useful in your analysis: $\log(m/(m-1)) = \Theta(1/m).$)

(c) Give an algorithm to find a maximum bottleneck capacity augmenting path in a flow graph. (Hint: Modify Dijkstra’s algorithm.)

Solution:

(a) Intuitively, the decomposition of a flow into simple paths can be understood from the Ford-Fulkerson (FF), where at each step we augment by a simple path (there is positive flow only along the augmenting path and zero flow elsewhere). For this question, however, we have to show this, since the flow could have been constructed by any means (not necessarily by using FF). We do this by decomposing the flow using augmenting paths as follows.

Let $f$ be a flow in $G$. If $|f| = 0$, we are done. Otherwise, then $|f| > 0$. As given in the hint, define a new capacity function $c'(e) = \max\{f(e), 0\}$ and let $G'$ be the graph with these capacities. Then $f$ is also a flow in $G'$, and since $c'(.) \leq c(.)$, any flow in $G'$ is also a flow in $G$. By the Max flow-Min Cut Theorem, the zero flow in $G'$ must have an augmenting path, which is a (simple) path from $s$ to $t$ with positive capacities. Note that by construction, every edge of $G'$ is saturated by flow $f$. Take $p$ to the path flow on that path whose value is the bottleneck capacity, i.e., set every flow on that path equal to the bottleneck capacity of that path. Then
the two flows $p$ and $f - p$ are both flows in $G'$ and at least one edge on the path is saturated by $p$. Remove the saturated edge (but don't add the reverse edge — here we are not computing the residual graph, but simply subtracting the flow $p$ from flow $f$). Now we repeat this process with $f - p$ to get $e''(.) < e'(.)$ and so on. Note that $G''$ has strictly fewer edges than $G'$, since the bottleneck edge has disappeared. This process can be repeated at most $m$ times before the flow value becomes zero. The original flow $f$ is then the sum of the flow of value zero and the paths flows found in each step.

(b) By part (a) any flow $f$ can be decomposed into at most $m$ path flows. Thus the maximum value of these (at most) $m$ flows is at least $|f|/m$. Thus if we augment along the path with the maximum bottleneck capacity, we push a flow of at least $|f|/m$. Then the flow (in the residual graph) that remains to be pushed will be at most $|f| - |f|/m = |f|(1 - 1/m)$. We then repeat the process. Since we reduce the remaining flow by a factor of at least $1 - 1/m$, the number of augmenting path iterations is at most $\frac{\log |f|}{\log(1-1/m)} = O(m \log |f|)$ (using the fact that $\log(1-1/m) = \Theta(1/m)$).

(c) Although finding the path with the maximum bottleneck capacity appears to be very different from the shortest paths problem, there is a uniform way of looking at both the problems. In the shortest path problem, the “value” of a path is the sum of weights in the path, and we want to find (among all possible paths) the one with minimum value. In the maximum bottleneck problem, the “value” of a path is the minimum weight edge of that path and we want to find the path with the maximum value. Thus, the definition of “value” differs, and instead of finding a path of minimum value (as in shortest paths), we are finding a path with maximum value.

A modification of Dijkstra’s algorithm can be used to compute the maximum bottleneck value path between $s$ (the source) and all other vertices. Let

- $b(u,v)$ — the weight (capacity) of edge $(u,v) \in E$.
- $key[v]$ — the value of the maximum bottleneck path from $s$ to $v$.
- $parent[v]$ — the predecessor of $v$ in the path.

Algorithm Dijkstra($G$, $w$, $s$)

1. For all $v \in V$ do

   (a) $key[v] = 0$

   (b) $parent[v] = NIL$
2. \( key[s] = \infty \)
3. \( S = \emptyset \)
4. \( Q = V \)
5. While \( Q \neq \emptyset \) do
   (a) \( u = \text{Extract Max}(Q) \)
   (b) \( S = S \cup \{u\} \)
   (c) For all \( v \in \text{Adj}[u] \) do
      i. if \( key[v] < \text{Min}(key[u], b(u, v)) \) then
         A. \( key[v] = \text{Min}(key[u], b(u, v)) \) // increase-key of \( v \)
         B. \( \text{parent}[v] = u \)

We can prove the correctness of this algorithm by showing that the following two invariants hold by induction on the number of iterations (or the size of \( S \)). The proof is very similar to the proof of Dijkstra’s algorithm for computing shortest paths in a graph.

1. if \( v \in S \) then \( key[v] \) is the value of the maximum bottleneck value path of \( v \) from \( s \), and \( \text{parent}[v] \) encodes the last edge of that path.

2. if \( v \notin S \) then \( key[v] \) is the value of the maximum bottleneck path of \( v \) from \( s \) using only internal vertices of \( S \), and \( \text{parent}[v] \) encodes the last edge of that path.

Note that we can use a binary max-heap to implement the heap operations — extract-max and increase-key. The running time is the same as Dijkstra’s algorithm.

13.6 Exercises

Exercise 13.4. Given an undirected graph \( G = (V, E) \) and elements \( s, t \in V, s \neq t \), decide whether there exist \( k \) edge-disjoint paths from \( s \) to \( t \), and find them if so. Provide an efficient polynomial time algorithm.

Exercise 13.5. A unit network \( G = (V, E) \) is a network in which all edge capacities are integers and each vertex \( v \in V \) other than the source \( s \) and sink \( t \) has either a single entering edge, of capacity one, or a single outgoing edge, of capacity one.

a) Show that on a unit network, Diniz algorithm halts after at most \( O(\sqrt{|V|}) \) phases (each phase finds a blocking flow).
b) Show that the above result can be used to obtain an $O(\sqrt{|V||E|})$-time algorithm for finding a maximum matching in unweighted bipartite graphs.

**Exercise 13.6.** Let $G = (V, E)$ be a flow network with source $s$, sink $t$, and integer capacities. We know the maximum flow value in $G$. Suppose that the capacity of a single edge $(u, v) \in E$ is increased by 1. Give an $O(|V| + |E|)$ algorithm to update the maximum flow.

**Exercise 13.7.** Modify Karger’s algorithm so that it finds a minimum cut in a weighted graph where all weights are integers. (The min cut is the set of edges with the smallest total weight whose removal disconnects the graph.)

**Exercise 13.8.** Suppose you are given a deterministic algorithm for finding min cut that runs in $O(mn)$ time. Show how you can combine this algorithm with Karger’s algorithm (that runs till $t$ vertices remain, for some suitably chosen $t$) to obtain an algorithm that runs in time $O(n^{2+\varepsilon})$ time. How small can $\varepsilon$ be?

**Exercise 13.9.** Generalizing on the notion of a cut-set, we define an $r$-way cut-set in a graph as a set of edges whose removal breaks the graph into $r$ or more connected components. Explain how the randomized min-cut algorithm can be used to find minimum $r$-way cut-sets, and bound the probability that it succeeds in one iteration.

**Exercise 13.10.** Let $G = (V, E)$ be an undirected graph with $m$ edges and $n$ vertices. For any subset $U \subseteq V$, let $G[U]$ denote the subgraph induced on $U$, i.e., the graph with vertex set $U$ and whose edge set consists of all edges of $G$ with both ends in $U$. Given a number $k$ ($1 \leq k \leq n$), your task is to design a polynomial time algorithm that produces a set $U \subseteq V$ of $k$ nodes with the property that the induced subgraph $G[U]$ has at least $\frac{mk(k-1)}{n(n-1)}$ edges.

(a) Give a deterministic polynomial time algorithm for the problem. Show that your algorithm indeed produces the desired set. What is the running time of your algorithm? (Hint: Try a greedy strategy.)

(b) Give a Las Vegas randomized algorithm that has expected polynomial running time. Show correctness and running time analysis.

**Exercise 13.11.** Prove Lemma 13.6.

**Exercise 13.12.** Show that solution of the recurrence $P(n) = \frac{c}{\log n + 0.5}$ for some (large) constant $c > 0$. (Hint: Use induction).
318 13. GRAPH ALGORITHMS: MAXIMUM FLOW AND MINIMUM CUT
Appendices
Mathematical Induction

Mathematical induction is a simple but very powerful technique to prove mathematical theorems. It is also extremely useful in proving correctness of algorithms as well as in understanding the insight behind their design; it is also useful in analyzing the performance of algorithms. Before we illustrate mathematical induction with some examples, a few words about the general technique. Suppose we want to prove some mathematical statement that depends on some parameter $n$ which can be any natural number. Examples are: (1) the sum of the first $n$ numbers is $\frac{n(n + 1)}{2}$; (2) $\binom{2n}{n} \geq 2^n$; (3) the running time of an algorithm on input size $n$ is at most $2n$ etc. The goal is to show that the mathematical statement is true, i.e., it is true for all natural numbers. Note that it is generally easy, at least in principle, to verify the statement for a particular value of $n$, e.g., if $n = 100$, the first statement can be checked by simply summing up the first 100 numbers (may take a while) and verifying that it equals $\frac{100(101)}{2}$. However, this is not a valid approach to proving the statement for every natural number. Mathematical induction is a technique for precisely doing this. (Note that in many cases, there are alternative ways to prove such statements, but induction is the most straightforward and “automatic” way to do that; indeed there are important mathematical theorems where the only known proof uses induction.)

A.1 The General Technique

How does mathematical induction work? Suppose we want to show that a mathematical statement is true for all natural numbers $n \geq 0$. Mathematical induction consists of two steps: (1) Base step and (2) Induction Step. The base step is verifying whether the statement is true for $n = 0$. This is, of course, easy, since we have to verify for just one value. The induction step is the crucial step: assuming
that the statement is true for natural number \( k \), we have to prove that the statement is true for \( k + 1 \). It is important to note that \( k \) is a variable and not any fixed number, and thus proving the induction step involves proving the statement for any (general) \( k \). The assumption that the statement is true for natural number \( k \) is usually called the induction hypothesis. If we show these two steps — the base step and the induction step — then by mathematical induction it follows that the statement is true for all natural numbers. Why is this the case? This is not difficult to see if we iterate through the numbers. The base step shows that the statement is true for \( n = 0 \). Applying the induction step for \( k = 0 \) (since we show it for any number \( k \), it holds for \( k = 1 \). Since it holds for \( k = 1 \), we can again use the induction step to show the statement for \( k = 2 \). Again, since it holds for \( k = 2 \), we can again use the induction step to show the statement for \( k = 3 \) and so on. Repeating this argument, it is clear that the statement holds for every natural number. This is the essence of a proof by mathematical induction.

The power of mathematical induction — which naturally underlies recursive algorithms (and iterative algorithms as well) — comes from the induction hypothesis which simply assumes that the statement is true for \( k \) (somehow by “magic”!). This gives us a lot of “ammunition” to prove that the statement is true for \( k + 1 \).

### A.1.1 Sum of first \( n \) natural numbers

Let’s illustrate the technique of mathematical induction by proving a basic mathematical statement (which also arises frequently in analysis of algorithms): the sum of the first \( n \) natural numbers is \( n(n + 1)/2 \).

**Theorem A.1.**

\[
0 + 1 + 2 + \cdots + n = \sum_{i=0}^{n} i = \frac{n(n + 1)}{2}.
\]

**Proof.** We prove by using mathematical induction on the variable \( n \). (Note that it is a good idea to say the variable that we are applying induction to. In this example, it is obvious since there is only one variable, but in others, there may be more than one variable in the statement, and it better to be explicit about it. We will see more examples throughout this book.)

We have to show the following two steps:

**Base step:** \( n = 0 \): The statement is trivially true, since \( 0(1)/2 = 0 \) which is indeed the sum of the first number (0).

**Induction step:** Assume that the statement is true for \( k \) (induction hypothesis). We will prove that the statement is true for \( k + 1 \).
By the induction hypothesis, we have

\[ 0 + 1 + 2 + \cdots + k = \frac{k(k + 1)}{2}. \]

Using the above, we have to prove that the following statement holds:

\[ 0 + 1 + 2 + \cdots + k + k + 1 = \frac{(k + 1)(k + 2)}{2} \] (A.1)

How can we show that? We use the induction hypothesis that already tells us the value of the sum up to \( k \). Thus the left hand side of Equation A.1 becomes

\[ \frac{k(k + 1)}{2} + k + 1 \]

which simplifies to \( \frac{(k+1)(k+2)}{2} \) which is the right hand side of Equation A.1 as desired.

Thus we have proved the induction step and hence the statement is proved by mathematical induction.

A.1.2 Coloring regions formed by lines

We will prove a simple, but somewhat surprising result on coloring regions formed by lines. Consider \( n \) distinct (infinite) lines drawn in a plane. A region is an area of the plane that is enclosed by lines; note that a region can be finite (closed) or infinite (open). The goal is to assign colors to regions so that no two adjacent regions (i.e., regions that share a line as a boundary) have the same color — such a coloring is called valid. For example, if there is only one line, it partitions the plane into two (adjacent) regions and if there are two intersecting lines then it partitions the plane into four regions where each region is adjacent to two other regions. How many colors are needed to validly color regions formed by \( n \) lines?

The following theorem shows that only two colors are needed.

**Theorem A.2.** Two colors are needed to validly color all regions formed by any number of lines.

**Proof.** The proof is by induction on \( n \), the number of lines.

Base step: \( n = 1 \). Then the line partitions the plane into two regions; color one region by blue and the other by red. Clearly this coloring is valid.

Induction step: Assume that the statement is valid for \( k \) lines. We will prove the statement when there are \( k + 1 \) lines. Consider \( k + 1 \) distinct lines drawn in a plane. We will demonstrate the existence of a valid coloring of the regions formed by these \( k + 1 \) lines using two colors. Consider the set of \( k + 1 \) lines and
remove one line from the set. This leaves us with \( k \) lines and by the induction hypothesis, there is a valid coloring of the regions formed by it using 2 colors — red and blue. Assume this valid coloring. Now let’s add the \((k + 1)\)th line and we will show how to modify the colors of the regions to still have a valid coloring. We leave the colors of all regions that are on one side of the \((k + 1)\)th line unchanged and reverse the colors of the regions on the other side. We claim that this is a valid coloring. Consider the regions formed by \( k \) lines. When the \((k + 1)\)th line was added, it cuts some regions formed by the \( k \) lines into two parts — one on each side. Since we reverse the color on one side of the \((k + 1)\)th line, these “cut” regions have a valid coloring. Now consider the regions that are not cut by the \((k + 1)\)th line, i.e., those that are entirely on either side of the line. These are (already) validly colored by the induction hypothesis, since these are regions formed by \( k \) lines.

Algorithm via Induction Proof  The above induction proof also suggests a natural algorithm that gives a valid coloring of the regions formed by \( n \) lines. The algorithm is naturally incremental. Start with one line and use two colors to color the two regions on either side — this is the base case. Then add one line at a time. After adding a line, simply reverse the colors of the regions on one side of the added line, leaving the other side unchanged — this is the strategy used in the induction step. Clearly this algorithm gives a valid coloring and the proof of its correctness is indeed the induction proof!

This example illustrates how algorithms can be designed by using induction proofs; as an added bonus the proof of correctness of the algorithm follows from the induction proof.

Weak Induction  The above form of mathematical induction is sometimes called the “weak form of induction” as the induction hypothesis assumes only the validity of the statement for value \( k \) (and not for values less than \( k \) — as done in the “strong form” as explained in the next section). This is already useful in proving correctness of many algorithms, especially iterative algorithms, which operate incrementally on the input size. In particular, it is useful in proving “loop invariant” in a for or while loop. A loop invariant is a property that holds in every iteration of the for loop; if it fails to hold the loop terminates. Correctness of many algorithms with for and while loops involve showing statements on loop invariants. Such examples are shown throughout the book.
A.2 Strong Mathematical Induction

In many cases, the induction hypothesis can be strengthened for “free.” Instead of assuming that the statement is true for $k$, we can assume that the statement is true for all natural numbers $n \leq k$ and then proceeding to show, under this assumption, that the statement is true for $k + 1$. The strengthened hypothesis, called strong form of induction, is extremely useful, leading to simpler proofs. It is especially useful in proving correctness (and understanding) of recursive algorithms, e.g., see Chapter 4.

A.2.1 Fundamental Theorem of Arithmetic

We will illustrate the strong form of induction using the following example, which is an important theorem in mathematics: the fundamental theorem of arithmetic which states that every natural number greater than 1 can be expressed as a product of prime numbers. Recall that a prime number is a positive integer (greater than 1) that is divisible only by itself and by 1.

**Theorem A.3.** Every natural number $n > 1$ can be expressed as a product of prime numbers.

**Proof.** We will prove by induction on $n$.

Base case: $n = 2$ (note that this is the base case, as it is the smallest natural number for which the theorem is supposed to hold). Since 2 is prime, the statement is trivially true.

Induction step: Assume that the statement is true for all natural numbers from 2 to $k$ (induction hypothesis). Note that we are assuming the strong form of the induction which is needed for this proof. We will prove that the statement is true for $k + 1$.

There are two cases to consider. Suppose $k + 1$ is prime — in this case, the statement is trivially true. Suppose $k + 1$ is not a prime number. Then $k + 1$ can be decomposed into a product of two numbers $a$ and $b$, i.e., $k + 1 = ab$, where $1 < a < k + 1$ and $1 < b < k + 1$ (in fact they are both strictly smaller than $k$). Since $a$ and $b$ are strictly smaller than $k + 1$ and strictly greater than 1, induction hypothesis applies to both of them, and hence each of them can be written as a product of prime numbers. Hence $k + 1$ which is a product of $a$ and $b$ can be written as a product of prime numbers.

Note that the strong form was needed for the above proof, since $a$ and $b$ are numbers that are smaller than $k$. 


A.3 Using Induction for Algorithm Analysis

One weakness of induction is that one needs to know the exact statement that one needs to prove beforehand. For example, we need to know that the sum of the first $n$ numbers is $n(n+1)/2$ and then induction can be used to prove this as demonstrated in Theorem A.1. Induction does not seem to help to discover the fact that the sum of the first $n$ numbers is $n(n+1)/2$; it merely serves to verify it.

Can induction do better? In many cases, it can, as we show as follows. Suppose we want to solve a slightly easier problem that occurs very often in analysis of algorithms: we want a good bound (not the exact value) on some sum or, more generally, some function of $n$. To be concrete, suppose we want to show an upper bound on the sum of the first $n$ numbers (which actually arises in analysis of algorithms for sorting and selection) and assume that we don’t know the exact formula. We want to show that the bound is $O(n^a)$ which usually suffices for algorithm analysis. We want a tight upper bound as possible, i.e., we want $a$ to be as small as possible. We know from Theorem A.1 that it is true for $a = 2$ (since $n(n+1)/2 \leq n^2$), but let’s use induction to find this.

Let’s set up the statement to be shown:

$$0 + 1 + 2 + \cdots + n \leq cn^a$$

where, $c$ and $a$ are fixed constants. We want to find the the smallest $a$ for which the above statement is true ($c$ can be any fixed constant).

Base step: $n = 0$. The statement is clearly true for any fixed $a$ and $c$.

Induction step: Assume the statement is true for $k$. We will prove that it is true for $k + 1$ for a suitable $a$.

By induction hypothesis, $\sum_{i=0}^{k} i \leq ck^a$ and thus,

$$\sum_{i=0}^{k+1} i = \sum_{i=0}^{k} i + k + 1 \leq ck^a + k + 1.$$  \hspace{1cm} (A.2)

For the induction step, we need to show that

$$ck^a + k + 1 \leq c(k + 1)^a.$$  \hspace{1cm} (A.2)

The above inequality will not be satisfied for $a = 1$, since that will imply that $k + 1 \leq c$ which is clearly not true (the left hand side increases with $k$, while the right hand side is a constant). Let’s try $a = 2$. Plugging $a = 2$ into the inequality (A.2) and simplifying yields $\frac{k+1}{2k+1} \leq c$, which is true for all $k$ if $c = 1$. Hence if we take $c = 1$ and $a = 2$, the induction step is satisfied. Thus, we have shown, in fact, derived, that $\sum_{i=0}^{n} i \leq n^2$. 

A.4 Worked Exercises

Worked Exercise A.1. Show using mathematical induction that any integer whose sum of digits is divisible by 3 is divisible by 3 as well.

Solution:
Base Case: \( n = 0 \): Trivially true.
Induction hypothesis: Assume the statement is true for all integers less than \( k \).
Induction step: We prove the statement that it is true for \( k \).

Consider the number \( k \) and let its representation in decimal form (digits) be \( x_m x_{m-1} \ldots x_1 x_0 \) — consisting of \( m+1 \) digits with \( x_0 \) being the least significant digit and \( x_m \) being the most significant. In other words, \( k = 10^m x_m + 10^{m-1} x_{m-1} + \cdots + 10 x_1 + x_0 \). Let’s denote the sum of the digits of number \( k \) to be \( s_k = x_m + x_{m-1} + \cdots + x_1 + x_0 \). Let’s assume that \( s_k \) is divisible by 3. Then we have to show that \( k \) is divisible by 3 as well.

Consider the integer \( k - 3 \). We have two cases:

Case 1: The last digit of \( k \), i.e., \( x_0 \) is greater than or equal to 3. Then the decimal representation of \( k - 3 \) is \( x_m x_{m-1} \ldots x_1 x_0 - 3 \) and the sum of its digits is \( s_{k-3} = x_m + x_{m-1} + \cdots + x_1 + x_0 - 3 = s_k - 3 \). Since \( s_k \) is divisible by 3 (by the above assumption), \( s_{k-3} \) is divisible by 3 as well. Hence the sum of the digits of number \( k - 3 \) is divisible by 3. By induction hypothesis (which applies to numbers less than \( k \)), \( k - 3 \) is divisible by 3. Hence \( k - 3 + 3 = k \) is divisible by 3 as well.

Case 2: The last digit of \( k \), i.e., \( x_0 \) is less than 3. Then the decimal representation of \( k - 3 \) can be computed as follows. The idea is to figure out the digits of \( k - 3 \). Let \( x_i \) (\( i > 0 \)) be the least significant non-zero digit, i.e., it is the first digit starting from the left (except, \( x_0 \)) that is non-zero.

(Example: if \( k = 83001 \), then \( x_1 = 3 \), since that is the first non-zero digit from the left (apart from the leftmost digit). If we subtract 3 from 83001, then we get 82998, where \( x_i \) will be subtracted by 1, and then all 9s and then the last digit is gotten by subtracting 11 (10+1) from 3. This is because of the usual subtraction where we borrow 1 from \( x_i \) and then do it repeatedly till we reach \( x_0 \), that why have 9’s in between. Note that since the sum of the digits of 83001 is divisible by 3, the sum of the digits of 82998 is divisible by 3 as well. Next we generalize the above example.)

Then the sum of its digits of \( k - 3 \) is \( s_{k-3} = x_m + x_{m-1} + \cdots + x_1 - 1 + 9 + \cdots + 9 + 10 + x_0 - 3 = x_m + x_{m-1} + \cdots + x_1 + \cdots + x_0 - 1 + 9 + \cdots + 9 + 10 - 3 = s_k + 9 + \cdots + 9 + 9 - 3 \) which is divisible by 3 since \( s_k \) is divisible by 3 and the sum \( 9 + \cdots + 9 + 9 - 3 \) is divisible by 3 (since each term is). The rest of the proof is now the same as Case 1.

Hence the sum of the digits of number \( k - 3 \) is divisible by 3. By induction hypothesis (which applies to numbers less than \( k \)), \( k - 3 \) is divisible by 3. Hence
k - 3 + 3 = k is divisible by 3 as well.

A.5 Exercises

Exercise A.1. Prove that the sum of the squares of the first \( n \) natural numbers is \( \frac{n(n+1)(2n+1)}{6} \).

Exercise A.2. Using induction, give a tight bound on the following sum: \( \sum_{i=1}^{n} i^d \), where \( d \) is a fixed constant.

Exercise A.3. Prove by mathematical induction the following statement:

\[
\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}
\]

where \( a \neq 1 \) is some fixed real number.

By the way, this is the sum of a geometric series, a useful formula that comes across often in algorithmic analysis.

Exercise A.4. Use mathematical induction to prove that \( x^n - y^n \) is divisible by \( x - y \) for all natural numbers \( x, y (x \neq y) \), and \( n \). (Hint: You will need strong induction.)

Exercise A.5. Show that for all natural numbers \( n \geq 1 \), \( \binom{2n}{n} \geq 2^n \). Note that \( \binom{2n}{n} = \frac{(2n)!}{n!n!} \).

Exercise A.6. Consider the set of first \( 2n \) positive integers, i.e., \( A = \{1, 2, \ldots, 2n\} \).

Take any subset \( S \) of \( n + 1 \) distinct numbers from set \( A \). Show that there are two numbers in \( S \) such that one divides the other.
Appendix B

Basic Data Structures: Heap, Stack, Queue, and Union-Find

We discuss three basic data structures used in algorithm design: heap, stack, and queue. We also discuss the union-find data structure which is useful in certain algorithms such as Kruskal’s MST algorithm [12.1.1].

B.1 Priority Queue (Heap) Data Structure

In many applications, we want to maintain the highest (or lowest) priority item among a set of items that are dynamically changing — i.e., items are added and deleted. Priority queue (also commonly called as Heap) is a data structure that supports four operations:

- INSERT(P, k): Insert key k (along with its data) into priority queue P.
- MIN(P): Return the smallest key (along with its data) in P.
- EXTRACT-MIN(P): Return and remove the smallest key (along with its data) from P.
- DECREASE-KEY(P, k, v): Decrease the value of key k to value v.

The above is called a MIN heap. Alternatively, we can define a MAX heap which has INSERT, MAX, EXTRACT-MAX, INCREASE-KEY operations.

B.1.1 Implementing a priority queue

One simple way is using an array: store the keys in an array. There are two ways to store the keys in an array — sorted or unsorted which determines the time
needed for the 3 operations. Assume that there are \( n \) keys currently in the array \( A \) — \( A[1], A[2], \ldots, A[n] \). (We will assume that keys are all distinct.)

We now perform an INSERT, MIN, or EXTRACT-MIN operation. The costs are:

- **Unsorted array**: INSERT \( (O(1)) \), MIN \( (O(1)) \), EXTRACT-MIN \( (O(n)) \) — Why?

- **Sorted array**: INSERT \( (O(n)) \), MIN \( (O(1)) \), EXTRACT-MIN \( (O(1)) \) — Why?

As we will see next, a heap gives a much better way to perform the above operations.

### B.1.2 Binary heap

A binary heap store the keys in a **complete binary tree**, where each node stores one key. We recall that a binary tree is a tree where each node has \textit{at most two} children; it is \textit{complete} if all levels are “full” except (possibly) the last level, i.e., nodes in all levels, (except possibly the last) will have 2 children. The last level will be filled from the left. Figure B.1 shown a example of a complete binary tree.

The main property of the binary (min) heap is that the keys are stored so that they satisfy the following (min) heap property: The key stored in any node will be \textit{smaller} than the keys stored in its children (if any).

Figure B.2 shows an example of a (min) binary heap.
Figure B.2: A binary heap.

### B.1.3 Representing a binary heap

A binary heap can be conveniently represented using an array. The indices of the vector capture the parent-child relationship. Let $A$ be a vector. The number of the elements in the vector is given by $A.size()$. The root of the binary heap is stored in $A[0]$. Given element $A[i]$, the children of this element are stored in $A[2i+1]$ and $A[2i+2]$, if they exist. The left child of $i$ denoted as $left(i) = A[2i+1]$. The right child of $i$ denoted as $right(i) = A[2i+2]$. The parent of $A[i]$ is stored in $A[\lfloor \frac{i-1}{2} \rfloor]$. To navigate the tree we follow the corresponding indices of the vector. See Figure B.3.

### B.1.4 Maintaining the heap property

We use a recursive algorithm called MIN-HEAPIFY to maintain the heap property. It takes as input a vector $A$ and an index $i$ into the vector. MIN-HEAPIFY assumes that the binary trees rooted at $left(i)$ and $right(i)$ are min-heaps. But $A[i]$ might be larger than the values at its children — violating the min-heap property. MIN-HEAPIFY moves the value of $A[i]$ down in the min-heap so that the subtree rooted at index $i$ obeys the min-heap property. MIN-HEAPIFY will be used to build a heap and also implement the priority queue operations.

The pseudocode for MIN-HEAPIFY is given in Algorithm B.3. For an illustration see Figure B.4.
Figure B.3: Figure shows the indices of an array (from 0 to 9) corresponding to the nodes of a binary heap.

Running time of MIN-HEAPIFY. It is easy to show that the time taken by MIN-HEAPIFY($A, i$) is proportional to the height of node $i$ in the binary tree. If there are $n$ elements in the heap, the height of the complete binary tree is $O(\log n)$. Hence MIN-HEAPIFY takes $O(\log n)$ time in the worst case.

B.1.5 Building a heap

Suppose we are given $n$ elements in an array and we want to make it into a binary heap. We can do this by calling MIN-HEAPIFY in a bottom-up manner. The pseudocode is given in Algorithm 4. The main idea of the algorithm is as follows. Represent the elements of the array as a binary tree (B.1.3). That is, the root of the binary heap is $A[0]$ and for $i \geq 0$, the children of node $A[i]$ are stored in $A[2i+1]$ and $A[2i+2]$, if they exist. Initially, the values in the binary tree need not be organized as a heap. The algorithm MIN-HEAPIFY, will organize the elements to satisfy the (min) heap property starting from the leaves. The leaves trivially satisfy the heap property. As the algorithm works on nodes at higher levels, the nodes at the lower levels — and hence the subtree rooted at these nodes — already satisfy the heap property. Thus the conditions for MIN-HEAPIFY are satisfied. The formal correctness is argued next.
Figure B.4: Illustrating Min-Heapify. The algorithm fixes the heap property at a node if it is violated. In this example, there is a violation at node 16 which is fixed.
334APPENDIX B. BASIC DATA STRUCTURES: HEAP, STACK, QUEUE, AND UNION-FIND

Algorithm 43 Algorithm MIN-HEAPIFY(A, i). Input: Array A and index i, trees rooted at left(i) and right(i) are min-heaps. Output: Tree rooted at i is a min heap.

1: l = left(i); r = right(i)
3:     smallest = l
4: else
5:     smallest = i
7:     smallest = r
8: if smallest ≠ i then
10: MIN-HEAPIFY(A, smallest)


1: n = A.size()
2: for i = n to 1 do
3:     MIN-HEAPIFY(A, i)

Correctness Consider the complete binary tree representation of the array. MIN-HEAPIFY fixes the heap property level by level starting from the last level and going up. Thus when MIN-HEAPIFY is called at a node i, the subtrees rooted at left(i) and right(i) already satisfy the heap property. When MIN-HEAPIFY is finally called at the root, then whole array becomes a min heap.

Running time of Algorithm Build-Min-Heap It is easy to see to upper bound the run time of Algorithm Build-Min-Heap as O(n log n). Actually, by a more careful analysis, it can be shown to be faster — takes O(n) time. We will bound the total time taken by all the MIN-HEAPIFY calls. The time taken by MIN-HEAPIFY(A, i) is proportional to the height of node i (note that leaves are at height 0). Hence total time is bounded by:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

since there are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes at height h. Hence,

$$O(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}).$$
The sum $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$. Hence,

$$O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n).$$

### B.1.6 Implementing Priority Queue Operations

The operations INSERT, MIN, EXTRACT-MIN, and DECREASE-KEY can be implemented using a binary heap. Given a binary heap $A$, we implement the above operations as follows.

- **MIN($A$)** is simply returning $A[0]$. This takes constant time.
- **EXTRACT-MIN($A$)**: Takes $O(\log n)$ time.
  1. $min = A[0]$.
  3. Delete (last) element $A[A.size()]$ from vector.
  4. MIN-HEAPIFY($A$, 0).
  5. Return min.
- **INSERT(($A$, $k$))**: Insert key $k$ into $A$. Exercise B.1 asks how this can be done in $O(\log n)$ time.
- **DECREASE-KEY(($A$, $k$, $v$))**: Decrease the value of key $k$ to $v$. Exercise B.1 asks how this can be done in $O(\log n)$ time. Note that for this operation, we assume that pointer to key $k$ in the binary heap (i.e., its index in the array) is given as part of the input (one need not search for the key which can take linear time).

**Exercise B.1.** Show how an element can be inserted in a binary heap in $O(\log n)$ time, where $n$ is the number of elements in the heap. The heap property should be satisfied after the insertion. Show also how to perform DECREASE-KEY in $O(\log n)$ time. Assume that pointer to key $k$ in the binary heap (i.e., its index in the array) is given as part of the input of the DECREASE-KEY operation.

### B.2 Stack Data Structure

Stack is a data structure that supports the following operations:

1. **INSERT($S$, $k$) or PUSH($S$, $k$)**: Insert or push key $k$ into stack $S$.
2. **DELETE($S$) or POP($S$)**: Return and delete the most recently inserted key.
APPENDIX B. BASIC DATA STRUCTURES: HEAP, STACK, QUEUE, AND UNION-FIND

3. EMPTY($S$): Return true if $S$ is empty, otherwise return false.

A stack can be easily implemented using an array so that all the above operations take $O(1)$ time. A stack is a Last In First Out (LIFO) data structure, i.e., the item that is inserted last is the first one to be output (popped).

A stack is useful in a variety of algorithmic applications — a notable application is that they are useful in iteratively implementing recursive algorithms. In fact, a compiler typically implements a recursive code by converting it into an iterative code — stack is the data structure that is used. In particular, a recursive algorithm can be converted into an iterative algorithm and the stack data structure usually plays a useful role in this conversion. A good illustration of this depth first search which is naturally stated as a recursive algorithm. This can be converted into an iterative algorithm using a stack. In practice, iterative algorithms are faster, since there is overhead involved in implementing recursive algorithms (since the compiler has to convert them into iterative algorithms).

Exercise B.2. Show how to implement all the stack data structure operations in $O(1)$ time.

B.3 Queue Data Structure

A queue is a data structure that supports the following operations:

1. INSERT($Q$, $k$) or ENQUEUE($Q$, $k$): Insert or push key $k$ into Queue $Q$.
2. DELETE($Q$) or DEQUEUE($Q$): Return and delete the least recently inserted key, i.e., the key that was inserted the earliest.
3. EMPTY($Q$): Return true if $Q$ is empty, otherwise return false.

A queue is a First-in-First-Out (FIFO) data structure, i.e., the item that is inserted first is the first one to be output (dequeued).

B.3.1 Implementing a Queue

A Queue can be implemented using a vector (or an array) so that all the above operations take $O(1)$ time. However, one has to be careful about space used. Let's say we use a vector $A$ to implement a Queue: simply keep the elements in the order of insertion and keep track of the indices of the first and last element. ENQUEUE will insert the element as the last element in the vector. DEQUEUE will remove and return the first element in the vector; update the index of the first element. The problem with the above implementation is that space is wasted at the beginning of the array when elements are dequeued. Exercise B.4 asks
you a way to address the problem. Exercise B.5 asks you to give an alternate implementation of a Queue.

B.4 Union-Find Data Structure for Disjoint Sets

The union-find data structure is used to maintain a dynamic collection of disjoint sets. Each set has a unique representative (an arbitrary member of the set). The data structure supports the following three efficient operations:

- **Make-Set**(x) - Create a new set with one member x.
- **Union**(x, y) - Combine the two sets, represented by x and y into one set.
- **Find-Set**(x) - Find the representative of the set containing x.

We maintain each disjoint set as a directed rooted tree, rooted at some node belonging to the tree. See Figure B.4. Each node in the directed tree points to its parent (the root points to itself). The root of the tree is the representative of the set. We implement the data structure operations as follows:

- **Make-Set**: creates a rooted tree with just one node which points to itself (Algorithm 45).
- **Find-Set**: Given a pointer to an element u, find(u), simply follows parent pointers to the root, and returns the root which is the representative of the set that u belongs to (Algorithm 46).
- **Union**: Given two representative elements u and v (i.e., these are the roots of the respective trees), union(u, v), makes the root of the tree with a fewer number of nodes point to the root of the tree with the larger number of nodes (if the two trees have the same number of nodes, then one of the roots points to the other). The new root will be the representative of the combined set. This strategy of making the root of the smaller tree point to the larger is called union-by-rank heuristic (Algorithm 47). Note that instead of maintaining the size of the tree, the union-by-rank heuristic uses a parameter called rank which is equal to the height of the tree. Hence, the root of the tree of smaller height is made a child of the root of the tree with larger (or equal) height.

**B.4.1 Analysis of Union and Find operations**

We show that implementations of the union and find operations as given in Algorithms 47 and 46 respectively are efficient. In particular, any union or find operation on a set of a total of n elements takes O(log n) time. To show this we need
Figure B.5: Union-Find data structure as a collection of disjoint rooted trees. (Top) Make-Set creates $n$ singleton trees, where $n$ is the total number of elements in the set. (Middle) Two trees representing two sets after some union operations. Find operation finds the root of the tree that the node belongs to by parent pointer chasing to the root. (Bottom) Combing the two trees via union-by-rank heuristic attaches the tree with smaller height to a tree of larger height.
B.4. UNION-FIND DATA STRUCTURE FOR DISJOINT SETS

Algorithm 45 Algorithm Make-Set(x). *Input:* An element *x*. *Output:* A singleton set with element *x* represented as a tree with one node.

1: // Initialize parent and rank of each element.
2: \textit{parent}[x] = x
3: \textit{rank}[x] = 0.

Algorithm 46 Algorithm find(x). *Input:* An element *x* *Output:* Representative of the set containing *x*, i.e., root of the directed tree containing *x*.

1: if \( x \neq \text{parent}[x] \) then
2: \hspace{1em} return \( \text{find}(\text{parent}[x]) \)
3: else
4: \hspace{1em} return \( x \)

To only show that any find operation takes \( O(\log n) \) time. This is because, apart from the two find operations in the union algorithm, the rest of the algorithm (attaching the pointer of one root to the other) takes only constant time.

To show that any find operation takes only \( O(\log n) \) time, we show that the height (i.e., the rank) of every rooted tree does not exceed \( \log n \). The main idea for this is as follows. Note that initially the height of all trees is 0, since each element is a singleton tree. The height of a tree increases only when it is combined with another tree — note that this happens during the union operation and this happens only when the two trees that are combined have the same rank (Line 11 of Algorithm 47). We show that whenever the height increases the size of the tree, i.e., the number of nodes in the tree doubles. Since there are only \( n \) nodes, and each tree starts with one node, the doubling can happen only \( \log n \) times till we include all \( n \) nodes. Hence the height of any tree is bounded by \( \log n \). The following lemma shows the formal proof by induction.

**Lemma B.1.** Let \( \text{size}(x) \) be the number of nodes in the tree rooted at *x*. Then, for any tree with root \( x \), \( \text{size}(x) \geq 2^{\text{rank}[x]} \). Furthermore, the height of a tree is the same as its rank.

**Proof.** Proof is by induction on number of union operations, the only operation which changes the rank (height) or size of its tree.

- **Base case:** True for singleton trees, since ranks are initialized to 0 and size is 1.
- **Inductive step:** Assume that the lemma holds before performing union(\( x, y \)). Let \( \text{rank} \) denote the rank just before the union and let \( \text{rank}' \) denote the rank immediately following the union. Similarly, let \( \text{size} \) and \( \text{size}' \) be the size before and after the union, respectively.
Algorithm 47 Algorithm Union(x, y). \textbf{Input}: Elements x and y belonging to different sets maintained as disjoint directed rooted trees. \textbf{Output}: A union of the two sets containing x and y.

1: \texttt{r(x) = find(x)}
2: \texttt{r(y) = find(y)}
3: \texttt{if r(x) \neq r(y) then}
4: \quad // if they belong to different sets
5: \quad \texttt{if rank[x] > rank[y] then}
6: \quad \quad union-by-rank heuristic: make the tree with smaller rank, the child of the one with larger rank
7: \quad \quad \texttt{parent[y] = x}
8: \quad \texttt{else}
9: \quad \quad \texttt{parent[x] = y}
10: \quad \texttt{if rank[x] = rank[y] then}
11: \quad \quad \texttt{rank[y] = rank[y] + 1 //increment rank if both trees are of same rank}

We first address the case, \(\text{rank}[x] \neq \text{rank}[y]\). Without loss of generality, assume \(\text{rank}[x] < \text{rank}[y]\). Node y would be the root of the resulting tree and:

\[
\text{size}'(y) = \text{size}(x) + \text{size}(y) \\
\geq 2^{\text{rank}[x]} + 2^{\text{rank}[y]} \\
\geq 2^{\text{rank}[y]} \\
= 2^{\text{rank}'[y]}
\]

The other case is that \(\text{rank}[x] = \text{rank}[y]\). Node y would be the root of the new tree, and:

\[
\text{size}'(y) = \text{size}(x) + \text{size}(y) \\
\geq 2^{\text{rank}[x]} + 2^{\text{rank}[y]} \\
\geq 2^{\text{rank}[y]+1} \\
= 2^{\text{rank}'[y]}
\]

\[\square\]

\textbf{Theorem B.1.} The height of any tree in the collection of rooted trees representing a collection of disjoint sets of n elements is \(O(\log n)\). Thus any find or union operation takes \(O(\log n)\) time.

\textbf{Proof.} By Lemma [B.1] since the number of nodes in a tree rooted at x, \text{size}(x) is
B.5. EXERCISES

at least $2^{\text{rank}[x]} = 2^\text{height}[x]$, where \( \text{height}[x] \) is the height of the tree rooted at \( x \). Hence \( \text{height}[x] \leq \log(\text{size}[x]) \leq \log n \), since the maximum size can be at most \( n \), the total number of elements.

Since the find operations involve pointer chasing to the root, it is bounded by the height of the tree which is \( O(\log n) \). Union operations involve two find operations and other operations which takes constant time, hence it takes also \( O(\log n) \) time.

B.5 Exercises

Exercise B.3. Given a list of \( n \) numbers, the median is the number of rank \( \lceil n/2 \rceil \), i.e., it is the \( \lceil n/2 \rceil \)th smallest element. Suppose the list is changing dynamically. That is at every step, a number is added or deleted from the list. The goal is to output the median in the current list as fast as possible. In other words, at any point in time, we would like to answer the query: “What is the median of the current list of numbers?”. How fast can you answer this query?

Exercise B.4. Show how to efficiently implement a queue using a single array. Your array should not waste significant space. In particular, at any time, the total space used should be \( O(n) \) where \( n \) is the (current) number of elements in the queue. Also, the time needed for all operations should be \( O(1) \) on the average, i.e., the total time time for \( k \) operations (which can be any of the three operations — INSERT, DELETE, EMPTY) divided by \( k \) should be \( O(1) \).

Exercise B.5. Show how to implement a (FIFO) Queue using two stacks. Analyze the running time of your Queue operations as follows:

1. What is the worst case time for ENQUEUE, DEQUEUE, and EMPTY?
2. If there are a total of \( m \) ENQUEUE and DEQUEUE operations (the number of DEQUEUE operations is not more than the number of ENQUEUE operations), what is the total time for all the \( m \) operations. For full credit, your total time should be \( O(m) \).

Exercise B.6. A strange number is one whose only prime factors are in the set \( \{2, 3, 5\} \). Give an efficient algorithm (give pseudocode) that uses a binary heap data structure to output the \( n \)th strange number. Explain why your algorithm is correct and analyze the run time of your algorithm. (Hint: Consider generating the strange numbers in increasing order, i.e., \( 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, \) etc. Show how to generate efficiently the next strange number using a heap.)

Exercise B.7. Consider the following geometric problem. You are given \( n \) points in the plane, \((x_1, y_1), \ldots, (x_n, y_n)\) and you want to determine whether there exist four
points that are the vertices of a square. If there are no such four points, report that no square exists. Otherwise report four points belonging to a square. If there exist more than one square, report the four points corresponding to the square of maximum area. Your algorithm should solve this problem in $O(n^2 \log n)$ time. Explain your algorithm and its implementation (explain what data structure operations will be used) and analyze the run time. You can assume that computing the distance between any two given points takes constant time. (Hint: Comparing all possible quadruplets of points will result in an $\Theta(n^4)$ time algorithm. To obtain the desired running time, you can only consider pairs of points.)

Exercise B.8. Show how you can sort $n$ numbers (given in a vector) using a binary heap. Your algorithm should use the heap operations. What is the (worst-case) running time of your algorithm? You should show that your time bound is tight, by giving an example where your algorithm will require that much time (asymptotically). In other words, if you say that the running time of your algorithm is $O(T(n))$, then you should give an $n$-size input where the algorithm actually takes $\Theta(T(n))$ time.
Appendix C

Probability

We recall basic definitions and concepts of probability theory. We also present various probabilistic inequalities and bounds that are used throughout the book. These prove very useful in the analysis of randomized algorithms. We refer to [4, 2] for more details on the concepts presented here.

C.1 Events, Probability, Probability Space

Consider an experiment with a finite (or countably infinite) number of outcomes. Each outcome is a simple event (or a sample point). The sample space is the set of all possible simple (elementary) events. An event \( E \) is a union of simple events — a subset of the sample space.

Two events are said to be **mutually exclusive** if \( A \cap B = 0 \).

With each simple event \( s \) we associate a number \( Pr(s) \) which is called as the **probability** of \( s \).


**Definition C.1.1 (Probability Space).** A probability distribution $Pr$ on a sample space $S$ is a mapping from events of $S$ to real numbers such that it satisfies the following three *axioms*:

- $Pr(A) \geq 0$ for any event $A$.
- $Pr\{S\} = 1$.
- For any (finite or countably infinite) sequence of pairwise mutually exclusive events $A_1, A_2, \ldots$:

$$Pr\{\bigcup_i A_i\} = \sum_i Pr\{A_i\}$$

The pair $(S, Pr)$ is called a discrete *probability space*.

The probability of any event $E$ can be computed as the sum of the probabilities of the simple events that it is composed of; this follows from the third axiom:

$$Pr(E) = \sum_{s \in E} Pr(s).$$

### C.2 Principle of Inclusion-Exclusion

The *inclusion-exclusion* principle gives a formula for computing the probability of the union of a set of events in terms of the probabilities of the individual events and their intersections.

**Statement C.1 (Inclusion-Exclusion Principle).** Let $E_1, E_2, \ldots, E_n$ be arbitrary events. Then

$$Pr(\bigcup_{i=1}^n E_i) = \sum_i Pr(E_i) - \sum_{i<j} Pr(E_i \cap E_j) + \Pr_{i<j<k}Pr(E_i \cap E_j \cap E_k) - \cdots + (-1)^{i+1} \sum_{i_1<i_2<\ldots<i_l} Pr(\bigcap_{r=1}^l E_{i_r}) + \ldots.$$

An important consequence of the inclusion-exclusion principle is a simple upper bound on the probability of the union of events known as *Boole’s inequality* or *union bound*.

**Statement C.2 (Boole’s inequality (union bound)).** For any arbitrary sequence of events $E_1, E_2, \ldots, E_n$:

$$Pr(\bigcup_{i=1}^n E_i) \leq \sum_i Pr(E_i)$$

The union bound is used often in upper bounding the occurrence of the union of a set of “bad” events. If this upper bound is small, then it implies that none of the bad events occur with high probability.
C.3. Conditional Probability

**Definition C.3.1** (Conditional Probability). Given two events \( A \) and \( B \) conditional probability of \( A \) given \( B \) is defined as follows:

\[
Pr(A \mid B) = \frac{Pr(A \cap B)}{Pr(B)}
\]

In the above definition, by conditioning on \( B \) we restrict the sample space to the set \( B \). Thus the conditional probability can be considered as \( Pr(A \cap B) \) “normalized” by \( Pr(B) \).

Given two events \( E_1 \) and \( E_2 \), Bayes’ rule relates the conditional probability of the first given the second, to the conditional probability of the second given the first. This is useful to infer one conditional probability from the other.

**Statement C.3** (Bayes’ rule).

\[
Pr(E_1 \mid E_2) = \frac{Pr(E_1 \cap E_2)}{Pr(E_2)} = \frac{Pr(E_2 \mid E_1) Pr(E_1)}{Pr(E_2)}
\]

The following are some useful identities that involve computing the probability of the intersection of many events.

**Statement C.4.**

- \( Pr(A \cap B) = Pr(A \mid B)Pr(B) \).
- \( Pr(A \cap B \cap C) = Pr(A \mid B \cap C)Pr(B \cap C) = Pr(A \mid B \cap C)Pr(B \mid C)Pr(C) \).
- The following is a generalization of the above identity. Let \( A_1, \ldots, A_n \) be a sequence of events. Let \( E_i = \bigcap_{j=1}^i A_i \). Then
  \[
  Pr(E_n) = Pr(A_n \mid E_{n-1}) Pr(E_{n-1}) = Pr(A_n \mid E_{n-1}) Pr(A_{n-1} \mid E_{n-2}) \ldots Pr(A_2 \mid E_1) Pr(A_1).
  \]

A fundamental concept that follows from conditional probability is that of **independence**.

**Definition C.3.2** (Independence of events). Two events \( A \) and \( B \) are said to be independent if

\[
Pr(A \cap B) = Pr(A) \times Pr(B),
\]

or (when \( Pr(B) > 0 \))

\[
Pr(A \mid B) = \frac{Pr(A \cap B)}{Pr(B)} = Pr(A).
\]
C.4 The Birthday Paradox

We study a problem called the “Birthday Paradox”, that arises often in algorithm design and analysis. The problem also serves to illustrate basic concepts in probability theory.

Question: What is the probability that among \(m\) people no two have the same birthday? We make the following two assumptions: (1) All birthdays are equally likely and (2) Birthdays are independent events.

To compute the above probability, we first define the sample space of the experiment. The sample space is the set of all vectors \(S = \{(b_1, \ldots, b_m) | b_i \in [1, \ldots, N]\}\), where \(b_i\) denotes the birthday of the \(i\)th person, and \(N\) is the total number of different birthdays (\(N = 365\) in Earth). We need to compute \(\Pr(E)\) where the event \(E = \{(b_1, \ldots, b_m) | b_i \neq b_j \text{ for all } i \neq j\}\), i.e., the event \(E\) is the set of all events where no two birthdays are the same.

How many different atomic events are counted in \(E\)? The number of possible \(m\) different birthdays is \(N(N-1)(N-2)\ldots(N-m+1)\). Hence,

\[
\Pr(E) = \frac{N(N-1)(N-2)\ldots(N-m+1)}{N^m} = \prod_{i=0}^{m-1} \left(1 - \frac{i}{N}\right)
\]

\[
\leq \prod_{i=0}^{m-1} e^{-i/N} = e^{-\sum_{i=0}^{m-1} i/N} = e^{-m(m-1)/2N}.
\]

For \(m = \sqrt{2N} + 1 \leq 28\), \(\Pr(E) < 1/e < 1/2\).

The apparent “paradox” in this problem is that significantly less number of people, i.e., only about \(\sqrt{N}\) people (which is much smaller than \(N\)), are needed to have a good chance (about 50%) to have two people to have a common birthday.

Alternate Analysis Assume that we choose one birthday after the other independently and uniformly at random from \([1 \ldots N]\). Let the event \(E_i\) denote: "the \(i\)th choice is different from the first \(i-1\) choices". Then, we compute:

\[
\Pr(E) = \Pr(\cap_{i=1}^m E_i) = \Pr(E_1) \Pr(E_2 | E_1) \Pr(E_3 | E_2 \cap E_1) \ldots \Pr(E_m | \cap_{i=1}^{m-1} E_i)
\]

\[
= (1 - 1/N)(1 - 2/N)\ldots(1 - (m-1)/N)
\]

\[
= \prod_{i=1}^m \left(1 - \frac{i-1}{N}\right).
\]

This gives the same result as before.

This analysis uses a very useful principle called the Principle of Deferred Decisions. In this principle, the idea is to defer the fixing of choices of events to “when they are needed” and not a priori (i.e., all at once). Typically, this type
of analysis will use conditional probabilities. In the above analysis, to compute \( \Pr(E) \), we first computed \( \Pr(E_1) \), then \( \Pr(E_2|E_1) \) and so on. At each stage, we only fixed the event that determined the respective conditional probabilities, i.e., first \( E_1 \) which fixes \( b_1 \), and then the event \( E_2 \) conditioned on \( E_1 \), which fixes \( b_2 \neq b_1 \), and so on.

### C.5 Random Variable

Random variable is a very useful concept in probability; it allows one to quantify the parameters associated with the events in a probability space as we see fit to model.

Let \((S, \Pr)\) be a discrete probability space. Let \( V \) be a set of values. A random variable \( X \) defined on \((S, \Pr)\) is a function

\[
X : S \rightarrow V.
\]

The random variable \( X \) allows one to “transfer” the probability function that is defined on the set of events of \( S \) to one that is defined on the set of values \( V \), the range of \( X \) in the following manner. Let \( \mathcal{E}(r) = \{ s \in S \mid X(s) = r \} \). Then we can define:

\[
\Pr(X = r) = \Pr(\mathcal{E}(r)) = \sum_{s \in \mathcal{E}(r)} \Pr(s).
\]

Two random variables \( X \) and \( Y \) (defined on the same sample space) are called independent if for all \( x \) and \( y \)

\[
\Pr\{X = x \text{ and } Y = y\} = \Pr\{X = x\} \Pr\{Y = y\}
\]

### C.6 Expectation of a random variable

Random variable is a function (thus “variable” is actually a misnomer!), and thus it can take any value in its range. The probability function \( \Pr \) gives the probability that each such value is taken, i.e., it gives the probability distribution over the values in \( V \). In many applications, we are less interested in the entire distribution of the probability over the values, but rather than a single (or few) parameter(s) that characterizes the distribution. A standard such parameter is the expectation, also called as the mean or the average of the random variable.
Definition C.6.1 (Expectation of a random variable). The expectation of a discrete random variable \( X \) is

\[
E[X] = \sum_{i \in \text{range}(X)} i \Pr(X = i).
\]

Thus, the expectation is a weighted sum over all possible values of the random variable, weighted by the corresponding probabilities of the values.

A very useful property that comes in handy in computing the expected value of a random variable is the linearity property of expectation stated as follows.

Theorem C.1 (Linearity of Expectation). Given a sequence of random variables, \( X_1, X_2, \ldots, X_n \), we have

\[
E \left[ \sum_{i=1}^{n} X_i \right] = \sum_{i=1}^{n} E[X_i].
\]

The linearity property says that the expectation of the sum of random variables is equal to the sum of their individual expectations. Note that this is true, regardless whether the random variables are independent or not; in fact, it is the latter case that makes this property so useful in many applications. In many applications, this is commonly used to compute the expectation of a random variable (which is not directly easy to compute by applying the Definition C.6.1) by decomposing it into a sum of random variables whose expectations are easier to compute. A typical example is decomposing the random variable into a set of indicator (also called Bernoulli or 0-1) random variables, which take only two values 0 and 1. The following example illustrates this paradigm.

Problem C.1. Assume that \( N \) customers checked their jackets in a restaurant. The jackets get mixed and while leaving the restaurant, each customer gets a random jacket, i.e., probability of getting any particular jacket is \( 1/N \). The goal is to compute the expected number of customers who get their own jacket.

Solution: It is difficult to compute \( E[X] = \sum_{k=0}^{N} k \Pr(X = k) \). Instead, we define \( N \) indicator random variables \( X_i \), where \( X_i = 1 \) iff \( i \) got her own jacket. That is, \( X_i \) “indicates” whether customer \( i \) gets her own jacket or not. It is much simpler to compute \( E[X_i] \) for any \( i \), since it takes only two values 0 and 1. Note that \( X_i \)'s are not independent, but we can still apply linearity of expectation.

\[
E[X_i] = 1 \cdot Pr(X_i = 1) + 0 \cdot Pr(X_i = 0) = Pr(X_i = 1) = \frac{1}{N},
\]

\[
E[X] = \sum_{i=1}^{N} E[X_i] = 1.
\]
Problem C.2. Let \( a_1, a_2, \ldots, a_n \) be a sequence of \( n \) values. Each value \( a_i \) is independently and randomly chosen from a fixed distribution \( D \) (e.g., uniform distribution from the real interval \([0, 1]\)). Let \( m_i = \min\{a_1, a_2, \ldots, a_i\} \), i.e., the minimum of the first \( i \) values. Let \( r.v. Y \) denote the number of times the minimum value is updated, i.e., the number of times \( m_i \neq m_i + 1 \). Then \( E[Y] = O(\log n) \).

Solution: First, without loss of generality we can assume that all the values are distinct, since we are upper bounding \( Y \). Indeed, when a value repeats, there won’t be any update.

Let the indicator random variable \( Y_i \) denote the event that \( m_i \) is updated. The value \( m_i \) will be updated if \( a_{i+1} \) is the minimum value among the first \( i + 1 \) values. The probability that the above event happens is \( \frac{1}{i+1} \). This is because each of the first \( i + 1 \) values are chosen independently from the same distribution, and assuming that values are distinct, the probability that a particular value, (i.e., \( a_{i+1} \)) is the minimum is \( \frac{1}{i+1} \). Hence \( E[Y_i] = \Pr(Y_i = 1) = \frac{1}{i+1} \).

We have \( Y = \sum_{i=1}^{n-1} Y_i \). Thus, \( E[Y] = \sum_{i=1}^{n-1} E[Y_i] = \sum_{i=1}^{n-1} \frac{1}{i+1} \leq H_n = \Theta(\log n) \).

Note that \( H_n \) is the harmonic mean, defined as \( \sum_{i=1}^{n} 1/i \) and is \( \Theta(\log n) \).

C.7 Useful Types of Random Variables

We define two types of random variables that arise often in probabilistic analysis of algorithms — the binomial and the geometric random variables.

C.7.1 Binomial Random variables

Let \( X \) be a 0-1 random variable such that

\[
Pr(X = 1) = p, \quad Pr(X = 0) = 1 - p.
\]

Then:

\[
E[X] = 1 \cdot p + 0 \cdot (1 - p) = p.
\]

Consider a sequence of \( n \) independent Bernoulli trials \( X_1, \ldots, X_n \). Let \( X = \sum_{i=1}^{n} X_i \).

Then we say that the random variable \( X \) has a Binomial distribution with parameters \( n \) and \( p \) denoted as \( X \sim B(n, p) \). Equivalently, \( X \) denotes the number of successes obtained when doing \( n \) independent trials where each trial has a probability of success \( p \). The probability of getting \( k \) successes is given by:

\[
Pr(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}.
\]
and the mean is:

\[ E[X] = np. \]

### C.7.2 The Geometric Distribution

Assume that an experiment has probability \( p \) for success and \( 1 - p \) for failure. How many trials do we need till the first success? The number of trials needed is a random variable and is said to have the geometric distribution. Geometric random variable arises often in probabilistic analysis of algorithms, typically in the following fashion. A randomized step will be shown to succeed with some probability \( p \). The quantity that will be interest will be number of such steps needed to get one (or more) success(es).

Thus a random variable \( X \) is said to be geometrically distributed if:

\[ Pr(X = i) = (1 - p)^{i-1}p. \]

We say \( X \) has a geometric distribution with parameter \( p \). Clearly, this is a probability distribution, as the probabilities sum to 1:

\[
\sum_{i=1}^{\infty} Pr(X = i) = \sum_{i=1}^{\infty} (1 - p)^{i-1}p = p \frac{1}{1 - (1 - p)} = 1.
\]

Assume that \( X \) get values in \( \mathcal{N} \).

\[
E[X] = \sum_{i \geq 0} i Pr(X = i) = \sum_{i \geq 1} Pr(X \geq i).
\]

Let \( X \sim G(p) \). Then, the expectation of \( X \) is:

\[
E[X] = \sum_{i \geq 1} \text{Prob}(X \geq i) = \sum_{i \geq 1} (1 - p)^{i-1} = \frac{1}{1 - (1 - p)} = \frac{1}{p}.
\]

The geometric distribution is said to be memoryless: For any \( k > r \),

\[
Pr(X > k \mid X > r) = \frac{(1 - p)^k}{(1 - p)^r} = (1 - p)^{(k-r)} = Pr(X > (k - r)).
\]
C.8  Bounding Deviation of a Random Variable from its Expectation

In many applications, we are interested in bounding the deviation of a random variable from its expectation. This is useful in bounding the probability of success (or failure) of an algorithm. The typical way to do this is to first compute the expectation, and then showing that with good probability the random variable does not deviate “too far” from its expectation, i.e., it is “concentrated” around its mean.

C.8.1  Markov Inequality

There are two basic deviation inequalities: Markov and Chebyshev. Markov’s inequality uses only the value of the expectation (also called “first moment”). Hence it can be quite weak.

**Theorem C.2.** [Markov Inequality] For any non-negative random variable and for any $a > 0$

$$
\Pr(X \geq a) \leq \frac{E[X]}{a}.
$$

**Proof.** For any $a > 0$, let $I$ be an indicator r.v. for the event $X \geq a$. Then $I \leq X/a$. Taking expectations on both sides, we get $E[I] = \Pr(X \geq a) \leq E[X]/a$.  

Chebyshev’s inequality gives a stronger bound, which assumes that the variance (or the “second moment”) or standard deviation of the random variable is known. These are defined below.

**Definition C.8.1.** The variance of a random variable $X$ is

$$Var[X] = E[(X - E[X])^2].$$

The standard deviation of a random variable $X$ is

$$\sigma(X) = \sqrt{Var[X]}.$$

**Theorem C.3** (Chebyshev’s Inequality). For any random variable

$$
\Pr(|X - E[X]| \geq a) \leq \frac{Var[X]}{a^2}.
$$

**Proof.**

$$
\Pr(|X - E[X]| \geq a) = \Pr((X - E[X])^2 \geq a^2).
$$
APPENDIX C. PROBABILITY

By Markov inequality,
\[ \Pr((X - E[X])^2 \geq a^2) \leq \frac{E[(X - E[X])^2]}{a^2} = \frac{\text{Var}[X]}{a^2}. \]

A useful special case of the Markov and Chebyshev’s inequalities is given by the following theorem. This is typically useful in showing whether a non-negative integer-valued random variable takes 0 value or otherwise.

**Theorem C.4** (First and Second Moment Inequalities). Let \( X \) be a non-negative integer-valued r.v. Then

\[ \Pr(X \geq 1) \leq E(X). \]
\[ \Pr(X = 0) \leq \frac{\text{Var}(X)}{(E(X))^2} = \frac{E(X^2)}{(E(X))^2} - 1. \]

C.8.2 Chernoff Bound for Sum of Indicator R.V’s

Chernoff bound is a standard tool to show deviation bounds for a random variable that can be expressed as a sum of independent indicator (or Bernoulli) random variables. Such a random variable typically comes in many algorithmic applications. We will state these bounds here and then prove them in Section C.9.1. Typically, in many algorithmic applications, the proofs are not needed to use these bounds.

One way to understand Chernoff bounds is from the point of view of the Binomial random variable \( B(n, p) \). As seen from Section C.7.1, the mean of the binomial distribution is \( \mu = np \). What is the probability that \( B(n, p) \) takes values that are far away from its mean? The probability that \( B(n, p) \) takes a value \( k > \alpha \mu \), where \( \alpha > 1 \) is some parameter is:

\[ \Pr(B(n, p) > \alpha \mu) = \sum_{k=\lceil \alpha \mu \rceil}^{n} \Pr(B(n, p) = k) = \sum_{k=\lceil \alpha \mu \rceil}^{n} \binom{n}{k} p^k (1 - p)^{n-k}. \]

This probability is called the “upper tail”, i.e., the probability of deviating above the mean. Analogously, one can write an expression for calculating the “lower tail”, i.e., the probability of deviating below the mean. The above formula does not directly give a useful way of quantifying the tail probabilities; as such it is quite a cumbersome sum to manipulate. On the other hand, Chernoff bound approximates the above sum in a way that is very useful to interpret and apply in applications. Moreover, the Binomial tail bounds are a special case of Chernoff, where the success probabilities of all trials are the same (equal to \( p \); more
C.8. BOUNDING DEVIATION OF A RANDOM VARIABLE FROM ITS EXPECTATION

generally, in Chernoff, each trial can have a different success probability — see Theorem C.5

Theorem C.5 (Chernoff bounds). Let \( X_1, X_2, \ldots, X_n \) be independent indicator random variables such that, for \( 1 \leq i \leq n \), \( \Pr[X_i = 1] = p_i \), where \( 0 < p_i < 1 \).

Define \( X = \sum_{i=1}^{n} X_i \) and let \( \mu = E[X] = \sum_{i=1}^{n} p_i \).

1. Upper Tail Bounds:
   a. For any \( \delta > 0 \),
      \[
      \Pr (X > (1 + \delta)\mu) < \left( \frac{e^{\delta}}{(1 + \delta)^{(1+\delta)}} \right)^{\mu}
      \]
   b. For \( 0 < \delta < 1 \),
      \[
      \Pr (X > (1 + \delta)\mu) \leq e^{-\mu \delta^2 / 3}
      \]
   c. For \( R \geq 6\mu \),
      \[
      \Pr(X \geq R) \leq 2^{-R}
      \]

2. Lower Tail Bound:
   For \( 0 < \delta < 1 \),
   \[
   \Pr(X < (1 - \delta)\mu) < e^{-\mu \delta^2 / 2}
   \]

C.8.3 Chernoff-Hoeffding Bound for Sum of Indicator R.V’s

Sometimes the following variant of Chernoff bound is useful.

Theorem C.6 (Chernoff-Hoeffding bounds). Let \( X_1, X_2, \ldots, X_n \) be independent indicator random variables such that, for \( 1 \leq i \leq n \), \( \Pr[X_i = 1] = p \), where \( 0 < p < 1 \).

Define \( X = \frac{\sum_{i=1}^{n} X_i}{n} \) and let

\[
E[X] = E\left[\frac{\sum_{i=1}^{n} X_i}{n}\right] = \frac{\sum_{i=1}^{n} E[X_i]}{n} = \frac{n p}{n} = p.
\]

Then we have the following tail bounds:

Upper tail:
\[
\Pr(X > E[X] + \varepsilon) \leq e^{-2n \varepsilon^2}.
\]

Lower tail:
\[
\Pr(X < E[X] - \varepsilon) \leq e^{-2n \varepsilon^2}.
\]

Both tails:
\[
\Pr(|X - E[X]| > \varepsilon) \leq 2e^{-2n \varepsilon^2}.
\]
C.9  Moment Generating Function

The moment generating function \( M(t) \) of (discrete) random variable \( X \) is defined for all real values \( t \) by

\[
M(t) = E[e^{tX}] = \sum_x e^{tx} \Pr(X = x).
\]

All moments of \( X \) can be obtained from \( M(t) \) and then evaluating the result at \( t = 0 \), hence the name.

\[
M'(t) = \frac{d}{dt} E[e^{tX}] = E[\frac{d}{dt}(e^{tX})] = E[Xe^{tX}].
\]

Thus, \( M'(0) = E[X] \) and the \( n \)th derivative of \( M(t) \) is given by:

\[
M^n(t) = E[X^n e^{tX}] \quad n \geq 1
\]
and

\[
M^n(0) = E[X^n] \quad n \geq 1.
\]

**Example: Binomial Distribution** \( B(n, p) \)  We calculate the Moment generating function of the Binomial random variable.

\[
M(t) = E[e^{tX}] = \sum_{k=0}^n e^{tk} \binom{n}{k} p^k (1-p)^{n-k} = \sum_{k=0}^n \binom{n}{k} (pe^t)^k (1-p)^{n-k} = (pe^t+1-p)^n.
\]

**Example: Unit Normal Distribution** \( N(0, 1) \)  We calculate the Moment generating function of the unit normal distribution.

\[
M(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{tx} e^{-x^2/2} dx.
\]

\[
= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-(x-t)^2/2+t^2/2} dx = e^{t^2/2}.
\]

**Important Properties of MGFs**

1. The moment generating function of the sum of independent random variables equals the product of the individual moment generating functions.

2. MGF uniquely determines the distribution.
C.9. MOMENT GENERATING FUNCTION

Chernoff Bounds via MGF  Let $X$ be a random variable and $M$ be its MGF. Then, for all $t > 0$, via Markov’s inequality, we have:

$$\Pr(X \geq a) = \Pr(e^{tX} \geq e^{ta}) \leq \frac{E[e^{tX}]}{e^{ta}} = e^{-ta} M(t).$$

Similarly, for all $t < 0$,

$$\Pr(X \leq a) \leq e^{ta} M(t).$$

The above bounds are called Chernoff bounds. We obtain the best bound by using the $t$ that minimizes the right hand side.

Example  Let $X$ be the standard normal r.v. $P(X \geq a) \leq e^{-ta} e^{t^2/2}$ for all $t > 0$. The right hand side is minimized for $t = a$. Thus, for $a > 0$

$$P(X \geq a) \leq e^{-a^2/2}.$$

Similarly, for $a < 0$,

$$\Pr(X \leq a) \leq e^{-a^2/2}.$$

C.9.1  Proof of Chernoff Bound for Sum of Indicator R.V’s

Theorem C.7. Let $X_1, X_2, \ldots, X_n$ be independent indicator random variables such that, for $1 \leq i \leq n$, $\Pr[X_i = 1] = p_i$, where $0 < p_i < 1$. Then, for $X = \sum_{i=1}^{n} X_i$, $\mu = E[X] = \sum_{i=1}^{n} p_i$, and any $\delta > 0$,

$$\Pr(X > (1 + \delta)\mu) < \left(\frac{e^\delta}{(1 + \delta)^{(1+\delta)}}\right)^\mu.$$

For $0 < \delta < 1$,

$$\Pr(X > (1 + \delta)\mu) \leq e^{-\mu \delta^2/3}.$$

For $R \geq 6\mu$,

$$\Pr(X \geq R) \leq 2^{-R}.$$

For $0 < \delta < 1$,

$$\Pr(X < (1 - \delta)\mu) < e^{-\mu \delta^2/2}.$$

Proof. Upper tail: For any positive real $t$,

$$\Pr(X > (1 + \delta)\mu) = \Pr(e^{tX} > e^{t(1+\delta)\mu}).$$

By Markov’s inequality,

$$\Pr(X > (1 + \delta)\mu) < \frac{E[e^{tX}]}{e^{t(1+\delta)\mu}}.$$
\[ E[e^{\sum_{i=1}^{n} X_i}] = \frac{\prod_{i=1}^{n} E[e^{tX_i}]}{e^{t(1+\delta)\mu}} = \frac{\prod_{i=1}^{n} (1 + p_i(e^t - 1))}{e^{t(1+\delta)\mu}} \]

\[ \leq \left( \frac{e^\delta}{(1 + \delta)(1+\delta)} \right)^\mu \]

for \( t = \ln(1 + \delta) \).

Using \( \delta - (1 + \delta) \ln(1 + \delta) \leq -\delta^2/3 \) for \( 0 < \delta < 1 \) we get

\[ \Pr(X > (1 + \delta)\mu) \leq e^{-\mu \delta^2/3}. \]

For \( R \geq 6\mu, \delta \geq 5 \),

\[ \Pr(X \geq (1 + \delta)\mu) \leq \left( \frac{e^\delta}{(1 + \delta)(1+\delta)} \right)^\mu \]

\[ \leq \left( \frac{e}{6} \right)^R \]

\[ \leq 2^{-R}. \]

**Lower tail:**

\[ \Pr(X < (1 - \delta)\mu) = \Pr(e^{-tX} > e^{-t(1-\delta)\mu}). \]

By Markov’s inequality,

\[ \Pr(X < (1 - \delta)\mu) < \frac{E[e^{-tX}]}{e^{-t(1-\delta)\mu}}. \]

Similar calculations yield

\[ \frac{E[e^{-tX}]}{e^{-t(1-\delta)\mu}} \leq \frac{e(e^{-t-1})\mu}{e^{-t(1-\delta)\mu}}. \]

For \( t = \ln(1/(1-\delta)) \), the right hand side of the above is

\[ \leq \left( \frac{e^{-\delta}}{((1-\delta)(1-\delta))} \right)^\mu. \]

Since \((1 - \delta)^{(1-\delta)} > e^{-\delta^2/2}\), we have

\[ \Pr(X < (1 - \delta)\mu) < e^{-\mu \delta^2/2}. \]
C.9. MOMENT GENERATING FUNCTION

C.9.2 Example application of Chernoff bound

**Theorem C.8.** Consider $n$ coin flips, let $X$ be the number of heads,

$$\Pr(|X - n/2| > \frac{1}{2} \sqrt{6n \log n}) \leq \frac{2}{n}$$

**Proof.**

$$E[X] = n/2$$

We need

$$\frac{n}{2} - \frac{1}{2} \sqrt{6n \log n} \leq X \leq \frac{n}{2} + \frac{1}{2} \sqrt{6n \log n}$$

or

$$X = \frac{n}{2}(1 \pm \sqrt{\frac{6 \log n}{n}})$$

Fixing $\delta = \sqrt{\frac{6 \log n}{n}}$,

$$\Pr(X < (1 - \delta)n/2) \leq e^{-\frac{n}{2} \delta^2} \leq 1/n$$

$$\Pr(X > (1 + \delta)n/2) \leq e^{-\frac{n}{2} \delta^2} \leq 1/n$$

Comparing Markov, Chebyshev, and Chernoff

What is the probability of more than $\frac{3N}{4}$ heads in $N$ coin flips?

1. Using Markov Inequality:

$$Pr(X \geq 3N/4) \leq 2/3.$$  

2. Using Chebyshev’s Inequality:

$$Pr(X \geq 3N/4) \leq 4/N.$$  

3. Using the Chernoff bound:

$$Pr(X \geq 3N/4) \leq e^{-\frac{N}{3} \delta^2}.$$  

We see that Chernoff gives a stronger bound compared to Chebyshev which in turn is stronger compared to Markov. The reason is Markov’s inequality uses
only the first moment (i.e., the expectation) of the random variable in its bound, whereas, Chebyshev uses the first and second moment (equivalently, the variance) into account, while Chernoff uses all the moments (via the MGF).

### C.10 Conditional Expectation

The conditional expectation of a random variable is defined as

\[
E[Y | Z = z] = \sum_y \Pr(Y = y | Z = z)
\]

where the summation is over all \( y \) in the range of \( Y \).

A useful identity is:

\[
E[X] = \sum_y \Pr(Y = y) E[X | Y = y]
\]

The expression \( E[Y | Z] \) is a random variable \( f(Z) \) that takes the value \( E[Y | Z = z] \) when \( Z = z \).

**Theorem C.9.**

\[
E[X] = E[E[X | Y]].
\]
Appendix D

Number-theoretic Concepts

This appendix reviews basic concepts in number theory.

D.1 Modular Arithmetic

The algorithms in Chapters 9 and 10 heavily use modular arithmetic, i.e., arithmetic modulo some number $n$. The following properties of mod function are useful.

- **sum rule**: $(a + b) \mod n = (a \mod n + b \mod n) \mod n$.
- **product rule**: $(a \times b) \mod n = ((a \mod n) \times (b \mod n)) \mod n$.
- **exponentiation rule**: $(a^y \mod n) = ((a \mod n)^y) \mod n$. The exponentiation rule is very useful in avoiding large integers.

D.2 Group Theory

Group theory is needed to understand and prove the correctness of primality testing algorithms, and once we have the basic concepts in place, it gives a powerful, yet easier way, to make our arguments.

A group $(S, \oplus)$ consists of a set $S$ and a binary operation $\oplus$ (e.g., can be the usual $+$ or $\times$) where the following four properties hold:

- **Closure**: For all $a, b \in S$, $a \oplus b$ also belongs to $S$.
- **Identity**: There exists an identity element $e \in S$ such that $a \oplus e = e \oplus a = a$, for all $a \in S$.
- **Associativity**: $a \oplus (b \oplus c) = (a \oplus b) \oplus c$, for all $a, b, c \in S$. 
• Inverses: For each element \( a \in S \), there exists an unique inverse \( a^{-1} \in S \), such that \( a \oplus a^{-1} = a^{-1} \oplus a = e \).

Common examples of groups that we use everyday are the set of (all) integers with the binary operation of addition, i.e., the group \((\mathbb{Z}, +)\). It is easy to check that all four properties hold. In particular, the identity element is 0, and the inverse of an integer \( a \) is \(-a\). Another common example is the group \((\mathbb{Q}^+, \times)\), where \( \mathbb{Q}^+ \) is the set of positive rational numbers and \( \times \) is multiplication. Here, the identity element is 1, and the inverse element of a rational number \( a \) is \( \frac{1}{a} \). (Note that, however, the set of rational numbers that includes 0 is not a group under multiplication. Why?)

In number theory, two groups are fundamental: they are the groups defined by modular addition and modular multiplication. We discuss them below.

D.2.1 Additive group modulo \( n \)

Given a positive number \( n \), this group consists of the set \( \mathbb{Z}_n = \{0, 1, 2, \ldots, n - 1\} \) with the binary operation being modular addition, i.e., addition modulo \( n \). This is the same as the usual addition, but the sum is always taken modulo \( n \). Hence, the set consists of elements which are all possible reminders of \( n \), i.e., from 0 to \( n - 1 \). The identity element is 0, as usual. The inverse of an element \( a \in \mathbb{Z}_n \) is \(-a \mod n \) which is the same as \( n - a \mod n \).

D.2.2 Multiplicative group modulo \( n \)

The binary operation that we use here is modular multiplication. This is the same as the usual multiplication, but the product is always taken modulo \( n \). Given a positive number \( n \), this group consists of the set of all numbers in \( \mathbb{Z}_n \) that are relatively prime to \( n \) (two numbers are relatively prime to each other if they have no common factor other than 1, i.e., their \( \gcd \) is 1) — this set is denoted by \( \mathbb{Z}_n^* \). In other words,

\[
\mathbb{Z}_n^* = \{ a \in \mathbb{Z}_n | \gcd(a, n) = 1 \}.
\]

Why only numbers that are relatively prime to \( n \) belong to \( \mathbb{Z}_n^* \)? As we will see below, only such numbers will have a (unique) inverse under modular multiplication. It is easy to show that the set satisfies, closure and associativity properties. It is also clear that the identity element is 1 and this obviously belongs to \( \mathbb{Z}_n^* \), since it is relatively prime to \( n \). What about inverses? We would like to show that for every \( a \in \mathbb{Z}_n^* \), \( a^{-1} \) exists: that is \( aa^{-1} = a^{-1}a \equiv 1 \). This follows from the following lemma and its corollary (see e.g., the Discrete Math Notes — Section 6 for a proof).
Lemma D.1. Consider the modular equation in (integer) variable \( x \): \( ax \equiv b \pmod{n} \). This equation has a unique solution if and only if \( \gcd(a, n) \) divides \( b \).

By putting \( b = 1 \) in the above lemma, we have the following corollary.

Corollary D.1. The equation \( ax \equiv 1 \pmod{n} \) has a unique solution if \( \gcd(a, n) = 1 \).

Hence there is a unique inverse for every \( a \in \mathbb{Z}_n^* \), since \( \gcd(a, n) = 1 \). Thus \( (\mathbb{Z}_n^*, \times) \) is a group where the operation \( \times \) is multiplication modulo \( n \). In particular, if \( n \) is prime, then \( \gcd(a, n) = 1 \) for all \( 1 \leq a \leq n - 1 \). Hence \( \mathbb{Z}_n^* = \{1, 2, \ldots, n - 1\} \). In fact, in many applications, a prime \( n \) is the common case. In this case, \( |\mathbb{Z}_n^*| = n - 1 \).

Calculating the solution to the modular equation \( ax \equiv b \pmod{n} \), and in particular, calculating the inverse of \( a \pmod{n} \) can be done by using Euclid’s algorithm (using Exercise 4.6).

Exercise D.1. Show how to find the inverse of \( a \pmod{n} \) (assuming that \( \gcd(a, n) = 1 \)) using Euclid’s algorithm (see Exercise 4.6).

D.2.3 Subgroups and generator of a group

Given a group \( (S, \oplus) \), and set \( S' \subseteq S \), if \( (S', \oplus) \) is also a group, then it is called a subgroup of \( S \). If \( S' \) is a proper subset of \( S \), then the subgroup is called a proper subgroup. For example, given the group of integers with addition operations \( (\mathbb{Z}, +) \), the group of even integers with the addition operation is a subgroup of \( (\mathbb{Z}, +) \). It is easy to show the following property of a subgroup:

Lemma D.2 (Closure property of a subgroup). If \( (S, \oplus) \) is a finite group and \( S' \) is a non-empty subset of \( S \) that is closed under \( \oplus \) (i.e., \( a \oplus b \in S' \) for every \( a, b \in S' \)), then \( (S', \oplus) \) is a subgroup of \( S \).

There is an important property of the size of a subgroup given by Lagrange’s Theorem which we state without proof.

Theorem D.1 (Lagrange’s Theorem). If \( (S, \oplus) \) is a group and \( (S', \oplus) \) is a subgroup of \( (S, \oplus) \), then \( |S'| \) divides \( |S| \). In particular, if \( (S', \oplus) \) is a proper subgroup, then \( |S'| \leq |S|/2 \).

We next introduce a very useful way to create subgroups of a given group, that uses the closure property of a subgroup. Let \( (S, \oplus) \) be a finite group. Let \( a \in S \) and denote the element \( a^{(k)} \) to be the element obtained by applying the \( \oplus \) operation \( k \) times, i.e.,

\[
a^{(k)} = \underbrace{a \oplus a \oplus \cdots \oplus a}_{k \text{ times}}.
\]
Consider the set of all elements “generated” by \( a \), i.e., applying the \( \oplus \) operation as many times. Denote this set as \( < a > \), i.e.,

\[
< a > = \{ a^{(k)} : k \geq 1 \}.
\]

By the closure property, the group \( (< a >, \oplus) \) is a subgroup (why?). For example, in the group \( (\mathbb{Z}_n, +) \), \( < a > = \{ a^{(k)} = ak \mod n : k \geq 1 \} \) and in the group \( (\mathbb{Z}_n^*, \times) \), \( < a > = \{ a^{(k)} = a^k \mod n : k \geq 1 \} \). We say \( (< a >, \oplus) \) as the subgroup generated by \( a \), and \( a \) is called the generator of this subgroup. Since \( S \) is finite, it is clear that the size of the subgroup, i.e., \( |< a >| \) is finite and will divide \( |S| \), by Lagrange’s theorem. Furthermore, we can show the following lemma.

**Lemma D.3.** Let \( k \) be the smallest positive integer such that \( a^{(k)} = e \), where \( e \) is the identity element of the group. Then, the elements \( a, a^{(2)}, \ldots, a^{(k)} = e \) are all distinct and \( k = |< a >| \). In particular, the sequence \( a^{(1)}, a^{(2)}, \ldots \) is periodic with period \( k \). Furthermore, \( a^{(i)} = a^{(j)} \) if and only if \( i \equiv j \mod k \).

**Proof.** Suppose \( a^{(i)} = a^{(j)} \) for \( i < j < k \). Then \( a^{(i)}a^{(j-i)} = a^{(j)} = a^{(i)} \). This means that \( a^{(j-i)} = e \) and \( j - i < k \) which contradicts the fact that \( k \) is the smallest integer such that \( a^{(k)} = e \). Hence \( a = a^{(1)}, a^{(2)}, \ldots, a^{(k)} = e \) are all distinct elements. Furthermore, \( a^{(k+i)} = a^{(k)}a^{(i)} = ea^{(i)} = a^{(i)} \), hence there are only \( k \) distinct elements. This also implies that they are periodic with period \( k \) and \( a^{(i)} = a^{(j)} \) if and only if \( i \equiv j \mod k \).

From the above lemma, we can define \( a^{(0)} = a^{(k)} = e \).

We next state an important result that immediately proves Fermat’s Little Theorem (stated next), an important result that we use in primality testing and in the proof of the RSA cryptosystem.

**Theorem D.2.** Let \( (S, \oplus) \) be a finite group with identity element \( e \). Then for all \( a \in S \),

\[
[ S ] \equiv e.
\]

**Proof.** Consider \( (< a >, \oplus) \), the subgroup of \( (S, \oplus) \) generated by \( a \). Then, by Lagrange’s Theorem, \( |< a >| \) divides \( |S| \). Hence, \( |S| \equiv 0 \mod |< a >| \). Therefore, for all \( a, a^{[S]} \equiv a^{(0)} = e \). □

### D.3 Fermat’s Little Theorem

The Fermat’s Little Theorem (FLT) is stated as follows.

The Fermat’s Little Theorem (FLT) is stated as follows.
Theorem D.3 (Fermat’s Little Theorem (FLT)). If $p$ is a prime, then for every $1 \leqslant a < p$,
\[
a^{p-1} \equiv 1 \pmod{p}.
\]

Proof. Apply Theorem D.2 with the group $\mathbb{Z}_p^*$. Since $p$ is prime, $|\mathbb{Z}_p^*| = p - 1$ and the identity element is 1. Hence for every $1 \leqslant a < p$, $a^{(p-1)} \equiv 1$ or equivalently, $a^{p-1} \equiv 1 \pmod{p}$ (since all operations in $\mathbb{Z}_p^*$ are modulo $p$). \qed
Appendix E

Graph-theoretic Concepts

This appendix reviews basic concepts in graph theory.

E.1 Definition

We formally define a graph as follows.

**Definition E.1.1.** An undirected graph $G$ is a pair $(V, E)$ where

- $V$ is the set of vertices.
- $E \subseteq V^2$ is the set of edges (unordered pairs) $E = \{(u, v) \mid u, v \in V\}$.

In a directed graph the edges have directions, i.e., ordered pairs — $(u, v)$ means that $u$ points to $v$ and $(v, u)$ means that $v$ points to $u$. A weighted graph includes a weight function $w : E \to \mathbb{Q}$ attaching a (rational) number (weight) to each edge.

Running time of a graph algorithm on a graph $G = (V, E)$ is usually in terms of $|V| = n$ and $|E| = m$. (We will use this notation throughout.) Sometimes, the running time can be a function of weights on the edges.

Figure E.1 shows an example of an undirected graph and an undirected weighted graph. Note that there can be at most $\binom{n}{2}$ edges in an undirected graph with $n$ nodes.

E.2 Path, Cycle, and Tree

These are fundamental structures in graphs and are defined as follows.
APPENDIX E. GRAPH-THEORETIC CONCEPTS

Figure E.1: A connected undirected graph (left) and the same graph with weights (right).

**Definition E.2.1.** A path in a graph $G = (V, E)$ is a sequence of vertices $v_1, v_2, ..., v_k$ such that for $1 \leq i \leq k - 1$, $(v_i, v_{i+1}) \in E$.

A cycle in a graph $G = (V, E)$ is a path $v_1, v_2, ..., v_k$ such that $(v_k, v_1) \in E$.

A tree is a graph with no cycles. A disjoint (i.e., the vertices are disjoint) collection of trees is a forest.

**E.3 Distance**

Given a graph $G = (V, E)$, and two vertices $u$ and $v$ belonging to $V$, and a path $P$ between $u$ and $v$, the length of the path is the number of edges between $u$ and $v$ in $P$. The distance between $u$ and $v$ is the length of the shortest path between $u$ and $v$. The shortest path between $u$ and $v$ will have minimum length among all other paths between the two vertices.

**E.4 Subgraph and Induced Subgraph**

**Definition E.4.1.** A graph $H = (V', E')$ is a subgraph of $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

If $V' = V$, then $H$ is a spanning subgraph.

Given a subset $U \subseteq V$ of vertices, the subgraph induced by $U$ in $G$ is the subgraph consisting of all vertices in $U$ and all the edges between them in $G$.

Figure E.4 shows an example of a graph and an induced subgraph.
Figure E.2: A graph and an induced subgraph of the graph.

**E.5 Degree of a vertex**

Given an undirected graph $G(V,E)$, the degree of a vertex is the number of its neighbors, i.e., the number of vertices that it shares an edge with. The degree of $v$ is denoted by $deg(v)$.

**Lemma E.1.** In an undirected graph $G = (V,E)$: $\sum_{v \in V} deg(v) = 2|E| = 2m$.

**Proof.** Consider an edge $(u,v) \in E$. This contributes one to $deg(u)$ and one to $deg(v)$. \qed

**E.6 Connectivity**

Connectivity is a basic concept in graph theory.

**Definition E.6.1.** An undirected graph is **connected** if there is a path between every pair of vertices. A directed graph is **connected** if there is a directed path from $u$ to $v$ or a directed path from $v$ to $u$, for every pair of vertices $u, v$.

A directed graph is **strongly connected** if there is a directed path from $u$ to $v$ and a directed path from $v$ to $u$, for every pair of vertices $u, v$.

A directed graph is **weakly connected** if the undirected graph obtained by ignoring the directions of the edges is connected.

A **connected component** of an undirected graph $G$ is a **maximal** induced subgraph of $G$ that is connected. It is maximal in the sense that no more vertices can be added to the subgraph, while still keeping the subgraph connected. Figure E.6 shows the example of a disconnected graph.
E.7 Spanning Tree (ST)

Definition E.7.1. Given a connected graph \( G = (V, E) \) a spanning tree \( T \) in \( G \) is a spanning subgraph of \( G \) which is a tree.

If the graph is not connected, then we can define a spanning tree on each connected component. A (disjoint) collection of spanning trees is called a spanning forest.

Maximality of a Spanning Tree Note that a spanning tree \( T \) is “maximal” in the sense that you cannot add additional edges — any edge added to \( T \) will close a cycle.

The reason of the above is as follows. Assume that \( T \) is a spanning tree in \( G \), and we add add an edge \((v, u)\) to \( T \). There is a path \( p \) from \( u \) to \( v \) using only edges of \( T \) (since it's a spanning tree). Thus, the path \( p \) and the edge \((v, u)\) must include a cycle.

E.7.1 The Size of a Spanning Tree

Theorem E.1. A spanning tree of a connected graph \( G = (V, E) \) has \(|V| - 1 = n - 1\) edges.

Proof. The proof is by induction on \(|V|\). Base case (\(|V| = 1\)) is trivial. Assume that the lemma is true for all graphs of size \(< |V|\). Consider a connected graph \( G \) of size \(|V|\) and a spanning tree \( T \) of \( G \). Remove an edge \((u, v)\) of \( T \). This breaks \( T \) into two components — \( T_1 \) and \( T_2 \). \( T_1 \) (\( T_2 \)) is a spanning tree of the subgraph induced by the vertices of \( T_1 \) (\( T_2 \)). The number of edges in \( T \) is \(|T_1| - 1 + |T_2| - 1 + 1 = |T| - 1 = |V| - 1\). \( \Box \)
Appendix F

Matrices

This appendix reviews basic concepts in theory of matrices.

F.1 Basic Facts

We recall some basic facts about matrices. If the entries of the matrices are numbers then the matrix operations \{+ , \ast\} define a **Ring** as seen below.

1. There is an **identity** element \(I\), such that for any matrix \(A\)
   \[AI = IA = A.\]

2. There is a **zero** element \(O\), s.t. for any matrix \(A\)
   \[AO = OA = O.\]

3. Every item has an additive inverse, i.e., every matrix \(A\) has an inverse \(-A\). Hence subtraction is simply adding one matrix with the additive inverse of the other.

4. Matrix multiplication is **associative**:
   \[A(BC) = (AB)C.\]

5. Matrix multiplication is **distributive** over addition:
   \[A(B + C) = AB + AC\]
   \[(B + C)D = BD + CD\]
However, one should note that matrix multiplication is NOT commutative:

\[
\begin{pmatrix}
0 & 1 \\
0 & 0
\end{pmatrix}
\cdot
\begin{pmatrix}
0 & 0 \\
1 & 0
\end{pmatrix}
= 
\begin{pmatrix}
1 & 0 \\
0 & 0
\end{pmatrix}
\]

\[
\begin{pmatrix}
0 & 0 \\
1 & 0
\end{pmatrix}
\cdot
\begin{pmatrix}
0 & 1 \\
0 & 0
\end{pmatrix}
= 
\begin{pmatrix}
0 & 0 \\
0 & 1
\end{pmatrix}
\]
Appendix G

Commonly Used Math Formulas in Algorithm Analysis

We collect commonly used math formulas in algorithm analysis. For an elaborate list see the theoretical computer cheat sheet by Steve Seiden (https://www.tug.org/texshowcase/cheat.pdf)

G.1 Geometric series

Summing a geometric series occurs very often in algorithmic analysis. The formula for the sum of a geometric series:

\[
\sum_{i=0}^{k-1} x^i = 1 + x + \ldots + x^{k-1} = \frac{x^k - 1}{x - 1}.
\]

where, \(x\) is called the (common) ratio.

Thus:

\[
\sum_{k=0}^{n} x^k = \Theta(x^n) \text{ if } x > 1 \text{ (increasing series)}. \\
\sum_{k=0}^{n} x^k = \Theta(1) \text{ if } x < 1 \text{ (decreasing series)}. \\
\text{If the series is infinite and } |x| < 1, \text{ then}
\]

\[
\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}.
\]

371
G.2 Combinatorial Inequalities

1. \[ \left( \frac{n}{k} \right)^k \leq \binom{n}{k} \leq \left( \frac{ne}{k} \right)^k. \]

2. For all \( x \in \mathbb{R} \), \( 1 + x \leq e^x. \)

3. For all \( x, n \in \mathbb{R} \), such that \( n \geq 1 \) and \( |x| \leq n \),
   \[ e^x \left( 1 - \frac{x^2}{n} \right) \leq \left( 1 + \frac{x}{n} \right)^n \leq e^x. \]
   The above holds even for negative values of \( x \).

4. For all \( x, n \in \mathbb{R}^+ \),
   \[ \left( 1 + \frac{x}{n} \right)^n \leq e^x \leq \left( 1 + \frac{x}{n} \right)^{n+x/2}. \]
Bibliography


