



# **6B23ICSC: DESIGN AND ANALYSIS OF ALGORITHMS**

**Lecture Notes**

**Mithun A V**





# Contents

I	Unit I	
<b>1</b>	<b>Introduction to Algorithms</b>	<b>9</b>
<b>1.1</b>	<b>Algorithm Definition</b>	<b>9</b>
1.1.1	Algorithm Example - Finding GCD	9
<b>1.2</b>	<b>Fundamentals of Algorithmic Problem Solving</b>	<b>11</b>
1.2.1	Understanding the Problem	11
1.2.2	Ascertaining the Capabilities of the Computational Device	11
1.2.3	Choosing between Exact and Approximate Problem Solving	11
1.2.4	Algorithm Design Techniques	11
1.2.5	Designing an Algorithm and Data Structures	11
1.2.6	Methods of Specifying an Algorithm	12
1.2.7	Proving an Algorithm's Correctness	12
1.2.8	Analyzing an Algorithm	12
1.2.9	Coding an Algorithm	13
<b>1.3</b>	<b>Important Problem Types</b>	<b>13</b>
1.3.1	Sorting	13
1.3.2	Searching	13
1.3.3	String Processing	13
1.3.4	Graph Problems	13
1.3.5	Combinatorial Problems	14
1.3.6	Geometric Problems	14
1.3.7	Numerical Problems	14
<b>1.4</b>	<b>Fundamental Data Structures</b>	<b>14</b>
1.4.1	Linear Data Structures	14
1.4.2	Graphs	15

1.4.3	Trees . . . . .	19
1.4.4	Sets and Dictionaries . . . . .	21
1.4.5	Abstract Data Type (ADT) . . . . .	22
<b>2</b>	<b>Fundamentals of Analysis of Algorithms . . . . .</b>	<b>23</b>
<b>2.1</b>	<b>The Analysis Framework . . . . .</b>	<b>23</b>
2.1.1	Measuring an Input's Size . . . . .	23
2.1.2	Units for Measuring Running Time . . . . .	23
2.1.3	Orders of Growth . . . . .	24
2.1.4	Worst-Case, Best-Case and Average-Case Efficiencies . . . . .	24
<b>2.2</b>	<b>Asymptotic Notations and Basic Efficiency Classes . . . . .</b>	<b>25</b>
2.2.1	Informal Introduction . . . . .	25
2.2.2	Formal Definitions . . . . .	26
2.2.3	Useful Property Involving the Asymptotic Notations . . . . .	28
2.2.4	Using Limits for Comparing Orders of Growth . . . . .	29
2.2.5	Basic Efficiency Classes . . . . .	30

## II

## Unit II

<b>3</b>	<b>Mathematical Analysis of Algorithms . . . . .</b>	<b>33</b>
<b>3.1</b>	<b>Mathematical Analysis of Nonrecursive Algorithms . . . . .</b>	<b>33</b>
<b>3.2</b>	<b>Mathematical Analysis of Recursive Algorithms . . . . .</b>	<b>36</b>
3.2.1	Example: Computing the $n^{\text{th}}$ Fibonacci Number . . . . .	38
<b>4</b>	<b>Brute Force and Exhaustive Search . . . . .</b>	<b>41</b>
<b>4.1</b>	<b>Brute Force Algorithms . . . . .</b>	<b>41</b>
4.1.1	Selection Sort . . . . .	41
4.1.2	Bubble Sort . . . . .	42
4.1.3	Sequential Search . . . . .	43
4.1.4	Brute-Force String Matching . . . . .	43
<b>4.2</b>	<b>Exhaustive Search . . . . .</b>	<b>44</b>
4.2.1	Traveling Salesman Problem (TSP) . . . . .	44
4.2.2	Knapsack Problem . . . . .	45
4.2.3	Assignment Problem . . . . .	46
4.2.4	Depth First Search Algorithm (DFS) . . . . .	46
4.2.5	Breadth First Search Algorithm (BFS) . . . . .	47

## III

## Unit III

<b>5</b>	<b>Divide and Conquer . . . . .</b>	<b>51</b>
<b>5.1</b>	<b>Introduction . . . . .</b>	<b>51</b>
5.1.1	Master Theorem . . . . .	52
<b>5.2</b>	<b>Mergesort . . . . .</b>	<b>52</b>
5.2.1	Analysis of Mergesort . . . . .	53

<b>5.3</b>	<b>Quicksort</b>	<b>54</b>
5.3.1	Analysis of Quicksort . . . . .	55
<b>5.4</b>	<b>Binary Search</b>	<b>56</b>
<b>5.5</b>	<b>Tree Traversals</b>	<b>57</b>
5.5.1	Preorder Traversal . . . . .	57
5.5.2	Inorder Traversal . . . . .	57
5.5.3	Postorder Traversal . . . . .	57
5.5.4	Analysis of Tree Traversal Algorithms . . . . .	58
<b>6</b>	<b>Dynamic Programming . . . . .</b>	<b>59</b>
<b>6.1</b>	<b>Introduction</b>	<b>59</b>
<b>6.2</b>	<b>Basic Examples</b>	<b>59</b>
6.2.1	Coin-row Problem . . . . .	59
<b>6.3</b>	<b>The Knapsack Problem and Memory Functions</b>	<b>61</b>
6.3.1	Memory Functions . . . . .	62
<b>6.4</b>	<b>Warshall's Algorithm</b>	<b>62</b>
<b>6.5</b>	<b>Floyd's Algorithm for the All-Pairs Shortest-Paths Problem</b>	<b>63</b>
<b>7</b>	<b>Greedy Technique . . . . .</b>	<b>65</b>
<b>7.1</b>	<b>Prim's Algorithm</b>	<b>65</b>
7.1.1	Kruskal's Algorithm . . . . .	67
<b>7.2</b>	<b>Dijkstra's Algorithm</b>	<b>68</b>

## IV

## Unit IV

<b>8</b>	<b>P, NP, and NP-Complete Problems . . . . .</b>	<b>73</b>
<b>8.1</b>	<b>Tractable and Intractable Problems</b>	<b>73</b>
<b>8.2</b>	<b>Class P Problems</b>	<b>73</b>
<b>8.3</b>	<b>Decidable and Undecidable Problems</b>	<b>74</b>
8.3.1	Halting Problem . . . . .	74
8.3.2	Proof for Undecidability of Halting Problem . . . . .	74
<b>8.4</b>	<b>Class NP Problems</b>	<b>74</b>
<b>8.5</b>	<b>NP-Complete Problems</b>	<b>75</b>
<b>9</b>	<b>Backtracking . . . . .</b>	<b>77</b>
<b>9.1</b>	<b>Introduction</b>	<b>77</b>
<b>9.2</b>	<b>n-Queens Problem</b>	<b>77</b>
<b>9.3</b>	<b>Subset-Sum Problem</b>	<b>79</b>
<b>9.4</b>	<b>Approximation Algorithms</b>	<b>79</b>
	<b>Books</b>	<b>81</b>





# Unit I

<b>1</b>	<b>Introduction to Algorithms .....</b>	<b>9</b>
1.1	Algorithm Definition	
1.2	Fundamentals of Algorithmic Problem Solving	
1.3	Important Problem Types	
1.4	Fundamental Data Structures	
<b>2</b>	<b>Fundamentals of Analysis of Algorithms</b>	<b>23</b>
2.1	The Analysis Framework	
2.2	Asymptotic Notations and Basic Efficiency Classes	





# 1. Introduction to Algorithms

## 1.1 Algorithm Definition

An algorithm is defined as a finite sequence of explicit instructions that, when provided with a set of input values, produces an output and then terminates. An algorithm should have the following properties:

- There must be no ambiguity in any instruction
- There should not be any uncertainty about which instruction should be executed next
- The execution of the algorithm should conclude after a finite number of steps
- Algorithms must be general enough to deal with any circumstances

The range of inputs for which an algorithm works has to be specified carefully. The same algorithm can be represented in several different ways (Using simple English sentences or pseudo code). There may exist several algorithms for solving the same problem. Different algorithms for solving the same problem may differ in their speed (Figure 1.1).

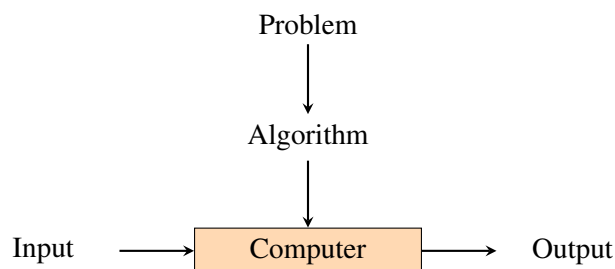


Figure 1.1: Algorithm

### 1.1.1 Algorithm Example - Finding GCD

The greatest common divisor of two nonnegative integers  $m$  and  $n$  (One of the numbers should be non-zero), denoted by  $\text{gcd}(m, n)$ , is defined as the largest integer that divides both  $m$  and  $n$ .

**Euclid's Algorithm**

According to Euclid's algorithm,

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

$$\gcd(m, 0) = m$$

Complete algorithm is given in Listing 1.1.

```

1 while n ≠ 0 do
2   r = m mod n
3   m = n
4   n = r
5 return m

```

Listing 1.1: Euclid's Algorithm

**Consecutive Integer Checking Algorithm**

Another method to find the gcd is given in Listing 1.2.

```

1 t = min(m, n)
2 while m mod t ≠ 0 or n mod t ≠ 0
3   t = t - 1
4 return t

```

Listing 1.2: Consecutive integer checking algorithm

Unlike Euclid's algorithm, this algorithm does not work correctly when one of its input numbers is zero.

**By finding prime factors**

```

1 Find the prime factors of m.
2 Find the prime factors of n.
3 Identify all the common factors in the two prime expansions found in
  Step 1 and Step 2. (If p is a common factor occurring  $p_m$  and  $p_n$ 
  times in m and n, respectively, it should be repeated  $\min\{p_m, p_n\}$ 
  times.)
4 Compute the product of all the common factors and return it as the gcd
  of m and n.

```

Listing 1.3: From prime factors

For example  $\gcd(24, 16)$  can be found as follows

$$\begin{aligned}
 24 &= 2.2.2.3 \\
 16 &= 2.2.2.2 \\
 \gcd(24, 16) &= 2.2.2 = 8
 \end{aligned}$$

In this algorithm, the prime factorization steps are not defined unambiguously. Hence this may not be treated as a legitimate algorithm.

Consecutive primes not exceeding any given integer  $n > 1$ , can be found using an algorithm known as **sieve of Eratosthenes**.

```

1 for p = 2 to n do
2   A[p] = p
3 for p = 2 to  $\sqrt{n}$ 
4   if A[p] ≠ 0
5     j = p × p
6     while j ≤ n do
7       A[j] = 0 //mark element as eliminated
8       j = j + p

```

```
9 //copy the remaining elements of A to array L of the primes
10 i = 0
11 for p = 2 to n do
12     if A[p] ≠ 0
13         L[i] = A[p]
14         i = i+1
15 return L
```

Listing 1.4: Sieve of Eratosthenes

## 1.2 Fundamentals of Algorithmic Problem Solving

The sequence of steps in design and analyzing an algorithm includes the following.

### 1.2.1 Understanding the Problem

- Read the problem description carefully
- Ask questions for any doubts about the problem
- Do small examples by hand
- think about special cases
- Ask questions again if needed.

An input to an algorithm specifies an **instance** of the problem the algorithm solves. The set of instances the algorithm needs to handle must be specified.

### 1.2.2 Ascertaining the Capabilities of the Computational Device

Computers today are extremely fast. Hence, in most situations one does not have to worry about a computer being too slow for the task. For problems that are very complex, or have to process huge volumes of data, or deal with applications where the time is critical, it is better to be aware of the speed and memory available on a particular computer system.

### 1.2.3 Choosing between Exact and Approximate Problem Solving

Approximate algorithms may be suitable for following scenarios

- Problems that simply cannot be solved exactly for most of their instances (e.g. extracting square roots)
- Available algorithms for solving a problem exactly are unacceptably slow because of the problem's intrinsic complexity
- An approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

### 1.2.4 Algorithm Design Techniques

An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

- Algorithm design techniques provide guidance for designing algorithms for new problems.
- Algorithm design techniques make it possible to classify algorithms according to an underlying design idea.

### 1.2.5 Designing an Algorithm and Data Structures

Algorithm design techniques offer helpful strategies for solving algorithmic problems, but designing an algorithm for a specific problem can still be difficult. Certain design techniques may not be suitable for the problem at hand. In some cases, it's necessary to combine multiple techniques. Choosing appropriate data structures for the operations performed by the algorithm is also very important since it may affect the speed of the algorithm.

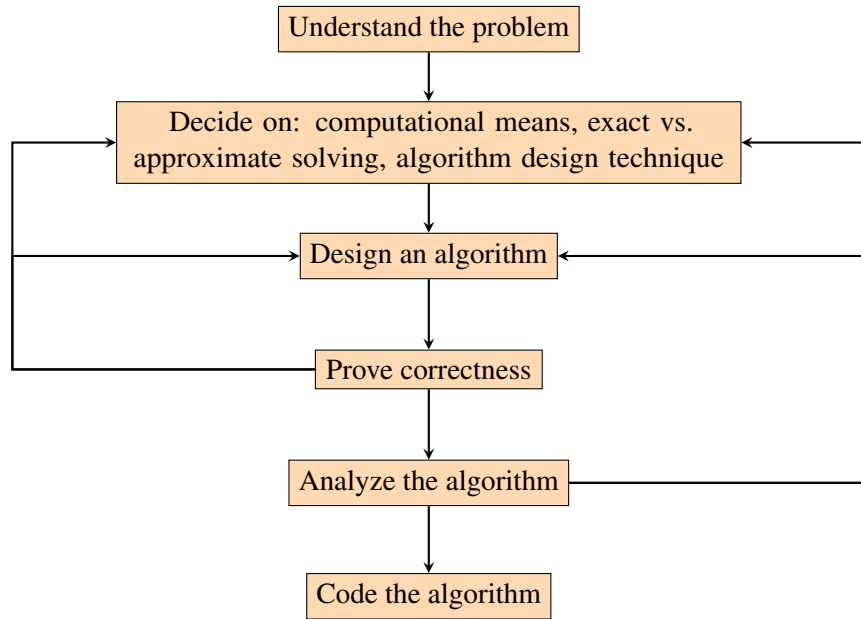


Figure 1.2: Algorithm design and analysis process

### 1.2.6 Methods of Specifying an Algorithm

Once an algorithm is designed it needs to be specified in some form. Two most common methods for specifying an algorithm are

- Writing sequence of steps in English
- Pseudocode, which is a mixture of a natural language and programming language-like constructs.

### 1.2.7 Proving an Algorithm's Correctness

Once an algorithm has been specified, it has to be ensured that the algorithm yields a required result for every legitimate input in a finite amount of time. A common technique for proving correctness is to use mathematical induction.

The concept of correctness in approximation algorithms is more complex compared to exact algorithms. In the case of an approximation algorithm, the objective is typically to demonstrate that the algorithm's error remains within a predetermined limit.

### 1.2.8 Analyzing an Algorithm

#### Efficiency

There are two kinds of algorithm efficiency:

- **time efficiency**, indicating how fast the algorithm runs
- **space efficiency**, indicating how much extra memory it uses

#### Simplicity

Simpler algorithms are easier to understand and easier to program. Consequently, the resulting programs usually contain fewer bugs. But simpler algorithm need not always be efficient.

#### Generality

To determine if two numbers are relatively prime (means they only have 1 as a common divisor), it is simpler to create an algorithm for a broader problem of finding the gcd of two numbers. The initial problem can then be solved by checking if the gcd of the two numbers is equal to 1 or not.

### 1.2.9 Coding an Algorithm

Having a functioning program opens up the possibility to empirically analyze the underlying algorithm. This analysis involves measuring the time it takes for the program to run with different inputs and then examining the results obtained.

## 1.3 Important Problem Types

### 1.3.1 Sorting

The sorting problem is to rearrange the items of a given list in nondecreasing order. Most common requirements are to sort lists of numbers, characters from an alphabet, character strings, records etc. In the case of records, a piece of information, known as a **key**, is to be chosen to guide sorting. Sorting makes many questions about the list easier to answer such as searching for an item in a list.

There are different types of sorting algorithms. Some are simple but slower, while others are faster but more complex. Certain algorithms perform well with randomly ordered inputs, while others excel with almost-sorted lists. Some algorithms are specifically designed for sorting small lists in fast memory, while others can be modified to sort large files stored on a disk.

A sorting algorithm is called **stable** if it preserves the relative order of any two equal elements in its input. Another notable feature of a sorting algorithm is the amount of extra memory the algorithm requires. An algorithm is said to be **in-place** if it does not require extra memory, except, possibly, for a few memory units.

### 1.3.2 Searching

The searching problem deals with finding a given value, called a **search key**, in a given set. Some searching algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays.

In certain applications, where the data changes frequently compared to the number of searches, searching needs to be considered alongside two other operations: adding an item to the data set and removing an item from it. In these situations, it is important to select data structures and algorithms that can meet the requirements of each operation in a balanced way.

### 1.3.3 String Processing

A string is a sequence of characters from an alphabet. Researchers have given special attention to a specific problem known as **string matching**, which involves searching for a given word in a text.

### 1.3.4 Graph Problems

A graph is a collection of points called vertices connected by line segments called edges. Graphs are used for modeling various applications like transportation, communication, social and economic networks, project scheduling, and games.

Basic graph algorithms include

- Graph-traversal algorithms - determine how to reach all the points in a network.
- Shortest-path algorithms- find the best route between two cities.
- Topological sorting (for graphs with directed edges) - to check if a set of courses and their prerequisites is consistent or self-contradictory.

Some graph problems are computationally challenging, such as the traveling salesman problem (TSP). TSP involves finding the shortest tour through multiple cities, visiting each city exactly once. Another difficult problem is graph coloring, which aims to assign the fewest number of colors to graph vertices so that no adjacent vertices have the same color. Graph coloring is applicable in event scheduling, where an optimal schedule can be achieved by solving the graph-coloring problem.

### 1.3.5 Combinatorial Problems

The traveling salesman problem and the graph-coloring problem are combinatorial problems. Combinatorial problems involve finding a combinatorial object, like a permutation, combination, or subset, that meets specific constraints. These problems may also require the combinatorial object to have additional properties, such as a maximum value or minimum cost.

Combinatorial problems are generally considered the most challenging in computing, both theoretically and practically. This difficulty arises from several factors. Firstly, the number of combinatorial objects grows rapidly as the problem size increases, even for moderate-sized instances. Secondly, there are no known algorithms that can solve most combinatorial problems within a reasonable timeframe. Many computer scientists believe that such algorithms may not even exist.

### 1.3.6 Geometric Problems

Geometric algorithms deal with geometric objects such as points, lines, and polygons. Two classic problems in computational geometry are

- **The closest-pair problem** - Finding the pair of points that are closest to each other among a given set of points.
- **The convex-hull problem** - Finding the smallest convex polygon that encompasses all the points in a given set.

### 1.3.7 Numerical Problems

Numerical problems are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on.

Most mathematical problems in computing can only be solved approximately. This is due to the inherent difficulty in manipulating real numbers, which can only be represented approximately in a computer. Additionally, when performing numerous arithmetic operations on approximately represented numbers, round-off errors can accumulate and significantly distort the output of an algorithm that initially seemed correct.

## 1.4 Fundamental Data Structures

A data structure can be defined as a particular scheme of organizing related data items.

### 1.4.1 Linear Data Structures

The two most important elementary data structures are the array and the linked list.

#### Arrays

A (one-dimensional) array is a sequence of  $n$  items of the same data type that are stored contiguously in computer memory. Elements inside an array can be accessed using an index. Each and every element of an array can be accessed in the same constant amount of time, regardless of its location within the array. Arrays are used for implementing a variety of other data structures such as **strings**. The operations performed on strings include computing the length of a string, comparing two strings to determine their lexicographic order (alphabetical order), and concatenating two strings to form a new string.

#### Linked List

A linked list is a sequence of zero or more elements called **nodes**. Each node contains two kinds of information: some data and one or more links to other nodes of the linked list. In a singly linked list, each node except the last one contains a single pointer to the next element.



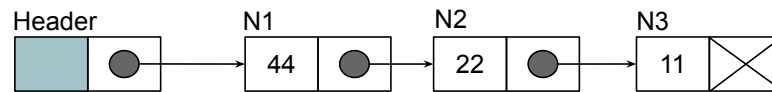


Figure 1.3: Single Linked List

To access a specific node in a linked list, the list must be traversed starting from the first node and following the pointers until the desired node is reached. Therefore, the time required to access an element in a singly linked list depends on its position within the list.

Linked lists do not require any prior reservation of computer memory, meaning that memory can be dynamically allocated as needed. This is in contrast to other data structures like arrays, which require a fixed amount of memory upfront.

Additionally, insertions and deletions can be efficiently performed in a linked list by adjusting a few appropriate pointers. This flexibility is one of the advantages of linked lists compared to other data structures.

In a doubly linked list, every node except the first and the last, contains pointers to both its successor and its predecessor.

## Stack

A **stack** is a list in which insertions and deletions can only be done at the end. This end is called the **top**. The insertion operation in a stack is called **push**, and the deletion operation is called **pop**. Since the last pushed element is the first one to be popped, a stack is also known as a **LIFO** (Last In First Out) list. Stacks are used for implementing recursive algorithms, evaluating postfix expression etc.

## Queue

A **queue** is a list in which insertions takes place at one end called **rear** and deletions takes place at the other end called **front**. The insertion operation in a queue is called **enqueue**, and the deletion operation is called **dequeue**. Since the first enqueued element is the first one to be dequeued, a queue is also known as a **FIFO** (First In First Out) list. Queues are used for implementing several graph algorithms.

### 1.4.2 Graphs

A graph  $G$  consists of 2 sets:

1. A set  $V$ , the set of all vertices (nodes).
2. A set  $E$ , the set of all edges (arcs).  $E$  is a set of pairs of elements from  $V$ .

For example, consider the graph  $G$  shown in Figure 1.4. For this graph,

- $V = \{a, b, c, d, e\}$
- $E = \{(a, b), (b, c), (b, d), (c, e), (b, b), (a, d), (e, d)\}$

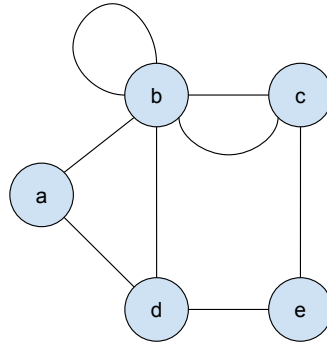


Figure 1.4: Example for graph

A **directed graph (Digraph)** is a graph  $G$ , such that  $G = \langle V, E \rangle$ , where  $V$  is the set of all vertices and  $E$  is the set of ordered pairs of elements from  $V$ . If  $(a)$

For example, consider the digraph  $G$  shown in Figure 1.5. For this graph,

- $V = \{a, b, c, d, e\}$
- $E = \{(a, b), (b, c), (b, d), (c, e), (b, b), (d, a), (d, e)\}$

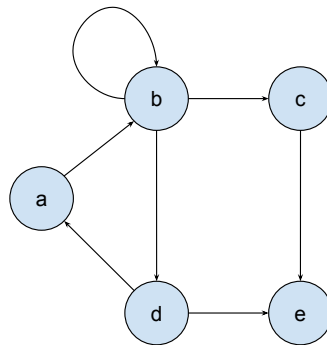


Figure 1.5: Example for digraph

A graph or digraph is called a **weighted graph** or **weighted digraph** if all the edges in it are labelled with some weights.

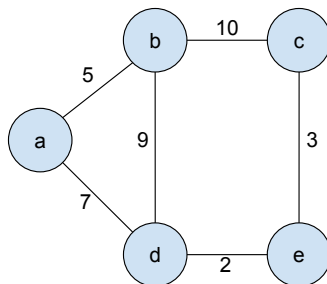


Figure 1.6: Example for weighted graph

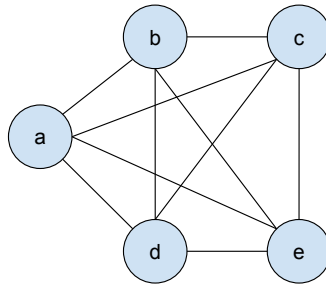


Figure 1.7: Example for complete graph

**Adjacent Vertices**

In a graph, a vertex  $a$  is adjacent to (or neighbour of) vertex  $b$ , if there is an edge from  $a$  to  $b$ . For example, in Figure 1.4, vertices  $b$  and  $d$  are adjacent to vertex  $a$ .

**Self Loop**

If there is an edge with same starting and ending vertex, then it is called a self loop or loop. For example, in Figure 1.4, edge  $(b, b)$  is a self loop.

**Parallel Edges**

If there is more than one edge between the same pair of vertices, then they are known as parallel edges. For example, in Figure 1.4, there are two parallel edges between vertices  $b$  and  $c$ .

**Multigraph**

If a graph has either self loop or parallel edges then it is called a multigraph. Hence the graph in Figure 1.4 is a multigraph.

**Simple Graph**

If a graph has neither a self loop nor parallel edges then it is called a simple graph. Hence the graph in Figure 1.6 is a simple graph.

**Complete Graph**

A graph or digraph  $G$  is called a complete graph, if each vertex  $v_i$  is adjacent to every other vertex  $v_j$  in  $G$ . Hence the graph in Figure 1.7 is a complete graph. A standard notation for the complete graph with  $n$  vertices is  $K_n$ .

**Path**

A path from vertex  $u$  to vertex  $v$  of a graph  $G$  can be defined as a sequence of adjacent vertices that starts with  $u$  and ends with  $v$ . If all vertices of a path are distinct, the path is said to be **simple**.

**Acyclic Graph**

If there is a path containing one or more edges which starts at vertex  $v_i$  and ends at  $v_i$ , then the path is known as a cycle. If a graph does not have any cycle, then the graph is called an acyclic graph.

**Degree of vertex**

In an undirected graph, the number of edges connected to with vertex  $v_i$  is called the degree of vertex  $v_i$  and is denoted by  $degree(v_i)$ . For example, in Figure 1.4,  $degree(a) = 2$ ,  $degree(d) = 3$ .

For a directed graph there are two degrees,

- **Indegree** of a node is the number of edges incident on the node. For example, in Figure 1.5,  $indegree(d) = 1$ .

- **Outdegree** of a node is the number of edges emanating from the node. For example, in Figure 1.5,  $\text{outdegree}(d) = 2$ .

### Dense and Sparse Graphs

A graph with relatively few possible edges missing is called dense; a graph with few edges relative to the number of its vertices is called sparse.

### Connected Graph

Two vertices  $v_i$  and  $v_j$  in a graph  $G$  are said to be connected, if there is a path in  $G$  from  $v_i$  to  $v_j$ . A graph is said to be connected, if there is path between every pair of vertices in  $G$ .

A digraph is said to be **strongly connected**, if there is directed path between every pair of vertices. If a digraph is not strongly connected, but the underlying undirected graph is connected, then the graph is said to be **weakly connected**.

### Representation of Graphs

#### • Set Representation

Two sets are maintained

1. The set of vertices  $V$ .
2. The set of edges,  $E$ , which is the subset of  $V \times V$ .

If the graph is weighted, the set  $E$  is the subset of  $W \times V \times V$ , where  $W$  is the set of weights.

For example, the set representation of graph in Figure 1.4 is

- $V = \{a, b, c, d, e\}$
- $E = \{(a, b), (b, c), (b, d), (c, e), (b, b), (a, d), (e, d)\}$

The set representation of the weighted graph in Figure 1.6 is

- $V = \{a, b, c, d, e\}$
- $E = \{(5, a, b), (10, b, c), (9, b, d), (3, c, e), (7, a, d), (2, e, d)\}$

#### • Linked Representation

In the linked representation, two types of nodes are used, one for unweighted graphs and the other for weighted graphs.



(a) For weighted graph



(b) For unweighted graph

Figure 1.8: Node structures for graph

For example, the linked representation of graph in Figure 1.4 is shown below.

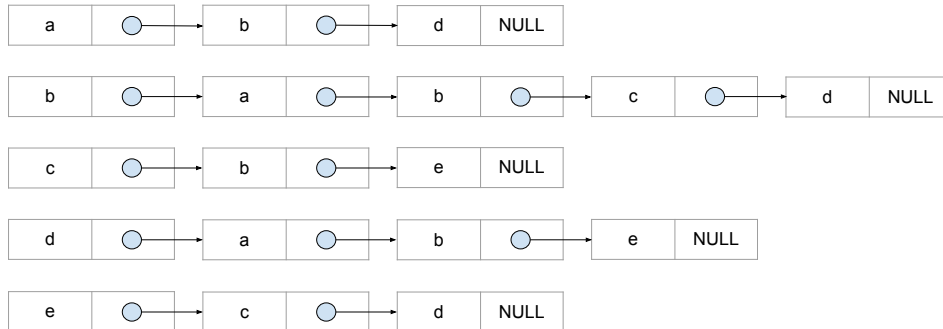


Figure 1.9: Linked representation of graph in Figure 1.4

### • Matrix Representation

A square matrix of order  $n \times n$ , where  $n$  is the number of vertices, is used to represent a graph. This matrix is known as the adjacency matrix. Entries in the adjacency matrix are defined as follows.

$$a_{ij} = \begin{cases} 1, & \text{If there is an edge from } v_i \text{ to } v_j \\ 0, & \text{otherwise} \end{cases}$$

In case of a multigraph, the entries in adjacency matrix will be the number of edges between the vertices, instead of 1. In case of a weighted graph, the entries in the adjacency matrix are weights of the edges between two vertices.

The adjacency matrix representation of the graph in Figure 1.4 is shown below

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
<i>a</i>	0	1	0	1	0
<i>b</i>	1	1	2	1	0
<i>c</i>	0	2	0	0	1
<i>d</i>	1	1	0	0	1
<i>e</i>	0	0	1	1	0

The adjacency matrix representation of the weighted graph in Figure 1.6 is shown below

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
<i>a</i>	0	5	0	7	0
<i>b</i>	5	0	10	9	0
<i>c</i>	0	10	0	0	3
<i>d</i>	7	9	0	0	2
<i>e</i>	0	0	3	2	0

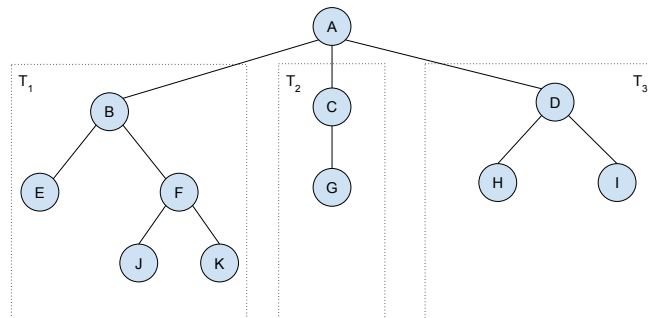
### 1.4.3 Trees

A tree (a free tree) is a connected acyclic graph. A graph that has no cycles but is not necessarily connected is called a **forest**: each of its connected components is a tree.

The number of edges in a tree is always one less than the number of its vertices. i.e.  $|E| = |V| - 1$ .

A rooted tree is a finite set of one or more nodes such that

- one of the nodes is special node called the **root** of the tree.
- the remaining nodes are partitioned into  $n$  ( $n > 0$ ) disjoint sets  $T_1, T_2, \dots, T_n$ , where each  $T_i$  is a tree.  $T_1, T_2, \dots, T_n$  are called subtrees of the root.



- **Node:** A tree is made up nodes. A node contains the data and links to other nodes.

- **Parent:** Parent of a node is the immediate predecessor of a node.
- **Child:** All the immediate successors of a node are called its children. The child on the left side is called the left child and the child on the right side is called the right child.
- **Root:** Root is a special node which has no parent.
- **Leaf:** The node which is at the end and does not have any child is called a leaf node.
- **Level:** Level is the rank of a node in the hierarchy. Root node is at level 0. If a node is at level  $l$  then all its children are at level  $l + 1$  and the parent is at level  $l - 1$ .
- **Height:** The maximum number of nodes that is possible in a path starting from the root node to the leaf node is called the height of the tree.
- **Degree:** The maximum number of children that is possible for a node in a tree is called the degree of the node.
- **Sibling:** The nodes having the same parent are called siblings.
- **Ancestors:** For any vertex  $v$  in a tree  $T$ , all the vertices on the simple path from the root to that vertex are called ancestors of  $v$ .
- **Descendants:** All the vertices for which a vertex  $v$  is an ancestor are said to be descendants of  $v$ .

### Binary Trees

A binary tree  $T$  is a finite set of nodes such that

- $T$  is empty (called the empty binary tree) or
- $T$  contains a special node called the root of  $T$ , and the remaining nodes of  $T$  form two disjoint binary tree  $T_1$  and  $T_2$  called left sub tree and right subtree respectively.

Each node in a binary tree can have at most 2 children.

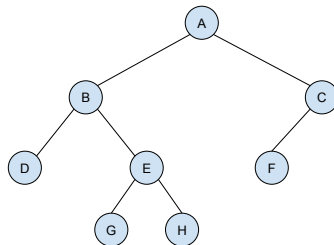


Figure 1.10: Binary Tree

### Full Binary Tree

A binary tree is a full binary tree if it contains the maximum possible number of nodes at all levels.

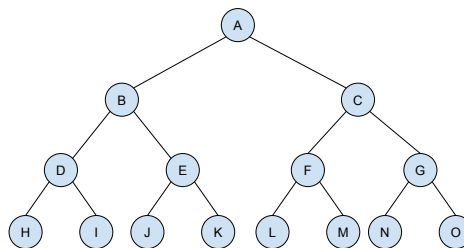


Figure 1.11: Full Binary Tree



### Complete Binary Tree

A binary tree is a complete binary tree if it contains the maximum possible number of nodes at all levels except possibly the last level and all the nodes in the last level appear as far left as possible. All full binary trees are complete binary trees, but not vice versa.

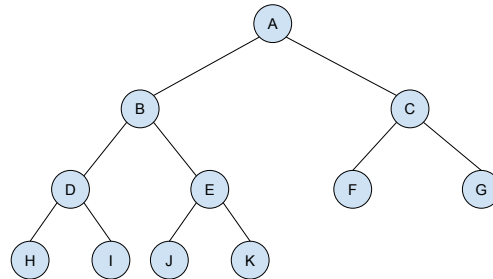


Figure 1.12: Complete Binary Tree

### Binary Search Tree (BST)

A binary tree  $T$  is a binary search tree, if each node  $N$  of  $T$  satisfies the following property:

- The value of  $N$  is greater than every value in the left subtree of  $N$  and is less than every value in the right subtree of  $N$ .

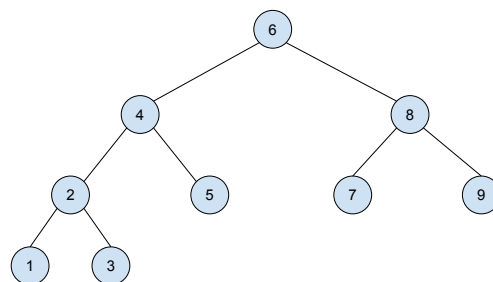


Figure 1.13: Example for Binary Search Tree

#### 1.4.4 Sets and Dictionaries

A set is an unordered collection (possibly empty) of distinct items called elements of the set. A set can be defined in either by

- Listing of its elements (e.g.,  $S = \{1, 3, 5, 7, 9\}$ ) or by
- Specifying a property that all the set's elements and only they must satisfy  
e.g.,  $S = \{n: n \text{ is an odd number smaller than } 10\}$ .

The most important set operations are: checking membership of a given item in a given set; finding the union of two sets, which comprises all the elements in either or both of them; and finding the intersection of two sets, which comprises all the common elements in the sets. Sets can be implemented in computer applications in two ways.

- Consider only sets that are subsets of some large set  $U$ , called the universal set. If set  $U$  has  $n$  elements, then any subset  $S$  of  $U$  can be represented by a bit string of size  $n$ , called a **bit vector**, in which the  $i^{\text{th}}$  element is 1 if and only if the  $i^{\text{th}}$  element of  $U$  is included in set  $S$ . For example, if  $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ , then  $S = \{1, 3, 5, 7, 9\}$  can be represented as 1010101010.
- Use the list structure to indicate the set's elements.

A **multiset**, or **bag**, an unordered collection of items that are not necessarily distinct.

In computing, the most common operations to be performed for a set or a multiset are searching for a given item, adding a new item, and deleting an item from the collection. A data structure that implements these three operations is called the **dictionary**.

#### 1.4.5 Abstract Data Type (ADT)

An ADT is a set of abstract objects representing data items with a collection of operations that can be performed on them.

## 2. Fundamentals of Analysis of Algorithms

### 2.1 The Analysis Framework

For an algorithm, there are two kinds of efficiencies:

- **Time efficiency (Time Complexity):** Indicates how fast the algorithm runs.
- **Space efficiency (Space Complexity):** Indicates the amount of memory required by the algorithm in addition to the space needed for the input and output.

#### 2.1.1 Measuring an Input's Size

Most algorithms take more time to execute with larger input sizes. Therefore, an algorithm's efficiency can be measured as a function of some parameter  $n$ , indicating the algorithm's input size.

- For searching and sorting algorithms,  $n$  could be the size of the list.
- For algorithms involving matrices, it could be the number of elements in the matrix.
- For algorithms solving problems such as checking primality of a positive integer  $n$ , it could be the number of bits  $b$  required to represent the number.

$$b = \lfloor \log_2 n \rfloor + 1$$

#### 2.1.2 Units for Measuring Running Time

One method is to measure the time taken to run the program implementing the algorithm. But this method has following drawbacks.

- The time taken depends on the specifications of the computer.
- Time depends on the quality of the program written and the compiler.
- It is difficult to clock the actual running time of the program.

Better method is to identify the most important operation of the algorithm, called the **basic operation**, which is the operation contributing to most of the total running time, and then compute the number of times the basic operation is executed.

The basic operation is usually the most time consuming operation in an algorithm's inner most loop. For example, in sorting algorithms, basic operation is the key comparison.

The running time  $T(n)$  of an algorithm with input size  $n$  can be written as

$$T(n) \approx c_{op}C(n)$$

where  $c_{op}$  is the execution time of the basic operation of the algorithm and  $C(n)$  is the number of times this basic operation is needed to be executed for the algorithm.

This equation can be used to evaluate how much faster an algorithm will run if the input size is doubled. For example, let  $C(n) = \frac{1}{2}n(n-1)$ . For large values of  $n$

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

If the input size is doubled,

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4$$

i.e. the algorithm will run 4 times longer if the input size is doubled. This can be calculated without knowing the value of  $c_{op}$  since it gets cancelled out.

### 2.1.3 Orders of Growth

Efficient algorithms are not primarily distinguished by their differences in running times on small inputs. For large values of  $n$ , it is the function's order of growth that counts. Table 2.1 shows order of growth of some important functions.

$n$	$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$	$n!$
10	3.3	10	$3.3 \times 10^1$	$10^2$	$10^3$	$10^3$	$3.6 \times 10^6$
$10^2$	6.6	$10^2$	$6.6 \times 10^2$	$10^4$	$10^6$	$1.3 \times 10^{30}$	$9.3 \times 10^{157}$
$10^3$	10	$10^3$	$1.0 \times 10^4$	$10^6$	$10^9$		
$10^4$	13	$10^4$	$1.3 \times 10^5$	$10^8$	$10^{12}$		
$10^5$	17	$10^5$	$1.7 \times 10^6$	$10^{10}$	$10^{15}$		
$10^6$	20	$10^6$	$2.0 \times 10^7$	$10^{12}$	$10^{18}$		

Table 2.1: Order of growth some important functions

The logarithmic function ( $\log n$ ) is the slowest growing functions in the Table 2.1. The exponential function  $2^n$  and the factorial function  $n!$  functions grow extremely fast, and even for relatively small values of  $n$ , their values become astronomically large. Algorithms that require an exponential number of operations can be used only for solving problems of very small sizes.

### 2.1.4 Worst-Case, Best-Case and Average-Case Efficiencies

There are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input. For example, in linear search algorithm, running time can be quite different for the same list size  $n$ . In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size  $n$ .

$$C_{worst}(n) = n$$

The **worst-case efficiency** of an algorithm is its efficiency for the worst-case input of size  $n$ , which is an input of size  $n$  for which the algorithm runs the longest among all possible inputs of that size. The worst-case analysis provides an upper bound on the running time of an algorithm.

The **best-case efficiency** of an algorithm is its efficiency for the best-case input of size  $n$ , which is an input of size  $n$  for which the algorithm runs the fastest among all possible inputs of that size. The best-case analysis provides a lower bound on the running time of an algorithm.

$$C_{best}(n) = 1$$

The **average-case efficiency** yields the necessary information about an algorithm's behaviour on a "typical" or "random" input. To analyze the algorithm's average case efficiency, some assumptions are to be made about possible inputs of size  $n$ . For example, in linear search, the assumptions are

- the probability of a successful search is  $p$  ( $0 \leq p \leq 1$ )
- the probability of the first match occurring in the  $i^{\text{th}}$  position of the list is the same for every  $i$ , which is equal to  $\frac{p}{n}$

In the case of a successful search, the probability of the first match occurring in the  $i^{\text{th}}$  position of the list is  $\frac{p}{n}$  for every  $i$ , and the number of comparisons made by the algorithm in such a situation is  $i$ . In the case of an unsuccessful search, the number of comparisons will be  $n$  with the probability of such a search being  $(1 - p)$ .

$$\begin{aligned} C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + 3 \cdot \frac{p}{n} + \dots + i \cdot \frac{p}{n} + \dots + n \cdot \frac{p}{n}] + n \cdot (1 - p) \\ &= \frac{p}{n} [1 + 2 + 3 + \dots + i + \dots + n] + n(1 - p) \\ &= \frac{p}{n} \cdot \frac{n(n+1)}{2} + n(1 - p) \\ &= \frac{p(n+1)}{2} + n(1 - p) \end{aligned}$$

From this equation, if  $p = 1$  (the search is successful), the average number of key comparisons made by sequential search is  $\frac{n+1}{2}$ , that is, the algorithm will inspect, on average, about half of the list's elements. If  $p = 0$  (the search is unsuccessful), average number of key comparisons will be  $n$ , because the algorithm will inspect all  $n$  elements on all such inputs.

## 2.2 Asymptotic Notations and Basic Efficiency Classes

Three notations used to compare and rank order of growth of functions are

- $O$  (big oh)
- $\Omega$  (big omega)
- $\Theta$  (big theta)

### 2.2.1 Informal Introduction

Informally,  $O()$  is the set of all functions with a lower or same order of growth as  $g(n)$  (to within a constant multiple, as  $n$  goes to infinity). For example,

$$\begin{aligned} n &\in O(n^2) \\ 5n + 3 &\in O(n^2) \\ \frac{1}{2}n(n-1) &\in O(n^2) \end{aligned}$$

But the following function does not belong to  $O(n)$

$$\begin{aligned} n^3 &\notin O(n^2) \\ 0.00001n^3 &\notin O(n^2) \\ n^4 + n + 6 &\notin O(n^2) \end{aligned}$$

Informally,  $\Omega(g(n))$  is the set of all functions with a higher or same order of growth as  $g(n)$  (to within a constant multiple, as  $n$  goes to infinity). For example,

$$\begin{aligned} n^3 &\in \Omega(n^2) \\ \frac{1}{2}n(n-1) &\in \Omega(n^2) \\ 5n+3 &\notin \Omega(n^2) \end{aligned}$$

Informally,  $\Theta(g(n))$  is the set of all functions with same order of growth as  $g(n)$  (to within a constant multiple, as  $n$  goes to infinity). For example,

$$\begin{aligned} \frac{1}{2}n(n-1) &\in \Theta(n^2) \\ 5n^2+3 &\in \Theta(n^2) \end{aligned}$$

### 2.2.2 Formal Definitions

#### *O*-notation

A function  $t(n)$  is said to be in  $O(g(n))$ , denoted  $t(n) \in O(g(n))$ , if  $t(n)$  is bounded above by some constant multiple of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant  $c$  and some nonnegative integer  $n_0$  such that

$$t(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0$$

For example to prove that  $5n+3 \in O(n^2)$ ,

$$\begin{aligned} 5n+3 &\leq 5n+n \quad (\text{for all } n \geq 3) = 6n \\ &\leq 6n^2 \end{aligned}$$

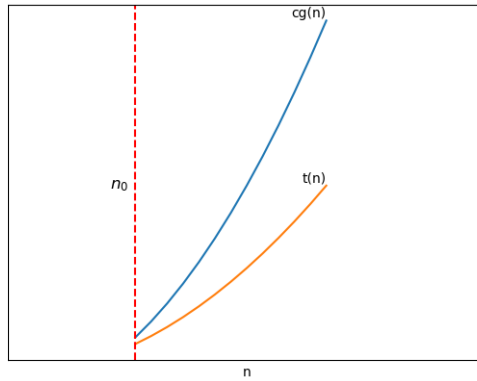
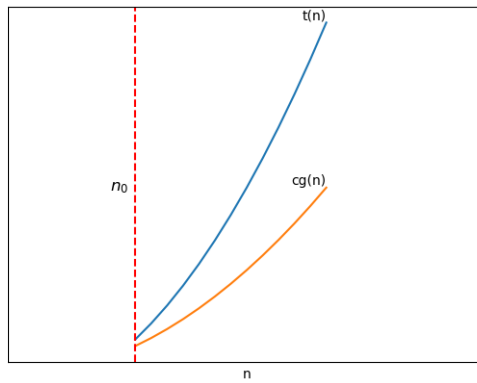
In this case, the value of  $c$  and  $n_0$  are 6 and 3 respectively.

This can also be proved with another set of values for  $c$  and  $n_0$  as shown below.

$$\begin{aligned} 5n+3 &\leq 5n+3n \quad (\text{for all } n \geq 1) = 8n \\ &\leq 8n^2 \end{aligned}$$

In this case, the value of  $c$  and  $n_0$  are 8 and 1 respectively.



Figure 2.1: Big-oh notation:  $t(n) \in O(g(n))$ Figure 2.2: Big-omega notation:  $t(n) \in \Omega(g(n))$  **$\Omega$ -notation**

A function  $t(n)$  is said to be in  $\Omega(g(n))$ , denoted  $t(n) \in \Omega(g(n))$ , if  $t(n)$  is bounded below by some constant multiple of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant  $c$  and some nonnegative integer  $n_0$  such that

$$t(n) \geq c \cdot g(n) \quad \text{for all } n \geq n_0$$

For example to prove that  $n^3 \in \Omega(n^2)$ ,

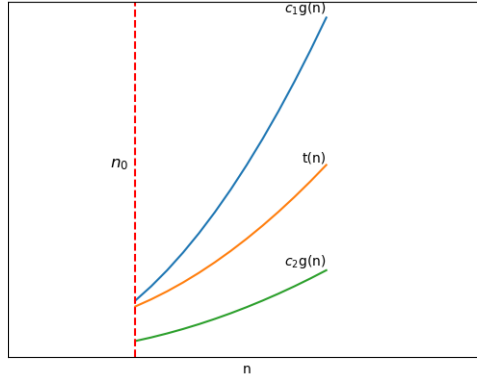
$$n^3 \geq n^2 \quad (\text{for all } n \geq 0)$$

In this case, the value of  $c$  and  $n_0$  are 1 and 0 respectively.

 **$\Theta$ -notation**

A function  $t(n)$  is said to be in  $\Theta(g(n))$ , denoted  $t(n) \in \Theta(g(n))$ , if  $t(n)$  is bounded above and below by some constant multiples of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constants  $c_1$  and  $c_2$  and some nonnegative integer  $n_0$  such that

$$c_2 \cdot g(n) \leq t(n) \leq c_1 \cdot g(n) \quad \text{for all } n \geq n_0$$

Figure 2.3: Big-theta notation:  $t(n) \in \Theta(g(n))$ 

For example to prove that  $\frac{1}{2}n(n-1) \in \Theta(n^2)$ , first prove the right inequality

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \text{ (for all } n \geq 0)$$

Then prove the left inequality

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n \cdot \frac{1}{2}n \text{ (for all } n \geq 2) = \frac{1}{4}n^2$$

In this case, the value of  $c_1 = \frac{1}{2}$ ,  $c_2 = \frac{1}{4}$  and  $n_0 = 2$ .

### 2.2.3 Useful Property Involving the Asymptotic Notations

**Theorem 2.2.1** If  $t_1(n) \in O(g_1(n))$  and  $t_2(n) \in O(g_2(n))$ , then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

**Proof**

Since  $t_1(n) \in O(g_1(n))$ , there exist some positive constant  $c_1$  and some non-negative integer  $n_1$  such that

$$t_1(n) \leq c_1 \cdot g_1(n) \text{ for all } n \geq n_1$$

Similarly, since  $t_2(n) \in O(g_2(n))$ ,

$$t_2(n) \leq c_2 \cdot g_2(n) \text{ for all } n \geq n_2$$

Let  $c_3 = \max\{c_1, c_2\}$  and consider  $n \geq \max\{n_1, n_2\}$ . Adding both inequalities above will yield

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n) \\ &\leq c_3 \cdot g_1(n) + c_3 \cdot g_2(n) = c_3(g_1(n) + g_2(n)) \\ &\leq c_3 \cdot 2(\max\{g_1(n), g_2(n)\}) \end{aligned}$$

Hence,  $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$ , with  $c = 2c_3 = 2\max\{c_1, c_2\}$  and  $n_0 = \max\{n_1, n_2\}$

This theorem can be used to check for example, whether an array has equal elements by the following two-part algorithm: first, sort the array by applying some known sorting algorithm; second, scan the sorted array to check its consecutive elements for equality. If, for example, a sorting algorithm used in the first part is in  $O(n^2)$  while the second part is in  $O(n)$ , the efficiency of the entire algorithm will be in  $O(\max\{n^2, n\}) = O(n^2)$ .

### 2.2.4 Using Limits for Comparing Orders of Growth

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n) \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n) \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n) \end{cases}$$

The limit-based approach is often more convenient than the one based on the definitions because it can take advantage of the powerful calculus techniques developed for computing limits, such as L'Hopital's rule

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{t'(n)}{g'(n)}$$

and Stirling's formula

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \text{ for large values of } n$$

■ **Example 2.1** Compare the orders of growth of  $\frac{1}{2}n(n-1)$  and  $n^2$ . ■

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{\frac{1}{2}n(n-1)}{n^2} &= \frac{1}{2} \lim_{n \rightarrow \infty} \frac{n^2 - n}{n^2} \\ &= \frac{1}{2} \lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right) \\ &= \frac{1}{2} \end{aligned}$$

Since the limit is equal to a positive constant, the functions have the same order of growth, i.e.  $\frac{1}{2}n(n-1) \in \Theta(n^2)$

■ **Example 2.2** Compare the orders of growth of  $\log_2 n$  and  $\sqrt{n}$  ■

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} &= \lim_{n \rightarrow \infty} \frac{(\log_2 n)'}{(\sqrt{n})'} \\ &= \lim_{n \rightarrow \infty} \frac{\log_2 e \frac{1}{n}}{\frac{1}{2\sqrt{n}}} \\ &= 2\log_2 e \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} \\ &= 0 \end{aligned}$$

Since the limit is equal to zero,  $\log_2 n$  has a smaller order of growth than  $\sqrt{n}$ .

■ **Example 2.3** Compare the orders of growth of  $n!$  and  $2^n$  ■

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{n!}{2^n} &= \lim_{n \rightarrow \infty} \frac{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n}{2^n} \\ &= \lim_{n \rightarrow \infty} \frac{\sqrt{2\pi n} n^n}{2^n e^n} \\ &= \lim_{n \rightarrow \infty} \sqrt{2\pi n} \left(\frac{n}{2e}\right)^n \\ &= \infty \end{aligned}$$

Since the limit is equal to  $\infty$ ,  $n!$  has a larger order of growth than  $2^n$ .

### 2.2.5 Basic Efficiency Classes

Class	Name	Comments
1	<i>constant</i>	Short of best-case efficiencies
$\log n$	<i>logarithmic</i>	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm
$n$	<i>linear</i>	Algorithms that scan a list of size $n$ (e.g., sequential search) belong to this class.
$n \log n$	<i>linearithmic</i>	Many divide-and-conquer algorithms including mergesort and quicksort in the average case, fall into this category.
$n^2$	<i>quadratic</i>	Typically, characterizes efficiency of algorithms with two embedded loops. Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
$n^3$	<i>cubic</i>	Typically, characterizes efficiency of algorithms with three embedded loops. Several nontrivial algorithms from linear algebra fall into this class.
$2^n$	<i>exponential</i>	Typical for algorithms that generate all subsets of an $n$ -element set.
$n!$	<i>factorial</i>	Typical for algorithms that generate all permutations of an $n$ -element set.

Table 2.2: Basic Efficiency Classes



# Unit II

<b>3</b>	<b>Mathematical Analysis of Algorithms . . .</b>	<b>33</b>
3.1	Mathematical Analysis of Nonrecursive Algorithms	
3.2	Mathematical Analysis of Recursive Algorithms	
<b>4</b>	<b>Brute Force and Exhaustive Search . . .</b>	<b>41</b>
4.1	Brute Force Algorithms	
4.2	Exhaustive Search	





## 3. Mathematical Analysis of Algorithms

### 3.1 Mathematical Analysis of Nonrecursive Algorithms

Steps for analyzing the time efficiency of nonrecursive algorithms are

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation, which is located in the innermost loop.
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

Some of the commonly used summation formulas and rules used in algorithm analysis are given below.

$$\begin{aligned}\sum_{i=l}^u c a_i &= c \sum_{i=l}^u a_i \\ \sum_{i=l}^u (a_i \pm b_i) &= \sum_{i=l}^u (a_i) \pm \sum_{i=l}^u (b_i) \\ \sum_{i=l}^u 1 &= u - l + 1 \\ \sum_{i=0}^n i &= \frac{n(n+1)}{2}\end{aligned}$$

■ **Example 3.1** Analyse the algorithm for finding the largest element in a list of  $n$  numbers. ■

```
1 maxval = A[0]
2 for i = 1 to n - 1 do
3     if A[i] > maxval
4         maxval = A[i]
```

```
5 return maxval
```

Listing 3.1: Pseudo code to find the largest number in a list

### Steps

1. The best choice for input size is  $n$ , the number of elements in the list.
2. Since the comparison is executed on each repetition of the loop and the assignment is not, comparison operation is selected as algorithm's basic operation.
3. Since the number of comparisons will be the same for all arrays of size  $n$ , there is no need to distinguish among the worst, average, and best cases.
4. The algorithm makes one comparison on each execution of the loop (i.e. for each value of the loop's variable  $i$  within the bounds 1 and  $n - 1$ ). Hence,

$$\begin{aligned}
 C(n) &= \sum_{i=1}^{n-1} 1 \\
 &= (n-1) - 1 + 1 \\
 &= n-1 \in \Theta(n)
 \end{aligned}$$

■ **Example 3.2** Analyse the algorithm for element uniqueness problem (check whether all the elements in a given array of  $n$  elements are distinct). ■

```
1 for i = 0 to n - 2 do
2   for j = i + 1 to n - 1 do
3     if A[i] = A[j]
4       return false
5 return true
```

Listing 3.2: Pseudo code for element uniqueness problem

### Steps

1. The best choice for input size is  $n$ , the number of elements in the list.
2. Since the innermost loop contains a single operation (the comparison of two elements), it is considered as the algorithm's basic operation..
3. The number of element comparisons depends not only on  $n$ , but also on whether there are equal elements in the array and, if there are, which array positions they occupy. The analysis will be done on worst case.
4.  $C_{worst}(n)$  is the largest among all arrays of size  $n$ , which happens when there are no equal elements or when the last two elements are the only pair of equal elements. For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of  $j$  between  $i + 1$  and  $n - 1$ . This is repeated for each value of the outer loop, i.e., for each value of  $i$  between 0 and  $n - 2$ . Hence,

$$\begin{aligned}
C_{\text{worst}}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\
&= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] \\
&= \sum_{i=0}^{n-2} (n-1-i) \\
&= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \\
&= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\
&= (n-1)^2 - \frac{(n-2)(n-1)}{2} \\
&= \frac{n(n-1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2)
\end{aligned}$$

■ **Example 3.3** Analyse the algorithm for multiplying two  $n \times n$  matrices. ■

```

1 for i = 0 to n - 1 do
2   for j = 0 to n - 1 do
3     C[i,j] = 0.0
4     for k = 0 to n - 1 do
5       C[i,j] = C[i,j] + A[i,k] * B[k,j]
6 return C

```

Listing 3.3: Pseudo code for matrix multiplication

#### Steps

1. The best choice for input size is  $n$ , the order of the matrix.
2. The innermost loop contains 2 operations (addition and subtraction). But each is executed exactly once. So multiplication can be taken as the basic operation.
3. Since the number of times basic operation is executed depends only on the size of the input matrices, there is no need to investigate the worst-case, average-case, and best-case efficiencies separately.
4. One multiplication operation is done for each repetition of the innermost loop, i.e., for each value of  $k$  between 0 and  $n-1$ . This is repeated for each pair of values of the outer loops, i.e., for each value of  $i$  between 0 and  $n-1$  and for each value of  $j$  between 0 and  $n-1$ . Hence,

$$\begin{aligned}
C(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 \\
&= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n \\
&= \sum_{i=0}^{n-1} n^2 \\
&= n^3 \in \Theta(n^3)
\end{aligned}$$

■ **Example 3.4** Analyse the algorithm for finding the number of binary digits in the binary representation of a positive decimal integer. ■

```

1 count = 1
2 while n > 1 do
3   count = count + 1
4   n = ⌊n/2⌋
5 return count

```

Listing 3.4: Pseudo code for finding the number of bits required to represent a decimal number

In this case, the loop variable takes only a few values between its lower and upper limits. Since the value of  $n$  is about halved on each repetition of the loop, the number of times the loop is executed is about  $\log_2 n$ . So this algorithm falls in  $\Theta(\log n)$ .

### 3.2 Mathematical Analysis of Recursive Algorithms

Steps for analyzing the time efficiency of recursive algorithms are

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation, which is located in the innermost loop.
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

■ **Example 3.5** Analyse the algorithm to find the factorial of a non negative integer  $n$ . ■

```

1 if n=0
2   return 1
3 else
4   return F(n-1)*n

```

Listing 3.5: Pseudo code for factorial -  $F(n)$

Let  $n$  be the measure of input size. The basic operation of the algorithm is multiplication. The number of times the basic operation is executed can be represented using the recurrence relation

$$M(n) = M(n-1) + 1, \text{ for } n > 0$$

When  $n = 0$ , the algorithm performs no multiplications. So, the initial condition can be written as

$$M(0) = 0$$

This recurrence relation can be solved using the method of *backward substitution* as given below.

$$\begin{aligned}
 M(n) &= M(n-1) + 1 \\
 &= [M(n-2) + 1] + 1 = M(n-2) + 2 \\
 &= [M(n-3) + 1] + 2 = M(n-3) + 3
 \end{aligned}$$

General formula for this pattern can be written as

$$M(n) = M(n-i) + i$$

To take advantage of the initial condition  $M(0) = 0$ , substitute  $i$  with  $n$ . Then

$$\begin{aligned} M(n) &= M(n-i) + i \\ &= M(n-n) + n \\ &= M(0) + n \\ &= n \end{aligned}$$

■ **Example 3.6** Analyse the algorithm to solve the tower of hanoi puzzle. ■

In this puzzle, there are  $n$  disks of different sizes that can slide onto any of three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. Only one disk can be moved at a time, and it is forbidden to place a larger disk on top of a smaller one.

Let  $H(n, \text{source}, \text{target}, \text{auxiliary})$  denotes, move  $n$  disks from source peg to target peg using auxiliary peg.

```
1 H(n-1, source, auxiliary, target)
2 Mov 1 disk from source to target
3 H(n-1, auxiliary, target, source)
```

Listing 3.6: Pseudo code for Tower of Hanoi Puzzle -  $H(n, \text{source}, \text{target}, \text{auxiliary})$

The number of disks,  $n$ , is the input's size indicator and moving one disk is the algorithm's basic operation. The number of moves  $M(n)$  required can be represented using the recurrence relation given below.

$$M(n) = M(n-1) + 1 + M(n-1), \text{ for } n > 1$$

The initial condition is

$$M(1) = 1$$

This recurrence can be solved as shown below.

$$\begin{aligned} M(n) &= M(n-1) + 1 + M(n-1) = 2M(n-1) + 1 \\ &= 2[2M(n-2) + 1] + 1 = 2^2M(n-2) + 2 + 1 \\ &= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3M(n-3) + 2^2 + 2 + 1 \end{aligned}$$

General formula for this pattern can be written as

$$\begin{aligned} M(n) &= 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \dots + 2 + 1 \\ &= 2^i M(n-i) + 2^i - 1 \end{aligned}$$

To take advantage of the initial condition  $M(1) = 1$ , substitute  $i$  with  $n-1$ . Then

$$\begin{aligned} M(n) &= 2^i M(n-i) + 2^i - 1 \\ &= 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1 \\ &= 2^{n-1} M(1) + 2^{n-1} - 1 \\ &= 2^{n-1} + 2^{n-1} - 1 \\ &= 2^n - 1 \end{aligned}$$

Hence this algorithm has an exponential running time.

■ **Example 3.7** Analyse the recursive algorithm to find the number of bits required to represent a decimal number. ■

```

1 if n=1
2   return 1
3 else
4   return Bin( $\lfloor \frac{n}{2} \rfloor$ ) + 1

```

Listing 3.7: Pseudo code for finding the number of bits - Bin(n)

The basic operation in this algorithm is the addition operation. The number of addition operations required can be represented using the recurrence relation.

$$A(n) = A(\lfloor \frac{n}{2} \rfloor) + 1, \text{ for } n > 1$$

The initial condition is

$$A(1) = 0$$

This recurrence can be solved as follows.

$$\begin{aligned}
 A(n) &= A(\lfloor \frac{n}{2} \rfloor) + 1 \\
 &= A(\lfloor \frac{n}{4} \rfloor) + 1 + 1 = A(\lfloor \frac{n}{4} \rfloor) + 2 \\
 &= A(\lfloor \frac{n}{8} \rfloor) + 1 + 2 = A(\lfloor \frac{n}{8} \rfloor) + 3
 \end{aligned}$$

General formula for this pattern can be written as

$$A(n) = A(\lfloor \frac{n}{2^i} \rfloor) + i$$

To take advantage of the initial condition  $A(1) = 0$ , substitute  $i$  with  $\log_2 n$ . Then

$$\begin{aligned}
 A(n) &= A(\lfloor \frac{n}{2^i} \rfloor) + i \\
 &= A(\lfloor \frac{n}{2^{\log_2 n}} \rfloor) + \log_2 n \\
 &= A(1) + \log_2 n \\
 &= \log_2 n \in \Theta(\log_2 n)
 \end{aligned}$$

### 3.2.1 Example: Computing the $n^{\text{th}}$ Fibonacci Number

Fibonacci sequence is

$$0, 1, 1, 2, 3, 5, 8, 13, \dots$$

This sequence can be represented using the recurrence

$$F(n) = F(n-1) + F(n-2), \text{ for } n > 1$$

and two initial conditions

$$F(0) = 0, F(1) = 1$$

The method of backward substitutions to solve this recurrence will fail to give an easily discernible pattern.

Instead, a theorem that describes solutions to a homogeneous second-order linear recurrence with constant coefficients can be used.

$$ax(n) + bx(n-1) + cx(n-2) = 0$$

where  $a$ ,  $b$ , and  $c$  are some fixed real numbers ( $a \neq 0$ ) called the coefficients of the recurrence and  $x(n)$  is the generic term of an unknown sequence to be found.

Applying this theorem to Fibonacci recurrence with the initial conditions given,

$$F(n) = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$$

where

$$\phi = \frac{(1 + \sqrt{5})}{2} \approx 1.61803$$

and

$$\hat{\phi} = \frac{-1}{\phi} \approx -0.61803$$

The algorithm to find the  $n^{\text{th}}$  Fibonacci number is

```

1 if n ≤ 1
2   return n
3 else
4   return Fib(n - 1) + Fib(n - 2)
```

Listing 3.8: Pseudo code for finding  $n^{\text{th}}$  Fibonacci number - Fib( $n$ )

The algorithm's basic operation is addition. So let  $A(n)$  be the number of additions performed by the algorithm in computing Fib( $n$ ). Then the recurrence for computing the number of additions can be written as

$$A(n) = A(n-1) + A(n-2) + 1, \text{ for } n > 1$$

and two initial conditions

$$A(0) = 0, A(1) = 0$$

Since the right hand side of the recurrence  $A(n) - A(n-1) - A(n-2) = 1$  is not zero the recurrence is called inhomogeneous. Hence, this recurrence can be rewritten as

$$[A(n) + 1] - [A(n-1) + 1] - [A(n-2) + 1] = 0$$

Let  $B(n) = A(n+1)$ . Hence

$$B(n) - B(n-1) - B(n-2) = 0$$

$$B(0) = 1, B(1) = 1$$

$B(n)$  is the same recurrence as  $F(n)$  except that it starts with two 1's and thus runs one step ahead of  $F(n)$ . So  $B(n) = F(n+1)$ .

$$\begin{aligned}
A(n) &= B(n) - 1 \\
&= F(n+1) - 1 \\
&= \frac{1}{\sqrt{5}}(\phi^{n+1} - \hat{\phi}^{n+1}) - 1
\end{aligned}$$

Hence,  $A(n) \in \Theta(\phi^n)$ , and if size of  $n$  is measured by the number of bits  $b = \lfloor \log_2 n + 1 \rfloor$  in its binary representation, the efficiency class will be even worse, doubly exponential:  $A(b) \in \Theta(\phi^{2^b})$ .

A much faster algorithm for finding  $n^{\text{th}}$  Fibonacci number is given below.

```
1 F[0] = 0
2 F[1] = 1
3 for i = 2 to n do
4   F[i] = F[i - 1] + F[i - 2]
5 return F[n]
```

Listing 3.9: Pseudo code for finding  $n^{\text{th}}$  Fibonacci number - Nonrecursive

This algorithm clearly makes  $n - 1$  additions. Hence,

$$A(n) \in \Theta(n)$$

$$A(b) \in \Theta(2^b)$$



## 4. Brute Force and Exhaustive Search

### 4.1 Brute Force Algorithms

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

- Brute force is applicable to a very wide variety of problems.
- The expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed.
- Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem.

#### 4.1.1 Selection Sort

Selection sorting algorithm is given below.

```
1 for i = 0 to n - 2 do
2   min = i
3   for j = i + 1 to n - 1 do
4     if A[j] < A[min]
5       min = j
6   swap A[i] and A[min]
```

Listing 4.1: Selection sort

#### Analysis of selection sort

The input size is given by the number of elements  $n$ . The basic operation is the key comparison  $A[j] < A[\text{min}]$ . The number of times it is executed depends only on the array size and is given by

the following sum

$$\begin{aligned}
 C(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\
 &= \sum_{i=0}^{n-2} (n-1-(i+1)+1) \\
 &= \sum_{i=0}^{n-2} (n-1-i) \\
 &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} (i) \\
 &= (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} (i) \\
 &= (n-1)(n-1) - \frac{(n-1)(n-2)}{2} \\
 &= (n-1) \left[ (n-1) - \frac{(n-2)}{2} \right] \\
 &= (n-1) \left[ \frac{2n-2-n+2}{2} \right] \\
 &= (n-1) \left[ \frac{n}{2} \right] \\
 &= \frac{n(n-1)}{2}
 \end{aligned}$$

Thus, selection sort is a  $\Theta(n^2)$  algorithm on all inputs. The number of key swaps is only  $\Theta(n)$ .

#### 4.1.2 Bubble Sort

```

1 for i = 0 to n - 2 do
2   for j = 0 to n - 2 - i do
3     if A[j] > A[j+1]
4       swap A[j] and A[j+1]

```

Listing 4.2: Bubble sort

##### Analysis of bubble sort

The input size is given by the number of elements  $n$ . The basic operation is the key comparison  $A[j] > A[j+1]$ . The number of times it is executed depends only on the array size and is given by

the following sum

$$\begin{aligned}
 C(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 \\
 &= \sum_{i=0}^{n-2} (n-2-i-0+1) \\
 &= \sum_{i=0}^{n-2} (n-1-i) \\
 &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} (i) \\
 &= (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} (i) \\
 &= (n-1)(n-1) - \frac{(n-1)(n-2)}{2} \\
 &= (n-1) \left[ (n-1) - \frac{(n-2)}{2} \right] \\
 &= (n-1) \left[ \frac{2n-2-n+2}{2} \right] \\
 &= (n-1) \left[ \frac{n}{2} \right] \\
 &= \frac{n(n-1)}{2}
 \end{aligned}$$

Thus, bubble sort is a  $\Theta(n^2)$  algorithm on all inputs. The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons

$$S_{\text{worst}}(n) = C(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

### 4.1.3 Sequential Search

The sequential search algorithm compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search).

```

1 A[n] = K
2 i = 0
3 While A[i] ≠ K do
4   i = i + 1
5 if i < n
6   return i
7 else
8   return -1

```

Listing 4.3: Sequential Search

The sequential search algorithm is linear ( $\Theta(n)$ ) in both worst and average cases.

### 4.1.4 Brute-Force String Matching

Given a string of  $n$  characters called the **text** and a string of  $m$  characters ( $m < n$ ) called the **pattern**, find a substring of the **text** that matches the **pattern**. The algorithm should return  $i$  - the index of

the leftmost character of the first matching substring in the **text** - such that  $t_i = p_0, \dots, t_{i+j} = p_j, \dots, t_{i+m-1} = p_{m-1}$ .

```

1 for i = 0 to n - m do
2   j = 0
3   while j < m and P[j] = T[i + j] do
4     j = j + 1
5   if j = m
6     return i
7 return -1

```

Listing 4.4: String matching

The algorithm may have to make all  $m$  comparisons before shifting the pattern, and this can happen for each of the  $n - m + 1$  tries. Thus, in the worst case, the algorithm makes  $m(n - m + 1)$  character comparisons, which puts it in the  $O(nm)$  class. However, for a typical word search in a natural language text, it has been shown to be linear, i.e.,  $\Theta(n)$ .

## 4.2 Exhaustive Search

Exhaustive search is a brute-force approach to combinatorial problems. It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element (e.g., the one that optimizes some objective function).

### 4.2.1 Traveling Salesman Problem (TSP)

The objective of TSP is to find the shortest tour through a given set of  $n$  cities that visits each city exactly once before returning to the city where it started. The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph. A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

A Hamiltonian circuit can also be defined as a sequence of  $n + 1$  adjacent vertices  $v_0, v_1, \dots, v_{n-1}, v_0$ , where the first vertex of the sequence is the same as the last one and all the other  $n - 1$  vertices are distinct. The shortest tour can be obtained by generating all the permutations of  $n - 1$  intermediate cities, calculating the tour lengths for each permutation, and then identifying the shortest among them. For example, consider the graph in Figure 4.1.

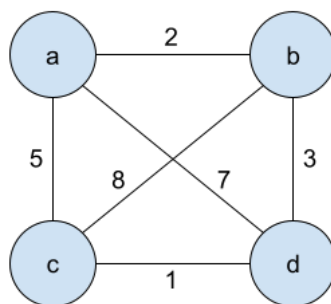


Figure 4.1: TSP example

Since three pairs of tours differ only by their direction, the number of vertex permutations can be cut by half. The total number of permutations needed is still  $\frac{1}{2}(n - 1)!$ , which makes the exhaustive search approach impractical for all but very small values of  $n$ .

Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	18	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	11	Optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	23	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	11	Optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	23	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	18	

Table 4.1: Solution to TSP in Figure 4.1

### 4.2.2 Knapsack Problem

Given  $n$  items of known weights  $w_1, w_2, \dots, w_n$  and values  $v_1, v_2, \dots, v_n$  and a knapsack of capacity  $W$ , find the most valuable subset of the items that fit into the knapsack. The exhaustive-search approach to this problem is as follows:

1. Generate all the subsets of the set of  $n$  items given.
2. Compute the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity).
3. Find a subset of the largest value among them.

Since the number of subsets of an  $n$ -element set is  $2^n$ , the exhaustive search leads to a  $\Omega(2^n)$  algorithm.

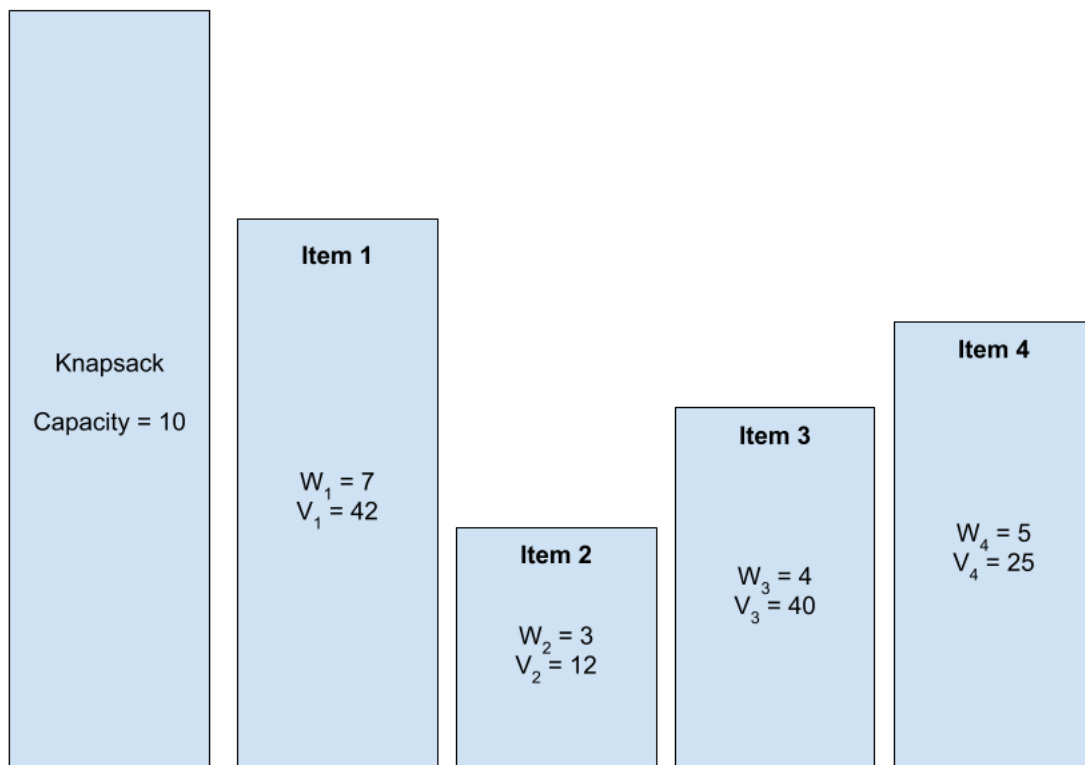


Figure 4.2: Knapsack example

TSP and Knapsack problems are the best-known examples of so-called **NP-hard** problems. No polynomial-time algorithm is known for any NP-hard problem.

Subset	Total Weight	Total Value
$\phi$	0	0
{1}	7	42
{2}	3	12
{3}	4	40
{4}	5	25
{1,2}	10	54
{1,3}	11	not feasible
{1,4}	12	not feasible
{2,3}	7	52
{2,4}	8	37
<b>{3,4}</b>	<b>9</b>	<b>65</b>
{1,2,3}	14	not feasible
{1,2,4}	15	not feasible
{1,3,4}	16	not feasible
{2,3,4}	12	not feasible
{1,2,3,4}	19	not feasible

Table 4.2: Solution to Knapsack problem in Figure 4.2

### 4.2.3 Assignment Problem

There are  $n$  people who need to be assigned to execute  $n$  jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the  $i^{\text{th}}$  person is assigned to the  $j^{\text{th}}$  job is a known quantity  $C[i, j]$  for each pair  $i, j = 1, 2, \dots, n$ . The problem is to find an assignment with the minimum total cost.

Feasible solutions to the assignment problem can be considered as  $n$ -tuples  $\langle j_1, \dots, j_n \rangle$  in which the  $i^{\text{th}}$  component,  $i = 1, \dots, n$ , indicates the column of the element selected in the  $i^{\text{th}}$  row (i.e., the job number assigned to the  $i^{\text{th}}$  person).

For example, consider the cost matrix given below.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

Table 4.3: Sample cost matrix for Assignment problem

Here,  $\langle 2, 3, 4, 1 \rangle$  indicates the assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1.

The exhaustive-search approach to the assignment problem would require generating all the permutations of integers  $1, 2, \dots, n$ , computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum.

Since the number of permutations to be considered for the general case of the assignment problem is  $n!$ , exhaustive search is impractical for all but very small instances of the problem.

### 4.2.4 Depth First Search Algorithm (DFS)

DFS algorithm is a graph traversal algorithm which visits each vertex in a graph which works as follows.

1. Start at an arbitrary vertex in the graph.
  2. Mark the current vertex as visited.
  3. Find an unvisited vertex adjacent to the current vertex. If there are multiple unvisited adjacent vertices, choose the one with the lowest alphabetical order (or any arbitrary tie-breaking rule).
  4. Move to the chosen unvisited vertex and repeat steps 2 and 3 (recursive call).
  5. If the chosen unvisited vertex has no adjacent unvisited vertices, backtrack (back up one edge) to the previous vertex.
  6. Try to continue visiting unvisited vertices from the previous vertex by going back to step 3.
  7. Continue this process until a dead end is reached.
  8. When a dead end is reached, backtrack again to the previous vertex and try to continue visiting unvisited vertices from there.
  9. Repeat steps 7 and 8 until starting vertex is reached.
  10. When starting vertex is reached again, the algorithm halts.
- The pseudo code is given below.

```

1 DFS(G)
2   mark each vertex in V with 0 as a mark of being "unvisited"
3   count = 0
4   for each vertex v in V do
5     if v is marked with 0
6       dfs(v)
7
8 dfs(v)
9   count = count + 1
10  mark v with count
11  for each vertex w in V adjacent to v do
12    if w is marked with 0
13      dfs(w)

```

Listing 4.5: DFS Algorithm

The algorithm takes just the time proportional to the size of the data structure used for representing the graph. Thus, for the adjacency matrix representation, the traversal time is in  $\Theta(|V|^2)$ , and for the adjacency list representation, it is in  $\Theta(|V| + |E|)$ , where  $|V|$  and  $|E|$  are the number of the vertices and edges of the graph, respectively.

Working of DFS algorithm on a graph is explained in Figure 4.3.

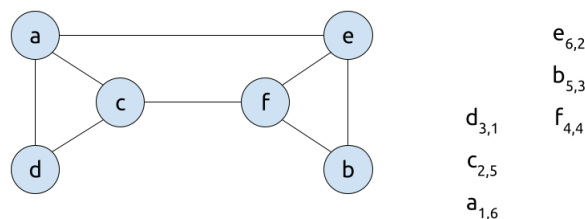


Figure 4.3: Working of DFS Algorithm

In the traversal's stack, shown along with the graph, the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack and the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack.

#### 4.2.5 Breadth First Search Algorithm (BFS)

BFS algorithm visits all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the

starting vertex are visited. The pseudo code is given below.

```

1 BFS(G)
2   mark each vertex in V with 0 as a mark of being "unvisited"
3   count = 0
4   for each vertex v in V do
5       if v is marked with 0
6           bfs(v)
7
8 bfs(v)
9   count = count + 1
10  mark v with count and initialize a queue with v
11  while the queue is not empty do
12      for each vertex w in V adjacent to the front vertex do
13          if w is marked with 0
14              count = count + 1
15              mark w with count
16              add w to the queue
17  remove the front vertex from the queue

```

Listing 4.6: BFS Algorithm

The algorithm takes just the time proportional to the size of the data structure used for representing the graph. Thus, for the adjacency matrix representation, the traversal time is in  $\Theta(|V|^2)$ , and for the adjacency list representation, it is in  $\Theta(|V| + |E|)$ , where  $|V|$  and  $|E|$  are the number of the vertices and edges of the graph, respectively.

Working of BFS algorithm on a graph is explained in Figure 4.4. BFS yields a single ordering

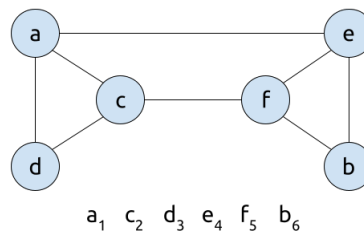


Figure 4.4: Working of BFS

of vertices because the queue is a FIFO (first-in first-out) structure and hence the order in which vertices are added to the queue is the same order in which they are removed from it.





# Unit III

<b>5</b>	<b>Divide and Conquer</b> .....	<b>51</b>
5.1	Introduction	
5.2	Mergesort	
5.3	Quicksort	
5.4	Binary Search	
5.5	Tree Traversals	
<b>6</b>	<b>Dynamic Programming</b> .....	<b>59</b>
6.1	Introduction	
6.2	Basic Examples	
6.3	The Knapsack Problem and Memory Functions	
6.4	Warshall's Algorithm	
6.5	Floyd's Algorithm for the All-Pairs Shortest-Paths Problem	
<b>7</b>	<b>Greedy Technique</b> .....	<b>65</b>
7.1	Prim's Algorithm	
7.2	Dijkstra's Algorithm	



## 5. Divide and Conquer

### 5.1 Introduction

Divide-and-conquer algorithms work as follows:

1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

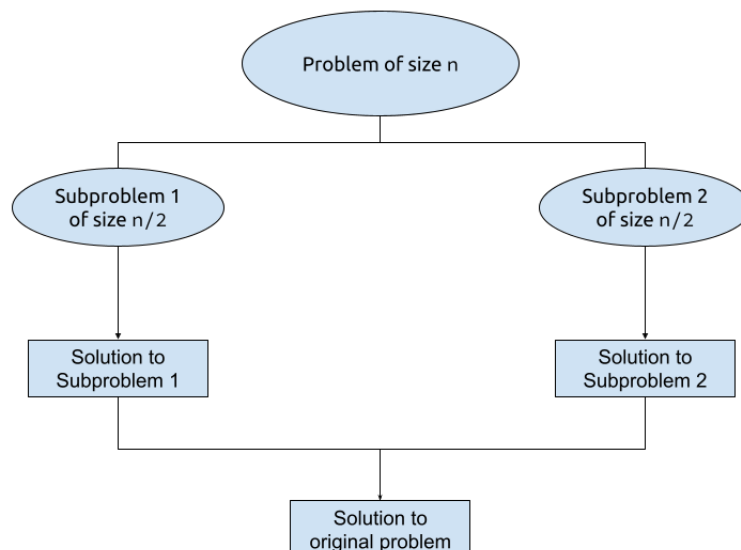


Figure 5.1: Divide and conquer technique

An instance of size  $n$  can be divided into  $b$  instances of size  $\frac{n}{b}$ , with  $a$  of them needing to be solved, where  $a$  and  $b$  are constants and  $a \geq 1$ ,  $b > 1$ . Assuming that size  $n$  is a power of  $b$ , the following recurrence can be written for the running time  $T(n)$ :

$$T(n) = aT(n/b) + f(n)$$

where  $f(n)$  is a function that accounts for the time spent on dividing an instance of size  $n$  into instances of size  $\frac{n}{b}$  and combining their solutions. This recurrence is called the **general divide-and-conquer recurrence**. The efficiency analysis of many divide-and-conquer algorithms can be obtained using **Master theorem**.

### 5.1.1 Master Theorem

If  $f(n) \in \Theta(n^d)$ , where  $d \geq 0$  in the general divide-and-conquer recurrence, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

For example, consider problem of computing the sum of  $n$  numbers  $a_0, \dots, a_{n-1}$ . If  $n > 1$ , the problem can be divided into two instances of the same problem: to compute the sum of the first  $\frac{n}{2}$  numbers and to compute the sum of the remaining  $\frac{n}{2}$  numbers. Once each of these two sums is computed by applying the same method recursively, their values can be added to get the sum in question. The recurrence for the number of additions  $A(n)$  made by this algorithm on inputs of size  $n = 2^k$  is

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

For applying Master theorem to this recurrence,  $a = 2$ ,  $b = 2$  and  $d = 0$ . Since  $a > b^d$

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

## 5.2 Mergesort

Mergesort sorts a given array  $A[0..n-1]$  by dividing it into two halves  $A[0..\lfloor n/2 \rfloor - 1]$  and  $A[\lfloor n/2 \rfloor..n-1]$ , sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

```

1 Mergesort(A[0..n-1])
2   if n > 1
3     copy A[0..⌊n/2⌋ - 1] to B[0..⌊n/2⌋ - 1]
4     copy A[⌊n/2⌋..n-1] to C[0..⌊n/2⌋ - 1]
5     Mergesort(B[0..⌊n/2⌋ - 1])
6     Mergesort(C[0..⌊n/2⌋ - 1])
7     Merge(B, C, A)
```

Listing 5.1: Mergesort Algorithm

```

1 Merge(B[0..p - 1], C[0..q - 1], A[0..p + q - 1])
2   i = 0
3   j = 0
4   k = 0
5   while i < p and j < q do
6     if B[i] <= C[j]
7       A[k] = B[i];
8       i = i + 1
9     else
10      A[k] = C[j]
11      j = j + 1
```

```

12   k=k+1
13   if i = p
14       copy C[j..q - 1] to A[k..p + q - 1]
15   else
16       copy B[i..p - 1] to A[k..p + q - 1]

```

Listing 5.2: Merge Algorithm

The operation of the mergesort algorithm on the list 9, 4, 3, 8, 6, 2, 6, 5 is illustrated in Figure 5.2

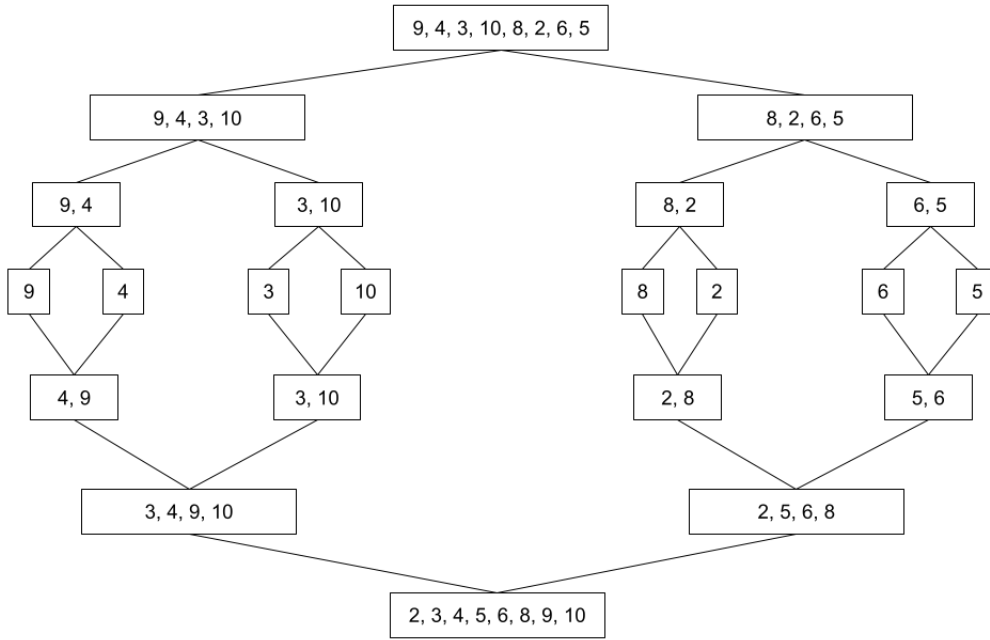


Figure 5.2: Operation of Mergesort algorithm

### 5.2.1 Analysis of Mergesort

The recurrence relation for the number of key comparisons  $C(n)$  is

$$C(n) = 2 C\left(\frac{n}{2}\right) + C_{\text{merge}}(n) \quad \text{for } n > 1$$

$$C(1) = 0$$

$C_{\text{merge}}(n)$  is the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two arrays becomes empty before the other one contains just one element. Therefore, for the worst case,  $C_{\text{merge}}(n) = n - 1$ . Therefore the recurrence is

$$C_{\text{worst}}(n) = 2 C_{\text{worst}}\left(\frac{n}{2}\right) + n - 1 \quad \text{for } n > 1$$

$$C_{\text{worst}}(1) = 0$$

#### Using Master Theorem

In the above recurrence  $a = 2$ ,  $b = 2$ ,  $d = 1$ . Therefore  $a = b^d$ . Hence

$$C_{\text{worst}}(n) \in \Theta(n \log n)$$

## Using Backward Substitution

$$\begin{aligned}
C_{\text{worst}}(n) &= 2 C_{\text{worst}}\left(\frac{n}{2}\right) + n - 1 \\
&= 2 \left[ 2 C_{\text{worst}}\left(\frac{n}{4}\right) + \frac{n}{2} - 1 \right] + n - 1 \\
&= 4 C_{\text{worst}}\left(\frac{n}{4}\right) + 2n - 3 \\
&= 4 \left[ 2 C_{\text{worst}}\left(\frac{n}{8}\right) + \frac{n}{4} - 1 \right] + 2n - 3 \\
&= 8 C_{\text{worst}}\left(\frac{n}{8}\right) + 3n - 7 \\
&= 2^i C_{\text{worst}}\left(\frac{n}{2^i}\right) + in - (2^i - 1)
\end{aligned}$$

Let  $n = 2^i$

$$\begin{aligned}
C_{\text{worst}}(n) &= n C_{\text{worst}}(1) + n \log_2 n - (n - 1) \\
&= n \log_2 n - n + 1 \\
&\in \Theta(n \log n)
\end{aligned}$$

### 5.3 Quicksort

Quicksort divides the elements of an array into partitions according to their value. A partition is an arrangement of the array's elements so that all the elements to the left of some element  $A[s]$  (known as the pivot element) are less than or equal to  $A[s]$ , and all the elements to the right of  $A[s]$  are greater than or equal to it. After a partition is achieved,  $A[s]$  will be in its final position in the sorted array, and two subarrays to the left and to the right of  $A[s]$  are sorted recursively.

$$\underbrace{A[0] \dots A[s-1]}_{\leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\geq A[s]}$$

```

1 Quicksort(A[l..r])
2   if l < r
3     s = Partition(A[l..r])
4     Quicksort(A[l..s - 1])
5     Quicksort(A[s + 1..r])

```

Listing 5.3: Quicksort Algorithm

The partition algorithm given below uses the first element of the array as the pivot element based on which the array is partitioned.

```

1 Partition(A[l..r])
2 p = A[l]
3 i = l
4 j = r + 1
5 repeat
6   repeat i = i + 1
7     until A[i] ≥ p
8   repeat j = j - 1
9     until A[j] ≤ p
10  swap(A[i], A[j])

```

```

11 until i ≥ j
12 swap(A[i], A[j]) //undo last swap when i ≥ j
13 swap(A[l], A[j])
14 return j

```

Listing 5.4: Partition Algorithm

Working of quick sort on an array with elements 5, 3, 1, 9, 8, 2, 4, 7 is shown below.

0	1	2	3	4	5	6	7
	<i>i</i>						<i>j</i>
5	3	1	9	8	2	4	7
			<i>i</i>			<i>j</i>	
5	3	1	9	8	2	4	7
			<i>i</i>			<i>j</i>	
5	3	1	4	8	2	9	7
				<i>i</i>	<i>j</i>		
5	3	1	4	8	2	9	7
				<i>i</i>	<i>j</i>		
5	3	1	4	2	8	9	7
				<i>j</i>	<i>i</i>		
5	3	1	4	2	8	9	7
2	3	1	4	5	8	9	7
	<i>i</i>		<i>j</i>				
2	3	1	4				
	<i>i</i>	<i>j</i>					
2	3	1	4				
	<i>i</i>	<i>j</i>					
2	1	3	4				
	<i>j</i>	<i>i</i>					
2	1	3	4				
1	2	3	4				
1							
			<i>ij</i>				
		3	4				
		<i>j</i>	<i>i</i>				
		3	4				
			4				
					<i>i</i>	<i>j</i>	
					8	9	7
					<i>i</i>	<i>j</i>	
					8	7	9
					<i>j</i>	<i>i</i>	
					8	7	9
					7	8	9
					7		
						9	

### 5.3.1 Analysis of Quicksort

The number of key comparisons made before a partition is achieved is  $n + 1$  if the scanning indices cross over and  $n$  if they coincide. If all the splits happen in the middle of corresponding

subarrays, then that will be the best case. The number of key comparisons in the best case satisfies the recurrence

$$C_{best}(n) = 2 C_{best}\left(\frac{n}{2}\right) + n \quad \text{for } n > 1$$

Applying Master theorem  $a = 2$ ,  $b = 2$  and  $d = 1$ ,  $a = b^d$ . Therefore

$$C_{best}(n) \in \Theta(n \log_2 n)$$

In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. This situation will happen if the array is already sorted. So, after making  $n + 1$  comparisons to get to this partition and exchanging the pivot  $A[0]$  with itself, the algorithm will be left with the strictly increasing array  $A[1..n - 1]$  to sort. This will continue until  $A[n - 2..n - 1]$  has been processed. The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n + 1) + n + \dots + 3 = \frac{(n + 1)(n + 2)}{2} - 3 \in \Theta(n^2)$$

In the average case, a partition can happen in any position  $s$  ( $0 \leq s \leq n - 1$ ) after  $n + 1$  comparisons are made to achieve the partition. After the partition, the left and right subarrays will have  $s$  and  $n - 1 - s$  elements, respectively. Assuming that the partition split can happen in each position  $s$  with the same probability  $1/n$ ,

$$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(n + 1) + C_{avg}(s) + C_{avg}(n - 1 - s)] \quad \text{for } n > 1$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0$$

Solving this yields

$$C_{avg}(n) \approx 2n \ln n \approx 1.39 n \log_2 n$$

Quicksort algorithm can be refined by better pivot selection methods such as randomized quicksort that uses a random element or the median-of-three method that uses the median of the leftmost, rightmost, and the middle element of the array.

## 5.4 Binary Search

Binary search algorithm is used to search for a key element in a sorted array. It works by comparing a search key  $K$  with the array's middle element  $A[m]$ . If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if  $K < A[m]$ , and for the second half if  $K > A[m]$ .

```

1 BinarySearch(A[0..n - 1], K)
2   l = 0
3   r = n - 1
4   while l ≤ r do
5     m = ⌊(l + r)/2⌋
6     if K = A[m]
7       return m
8     else if K < A[m]
9       r = m - 1
10    else
11      l = m + 1
12  return -1

```

Listing 5.5: Binary Search Algorithm



Worst case happens when the key is not in the array. The recurrence for binary search is

$$C_{\text{worst}}(n) = C_{\text{worst}}(n/2) + 1$$

Applying Master theorem,  $a = 1$ ,  $b = 2$ ,  $d = 0$  gives  $a = b^d$ .

$$C_{\text{worst}}(n) = \Theta(\log n)$$

## 5.5 Tree Traversals

Traversal operation visits each node in the tree exactly once. Three types of traversals are

- Preorder Traversal
- Inorder Traversal
- Postorder Traversal

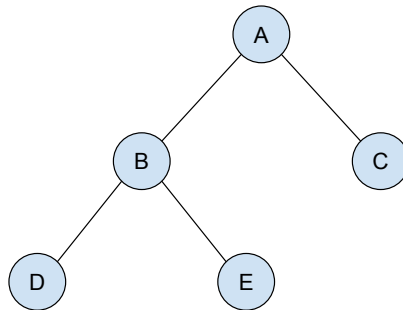


Figure 5.3: Binary Tree for Traversal

### 5.5.1 Preorder Traversal

Steps of preorder traversal are

1. Visit the root node  $R$ .
2. Traverse the left subtree of  $R$  in preorder
3. Traverse the right subtree of  $R$  in preorder

The preorder traversal of the tree shown in Figure 5.3 is  $A, B, D, E, C$ .

### 5.5.2 Inorder Traversal

Steps of inorder traversal are

1. Traverse the left subtree of  $R$  in inorder.
2. Visit the root node  $R$ .
3. Traverse the right subtree of  $R$  in inorder.

The inorder traversal of the tree shown in Figure 5.3 is  $D, B, E, A, C$ .

### 5.5.3 Postorder Traversal

Steps of postorder traversal are

1. Traverse the left subtree of  $R$  in postorder.
2. Traverse the right subtree of  $R$  in postorder.
3. Visit the root node  $R$ .

The postorder traversal of the tree shown in Figure 5.3 is  $D, E, B, C, A$ .

### 5.5.4 Analysis of Tree Traversal Algorithms

The time taken to traverse a tree with  $n$  nodes is

$$T(n) = T(k) + T(n - k - 1) + c$$

Where  $k$  is the number of nodes in the left subtree and  $c$  is a constant. Analysis can be done on two cases

#### Case 1: Skewed tree

This case occurs when one of the subtree is empty. In this case,  $k = 0$ .

$$\begin{aligned} T(n) &= T(0) + T(n - 1) + c \\ &= T(0) + (T(0) + T(n - 2) + c) + c \\ &= 2T(0) + T(n - 2) + 2c \\ &= 2T(0) + (T(0) + T(n - 3) + c) + 2c \\ &= 3T(0) + T(n - 3) + 3c \\ &= \dots \\ &= nT(0) + T(0) + nc \\ &= nc \\ &\in \Theta(n) \end{aligned}$$

#### Case 2: When both subtrees have same size

$$T(n) = 2T\left(\frac{n}{2}\right) + c$$

Applying Master's theorem,  $a = 2$ ,  $b = 2$  and  $d = 0$ .  $a > b^d$ . Hence

$$T(n) \in \Theta(n^{\log_2 2}) \in \Theta(n)$$

## 6. Dynamic Programming

### 6.1 Introduction

Dynamic programming was invented by U.S. mathematician, Richard Bellman, in the 1950s. Dynamic programming is a technique for solving problems with overlapping subproblems. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

### 6.2 Basic Examples

#### 6.2.1 Coin-row Problem

There is a row of  $n$  coins whose values are some positive integers  $c_1, c_2, \dots, c_n$ , not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

Let  $F(n)$  be the maximum amount that can be picked up from the row of  $n$  coins. To derive a recurrence for  $F(n)$ , all the allowed coin selections can be partitioned into two groups:

1. Those that include the last coin  $c_n$ . In this case, the largest amount that can be obtained is

$$c_n + F(n-2)$$

2. Those that does not include the last coin  $c_n$ . In this case, the largest amount that can be obtained is

$$F(n-1)$$

This gives the following recurrence

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \quad \text{for } n > 1$$

$$F(0) = 0, \quad F(1) = c_1$$

Algorithm for Coin-row problem is given below where the input is the array  $C[1..n]$  of positive integers indicating the coin values.

```

1 CoinRow(C[1..n])
2   F[0] = 0
3   F[1] = C[1]
4   for i = 2 to n do
5     F[i] = max(C[i] + F[i - 2], F[i - 1])
6   return F[n]

```

Listing 6.1: Coin-row problem Algorithm

Steps for solving the coin-row problem by dynamic programming for the coin row 5, 1, 2, 10, 6, 2 is given below.

- $F[0] = 0, F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

- $F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

- $F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

- $F[4] = \max\{10 + 5, 5\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

- $F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

- $F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	<b>17</b>

To find the coins with the maximum total value found, computations can be back-traced to see which of the two possibilities,  $c_n + F(n - 2)$  or  $F(n - 1)$  produced the maxima.

### 6.3 The Knapsack Problem and Memory Functions

Given  $n$  items of known weights  $w_1, \dots, w_n$  and values  $v_1, \dots, v_n$  and a knapsack of capacity  $W$ , find the most valuable subset of the items that fit into the knapsack.

Let  $F(i, j)$  be the value of the most valuable subset of the first  $i$  items that fit into the knapsack of capacity  $j$ . All the subsets of the first  $i$  items that fit the knapsack of capacity  $j$  can be divided into two categories:

- Those that do not include the  $i^{\text{th}}$  item. In this case, the value of an optimal subset is  $F(i-1, j)$ .
- Those that include the  $i^{\text{th}}$  item. In this case, an optimal subset is made up of this item and an optimal subset of the first  $i-1$  items that fits into the knapsack of capacity  $j - w_i$ . The value of such an optimal subset is  $v_i + F(i-1, j - w_i)$ .

Hence the recurrence can be written as

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j - w_i)\} & \text{if } j - w_i \geq 0 \\ F(i-1, j) & \text{if } j - w_i < 0 \end{cases}$$

$$F(0, j) = 0 \text{ for } j \geq 0$$

$$F(i, 0) = 0 \text{ for } i \geq 0.$$

Table for solving the knapsack problem by dynamic programming is given below.

	0	$j - w_i$	$j$	$W$
0	0	0	0	0
$i-1$	0	$F(i-1, j - w_i)$	$F(i-1, j)$	
$i$	0		$F(i, j)$	
$n$	0			goal

For example, consider the following data assuming the capacity  $W = 5$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

$i$	capacity $j$					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	<b>37</b>

Table 6.1: Application of dynamic programming to knapsack sum problem

The time efficiency and space efficiency of this algorithm are both in  $\Theta(nW)$ .

### 6.3.1 Memory Functions

The direct top-down approach to finding a solution to a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient. The classic dynamic programming approach, on the other hand, works bottom up: it fills a table with solutions to all smaller subproblems, but each of them is solved only once. The drawback of this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given. A method that solves only subproblems that are necessary and does so only once is based on using **memory functions**. An algorithm implementing memory function for Knapsack sum problem is given below.

```

1 MFKnapsack(i, j)
2   if F[i, j] < 0
3     if j < Weights[i]
4       value = MFKnapsack(i - 1, j)
5     else
6       value = max(MFKnapsack(i - 1, j),
7                   Values[i] + MFKnapsack(i - 1, j - Weights[i]))
8   F[i, j] = value
9   return F[i, j]

```

Listing 6.2: Knapsack sum using Memory Function

Application of this algorithm on the example above shown below where Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed.

		capacity $j$					
$i$	0	1	2	3	4	5	
0	0	0	0	0	0	0	
1	0	0	12	12	12	12	
2	0	-	12	22	-	22	
3	0	-	-	22	-	32	
4	0	-	-	-	-	<b>37</b>	

Table 6.2: Application of dynamic programming with memory function to knapsack sum problem

## 6.4 Warshall's Algorithm

The transitive closure of a directed graph with  $n$  vertices can be defined as the  $n \times n$  boolean matrix  $T = \{t_{ij}\}$ , in which the element in the  $i^{th}$  row and the  $j^{th}$  column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the  $i^{th}$  vertex to the  $j^{th}$  vertex; otherwise,  $t_{ij}$  is 0.

Warshall's algorithm is used to find the transitive closure of a directed graph. Warshall's algorithm constructs the transitive closure through a series of  $n \times n$  boolean matrices:

$$R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}$$

The element  $r_{ij}^{(k)}$  in the  $i^{th}$  row and  $j^{th}$  column of matrix  $R^{(k)}$  ( $i, j = 1, 2, \dots, n, k = 0, 1, \dots, n$ ) is equal to 1 if and only if there exists a directed path of a positive length from the  $i^{th}$  vertex to the  $j^{th}$  vertex with each intermediate vertex, if any, numbered not higher than  $k$ .

The formula for generating the elements of matrix  $R^{(k)}$  from the elements of matrix  $R^{(k-1)}$ :

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } \left( r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)} \right)$$

```

1 Warshall(A[1..n, 1..n])
2   R(0) = A
3   for k = 1 to n do
4     for i = 1 to n do
5       for j = 1 to n do
6          $R^{(k)}[i, j] = R^{(k-1)}[i, j] \text{ or } (R^{(k-1)}[i, k] \text{ and } R^{(k-1)}[k, j])$ 
7   return  $R^{(n)}$ 

```

Listing 6.3: Warshall's Algorithm

In this algorithm,  $A$  is the adjacency matrix.

An example for working of Warshall's algorithm is given in Figure 6.1.

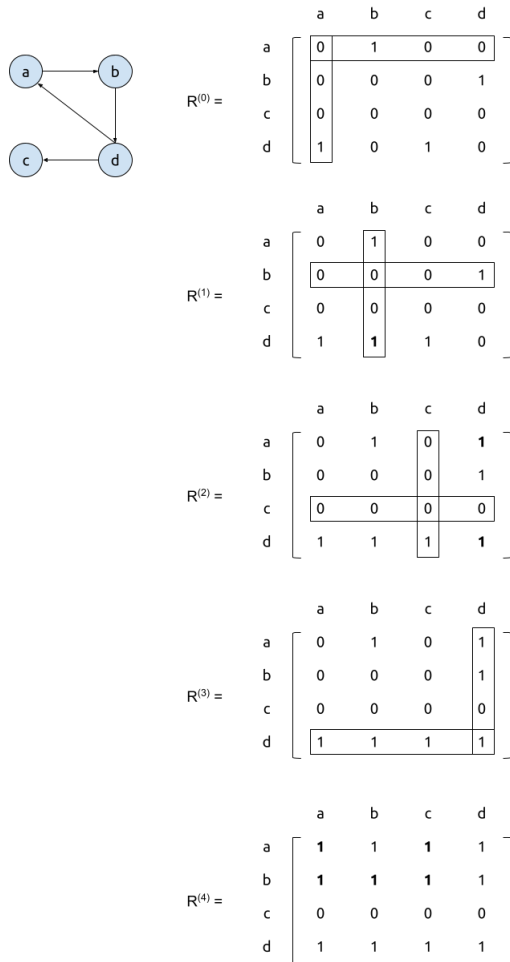


Figure 6.1: Operation of Warshall's algorithm

The time efficiency of Warshall's algorithm is  $\Theta(n^3)$ .

## 6.5 Floyd's Algorithm for the All-Pairs Shortest-Paths Problem

Given a weighted connected graph (undirected or directed), the all-pairs shortest-paths problem asks to find the distance - i.e., the lengths of the shortest paths from each vertex to all other vertices.

Floyd's algorithm constructs the transitive closure through a series of  $n \times n$  boolean matrices:

$$D^{(0)}, \dots, D^{(k-1)}, D^{(k)}, \dots, D^{(n)}$$

The element  $d_{ij}^{(k)}$  in the  $i^{th}$  row and  $j^{th}$  column of matrix  $D^{(k)}$  ( $i, j = 1, 2, \dots, n, k = 0, 1, \dots, n$ ) is equal to the length of the shortest path among all paths from  $i^{th}$  vertex to  $j^{th}$  vertex with each intermediate vertex, if any, numbered not higher than  $k$ .

The formula for generating the elements of matrix  $D^{(k)}$  from the elements of matrix  $D^{(k-1)}$ :

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \text{ for } k \geq 1$$

```

1 Floyd(W[1..n, 1..n])
2   D = W
3   for k = 1 to n do
4     for i = 1 to n do
5       for j = 1 to n do
6         D[i,j] = min{D[i,j], D[i,k] + D[k,j]}
7   return D

```

Listing 6.4: Floyd's Algorithm

An example for working of Floyd's algorithm is given in Figure 6.2.

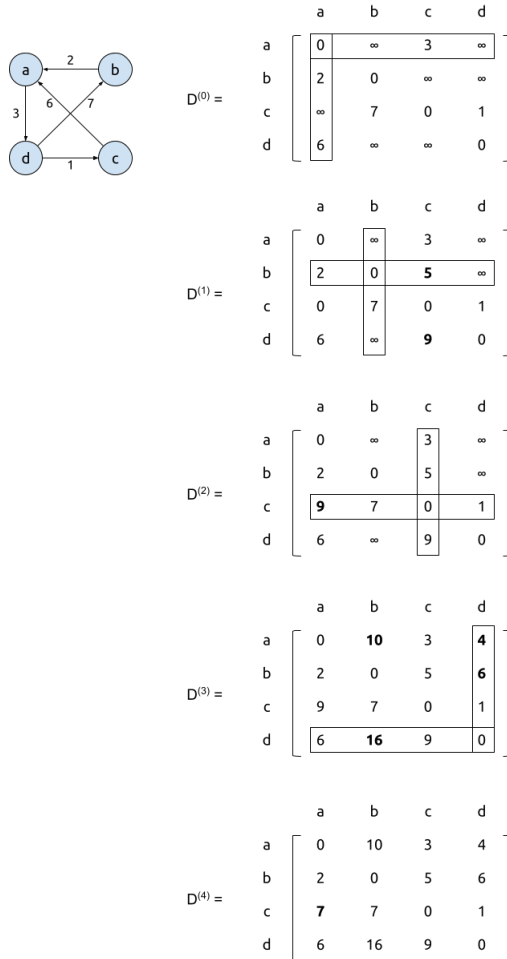


Figure 6.2: Operation of Floyd's algorithm

The time efficiency of Floyd's algorithm is  $\Theta(n^3)$ .



## 7. Greedy Technique

The greedy technique is suitable for optimization problems exclusively. It involves creating a solution gradually by taking steps, with each step expanding a partially built solution. This continues until a complete solution is achieved. The technique gets its name from this approach: at each step, it suggests making a "greedy" choice by picking the best available option, hoping that a sequence of locally best selections will lead to a globally optimal solution for the entire problem. On each step, the choice made must be:

- feasible, i.e., it has to satisfy the problem's constraints
- locally optimal, i.e., it has to be the best local choice among all feasible choices available on that step
- irrevocable, i.e., once made, it cannot be changed on subsequent steps of the algorithm problem.

### 7.1 Prim's Algorithm

A spanning tree of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a minimum spanning tree is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges. The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set  $V$  of the graph's vertices. On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed.

Two labels are attached to each vertex

- the name of the nearest tree vertex
- and the length (the weight) of the corresponding edge.

Vertices that are not adjacent to any of the tree vertices can be given the  $\infty$  label and a null label for the name of the nearest tree vertex.

```
1 Prim(G)
2    $V_T = v_0$ 
3    $E_T = \phi$ 
4   for i = 1 to  $|V| - 1$  do
5     find a minimum-weight edge  $e^* = (v^*, u^*)$  among all the edges  $(v, u)$ 
6     such that  $v$  is in  $V_T$  and  $u$  is in  $V - V_T$ 
7      $V_T = V_T \cup \{u^*\}$ 
8      $E_T = E_T \cup \{e^*\}$ 
9   return  $E_T$ 
```

Listing 7.1: Prim’s Algorithm

Tree Vertices	Remaining Vertices	Illustration
$a(-, -)$	$b(a, 3) \ c(-, \infty) \ d(-, \infty) \ e(a, 6) \ f(a, 5)$	
$b(a, 3)$	$c(b, 1) \ d(-, \infty) \ e(a, 6) \ f(b, 4)$	
$c(b, 1)$	$d(c, 6) \ e(a, 6) \ f(b, 4)$	
$f(b, 4)$	$d(f, 5) \ e(f, 2)$	
$e(f, 2)$	$d(f, 5)$	
$d(f, 5)$		

Table 7.1: Working of Prim’s algorithm

If a graph is represented by its weight matrix and the priority queue is implemented as an

unordered array, the algorithm’s running time will be in  $\Theta(|V|^2)$ .

7.1.1 Kruskal’s Algorithm

The algorithm starts by sorting the edges of the graph in ascending order based on their weights. At the beginning, the spanning tree is empty. The algorithm selects the first edge from the sorted list and includes it in the spanning tree if its inclusion does not create a cycle. If a cycle is formed, the algorithm disregards that edge and proceeds to the next one.

```
1 Kruskal(G)
2   sort E in increasing order of the edge weights,  $w(e_{i1}) \leq \dots \leq w(e_{i|E|})$ 
3    $E_T = \phi$ 
4    $ecounter = 0$ 
5    $k = 0$ 
6   while  $ecounter < |V| - 1$  do
7      $k = k + 1$ 
8     if  $E_T \cup \{e_{ik}\}$  is acyclic
9        $E_T = E_T \cup \{e_{ik}\}$ 
10       $ecounter = ecounter + 1$ 
11  return  $E_T$ 
```

Listing 7.2: Kruskal’s Algorithm

Tree Vertices	Sorted list of	Illustration
	<b>bc(1)</b> ef(2) ab(3) bf(4) cf(4) af(5) df(5) ae(6) cd(6) de(8)	
bc(1)	bc(1) <b>ef(2)</b> ab(3) bf(4) cf(4) af(5) df(5) ae(6) cd(6) de(8)	
ef(2)	bc(1) ef(2) <b>ab(3)</b> bf(4) cf(4) af(5) df(5) ae(6) cd(6) de(8)	
ab(3)	bc(1) ef(2) ab(3) <b>bf(4)</b> cf(4) af(5) df(5) ae(6) cd(6) de(8)	
bf(4)	bc(1) ef(2) ab(3) bf(4) <b>cf(4)</b> af(5) df(5) ae(6) cd(6) de(8)	
df(5)	bc(1) ef(2) ab(3) bf(4) cf(4) <b>af(5)</b> df(5) ae(6) cd(6) de(8)	

Table 7.2: Working of Kruskal’s algorithm

With an efficient union-find algorithm, the running time of Kruskal's algorithm will be dominated by the time needed for sorting the edge weights of a given graph. Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in  $O(|E|\log|E|)$ .

## 7.2 Dijkstra's Algorithm

Dijkstra's algorithm is a single-source shortest-paths problem: for a given vertex called the source in a weighted connected graph, find shortest paths to all its other vertices. This algorithm is applicable to undirected and directed graphs with nonnegative weights only.

Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source. First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. In general, before its  $i^{th}$  iteration commences, the algorithm has already identified the shortest paths to  $i - 1$  other vertices nearest to the source. The set of vertices adjacent to the vertices in  $T_i$  can be referred to as **fringe** vertices. To identify the  $i^{th}$  nearest vertex, the algorithm computes, for every fringe vertex  $u$ , the sum of the distance to the nearest tree vertex  $v$  and the length  $d_v$  of the shortest path from the source to  $v$  (previously determined by the algorithm) and then selects the vertex with the smallest such sum.

In the Dijkstra's algorithm given below, the set  $V_T$  is the set of vertices for which a shortest path has already been found and  $Q$  is the priority queue of the fringe vertices.

The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself. The algorithm is in  $\Theta(|V|^2)$  for graphs represented by their weight matrix and the priority queue implemented as an unordered array.

```

1 Dijkstra(G, s)
2   Initialize(Q)
3   for every vertex v in V
4      $d_v = \infty$ 
5      $p_v = null$ 
6     Insert(Q, v,  $d_v$ )
7    $d_s = 0$ 
8   Decrease(Q, s,  $d_s$ )
9    $V_T = \emptyset$ 
10  for i=0 to  $|V|-1$  do
11     $u^* = \text{DeleteMin}(Q)$ 
12     $V_T = V_T \cup \{u^*\}$ 
13    for every vertex u in  $V - V_T$  that is adjacent to  $u^*$  do
14      if  $d_u^* + w(u^*, u) < d_u$ 
15         $d_u = d_u^* + w(u^*, u)$ 
16         $p_u = u^*$ 
17        Decrease(Q, u,  $d_u$ )

```

Listing 7.3: Dijkstra's Algorithm

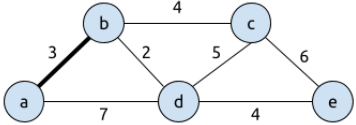
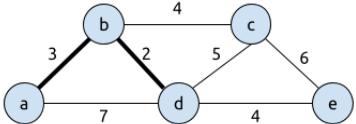
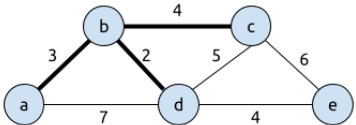
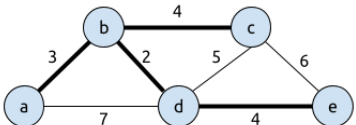
Tree Vertices	Remaining Vertices	Illustration
$a(-,0)$	$b(a,3) \ c(-,\infty) \ d(a,7) \ e(-,\infty)$	
$b(a,3)$	$c(b,3+4) \ d(b,3+2) \ e(-,\infty)$	
$d(b,5)$	$c(b,7) \ e(d,5+4)$	
$c(b,7)$	$e(d,9)$	
$e(d,9)$		

Table 7.3: Working of Dijkstra’s algorithm

The shortest paths are given below.

- from  $a$  to  $b$  :  $a - b$  of length 3
- from  $a$  to  $d$  :  $a - b - d$  of length 5
- from  $a$  to  $c$  :  $a - b - c$  of length 7
- from  $a$  to  $e$  :  $a - b - d - e$  of length 9



# IV

## Unit IV

<b>8</b>	<b>P, NP, and NP-Complete Problems . . . . .</b>	<b>73</b>
8.1	Tractable and Intractable Problems	
8.2	Class P Problems	
8.3	Decidable and Undecidable Problems	
8.4	Class NP Problems	
8.5	NP-Complete Problems	
<b>9</b>	<b>Backtracking . . . . .</b>	<b>77</b>
9.1	Introduction	
9.2	n-Queens Problem	
9.3	Subset-Sum Problem	
9.4	Approximation Algorithms	
	Books	





## 8. P, NP, and NP-Complete Problems

### 8.1 Tractable and Intractable Problems

An algorithm is said to solve a problem in polynomial time if its worst-case time efficiency belongs to  $O(p(n))$  where  $p(n)$  is a polynomial of the problem's input size  $n$  ( $O(1)$ ,  $O(\log n)$ ,  $O(n)$ ,  $O(n \log n)$ ,  $O(n^2)$ ,  $O(n^3)$ ). Problems that can be solved in polynomial time are called **tractable**, and problems that cannot be solved in polynomial time are called **intractable**.

Examples for tractable problems are

- Computing the product
- The greatest common divisor of two integers
- Sorting a list
- Searching for a key in a list
- Finding a minimum spanning tree and shortest paths in a weighted graph

Examples for intractable problems are

- Towers of Hanoi
- List all permutations (all possible orderings) of  $n$  numbers
- Travelling Salesmen Problem

### 8.2 Class P Problems

Class **P** is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called **polynomial**. A decision problem is a problem with answers yes/no.

Many important problems that are not decision problems, can be reduced to a series of decision problems that are easier to study. Consider the vertex coloring problem, which aims to determine the minimum number of colors required to color a graph's vertices in such a way that no adjacent vertices share the same color. This problem can be reformulated as a decision problem: Is it feasible to color the vertices using a maximum of  $m$  colors? Here, the value of  $m$  can take on values such as 1, 2, and so forth. This problem is known as the  $m$ -coloring problem. The solution to the decision problem for  $m$ -coloring, specifically for the smallest value of  $m$  in the series that has a feasible

solution, effectively addresses the optimization aspect of the graph-coloring problem.

### 8.3 Decidable and Undecidable Problems

Decision problems that cannot be solved at all by any algorithm are called undecidable problems and problems that can be solved by an algorithm are called decidable problems.

A famous example of an undecidable problem was given by Alan Turing in 1936 known as **halting problem**.

#### 8.3.1 Halting Problem

Given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

#### 8.3.2 Proof for Undecidability of Halting Problem

This can be proved by proof by contradiction method. Assume that  $A$  is an algorithm that solves the halting problem. That is, for any program  $P$  and input  $I$

$$A(P, I) = \begin{cases} 1 & \text{if program } P \text{ halts on input } I \\ 0 & \text{if program } P \text{ does not halt on input } I \end{cases}$$

If  $P$  is given as input to itself, the output of algorithm  $A$  for pair  $(P, P)$  can be used to construct a program  $Q$  as follows:

$$Q(P) = \begin{cases} \text{halts,} & \text{if } A(P, P) = 0 \text{ i.e. if program } P \text{ does not halt on input } P \\ \text{does not halt,} & \text{if } A(P, P) = 1 \text{ i.e. if program } P \text{ halts on input } P \end{cases}$$

On substituting  $Q$  for  $P$

$$Q(Q) = \begin{cases} \text{halts,} & \text{if } A(Q, Q) = 0 \text{ i.e. if program } Q \text{ does not halt on input } Q \\ \text{does not halt,} & \text{if } A(Q, Q) = 1 \text{ i.e. if program } Q \text{ halts on input } Q \end{cases}$$

This is a contradiction because neither of the two outcomes for program  $Q$  is possible, which completes the proof.

### 8.4 Class NP Problems

A nondeterministic algorithm is a two-stage procedure that takes as its input an instance  $I$  of a decision problem and does the following.

- **Nondeterministic (“guessing”) stage:** An arbitrary string  $S$  is generated that can be thought of as a candidate solution to the given instance  $I$ .
- **Deterministic (“verification”) stage:** A deterministic algorithm takes both  $I$  and  $S$  as its input and outputs yes if  $S$  represents a solution to instance  $I$ .

A nondeterministic algorithm is said to be **nondeterministic polynomial** if the time efficiency of its verification stage is polynomial.

**Class NP** is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called nondeterministic polynomial. This class includes all the problems in **P**, because if a problem is in **P**, then the deterministic polynomial-time algorithm can be used to solve it in the verification-stage of a nondeterministic algorithm that simply ignores string  $S$  generated in its nondeterministic stage.

$$P \subseteq NP$$

Examples for Class NP problems are

- **Hamiltonian circuit problem:** Determine whether a given graph has a Hamiltonian circuit - a path that starts and ends at the same vertex and passes through all the other vertices exactly once.
- **Traveling salesman problem:** Find the shortest tour through  $n$  cities with known positive integer distances between them (find the shortest Hamiltonian circuit in a complete graph with positive integer weights).
- **Knapsack problem:** Find the most valuable subset of  $n$  items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.
- **Graph-coloring problem:** For a given graph, find its chromatic number, which is the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color.

## 8.5 NP-Complete Problems

A decision problem  $D_1$  is said to be polynomially reducible to a decision problem  $D_2$ , if there exists a function  $t$  that transforms instances of  $D_1$  to instances of  $D_2$  such that:

- $t$  maps all *yes* instances of  $D_1$  to *yes* instances of  $D_2$  and all *no* instances of  $D_1$  to *no* instances of  $D_2$
- $t$  is computable by a polynomial time algorithm

A decision problem  $D$  is said to be NP-complete if:

- it belongs to class NP
- every problem in NP is polynomially reducible to  $D$

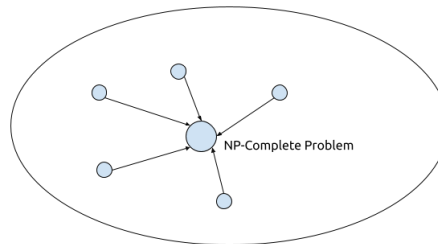


Figure 8.1: Polynomial-time reductions of NP problems to an NP-complete problem

Closely related decision problems are polynomially reducible to each other. For example, Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem. The graph  $G$  of a given instance of the Hamiltonian circuit problem can be mapped to a complete weighted graph  $G'$  representing an instance of the traveling salesman problem by assigning 0 as the weight to each edge in  $G$  and adding an edge of weight 1 between any pair of nonadjacent vertices in  $G$ .

Let  $G$  be a *yes* instance of the Hamiltonian circuit problem. Then  $G$  has a Hamiltonian circuit, and its image in  $G'$  will have length 0, making the image a *yes* instance of the decision traveling salesman problem. Conversely, if there is a Hamiltonian circuit of the length not larger than  $n$  in  $G$ , then its length must be exactly  $n$  and hence the circuit must be made up of edges present in  $G$ , making the inverse image of the *yes* instance of the decision traveling salesman problem be a *yes* instance of the Hamiltonian circuit problem. This completes the proof. In other words if there is a hamiltonian circuit in the graph  $G$ , then  $G'$  will have a path with minimum tour cost 0 and if there is no hamiltonian circuit in the graph  $G$ , then  $G'$  will have a path with minimum tour cost greater than 0 .

Showing that a decision problem is NP-complete can be done in two steps.

1. Show that the problem is in NP

2. Show that every problem in NP is reducible to the problem in question in polynomial time. Because of the transitivity of polynomial reduction, this step can be done by showing that a known NP-complete problem can be transformed to the problem in question in polynomial time.

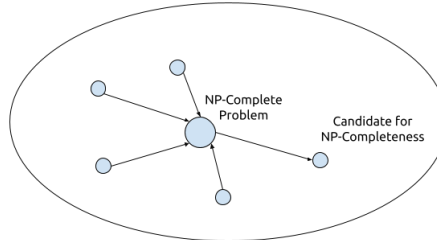


Figure 8.2: Proving NP-completeness by reduction

## 9. Backtracking

### 9.1 Introduction

Backtracking can be considered as a more intelligent variation of exhaustive search technique. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.

- If a partially constructed solution has the potential to be improved without violating the problem's constraints, the algorithm selects the first remaining valid option for the next component.
- When there's no right choice for the next part, the algorithm goes back. It changes the last part it added and tries a different choice.

This process can be made easier by creating a tree of choices. This tree is called a **state-space tree**. The starting point is at the root of the tree before searching for a solution. The first level of the tree shows choices for the first part of the solution. The second level shows choices for the second part, and so on. A node in this tree is called "**promising**" if it could lead to a complete solution, otherwise, it's called "**nonpromising**". The ends of the branches can represent either solutions or dead ends that won't work.

### 9.2 n-Queens Problem

**Definition 9.1.** Place  $n$  queens on an  $n \times n$  chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

For  $n = 1$ , the problem has a trivial solution. For  $n = 2$  and  $n = 3$  there is no solution. Backtracking can be used to solve 4-queens problem.

Since each of the four queens has to be placed in its own row, a column has to be assigned for each queen on the board presented in Figure 9.1. Start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable

position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem. State-space tree of solving the four-queens problem by backtracking is given in Figure 9.2.

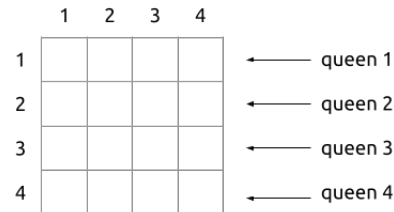


Figure 9.1: Board for the 4-queens problem

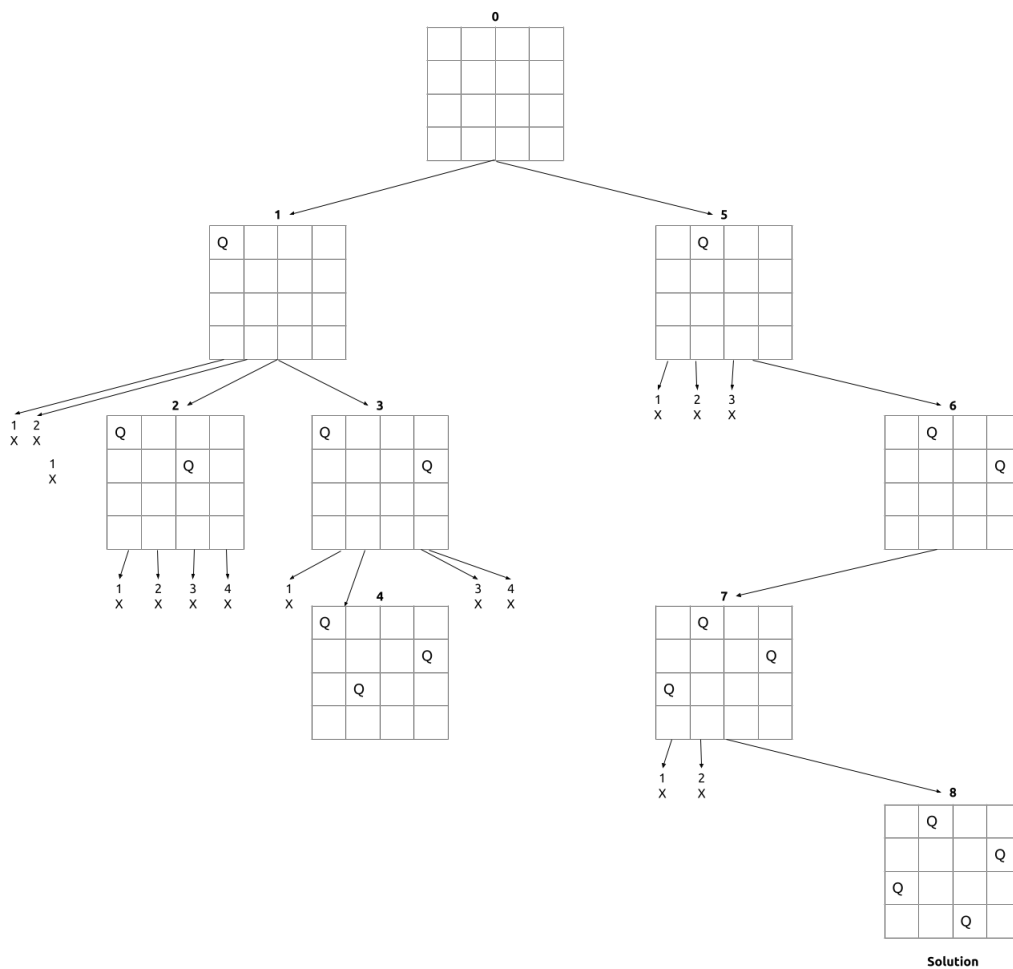


Figure 9.2: State-space tree of solving the four-queens problem by backtracking

### 9.3 Subset-Sum Problem

**Definition 9.2.** Find a subset of a given set  $A = \{a_1, \dots, a_n\}$  of  $n$  positive integers whose sum is equal to a given positive integer  $d$ .

For example, for  $A = \{1, 2, 5, 6, 8\}$  and  $d = 9$ , there are two solutions:  $\{1, 2, 6\}$  and  $\{1, 8\}$ . It is convenient to sort the set's elements in increasing order.

State space diagram for the instance  $A = \{3, 5, 6, 7\}$  and  $d = 15$  is given in Figure 9.3. The root of the tree represents the starting point, with no decisions about the given elements made as yet. Its left and right children represent, respectively, inclusion and exclusion of  $a_1$  in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of  $a_2$  while going to the right corresponds to its exclusion, and so on. Thus, a path from the root to a node on the  $i^{\text{th}}$  level of the tree indicates which of the first  $i$  numbers have been included in the subsets represented by that node.

The sum of the numbers,  $s$  are marked on the node. If  $s = d$ , then that is a solution to the problem. If  $s \neq d$ , the node can be terminated as nonpromising if either of the following two inequalities holds:

$$s + a_{i-1} > d$$

$$s + \sum_{j=i+1}^n a_j < d$$

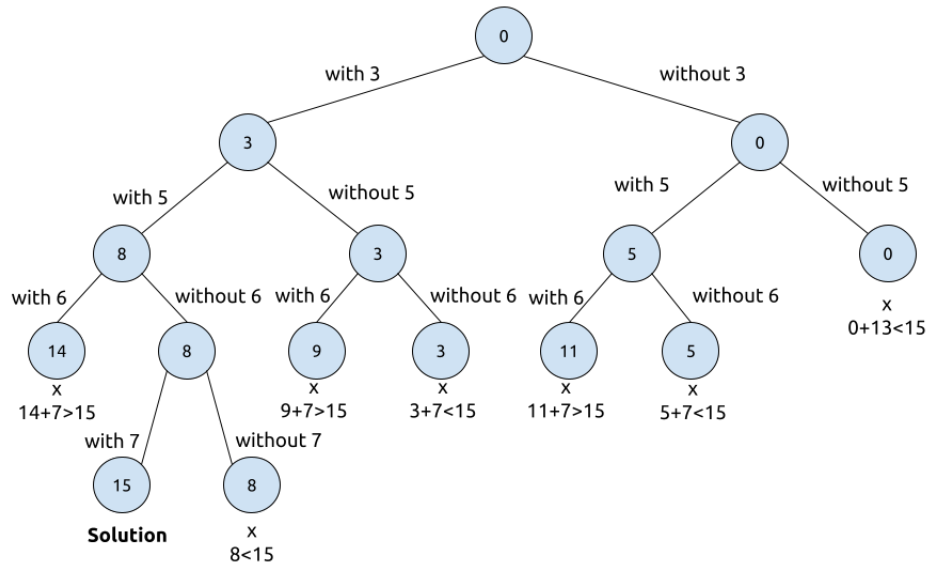


Figure 9.3: State-space tree of solving the subset-sum problem

### 9.4 Approximation Algorithms

Difficult optimization problems can be solved approximately by a fast algorithm. This approach is particularly appealing for applications where a good but not necessarily optimal solution will suffice. Besides, in real-life applications, often it is better to operate with inaccurate data to begin with. Under such circumstances, going for an approximate solution can be a sensible choice.

Most of the approximation algorithms are based on some problem-specific heuristic. A **heuristic** is a common-sense rule drawn from experience rather than from a mathematically proved assertion.

For example, going to the nearest unvisited city in the traveling salesman problem is a good illustration of this notion.

The accuracy of an approximate solution ( $s_a$ ) to a problem involving the minimization of a function (denoted as  $f$ ) can be evaluated by examining the magnitude of the relative error in this approximation.

$$re(s_a) = \frac{f(s_a) - f(s^*)}{f(s^*)}$$

where  $s^*$  is an exact solution to the problem. Since  $re(s_a) = f(s_a)/f(s^*) - 1$ , the accuracy ratio can be written as

$$r(s_a) = \frac{f(s_a)}{f(s^*)}$$

The closer  $r(s_a)$  is to 1, the better the approximate solution is.

For most instances, however, the accuracy ratio cannot be computed,  $f(s^*)$  is not known. Therefore, it is better to obtain a good upper bound on the values of  $r(s_a)$ .

**Definition 9.3.** A polynomial-time approximation algorithm is said to be a **c-approximation algorithm**, where  $c \geq 1$ , if the accuracy ratio of the approximation it produces does not exceed  $c$  for any instance of the problem in question.

$$r(s_a) \leq c$$

The best (i.e., the smallest) value of  $c$  for which inequality above holds for all instances of the problem is called the performance ratio of the algorithm and denoted  $R_A$ .

The performance ratio serves as the principal metric indicating the quality of the approximation algorithm. The closer the value of  $R_A$  is to 1 the better is the solution.



## References

- [1] Anany Levitin. *Introduction to the Design and Analysis of Algorithms*. Pearson, 2012.