CBCS SCHEME

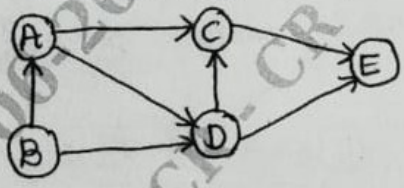
USN

BCS401

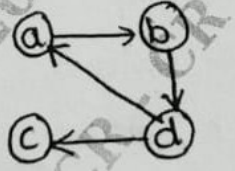
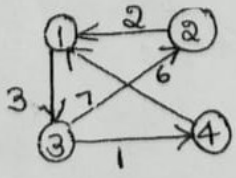
Fourth Semester B.E./B.Tech. Degree Examination, June/July 2026
Analysis & Design of Algorithm

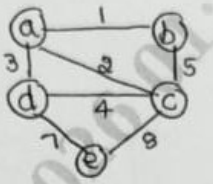
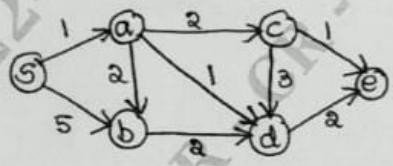
Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

			M	L	C
		Module – 1			
Q.1	a.	Define an Algorithm. With the help of a flowchart, explain the various steps in the algorithm design and analysis process.	8	L1	CO1
	b.	Explain Asymptotic notations Big O, Big Ω and Big θ that are used to compare the order of growth of algorithm with example.	8	L2	CO1
	c.	Discuss Brute Force string matching algorithm with example.	4	L2	CO1
		OR			
Q.2	a.	Write an algorithm to search a key using sequential search. Derive its time efficiency for best case, worst case and average case.	8	L2	CO1
	b.	Explain the general plan for analyzing time efficiency of recursive algorithm. Illustrate mathematical analysis of recursive algorithm for tower of Hanoi.	8	L2	CO1
	c.	Write selection sort algorithm with time complexity.	4	L2	CO1
		Module – 2			
Q.3	a.	What is topological sorting? Apply DFS and source removal method for below graph to solve topological sorting.  Fig. Q3 (a)	8	L3	CO2
	b.	Explain merge sort algorithm with example and discuss its best-case, average-case and worst case efficiency.	8	L2	CO2
	c.	Explain Exhaustive search technique for travelling salesman problem.	4	L2	CO2

BCS4

		OR			
Q.4	**a.**	Write an algorithm to sort 'n' numbers using Quick Sort. Trace the algorithm to sort the following list in ascending order 80, 60, 70, 40, 10, 30, 50, 20.	**8**	**L3**	**CO2**
	b.	Illustrate Strassen's matrix multiplication and derive its time complexity.	**8**	**L2**	**CO2**
	c.	Write an algorithm for insertion sort.	**4**	**L1**	**CO2**
		Module – 3			
Q.5	**a.**	Write an algorithm for Heap Sort. Sort the below given lists by heapsort using the array representation of heaps : SORTING (alphabetical order)	**10**	**L3**	**CO3**
	b.	(i) Construct an AVL tree for the list. 5, 6, 8, 3, 2, 4, 7 (ii) Construct an 2-3 tree for the list. 9, 5, 8, 3, 2, 4, 7	**10**	**L3**	**CO3**
		OR			
Q.6	**a.**	Discuss comparision counting sort with an algorithm. Sort the following numbers using comparision counting sort: 25 45 10 20 50 15	**10**	**L3**	**CO3**
	b.	Explain Horspool's algorithm for string matching. Trace Horspool's algorithm to find the pattern 'DIRECTORS' in text 'IN THE DIRECT TO DIRECTORS SITUATION'	**10**	**L3**	**CO3**
		Module – 4			
Q.7	**a.**	Define Transitive closure. Write Warshall's algorithm to compute transitive closure. Generate transitive closure of the graph given below:  Fig. Q7 (a)	**8**	**L3**	**CO4**
	b.	Apply Floyd's algorithm to find the all pair shortest path for the graph given below :  Fig. Q7 (b)	**8**	**L3**	**CO4**

	c.	Solve for the coin-row problem for the instance {7, 3, 9, 10, 8, 6}	4	L3	CO4
		OR			
Q.8	a.	Apply Prim's algorithm to find the minimal cost spanning tree for the graph given :  Fig.Q8 (a)	8	L3	CO4
	b.	Apply Dijkstra's algorithm to find single source shortest path for the given graph by considering S as the source vertex.  Fig. Q8 (b)	8	L3	CO4
	c.	Construct a Huffman code for the following data:	4	L3	CO4

Character	A	B	C	D	-
Probabiltiy	0.4	0.1	0.2	0.15	0.15

		Module – 5			
Q.9	a.	Explain the following with examples : (i) Class P problems (ii) NP complete problem.	10	L1	CO5
	b.	Give the problem statement of n-queens problem. Explain the solution for 4-queens problem using state space tree.	10	L2	CO6
		OR			
Q.10	a.	Explain how knapsack problem can be solved using branch and bound technique, with example.	10	L3	CO6
	b.	Discuss Decision tree for three element selection sort and binary search in a four element array.	10	L2	CO6

* * * * *

Solution:

Q.1.a. Define an Algorithm. With the help of a flowchart, explain the various steps in the algorithm design and analysis process.

Scheme:

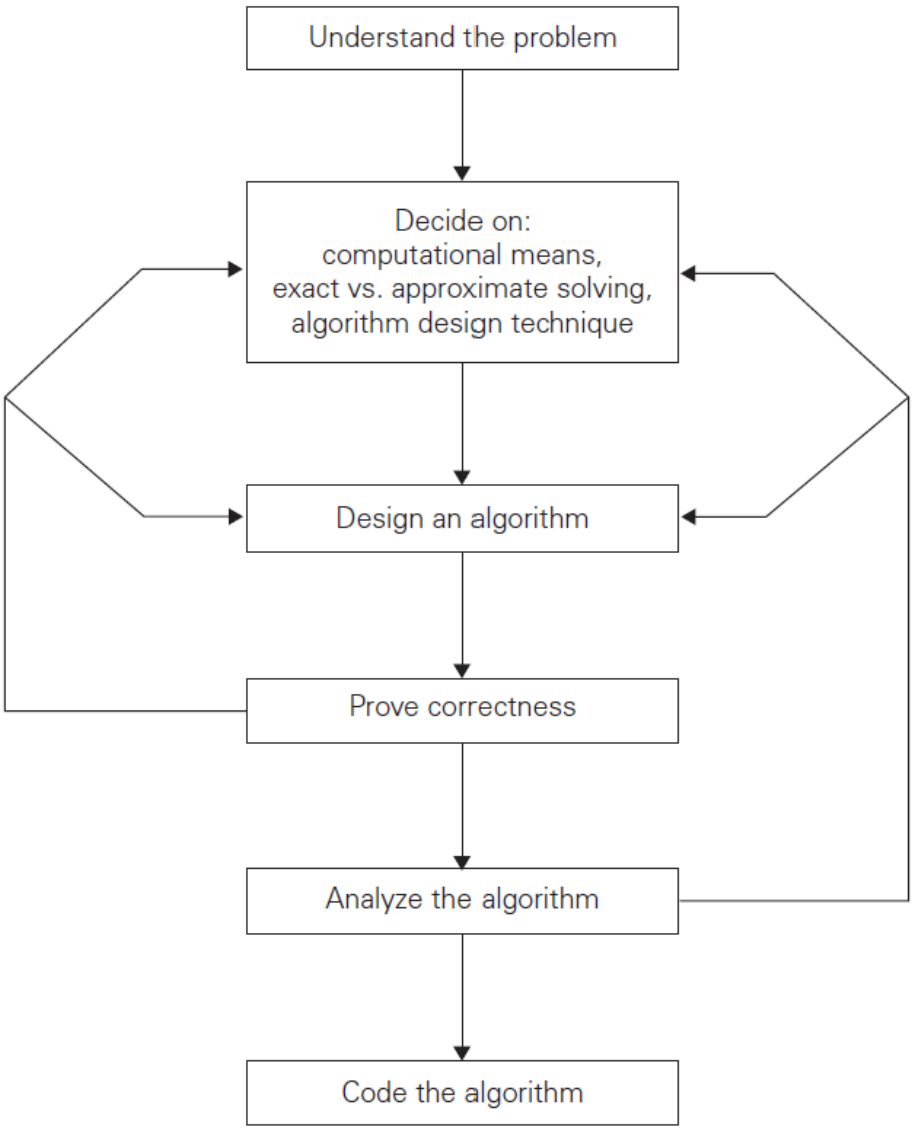
Algorithm Definition – 1 M

Flowchart – 3 M

Explanation – 4 M

Answer:

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



Step 1: Understanding the Problem

From a practical perspective, the first thing we need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if any doubts about the problem; do a few small examples by hand, think about special cases, and ask questions again if needed.

Often we will not find a readily available algorithm and will have to design our own.

An input to an algorithm specifies an instance of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle. If we fail to do this, our algorithm may work correctly for a majority of inputs, but crash on some **"boundary"** values.

A correct algorithm is not one that works most of the time, but one that works correctly for all legitimate inputs.

Step 2: Decide on Computational Means

Once we completely understand a problem, we need to ascertain the capabilities of the computational device the algorithm is intended for.

The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the **von Neumann machine**.

The essence of this architecture is the **Random Access Machine (RAM)**. Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called **Sequential algorithms**.

The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called **Parallel algorithms**.

The **speed** and **memory** available on a particular computer system are other factors to consider.

3) Exact vs Approximate Problem Solving

The next decision is to choose between solving the problem exactly or solving it approximately.

An algorithm which solves the problem exactly is called an **exact algorithm** *(Precise solution)*.

An algorithm which solves the problem approximately is called an **approximation algorithm** *(Approximate solution)*.

Why opt for an approximation algorithm?

1. There are important problems that simply cannot be solved exactly for most of their instances.

Examples: Calculating square roots, solving non-linear equations, evaluating definite integrals.

2. Available exact algorithms are unacceptably slow because of the problem's intrinsic complexity.
3. An approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

4) Algorithm Design Techniques

An algorithm design technique (or **"strategy"** or **"paradigm"**) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

- Design an algorithm to solve a given problem.
- Algorithm design techniques make it possible to classify algorithms according to an underlying design idea.
- They can serve as a natural way to both categorize and study algorithms.

5) Step 3: Design an Algorithm (and Data Structures)

Designing an algorithm for a particular problem may be a challenging task. Some design techniques can be simply inapplicable to the problem in question. Sometimes, several techniques need to be combined.

- Choosing data structures appropriate for the operations performed by the algorithm is important.

Algorithms + Data Structures = Programs.

In the new world of **object-oriented programming**, data structures remain crucially important for both design and analysis of algorithms.

6) Methods of Specifying an Algorithm

Once the algorithm is designed, it should be specified in some fashion. The two options that are most widely used for specifying algorithms are **Pseudocode** and **Flowchart**.

Pseudocode is a mixture of natural language and programming language-like constructs.

Pseudocode is a step-by-step description of an algorithm.

Pseudocode is the intermediate state between an idea and its implementation (code) in a high-level language.

Algorithm → Pseudocode → Program

Flowchart is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

Flowchart is a pictorial representation of the flow of an algorithm.

7) Step 4: Proving an Algorithm's Correctness

Once an algorithm has been specified, we have to prove its correctness, i.e., prove that the algorithm yields the required result for every legitimate input in a finite amount of time.

Step 5: Coding an Algorithm

Algorithms ultimately have to be implemented as computer programs. Programming an algorithm presents both **peril** and an **opportunity**.

The peril (threat) lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently.

The validity of the program is established by **testing and debugging** programs. Test and debug the program thoroughly when an algorithm is implemented.

Code optimization may be required. Such improvements can speed up a program (**running time**).

Q.1.b. Explain Asymptotic notations Big O, Big Ω and Big θ that are used to compare the order of growth of algorithm with example.

Scheme:

Big O, Big Ω and Big θ – Each 2 marks – 6 M

Order of growth – 2 M

Answer:

- **Mathematical notation** for determination of the running time (**time complexity**) of an algorithm.
- These are notations for determination of the order of magnitude of algorithms during **priori analysis** *(before execution of the algorithm)*. These notations are called **asymptotic notations**. These notations help to make approximate (but meaningful) assumptions about the **time** and **space complexity** of algorithms.
- To compare algorithms, we need to check the efficiency of each algorithm. The efficiency can be measured by computing the **time complexity** of each algorithm. **Asymptotic notation** is a shorthand way to represent the time complexity. Using asymptotic notations, we can give time complexity as **"fastest possible"**, **"slowest possible"**, or **"average time"**.
- Commonly used mathematical asymptotic notations are:
    1. **Big Oh notation – O**
    2. **Big Omega notation – Ω**
    3. **Theta notation – θ**
- To compare and rank orders of growth, we use three asymptotic notations: **O (Big Oh), Ω (Big Omega), and θ (Big Theta).**

Let **t(n)** and **g(n)** be any nonnegative functions defined on the set of natural numbers. **t(n)** will be an algorithm's running time (indicated by its basic operation count **c(n)**), and **g(n)** will be some simple function to compare the count with it.

- Informally, **O(g(n))** 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).

- **Ω(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).
- **Θ(g(n))** is the set of all functions that have the **same order of growth** as **g(n)** (to within a constant multiple) as **n** goes to infinity.

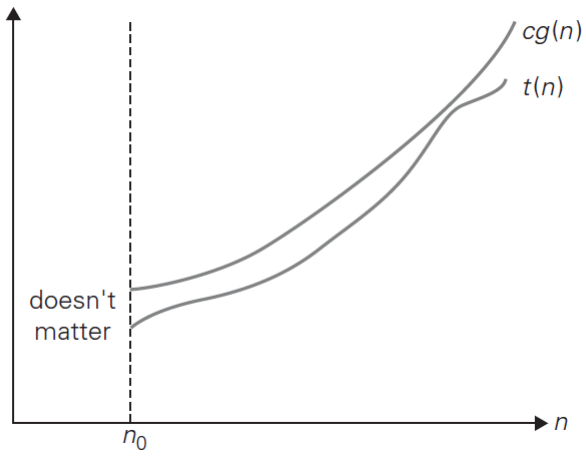
1) Big Oh Notation: O

Definition: A function **t(n)** is said to be in **O(g(n))**, denoted **t(n) ∈ 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 non-negative integer $\mathbf{n_0}$ (threshold problem size) such that

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

- **Big O** refers to the set of all functions whose growth rate is **higher than that of the algorithm's growth rate**.

- The **Big Oh notation** refers to the **"upper bound"** of resources required for solving the problem.
- **'O'** is a method of representing the **upper bound** of an algorithm's running time.
- Using **'O'**, we can give the **longest amount of time** taken by the algorithm to complete.

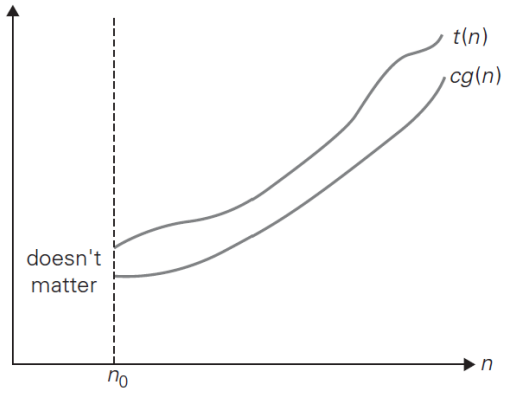


2) Omega Notation: Ω

Definition: A function **t(n)** is said to be in **Ω(g(n))**, denoted **t(n) ∈ Ω(g(n))**, if **t(n)** is **bounded below** by some positive constant multiple of **g(n)** for all large **n**, i.e., if there exist some positive constant **c** and some non-negative integer $\mathbf{n_0}$ *(threshold problem size)* such that

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

- **Omega** refers to the set of all functions whose growth rate is **lower than that of the algorithm's growth rate**.
- **Ω** represents the **lower bound** of resources required for solving the problem.
- **Ω** is a method of representing the **lower bound** of an algorithm's running time.
- Using **Ω**, we can give the **shortest amount of time** taken by the algorithm to complete.



3) Theta Notation: Θ

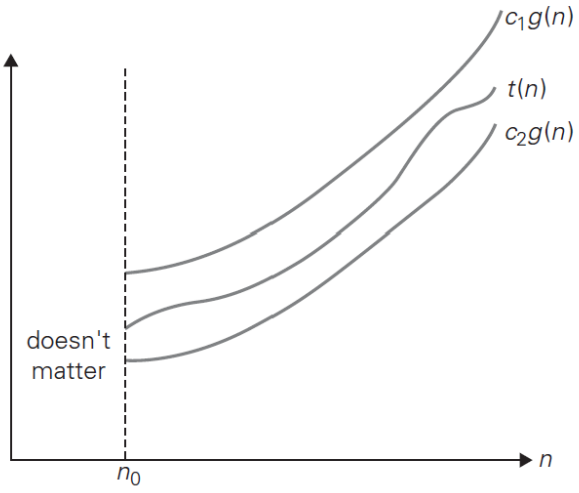
Definition: A function **t(n)** is said to be in **Θ(g(n))**, denoted

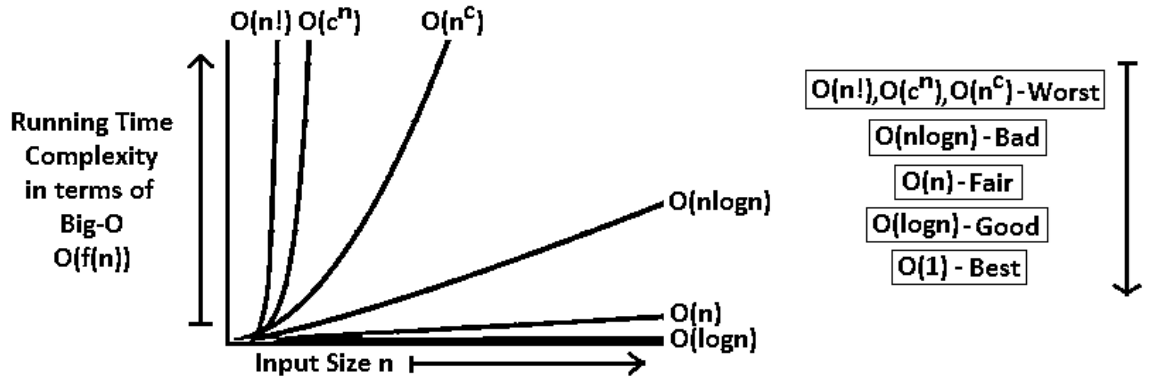
$$t(n) \in \Theta(g(n))$$

if **t(n)** is **bounded both above and below** by some positive constant multiples of **g(n)** for all large **n**, i.e., if there exist some positive constants $\mathbf{c_1}$ and $\mathbf{c_2}$, and some non-negative integer $\mathbf{n_0}$, such that

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

- **Theta (Θ)** refers to the set of all functions whose growth rate is **between the upper bound and lower bound** of the algorithm's growth rate.
- In **Θ**, the running time of the algorithm is **between the upper bound and the lower bound**.





Q.1.c. Discuss Brute Force string matching algorithm with example.

Scheme:

Algorithm – 2 M

Example with explanation – 2M

Answer:

A **string** is a sequence of characters, 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. Find **i**, the index of the leftmost character of the first matching substring in the text, such that

t0 ... ti ... tj ... ti+m−1 ... tn−1 Text T

↑ ↑ ↑

p0 ... pj ... pm−1 Pattern P

The **brute-force algorithm** for string matching aligns the pattern against the first **m** characters of the text and starts matching the corresponding pairs of characters from **left to right** until either all the **m** pairs of characters match (then the algorithm can stop) or a mismatching pair is encountered. It then shifts the pattern **one position to the right** and resumes the character comparisons, starting again with the first character of the pattern and its counterpart in the text.

The **simplest algorithm**, where we simply try to match the first character of the pattern with the first character of the text, and if it succeeds, try to match the second character, and so on. If we hit a failure point, slide the pattern over **one character** and try again. When we find a match, return its **starting location**.

Algorithm: Brute-Force String Match *(T[0…n−1], P[0…m−1])*

Implements brute-force string matching

Input:

- An array **T[0…n−1]** of **n** characters (**text**)
- An array **P[0…m−1]** of **m** characters (**pattern**)

Output:

- The index of the first character in the text that starts a matching substring, or **−1** if the search is unsuccessful.

```
for i ← 0 to n − m do
   j ← 0
   while j < m and P[j] = T[i + j] do
      j ← j + 1
   if j = m return i

return −1
```

The **time complexity** of the algorithm is:

$$O(m \times n)$$

where **n** is the length of the **text** and **m** is the length of the **pattern**.

Example

Text: NOBODY_NOTICED

Pattern: NOT

The pattern is aligned with each position in the text until a match is found:

Index : 0 1 2 3 4 5 6 7 8 9 10 11 12 13

Text : N O B O D Y _ N O T I C E D

Pattern:

N O T

N O T

N O T

N O T

N O T

N O T

N O T

N O T

Index 7 is returned.

Q.2.a. Write an algorithm to search a key using sequential search. Derive its time efficiency for best case, worst case and average case.

Scheme:

Algorithm – 3 M

Explanation – 2 M

Time efficiency – 3 M

Answer:

The algorithm simply compares successive elements of a 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**).

It is the **most basic search technique**. In this type of search, we go through the entire list and try to find a match for a single element (**key element**). If we find a match, then the **address (index)** of the matching element is returned.

Algorithm:

SequentialSearch(A[0...n], K)

Implements sequential search with a search key as a sentinel

Input: An array **A** of **n** elements and a search key **K**.

Output: The index of the first element in **A[0...n−1]** whose value is equal to **K**, or **−1** if no such element is found.

A[n] ← K

i ← 0

while A[i] ≠ K do

  i ← i + 1

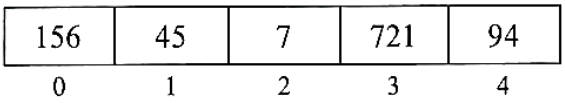
if i < n

  return i

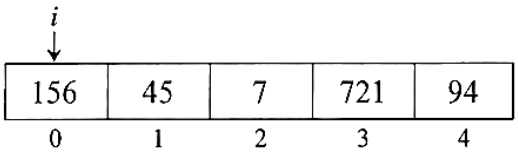
else

return −1

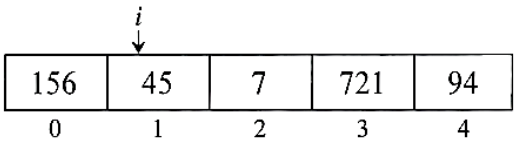
Example 1:

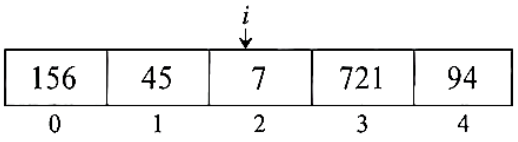


Data/element to be searched is (7).

Step 1:  A[i] ≠ K (i ← i + 1)

156 = 7? No

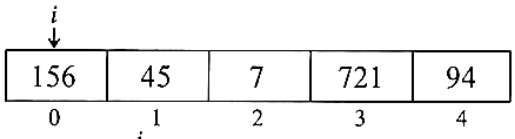
Step 2:  45 = 7? No

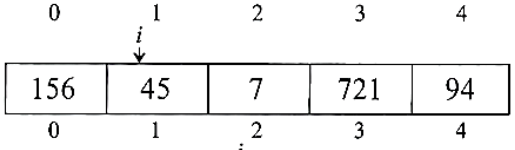
Step 3:  7 = 7? Yes

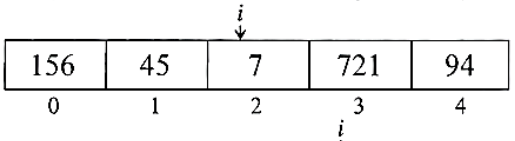
Element found at position 2.

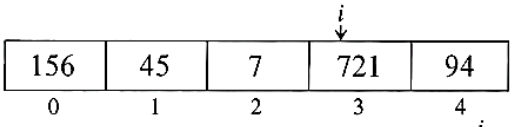
Example 2:

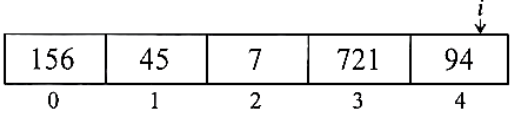
Data/element to be searched is (14).

Step 1:  156 = 14? No

Step 2:  45 = 14? No

Step 3:  7 = 14? No

Step 4:  721 = 14? No

Step 5:  94 = 14? No

Now the end of the list is reached. There are no more elements in the list.

So the item 14 is not found.

Analysis of Sequential Search Algorithm

- **Input size:** The input size is given by the number of elements **n**.

- **Basic operation:** Key comparison

$$A[i] \neq K$$

Basic operation count

Worst Case

The number to be searched is present as the **last element** or **not present at all**. The algorithm makes the **largest number of key comparisons** among all possible inputs of size **n**.

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

$$T(n) = O(n + k) = O(n)$$

Best Case

The number to be searched is present as the **first element** in the given set of numbers.

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

(Only one comparison)

$$T(n) = O(1)$$

Average Case

Each number in the array (and the number **not in the array**) is equally likely to be the number being searched.

$$C_{\text{avg}}(n) = \text{Probability of successful search} + \text{Probability of unsuccessful search}$$

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

Q.2.b. Explain the general plan for analyzing time efficiency of recursive algorithm. Illustrate mathematical analysis of recursive algorithm for Tower of Hanoi.

Scheme:

Plan 5 points – 4 M

Algorithm – 1 M

Mathematical analysis – 3 M

Answer:

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an **input size**.
2. Identify the algorithm's **basic operation**.

3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size. If it can, then the **worst-case, average-case, and best-case efficiencies** must 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.

While solving the recurrence, we will use the **forward and backward substitution method**. The correctness of the formula can then be proved with the help of the **mathematical induction method**.

Towers of Hanoi

We have **n disks** of different sizes and **three pegs**. Initially, all the disks are on the **first peg**, such that the **largest disk is at the bottom** and the **smallest disk is on the top**.

We have to move all the disks to the **third peg** using the **second peg as auxiliary**.

The following rules must be followed:

- We have to move **only one disk at a time**.
- It is **forbidden to place a larger disk on top of a smaller one**.

This problem can be solved by the **recursive technique**.

Algorithm: TOH(n, A, C, B)

// Move n disks from source to destination using auxiliary peg

Input: n disks and 3 pegs

Output: Moving n disks from source to destination such that the largest disk is at the bottom and the smallest disk is on top.

```
if (n = 1) then
    write ("The peg is moved from A to C")
    return
else
    TOH(n - 1, A, B, C)
    move disk from source to destination
    TOH(n - 1, B, C, A)
```

Mathematical Analysis

Step 1:

The input parameter is **n**, i.e., the total number of disks.

Step 2:

The basic operation of the algorithm is **moving disks from one peg to another**.

If **n = 1**, then we simply move the disk from **Peg A** to **Peg C**.

Step 3:

The number of disk movements depends only on the number of disks we have to move, i.e., **n**.

Step 4:

The recurrence relation for the basic operation count is given by:

The recurrence relation is

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

where :

$M(n-1)$: To move $(n-1)$ disks from Peg A to Peg B

1 : To move the largest disk from Peg A to Peg C

$M(n-1)$: To move $(n-1)$ disks from Peg B to Peg C

Therefore,

$$M(n) = 2M(n-1) + 1$$

Initial condition:

$$M(1) = 1$$

Step 5: Solving the recurrence relation using backward substitution method

$$M(n) = 2M(n-1) + 1 \quad —①$$
$$M(n-1) = 2M(n-2) + 1 \quad —②$$
$$M(n-2) = 2M(n-3) + 1 \quad —③$$

Substituting the above equations,

$$\begin{aligned} M(n) &= 2\left(2M(n-2)+1\right)+1 \\ &= 4M(n-2)+3 \\ &= 4\left(2M(n-3)+1\right)+3 \\ &= 8M(n-3)+7 \\ &= \cdots \\ &= 2^i M(n-i) + 2^i - 1 \end{aligned}$$

Put $i = n-1$

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

$$\therefore \quad \boxed{M(n) \in \Theta(2^n)}$$

Q.2.c. Write selection sort algorithm with time complexity.

Scheme:

Algorithm – 3 M

Time complexity – 1 M

Answer:

Selection Sort is a simple comparison-based sorting algorithm. It repeatedly selects the smallest element from the unsorted part of the array and places it at the beginning. After each pass, one element is placed in its correct sorted position.

Algorithm: Selection Sort

Input: An array A[0...n−1] of n elements.

Output: Array sorted in ascending order.

SelectionSort(A[0...n−1])

```
for i ← 0 to n−2 do
   min ← i
   for j ← i+1 to n−1 do
      if A[j] < A[min] then
         min ← j
   if min ≠ i then
      swap A[i] and A[min]

return A
```

Time Complexity Analysis

Input Size

The input size is **n**, the number of elements in the array.

Basic Operation

Comparison:

A[j] < A[min]

Number of Comparisons

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

Best Case

Even if the array is already sorted, Selection Sort still compares every element to find the minimum.

$$T(n) = \Theta(n^2)$$

Worst Case

Even if the array is in reverse order, the number of comparisons remains the same.

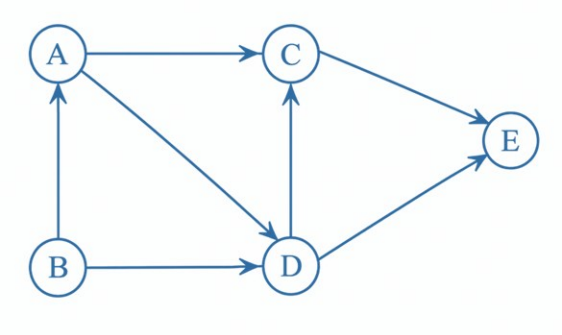
$$T(n) = \Theta(n^2)$$

Average Case

The number of comparisons does not depend on the initial arrangement of the elements.

$$T(n) = \Theta(n^2)$$

Q.3.a. What is topological sorting? Apply DFS and source removal method for below graph to solve topological sorting.



Scheme:

Topological sorting – 1 M

DFS – 3 M

Source removal method – 3 M

Time complexity – 1 M

Answer:

A topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge u→v, vertex u appears before vertex v. Topological sorting exists only for Directed Acyclic Graphs (DAGs).

Example Graph

Edges:
B→A
A→C
A→D
B→D
D→C

C→E
D→E

DFS Algorithm

```
DFS-Topo(G)
for each vertex u
   if u is WHITE
      DFS(u)

DFS(u)
   mark u visited
   for each adjacent vertex v
      if v is WHITE
         DFS(v)
   push u into stack

Reverse(stack)
```

Method 1: DFS-Based Topological Sorting

Step 1: Start DFS from B

Choose **B** as the starting vertex.

Visited:

B

Step 2: Visit A

From **B**, visit **A**.

B → A

Visited:

B, A

Step 3: Visit C

From **A**, move to **C**.

B → A → C

Visited:

B, A, C

Step 4: Visit E

From **C**, move to **E**.

B → A → C → E

Since **E** has no outgoing edges,

Push **E** into the stack.

Stack

Top

E

Step 5: Backtrack to C

Return to **C**.

All adjacent vertices are visited.

Push **C**.

Stack

Top

C

E

Step 6: Backtrack to A

The next adjacent vertex of **A** is **D**.

Visit **D**.

B → A → D

Both adjacent vertices (**C** and **E**) are already visited.

Push **D**.

Stack

Top

D

C

E

Step 7: Backtrack to A

All adjacent vertices are visited.

Push **A**.

Stack

Top

A

D

C

E

Step 8: Backtrack to B

All adjacent vertices are visited.

Push **B**.

Final Stack

Top

B

A

D

C

E

Step 9: Reverse the Stack

Reverse order gives

B → A → D → C → E

Method 2: Source Removal Method

Step 1: Calculate In-Degree

Vertex	In-degree
A	1
B	0
C	2
D	2
E	2

Source vertex = **B**

Delete **B**.

Order:

B

Remaining edges

A → C

A → D

D → C

C → E

D → E

Step 2

New in-degrees

Vertex	In-degree
A	0
C	2
D	1
E	2

Delete **A**.

Order

B → A

Remaining graph

D → C

C → E

D → E

Step 3

New in-degrees

Vertex	In-degree
D	0
C	1
E	2

Delete **D**.

Order

B → A → D

Remaining graph

C → E

Step 4

New in-degrees

Vertex In-degree

C 0

E 1

Delete **C**.

Order

B → A → D → C

Remaining graph

E

Step 5

Delete **E**.

Final order

B → A → D → C → E

Final Answer

DFS Method

B → A → D → C → E

Source Removal Method

B → A → D → C → E

Complexity

Method	Time	Space
DFS	O(V+E)	O(V)
Source Removal	O(V+E)	O(V)

Q.3.b. Explain merge sort algorithm with example and discuss its best-case, average-case and worst case efficiency.

Scheme:

Merge sort Algorithm – 4 M (2M each)

Example – 3 M

Time complexity – 2 M

Answer:

- **Merge Sort** is a perfect example of a successful application of the **divide-and-conquer** technique.
- It sorts a given array **A[0...n−1]** by dividing it into two halves:
  - **A[0...[n/2]−1]**
  - **A[[n/2]...n−1]**

sorting each of them recursively, and then **merging** the two smaller sorted arrays into a **single sorted array**.

Algorithm 1

MergeSort(A[0...n−1])

// Sorts array A[0...n−1] by recursive Merge Sort
// Input : An array A[0...n−1] of orderable elements
// Output: Array A[0...n−1] sorted in non-decreasing order

if (n > 1)

copy A[0...[n/2]−1] to B[0...[n/2]−1]

copy A[[n/2]...n−1] to C[0...[n/2]−1]

MergeSort(B[0...[n/2]−1])

MergeSort(C[0...[n/2]−1])

Merge(B, C, A)

- The **merging of two sorted arrays** can be done as follows:
- Two pointers (**array indices**) are initialized to point to the **first elements** of the arrays being merged.
- The elements pointed to are **compared**, and the **smaller** of them is added to a **new array** being constructed.

- The index of the **smaller element** is incremented to point to its **next immediate successor** in the array it was copied from.
- This operation is repeated until **one of the two given arrays is exhausted**, and then the **remaining elements of the other array** are copied to the end of the new array.

Algorithm 2

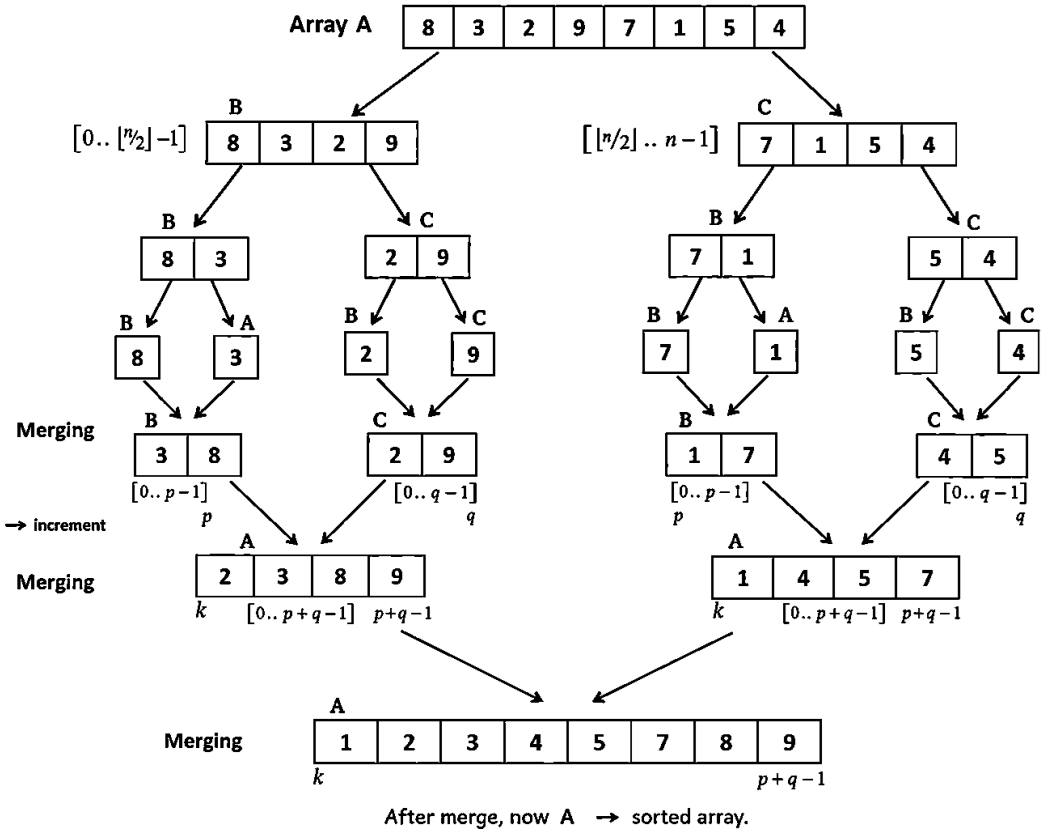
Merge(B[0...p−1], C[0...q−1], A[0...p+q−1])

// Merges two sorted arrays into one sorted array.

// Input : Arrays B[0...p−1] and C[0...q−1], both sorted.

// Output: Sorted array A[0...p+q−1] containing the elements of B and C.

```
i ← 0
j ← 0
k ← 0
while i < p and j < q do
    if B[i] ≤ C[j] then
        A[k] ← B[i]
        i ← i + 1
    else
        A[k] ← C[j]
        j ← j + 1
    k ← k + 1
if i = p then
    copy C[j...q−1] to A[k...p+q−1]
else
    copy B[i...p−1] to A[k...p+q−1]
```



It begins from **Algorithm 1**, where the whole array is called **A** and is divided into **two halves** as **B** and **C**, i.e.,

B[0 ... ⌈n/2⌉ − 1] and **C[0 ... ⌊n/2⌋ − 1]**.

The same process is **repeated until all the elements are divided separately**.

Once all the elements are divided, the function **Merge** is called, i.e., **Merge(B, C, A).**

- **Merge(B[0...p−1], C[0...q−1], A[0...p+q−1])**

i.e., **B** array values are from **0 to p−1**, **C** array values are from **0 to q−1**, and **A** array values are from **0 to p+q−1**.

- We have initialized **i, j, k = 0** for iteration of arrays **B**, **C**, and **A**, respectively.
- **While** i < p **and** j < q **do**

i.e., check the condition for arrays **B** and **C**.

Above, we have:

  - i = 0 and p = 1
  - j = 0 and q = 1

- **If** B[i] ≤ C[j]

i.e.,

B[0] ≤ C[0]

3 ≤ 2

Since the condition is **false**,

A[k] ← C[j]

j ← j + 1

As **2 is smaller**, j gets incremented and **2** is stored in A[k].

k ← k + 1

- **If** i = p

copy C[j...q−1] to A[k...p+q−1]

Else

copy B[i...p−1] to A[k...p+q−1]

Since **i ≠ p**, **B** gets copied to **A**.

- The same process is repeated for all the subarrays, and they are merged into array **A**. Hence, we obtain the **sorted list**.
- **The time complexity of Merge Sort is**

$$O(n \log n)$$

Step 1: Form the Recurrence Relation

Merge Sort divides the input into two equal halves, recursively sorts each half, and merges the two sorted halves in linear time.

Recurrence Relation:

T(n) = 2T(n/2) + Θ(n)

Where:

a = 2 (number of recursive calls)

b = 2 (problem size is reduced by a factor of 2)

f(n) = Θ(n) (time required for merging)

Step 2: Apply the Master Theorem

General form:

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

Compute:

n^(log_b a) = n^(log₂2) = n

Step 3: Compare f(n) and n^(log_b a)

f(n) = n

n^(log_b a) = n

Therefore, f(n) = Θ(n^(log_b a)).

This satisfies Case 2 of the Master Theorem.

Step 4: Apply Case 2

If f(n) = Θ(n^(log_b a) log^k n), where k = 0,

then T(n) = Θ(n^(log_b a) log^(k+1)n).

Hence,

T(n) = Θ(n log n)

Final Result

Case	Time Complexity
Best Case	Θ(n log n)
Average Case	Θ(n log n)
Worst Case	Θ(n log n)

Q.3.c. Explain Exhaustive search technique for travelling salesman problem.

Scheme:

Explanation – 2 M

Example – 2 M

Answer:

- **Exhaustive Search** is simply 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.

Travelling Salesman Problem (TSP)

- The **Travelling Salesman Problem (TSP)** asks 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.

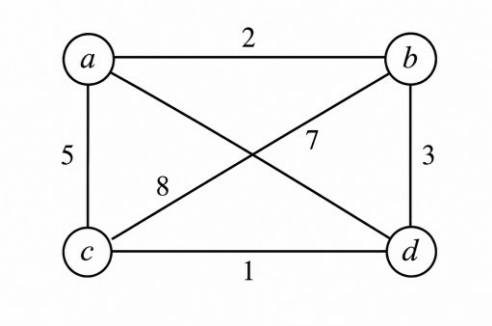
- **Hamiltonian Circuit:** A Hamiltonian circuit is defined as a **cycle that passes through all the vertices of a graph exactly once**. It is named after the **Irish mathematician Sir William Rowan Hamilton**.
- A Hamiltonian circuit can also be defined as a sequence of **n+1** adjacent vertices

$$v_0, v_1, \ldots, v_{n-1}, v_0$$

where the **first vertex of the sequence is the same as the last one**, and all the other **n−1** vertices are distinct.

- Thus, we can obtain all the tours by generating all the **permutations of the n−1 intermediate cities**, compute the tour lengths, and find the **shortest** among them.

Example



The weighted graph contains the following edge weights:

- a↔b=2
- a↔c=5
- b↔d=3
- c↔d=1
- a↔d=7
- b↔c=8
- The figure presents a **small instance of the problem**.
- An inspection of the figure reveals **8 pairs of tours** that differ only by their direction.
- Therefore, we can **cut the number of vertex permutations by half**.

Tour	**Length**
1) a → b → c → d → a	(l = 2 + 8 + 1 + 7 = 18)
2) a → b → d → c → a	(l = 2 + 3 + 1 + 5 = 11) **(Optimal)**
3) a → c → b → d → a	(l = 5 + 8 + 3 + 7 = 23)
4) a → c → d → b → a	(l = 5 + 1 + 3 + 2 = 11) **(Optimal)**

Tour	Length
5) a → d → b → c → a	(l = 7 + 3 + 8 + 5 = 23)
6) a → d → c → b → a	(l = 7 + 1 + 8 + 2 = 18)

Q.4.a. Write an algorithm to sort 'n' numbers using Quick Sort. Trace the algorithm to sort the following list in ascending order: 80, 60, 70, 40, 10, 30, 50, 20.

Scheme:

Algorithm – 4 M

Problem workout – 4 M

Answer:

- **Quick Sort** is another important sorting algorithm that is based on the **divide-and-conquer** approach.
- Quick Sort divides its input elements according to their values.
- It is divided into **partitions**, i.e., a partition is an arrangement of the array's elements such that all the elements to the **left** of some element **A[s]** are **less than or equal to A[s]**, and all the elements to the **right** of **A[s]** are **greater than or equal to A[s]**.

A[0] ... A[s−1] A[s] A[s+1] ... A[n−1]

all are ≤ A[s] all are ≥ A[s]

- After the partition is achieved, **A[s]** will be in its **final position** in the sorted array, and we can continue sorting the two subarrays to the **left** and **right** of **A[s]** independently.
- **Pseudocode of Quick Sort:** call QuickSort(A[0...n−1])

Algorithm 1

QuickSort(A[l...r])

// Sorts a subarray by Quick Sort

// Input : Subarray of array A[0...n−1], defined by its left and right indices l and r.

// Output : Subarray A[l...r] sorted in ascending order.

if l < r then

  s ← Partition(A[l...r]) // s is a split position

  QuickSort(A[l...s−1])

  QuickSort(A[s+1...r])

Partitioning

- We start by selecting a **pivot**, an element with respect to whose value we are going to divide the subarray.
- There are several different strategies for selecting a pivot. Here, we use the **simplest strategy** of selecting the **first element** of the subarray:

P = A[l]

- Make the **first element** the pivot element **P**.
- Set:
  - **i** (low index) to point to **0** (i.e., the first element of the subarray).
  - **j** to the **high index**, which is **n − 1** (i.e., the last element of the subarray).
- **Increment** i until A[i] > p. If this condition is met, **stop incrementing** i.
- **Decrement** j until A[j] ≤ p. If this condition is met, **stop decrementing** j.
- Compare the positions of **i** and **j**. Three cases may arise (**i < j**, **i > j**, **i = j**).

Case 1: If i < j

Swap A[i] and A[j], and continue scanning by **incrementing i** and **decrementing j** from the same positions.

Illustration:

P | all are ≤ p | > p | ..... | ≤ p | all are ≥ p |

↑ i j ↑

Case 2: If i > j

Swap the **pivot element P** with A[j] and return j as the **split position**.

When the scanning indices have crossed over (i > j), the subarray has been partitioned. After exchanging the pivot with A[j], the array is divided as follows:

P | all are ≤ p | ≤ p | ≥ p | all are ≥ p |

↑ j ↑ i

Here, j becomes the **final position of the pivot** (split position), and the array is partitioned into two subarrays that can be recursively sorted.

The extracted text from the image is:

- Finally, if the **scanning indices stop while pointing to the same element**, i.e.,

i = j

the value they are pointing to must be equal to the pivot **p**. Thus, we have the subarray partitioned, with the **split position**

s = i = j

Illustration:

P | all are ≤ p | = p | all are ≥ p |

↑

i = j

- We combine the last case with the case of **crossed-over indices** ((i > j)) by exchanging the pivot with **A[j]** whenever

i > j

Here, **j** is the **split position**.

Algorithm

HoarePartition(A[l...r])

// Partitions a subarray by Hoare's Algorithm, using the first element as a pivot.

// Input : Subarray of array A[0...n−1], defined by its left and right indices l and r (l < r).

// Output : Partition of A[l...r], with the split position returned as this function's value.

p ← A[l]

i ← l

j ← r + 1

repeat

repeat i ← i + 1 until A[i] ≥ p

repeat j ← j − 1 until A[j] ≤ p

Swap(A[i], A[j])

until i ≥ j

Swap(A[i], A[j]) // Undo last swap when i ≥ j

Swap(A[l], A[j])

return j.

Input Array

80 60 70 40 10 30 50 20

Pass 1

Pivot = **80**

All elements are smaller than 80.

After placing pivot

20 60 70 40 10 30 50 |80|

Pivot position = **7**

Left Subarray

20 60 70 40 10 30 50

Pivot = **20**

After partition

10 |20| 70 40 60 30 50 80

Pivot position = **1**

Right Subarray

70 40 60 30 50

Pivot = **70**

After partition

10 20 50 40 60 30 |70| 80

Pivot position = **6**

Left Subarray

50 40 60 30

Pivot = **50**

After partition

10 20 30 40 |50| 60 70 80

Pivot position = **4**

Left Subarray

30 40

Pivot = **30**

Already partitioned.

10 20 |30| 40 50 60 70 80

Right Subarray

40

Only one element.

No partition required.

Right Subarray

60

Only one element.

No partition required.

Final Sorted Array

10 20 30 40 50 60 70 80

Q.4.b. Illustrate Strassen's matrix multiplication and derive its time complexity.

Scheme:

Explanation with formulas – 5 M

Time complexity – 3 M

Answer:

By: Divide and Conquer Method

- **Multiplication of 2 × 2 matrices:** By using the **Divide and Conquer** method (or approach), we can reduce the **number of multiplications**. Such an algorithm was published by **V. Strassen** in **1969**.
- The principal insight of the algorithm lies in the discovery that we can find the product matrix **C** of two **2 × 2 matrices A** and **B** with **just seven multiplications**, as opposed to the **eight multiplications** required by the brute-force algorithm.
- This is accomplished by using the following formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

where,

$$\begin{aligned} c_{00} &= M_1 + M_4 - M_5 + M_7, \\ c_{01} &= M_3 + M_5, \\ c_{10} &= M_2 + M_4, \\ c_{11} &= M_1 + M_3 - M_2 + M_6. \end{aligned}$$

The seven intermediate products are:

$$\begin{aligned} M_1 &= (a_{00} + a_{11}) \times (b_{00} + b_{11}), \\ M_2 &= (a_{10} + a_{11}) \times b_{00}, \\ M_3 &= a_{00} \times (b_{01} - b_{11}), \\ M_4 &= a_{11} \times (b_{10} - b_{00}), \\ M_5 &= (a_{00} + a_{01}) \times b_{11}, \\ M_6 &= (a_{10} - a_{00}) \times (b_{00} + b_{01}), \\ M_7 &= (a_{01} - a_{11}) \times (b_{10} + b_{11}). \end{aligned}$$

- Thus, to multiply **2 × 2 matrices**, **Strassen's algorithm** makes **7 multiplications** and **18 additions/subtractions**, whereas the **brute-force algorithm** requires **8 multiplications** and **4 additions**.

Multiplication of n×n times n×n Matrices

- Let **A** and **B** be two n×n matrices, where **n** is a **power of 2**.
- Divide matrices **A**, **B**, and their product **C** into four n/2 x n/2 submatrices as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

- It is not difficult to verify that one can treat these submatrices exactly as in the **2 × 2 matrix multiplication** case to obtain the correct product.

Example

- C00 can be computed either as

$$C_{00} = A_{00}B_{00} + A_{01}B_{10}$$

or equivalently as

$$C_{00} = M_1 + M_4 - M_5 + M_7,$$

where the **seven products** are computed using the corresponding submatrices.

- If the **7 products** of n/2 x n/2 matrices are computed **recursively** using the same method, we obtain **Strassen's algorithm for matrix multiplication**.

Time Complexity Analysis

If $M(n)$ is the number of multiplications made by Strassen's algorithm in multiplying two $n \times n$ matrices, we get the following recurrence relation for it,

$$M(n) = 7M\left(\frac{n}{2}\right), \qquad \text{for } n > 1,\ M(1) = 1.$$

Since $n = 2^k$,

$$\begin{aligned} M(2^k) &= 7M(2^{k-1}) \\ &= 7\left[7M(2^{k-2})\right] \\ &= 7^2 M(2^{k-2}) \\ &= 7^i M(2^{k-i}) \\ &\vdots \\ &= 7^k M(2^{k-k}) \\ &= 7^k \end{aligned}$$

Since $k = \log_2 n$,

$$\begin{aligned} M(n) &= 7^{\log_2 n} \\ &= n^{\log_2 7} \\ &\approx n^{2.807} \end{aligned}$$

This implies $\boxed{M(n) = O\left(n^{2.807}\right)}$ which is smaller than n^3 required by the brute-force algorithm.

Q.4.c. Write an algorithm tor insertion sort.

Scheme:

Algorithm – 2 M

Example – 2 M

Answer:

Algorithm: InsertionSort(A[0...n−1])

Input: An array A[0...n−1] of n elements.

Output: Array A sorted in ascending order.

InsertionSort(A[0...n−1])

```
for i ← 1 to n − 1 do
   key ← A[i]
   j ← i − 1

   while j ≥ 0 and A[j] > key do
      A[j + 1] ← A[j]
      j ← j − 1

   A[j + 1] ← key

return A
```

Example

Input:
A = [8, 3, 2, 9, 7, 1, 5, 4]

Pass 1: [3, 8, 2, 9, 7, 1, 5, 4]
Pass 2: [2, 3, 8, 9, 7, 1, 5, 4]
Pass 3: [2, 3, 8, 9, 7, 1, 5, 4]
Pass 4: [2, 3, 7, 8, 9, 1, 5, 4]
Pass 5: [1, 2, 3, 7, 8, 9, 5, 4]
Pass 6: [1, 2, 3, 5, 7, 8, 9, 4]
Pass 7: [1, 2, 3, 4, 5, 7, 8, 9]

Sorted Array:
[1, 2, 3, 4, 5, 7, 8, 9]

Q.5.a. Write an algorithm for Heap Sort. Sort the below given lists by Heap Sort using the array representation of heaps: SORTING (alphabetical order)

Scheme:

Algorithm – 3 M

Solution to problem – 5 M (Tree and array representation is mandatory)

Time complexity – 2 M

Answer:

Heap Sort is a **comparison-based sorting algorithm** that uses the **Binary Heap** data structure. It first constructs a **Max Heap** from the given elements and then repeatedly removes the largest element (root), placing it at the end of the array. This process continues until all elements are sorted.

A **Max Heap** satisfies the property:

- Every parent node is **greater than or equal to** its children.

For an array representation of a heap:

- Parent of node i = $\lfloor (i-1)/2 \rfloor$
- Left child = 2i + 1
- Right child = 2i + 2

Algorithm 1: Heap Sort

```
HeapSort(A[0...n−1])
BuildMaxHeap(A)
for i ← n−1 downto 1 do
   swap(A[0], A[i])
   heapSize ← heapSize − 1
   Heapify(A, 0)
```

Algorithm 2: Max Heapify

```
Heapify(A, i)
largest ← i
left ← 2i + 1
right ← 2i + 2
if left < heapSize and A[left] > A[largest]
   largest ← left
if right < heapSize and A[right] > A[largest]
```

largest ← right

if largest ≠ i then

swap(A[i], A[largest])

Heapify(A, largest)

Trace the Algorithm

Input

SORTING

Step 1: Convert the characters into an array

[S, O, R, T, I, N, G]

Step 2: Build the Max Heap

Initial Array

S O R T I N G

After Heap Construction

T S R O I N G

(Array representation of Max Heap)

Step 3: Heap Sort

Pass 1

Swap root with last element.

G S R O I N | T

Heapify

S O R G I N | T

Pass 2

Swap

N O R G I | S T

Heapify

R O N G I | S T

Pass 3

Swap

I O N G | R S T

Heapify

O I N G | R S T

Pass 4

Swap

G I N | O R S T

Heapify

N I G | O R S T

Pass 5

Swap

G I | N O R S T

Heapify

I G | N O R S T

Pass 6

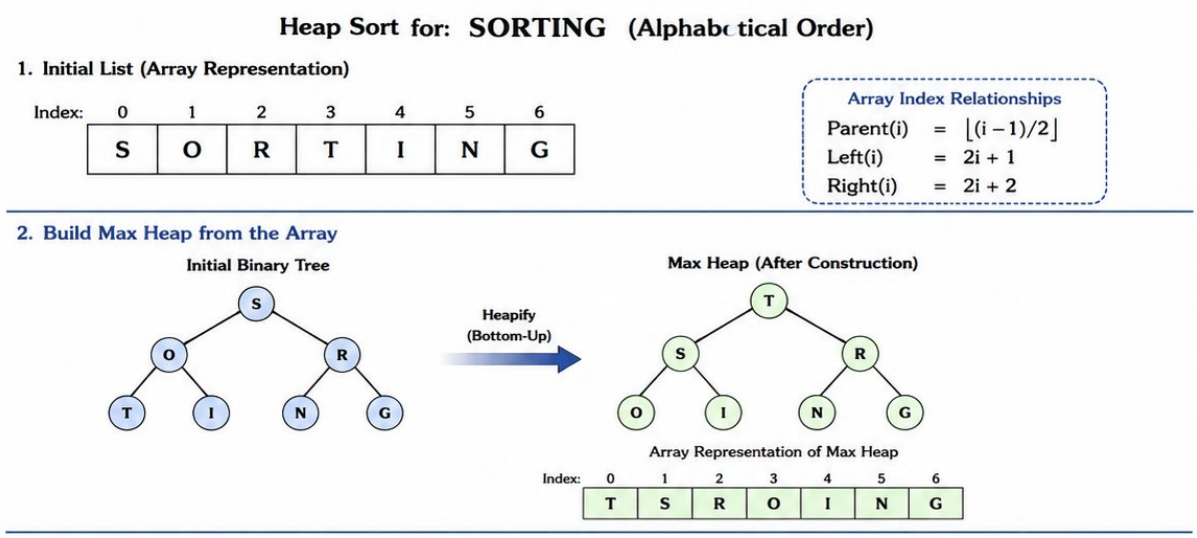
Swap

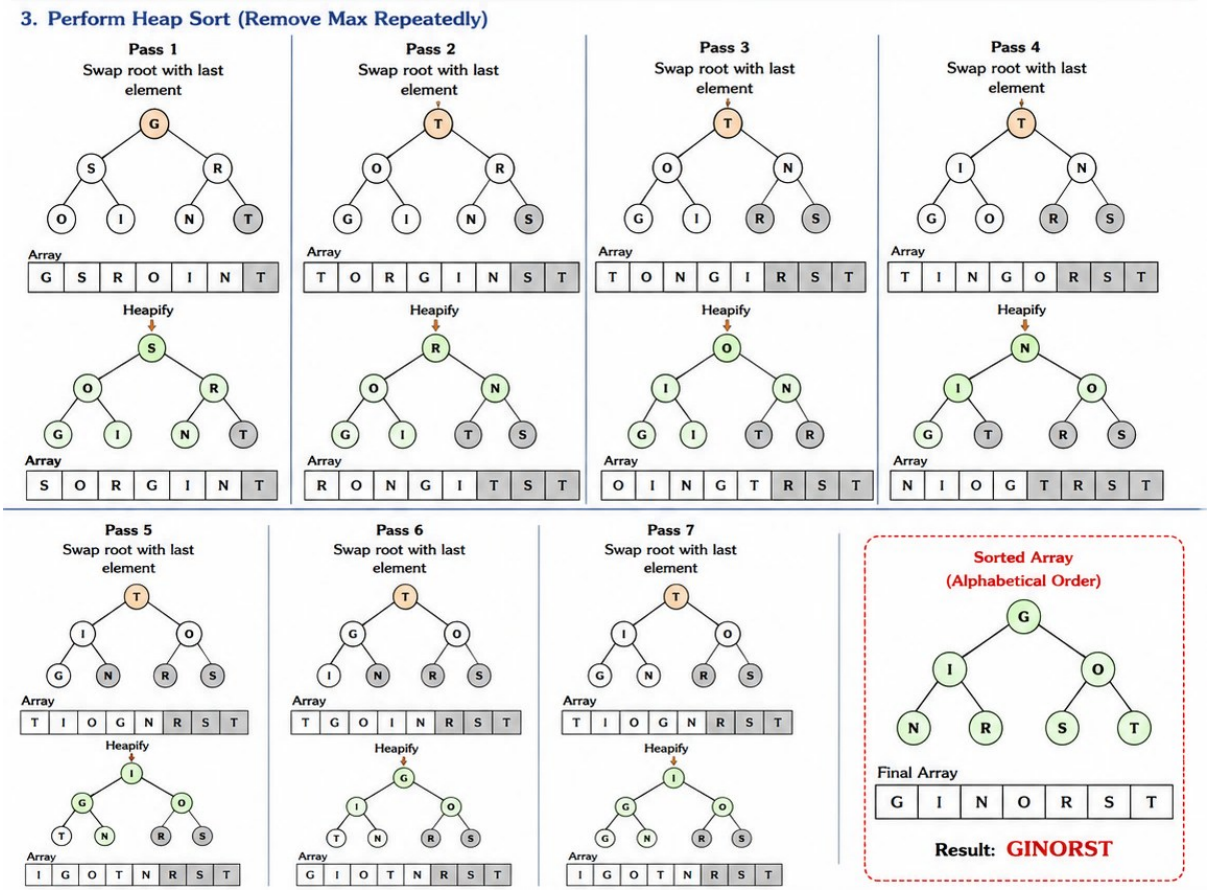
G | I N O R S T

Only one element remains in the heap.

Final Sorted Array

G I N O R S T





Time Complexity Analysis

Building the Heap

O(n)

Heapify Operation

Each Heapify operation takes
O(log n)
There are **(n − 1)** Heapify operations.

Hence,

T(n)=O(n)+O(nlog n)

Complexity Summary

Case	**Time Complexity**
Best Case	**O(n log n)**
Average Case	**O(n log n)**
Worst Case	**O(n log n)**

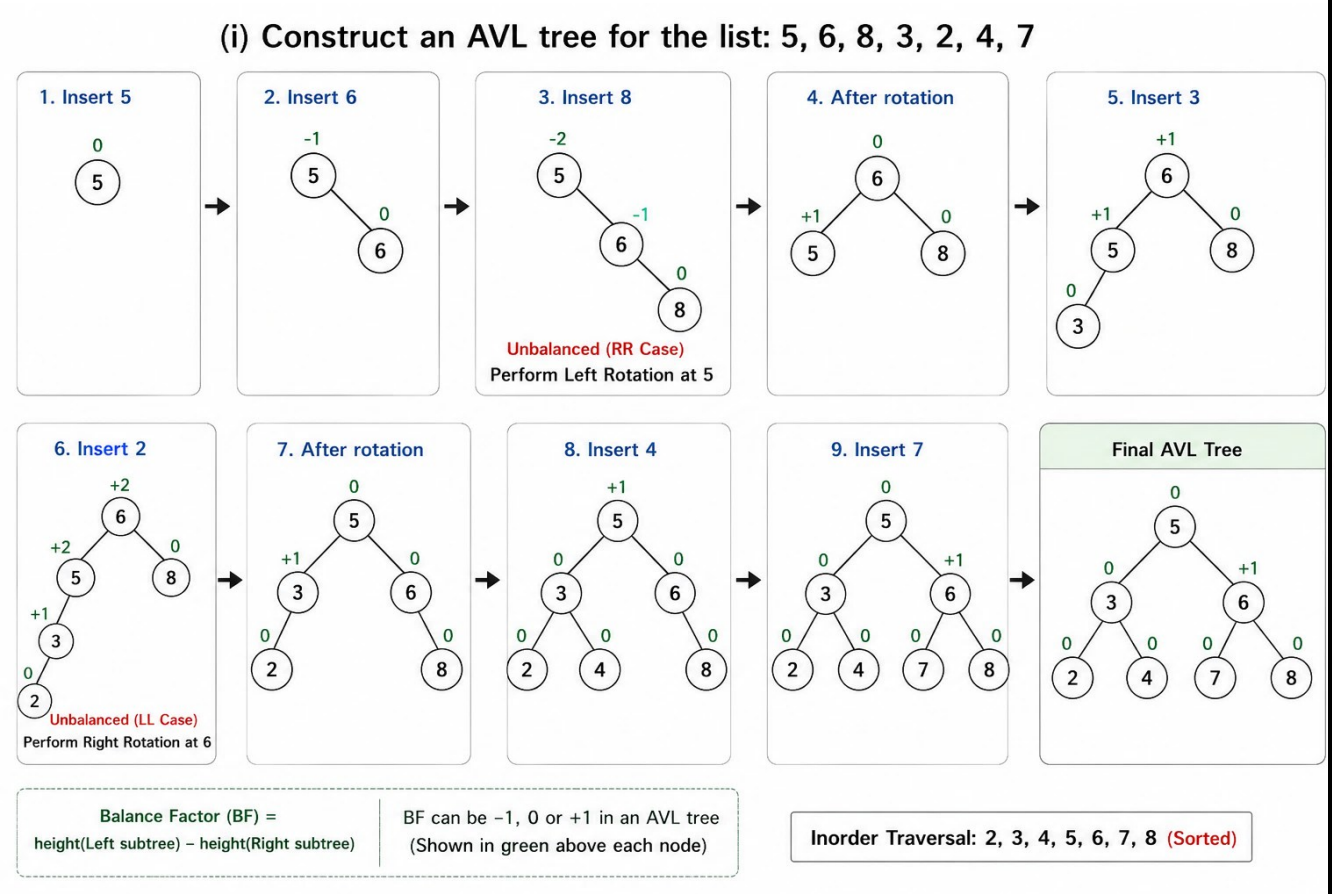
Q.5.b. (i) Construct an AVL tree for the list: 5, 6, 8, 3, 2, 4, 7
(ii) Construct a 2–3 tree for the list: 9, 5, 8, 3, 2, 4, 7

Scheme:

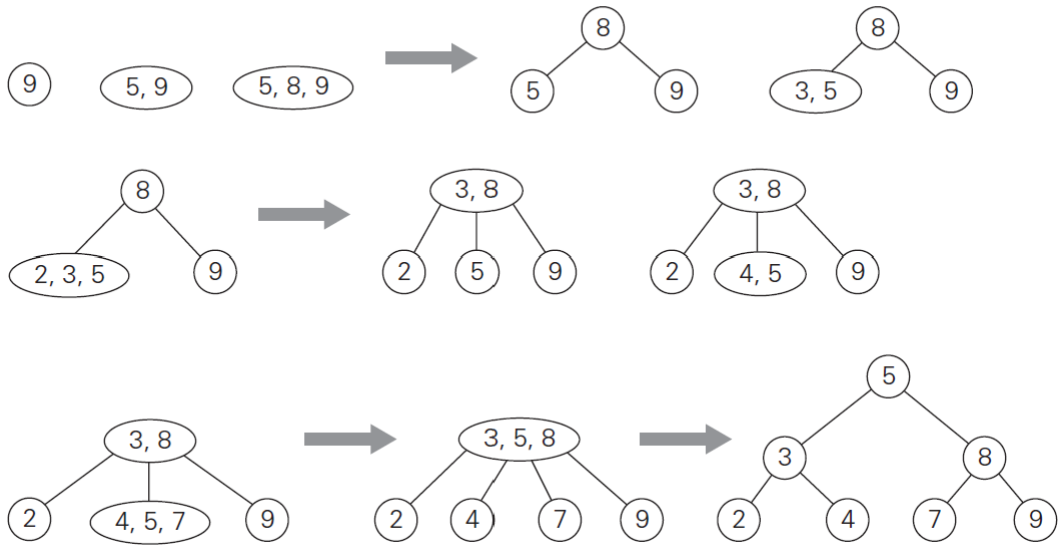
i) AVL Tree – 5 M

ii) 2-3 Tree – 5 M

Answer:



ii) 2-3 Trees:



Q.6.a. Discuss comparison counting sort with an algorithm. Sort the following numbers using comparison counting sort: 25, 45, 10, 20, 50, 15

Scheme:

Algorithm - 4 M

Problem solution – 5 M

Time complexity – 1 M

Answer:

Comparison Counting Sort is a sorting technique that determines the final position of each element by counting the number of elements that are smaller than it. Unlike Bubble Sort or Selection Sort, it does **not interchange elements during comparisons**. After all comparisons are completed, each element is placed directly into its correct position in the output array.

Algorithm: Comparison Counting Sort ((A[0 \ldots n-1]))

Algorithm ComparisonCountingSort(A[0...n−1])

// Sorts an array by comparison counting

// Input : An array A[0...n−1] of orderable elements

// Output : Array S[0...n−1] containing A's elements in non-decreasing order

for i ← 0 to n−1 do

  count[i] ← 0

for i ← 0 to n−2 do

  for j ← i+1 to n−1 do

    if A[i] < A[j] then

```
        count[j] ← count[j] + 1
    else
        count[i] ← count[i] + 1
for i ← 0 to n−1 do
    S[count[i]] ← A[i]
return S
```

EXAMPLE **Sort the following numbers using Comparison Counting Sort:**

25, 45, 10, 20, 50, 15

Step 1: Gilven array A and initialize count array.

Index	0	1	2	3	4	5
A[i]	25	45	10	20	50	15
Count[i]	0	0	0	0	0	0

Idea:
Count[i] will store the number of elements smaller than A[i]. This will be the final position of A[i] in the sorted array.

Step 2: Compare every pair of elements and update count array.

Comparison	Smaller Element	Updated Count	Comparison	Smaller Element	Updated Count
25 vs 45	25	Count(45) = 1	45 vs 15	15	Count(45) = 4
25 vs 10	10	Count(25) = 1	10 vs 20	10	Count(20) = 1
25 vs 20	20	Count(25) = 2	10 vs 50	10	Count(50) = 3
25 vs 50	25	Count(50) = 1	10 vs 15	10	Count(15) = 1
25 vs 15	15	Count(25) = 3	20 vs 50	20	Count(50) = 4
45 vs 10	10	Count(45) = 2	20 vs 15	15	Count(20) = 2
45 vs 20	20	Count(45) = 3	50 vs 15	15	Count(50) = 5
45 vs 50	45	Count(50) = 2			

Step 3: Final count array.

A[i]	25	45	10	20	50	15
Count[i]	3	4	0	2	5	1

Count[i] = number of elements smaller than A[i] → final index of A[i] in sorted array.

Step 4: Place each element A[i] at index Count[i] in output array S.

i	A[i]	Count[i] (final index)		Place in S
0	25	3	→	S[3] = 25
1	45	4	→	S[4] = 45
2	10	0	→	S[0] = 10
3	20	2	→	S[2] = 20
4	50	5	→	S[5] = 50
5	15	1	→	S[1] = 15

Output array S (after placing)

	0	1	2	3	4	5
S[i]	10	15	20	25	45	50

Step 5: Final sorted array.

Sorted Array (S) :	10	15	20	25	45	50

Time Complexity: The number of comparisons = $n(n-1)/2 \Rightarrow O(n^2)$

Space Complexity: $O(n)$ (for count [] and output array S[])

Q.6.b. Explain Horspool's algorithm for string matching. Trace Horspool's algorithm to find the pattern "DIRECTORS" in the text "IN THE DIRECT TO DIRECTORS SITUATION".

Scheme:

Algorithm - 4 M

Shift Table computing – 2 M

Comparison – 4 M

Answer:

Horspool's algorithm is an efficient pattern matching algorithm based on the **Brute Force String Matching** technique. It improves the searching process by using a **shift table**, which determines how far the pattern can be shifted whenever a mismatch occurs. The comparison always starts from the **rightmost character** of the pattern.

The preprocessing phase constructs a shift table based on the characters of the pattern. During searching, if a mismatch occurs, the pattern is shifted according to the value in the shift table instead of moving only one position.

Algorithm: Horspool(P, T)

Input:

- Text T[0...n−1]
- Pattern P[0...m−1]

Output:

- Starting index of the first occurrence of the pattern, otherwise -1.

```
ShiftTable(P)
i ← m − 1
while i < n do
   k ← 0
   while k < m and P[m−1−k] = T[i−k] do
      k ← k + 1
   if k = m then
      return i − m + 1
   i ← i + Shift[T[i]]
return -1
```

```
Algorithm Horspool(S, P)
{
   n ← length(S);
   m ← length(P);
   ShiftTable(P, ST);
   i ← m − 1;            // Start comparison from the last character
   while (i ≤ n − 1)
   {
      k ← 0;
      while (k ≤ m − 1) and (P[m − 1 − k] = S[i − k])
         k ← k + 1;
      if (k = m)
         return (i − m + 1);
      else
         i ← i + ST[S[i]];
   }
   return −1; }
```

Text (T) : IN THE DIRECT TO DIRECTORS SITUATION — Length of Text, n = 34

Pattern (P): DIRECTORS — Length of Pattern, m = 9

1) SHIFT TABLE CONSTRUCTION

For each character in the pattern (except the last character),
shift value = distance from that character to the last character.
Default shift value for all other characters = m = 9.

Index	0	1	2	3	4	6	7	8
Pattern	D	I	R	E	C	T	O	S
Shift Value	8	7	3	5	4	2	1	9*

* Last character 'S' and all other characters not in the pattern have shift value = 9.

Character	D	I	R	E	C	O	S	Others
Shift Value	8	7	3	5	4	2	1	9

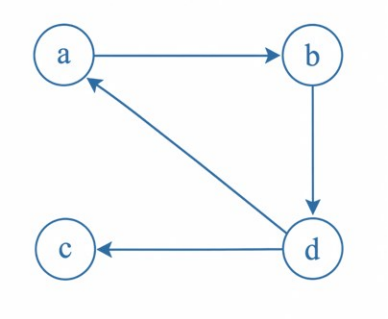
2) TRACE OF HORSPOOL'S ALGORITHM

Step	Alignment (pattern under text)	Comparison	Mismatch Character	Shift	New Position
1 i = 8 (start)	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 I N T H E D I R E C T T O D I R E C T O R S S I T U A T I O N D I R E C T O R S ------ Shift by 5 -----→	Compare from right: S vs E → Mismatch	Text[8] = 'E' Shift(E) = 5	5	i = 8 + 5 = 13
2 i = 13	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 I N T H E D I R E C T T O D I R E C T O R S S I T U A T I O N D I R E C T O R S --- Shift by 2 ----→	Compare from right: S vs T → Mismatch	Text[13] = 'T' Shift(T) = 2	2	i = 13 + 2 = 15

3 i = 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 I N T H E D I R E C T T O D I R E C T O R S S I T U A T I O N D I R E C T O R R S -- Shift by 1 --→	Compare from right: S vs O → Mismatch	Text[15] = 'O' Shift(O) = 1	1	i = 15 + 1 = 16
4 i = 16	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 I N T H E D I R E C T T O D I R E C T O R S S I T U A T I O N D I R E C T O R S S	All characters match from right to left. Pattern found!	–	–	Match found at index i - (m - 1) = 16 - 8 = 8 (0-based) = 9th position (1-based)

PATTERN 'DIRECTORS' FOUND AT INDEX 8 (0-BASED) = 9TH POSITION (1-BASED) IN THE TEXT.

Q.7.a. Define Transitive Closure. Write Warshall's algorithm to compute transitive closure. Generate the transitive closure of the graph given below:



Scheme:

Transitive Closure – 1 M

Warshall's algorithm – 2 M

Computing transitive closure – 5 M

Answer:

The transitive closure of a directed graph is a graph that contains an edge from vertex u to vertex v if and only if there exists a path (direct or indirect) from u to v in the original graph.

Given Graph

Edges: a→b, b→d, d→a, d→c

Algorithm

```
Warshall(A,n)
R ← A
for i=1 to n:
   R[i][i] ← 1
for k=1 to n:
   for i=1 to n:
      for j=1 to n:
```

R[i][j] = R[i][j] OR (R[i][k] AND R[k][j])
return R

Initial Adjacency Matrix A

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

R(0) - After Adding Self Reachability

	a	b	c	d
a	1	1	0	0
b	0	1	0	1
c	0	0	1	0
d	1	0	1	1

R(1) - After Considering Vertex a

	a	b	c	d
a	1	1	0	0
b	0	1	0	1
c	0	0	1	0
d	1	1	1	1

R(2) - After Considering Vertex b

	a	b	c	d
a	1	1	0	1
b	0	1	0	1
c	0	0	1	0
d	1	1	1	1

R(3) - After Considering Vertex c

	a	b	c	d
a	1	1	0	1
b	0	1	0	1

c	0	0	1	0
d	1	1	1	1

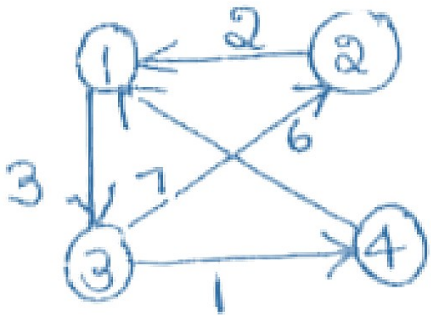
R(4) - After Considering Vertex d (Final Transitive Closure)

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	1	0
d	1	1	1	1

Time Complexity

The algorithm uses three nested loops.
$T(n)=O(n^3)$.

Q.7.b. Apply Floyd's algorithm to find the all pair shortest path for the graph given below:



Scheme:

Algorithm – 2 M

Problem workout – 6 M

Answer:

Algorithm:

Algorithm Floyd(A[0...n−1, 0...n−1])

// Input : Weighted adjacency matrix A

// Output: Matrix D containing all-pairs shortest paths

D ← A

for k ← 0 to n − 1 do

  for i ← 0 to n − 1 do

    for j ← 0 to n − 1 do

D[i][j] ← min(D[i][j], D[i][k] + D[k][j])

return D

Graph edges:
2→1 (2)
1→3 (3)
3→2 (7)
1→4 (6)
3→4 (1)

D(0) Initial Distance Matrix

	1	2	3	4
1	0	∞	3	6
2	2	0	∞	∞
3	∞	7	0	1
4	∞	∞	∞	0

Iteration k=1 (Vertex 1 as Intermediate)

New shortest paths found: 2→3 = 5, 2→4 = 8

D(1)

	1	2	3	4
1	0	∞	3	6
2	2	0	5	8
3	∞	7	0	1
4	∞	∞	∞	0

Iteration k=2 (Vertex 2 as Intermediate)

New shortest paths found: 3→1 = 9

D(2)

	1	2	3	4
1	0	∞	3	6
2	2	0	5	8
3	9	7	0	1
4	∞	∞	∞	0

Iteration k=3 (Vertex 3 as Intermediate)

Edge 3→2 has weight 7.
New shortest paths found: 1→2 = 10, 1→4 = 4, 2→4 = 6

D(3)

	1	2	3	4
1	0	10	3	4
2	2	0	5	6
3	9	7	0	1
4	∞	∞	∞	0

Iteration k=4 (Vertex 4 as Intermediate)

Edge 3→2 has weight 7.
New shortest paths found: No changes

Final Distance Matrix D(4)

	1	2	3	4
1	0	10	3	4
2	2	0	5	6
3	9	7	0	1
4	∞	∞	∞	0

Time Complexity

Three nested loops are used.
$T(n)=O(n^3)$.

Q.7.c. Solve for the coin-row problem for the instance {7, 3, 9, 10, 8, 6}.

Scheme:

DP formula – 1 M

Workout – 3 M

Answer:

Coin-Row Problem: Given a row of coins, select coins such that **no two adjacent coins** are selected and the total value is maximum.

Dynamic Programming Formula

Let $F(i)$ = maximum value that can be collected from the first i coins.

$$F(i) = \max(F(i-1), F(i-2) + C_i)$$

Base Cases:

$F(0) = 0$

$F(1) = C_1$

Given Coins

Coin Position (i)	1	2	3	4	5	6
Coin Value (C_i)	7	3	9	10	8	6

Step-by-Step Computation

i	Coin Value (C_i)	Formula	$F(i)$	Explanation
0	–	Initial	0	No coins, value = 0
1	7	$F(1) = 7$	7	Only first coin
2	3	$F(2) = \max(F(1), F(0) + 3) = \max(7, 0 + 3)$	7	Choose max of (exclude coin 2) or (include coin 2)
3	9	$F(3) = \max(F(2), F(1) + 9) = \max(7, 7 + 9)$	16	Choose max of (exclude coin 3) or (include coin 3)
4	10	$F(4) = \max(F(3), F(2) + 10) = \max(16, 7 + 10)$	17	Choose max of (exclude coin 4) or (include coin 4)
5	8	$F(5) = \max(F(4), F(3) + 8) = \max(17, 16 + 8)$	24	Choose max of (exclude coin 5) or (include coin 5)
6	6	$F(6) = \max(F(5), F(4) + 6) = \max(24, 17 + 6)$	24	Choose max of (exclude coin 6) or (include coin 6)

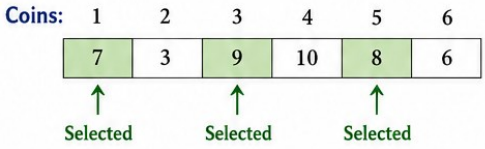
Dynamic Programming Table

i	0	1	2	3	4	5	6
Coin Value (C_i)	–	7	3	9	10	8	6
$F(i)$	0	7	7	16	17	24	24

Finding the Selected Coins (Backtracking)

Start from the last coin.

- $F(6) = F(5)$, so coin 6 is not selected.
- At $F(5)$, $24 = 16 + 8$, so coin 5 (8) is selected. Move to coin 3.
- At $F(3)$, $16 = 7 + 9$, so coin 3 (9) is selected. Move to coin 1.
- Coin 1 (7) is selected.



Final Answer:

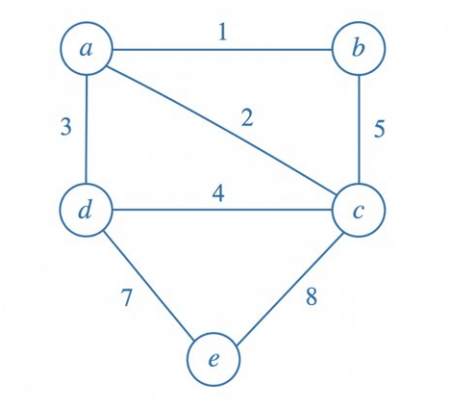
Selected Coins: 7, 9, 8 | **Maximum Amount Collected:** 24

Time Complexity: $O(n)$

Space Complexity: $O(n)$

(Can be optimized to $O(1)$)

Q.8.a. Apply Prim's algorithm to find the minimal cost spanning tree for the graph given below:



Scheme:

Algorithm – 2 M

Construction of MST – 6 M

Answer:

Prim's Algorithm

Start with vertex **a**.

At each step, choose the **minimum weight edge** that connects a visited vertex to an unvisited vertex.

Step 1

Start at vertex **a**.

Visited vertices:

{a}

Available edges:

Edge	Weight
a–b	**1**
a–c	2
a–d	3

Choose the minimum edge.

✓ Select a–b (1)

Current MST:

a -----1----- b

Cost = **1**

Step 2

Visited vertices:

{a,b}

Available edges:

Edge	Weight
a–c	**2**
a–d	3
b–c	5

Choose the minimum edge.

✓ Select a–c (2)

Current MST:

b

|

1

|

a -----2----- c

Cost

1+2=3

Step 3

Visited vertices:

{a,b,c}

Available edges

Edge	Weight
a–d	**3**
c–d	4
c–e	8

Choose the minimum edge.

✓ Select a–d (3)

Current MST

b

|

1

|

a -----2----- c

|

3

|

d

Cost
1+2+3=6

Step 4

Visited vertices

{a,b,c,d}

Available edges

Edge Weight

d–e **7**

c–e 8

Choose the minimum edge.

✓ Select d–e (7)

Current MST

b

|

1

|

a -----2----- c

|

3

|

d

|

7

|

e

Cost

1+2+3+7=13

All vertices are included.

Minimum Spanning Tree

Selected edges

Edge	**Weight**
a–b	1
a–c	2
a–d	3
d–e	7

Final MST

b

|

(1)

|

a -------(2)------- c

|

(3)

|

d

|

(7)

|

e

Total Minimum Cost

1+2+3+7=13

Final Answer

Minimum Spanning Tree (MST):

- a – b (1)
- a – c (2)
- a – d (3)
- d – e (7)

Minimum Cost = 13

Time Complexity

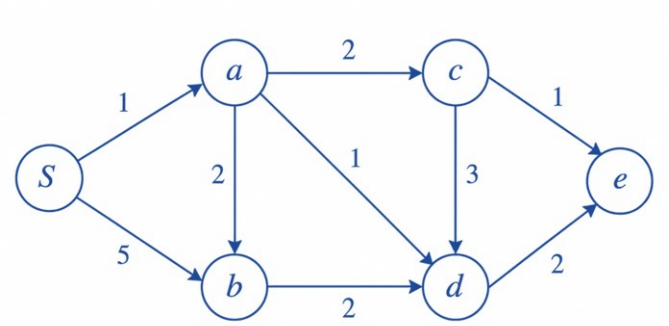
Using an adjacency matrix implementation:

O(V^2)

Using a priority queue (binary heap) and adjacency list:

O(E\log V)

Q.8.b. Apply Dijkstra's algorithm to find single source shortest path for the given graph by considering S as the source vertex.



Scheme:

Algorithm – 2 M

Problem workout – 6 M

Answer:

Algorithm:
Algorithm Dijkstra(G, s)

// Input : Weighted graph G(V, E) and source vertex s

// Output: Shortest distance from s to every vertex

for each vertex v ∈ V do

dist[v] ← ∞

```
    parent[v] ← NIL
    visited[v] ← FALSE
dist[s] ← 0
repeat |V| times do
    u ← unvisited vertex having minimum dist[u]
    visited[u] ← TRUE
    for each vertex v adjacent to u do
        if visited[v] = FALSE and
           dist[v] > dist[u] + weight(u, v) then
            dist[v] ← dist[u] + weight(u, v)
            parent[v] ← u
return dist[], parent[]
```

Source Vertex: S

Tree Vertex	Remaining Vertices (Predecessor, Distance)	Illustration
S(-,0)	a(S,1) _b(S,5)_ c(-,∞) d(-,∞) e(-,∞)	Tree: S └──1──▶ a
a(S,1)	_d(a,2)_ b(a,3) c(a,3) e(-,∞)	Tree: S └──1──▶ a ├──2──▶ b ├──2──▶ c └──1──▶ d
d(a,2)	_b(a,3)_ c(a,3) e(d,4)	Tree: S └──1──▶ a ├──2──▶ b ├──2──▶ c └──1──▶ d └──2──▶ e
b(a,3)	_c(a,3)_ e(d,4)	Tree unchanged

c(a,3)	_e(d,4)_	Tree unchanged (c→e gives distance 4, equal to current)
e(d,4)	All vertices included.	Final shortest-path tree

Final Shortest Paths

Path	Distance
S → a	1
S → a → d	2
S → a → b	3
S → a → c	3
S → a → d → e	4

Final Distance Table

Vertex	S	a	b	c	d	e
Distance	0	1	3	3	2	4

Time Complexity

Using adjacency matrix: $O(V^2)$
Using binary heap and adjacency list: O((V+E) log V).

Q.8.c. Construct a Huffman code for the following data:

Character A B C D -

Probability 0.40 0.10 0.20 0.15 0.15

Scheme:

Construction of Huffman tree – 2 M

Final code table – 2 M

Answer:

Step 1: Arrange the Probabilities in Ascending Order

Character Probability

B 0.10

D 0.15

– 0.15

C 0.20

Character	Probability
A	0.40

Step 2: Construct the Huffman Tree

Iteration 1

Combine the two smallest probabilities:

B(0.10)+D(0.15)=0.25
Remaining nodes:

Node	Probability
–	0.15
C	0.20
BD	0.25
A	0.40

Iteration 2

Combine

-(0.15)+C(0.20)=0.35
Remaining nodes

Node	Probability
BD	0.25
–C	0.35
A	0.40

Iteration 3

Combine

BD(0.25)+(-C)(0.35)=0.60
Remaining nodes

Node	Probability
A	0.40

Node Probability

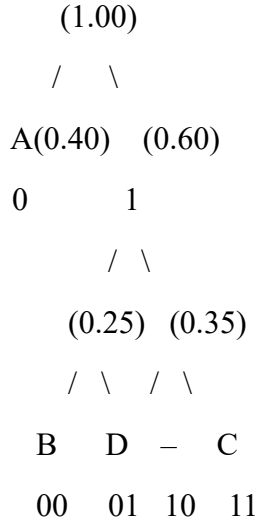
BD-C 0.60

Iteration 4

Combine

A(0.40)+BD-C(0.60)=1.00
The Huffman Tree is complete.

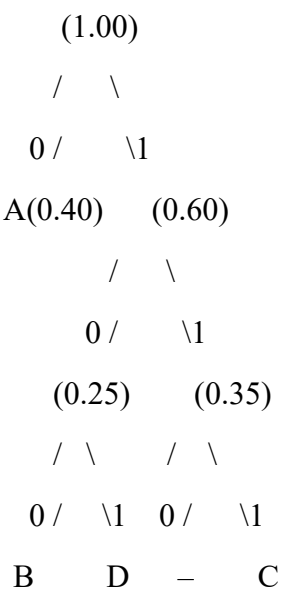
Huffman Tree



To ensure the prefix property, assign edge labels from the root:

- Left edge = **0**
- Right edge = **1**

Thus, the codes become:



Huffman Codes

Character	Huffman Code
A	**0**
B	**100**
D	**101**
–	**110**
C	**111**

Final Answer

Character	Probability	Huffman Code
A	0.40	**0**
B	0.10	**100**
C	0.20	**111**
D	0.15	**101**
–	0.15	**110**

Q.9.a. Explain the following with examples:

(i) Class P problems

(ii) NP complete problem

Scheme:

Class P – 5 M

NP Complete – 5 M

Answer:

(i) Class P Problems

Definition

Class P (Polynomial Time) is the class of decision problems that can be solved by a deterministic algorithm in **polynomial time**, i.e., the running time is bounded by a polynomial function of the input size.

If the input size is (n), then the running time is of the form

O(1),; O(n),; O(n^2),; O(n^3),; O(n^k)
where (k) is a constant.

Hence, problems in Class P are considered **efficiently solvable**.

Examples of Class P Problems

- Searching (Binary Search)
- Sorting (Merge Sort, Heap Sort)
- Minimum Spanning Tree (Prim's and Kruskal's algorithms)
- Single Source Shortest Path (Dijkstra's algorithm)
- Matrix Multiplication
- Breadth First Search (BFS)
- Depth First Search (DFS)

Characteristics of Class P

- Solvable in polynomial time.
- Efficient deterministic algorithms exist.
- Considered computationally tractable.
- Every P problem is also an NP problem.

(ii) NP-Complete Problems

Definition

A problem is **NP-Complete (NPC)** if

1. It belongs to **NP**, i.e., a proposed solution can be verified in polynomial time.
2. Every problem in NP can be reduced to it in polynomial time.

Thus, NP-Complete problems are the **hardest problems in NP**.

No polynomial-time algorithm is currently known for solving NP-Complete problems.

Characteristics of NP-Complete Problems

- Solution verification takes polynomial time.
- No known polynomial-time algorithm exists.
- Every NP problem can be transformed into an NP-Complete problem in polynomial time.
- If any NP-Complete problem is solved in polynomial time, then

P = NP

Examples of NP-Complete Problems

- Travelling Salesman Problem (Decision Version)
- Hamiltonian Cycle Problem
- Vertex Cover Problem
- Clique Problem
- Graph Coloring Problem
- Subset Sum Problem
- Knapsack Problem (Decision Version)
- SAT (Boolean Satisfiability Problem)
- 3-SAT Problem

Difference Between P and NP-Complete Problems

Class P	NP-Complete
Solvable in polynomial time	No known polynomial-time algorithm
Easy to solve	Difficult to solve
Deterministic algorithms exist	Only verification is polynomial-time
Considered tractable	Considered computationally intractable
Examples: Merge Sort, BFS, Dijkstra, Prim	Examples: TSP, SAT, Hamiltonian Cycle, Clique

Q.9.b. Give the problem statement of n-queens problem. Explain the solution for 4-queens problem using state space tree.

Scheme:

n- queens explanation – 3 M

State Space tree – 7 M

Answer:

- The problem is to place (n) queens on an (n \times n) chessboard so that no two queens attack each other by being in the same row, in the same column, or on the same diagonal.
- For (n = 1), the problem has a trivial solution, and for (n = 2) and (n = 3), there is no solution.
- So, let us consider the four-queens problem and solve it using the backtracking technique.
- Since each of the four queens has to be placed in its own row, assign one column for each queen on the board as shown in the figure.

Figure 1: Board for the four-queens problem

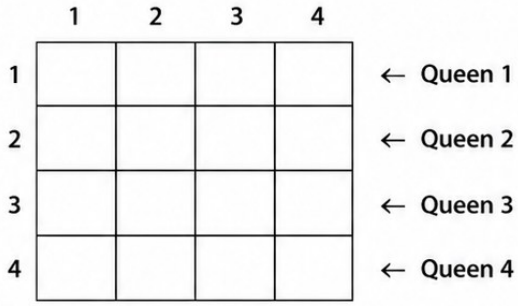
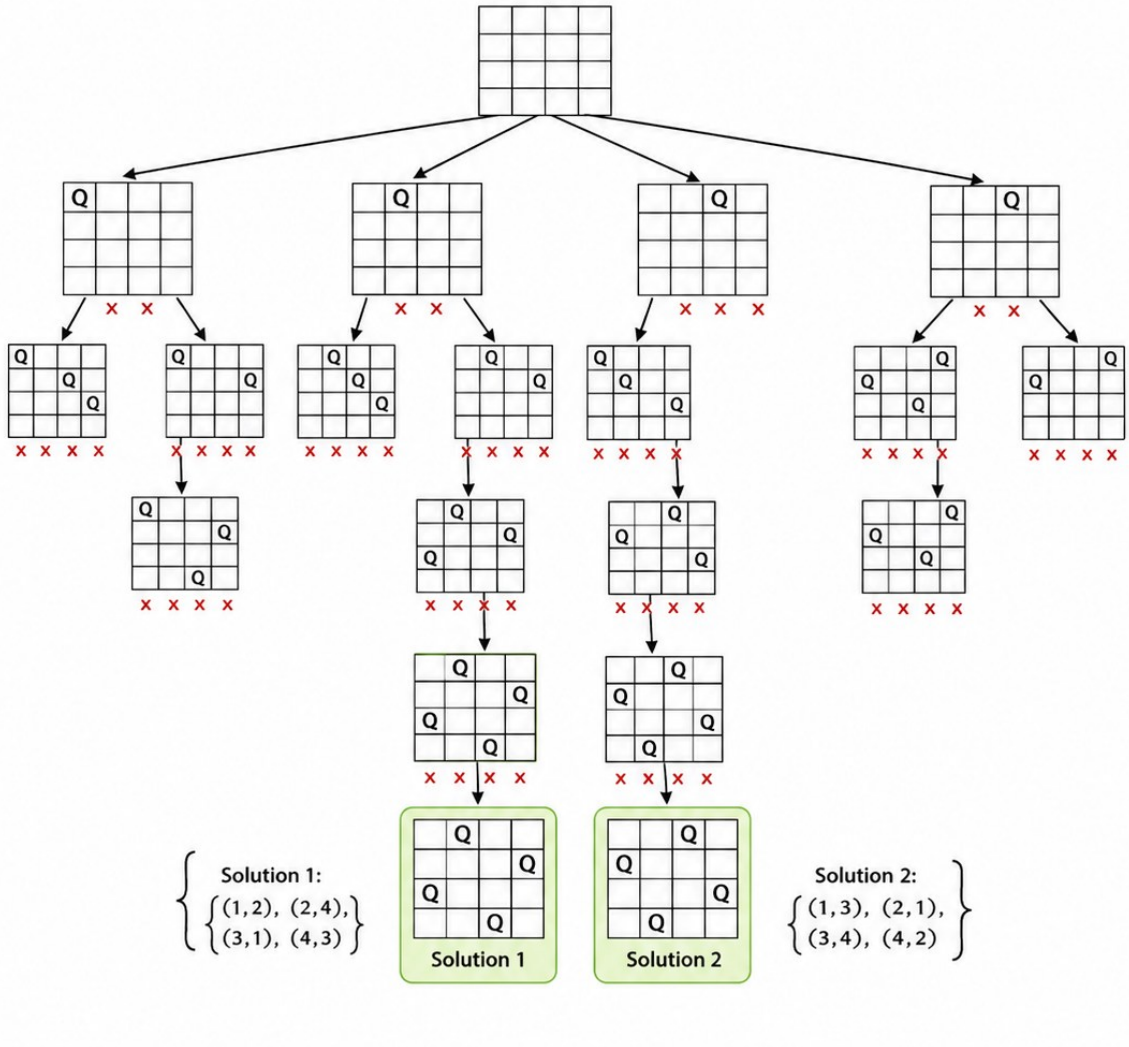


Figure 2: State space tree for solving 4-queens problem



- We start with an empty board and then place the first queen in the first possible position of its row, i.e., (1,1).
- Then place Queen 2. Columns 1 and 2 are unsuccessful, so place it in (2,3).
- Then placing Queen 3 leads to a dead end. The algorithm backtracks and moves Queen 2 to the next position, i.e., (2,4).
- Then Queen 3 is placed in (3,2), but again it leads to a dead end. The algorithm backtracks all the way to Queen 1 and moves it to (1,2).

- The queens are then placed as follows:
  - Queen 2 → (2,4)
  - Queen 3 → (3,1)
  - Queen 4 → (4,3)

This gives one valid solution to the 4-Queens problem.

Algorithm: N-Queens (Backtracking)

```
Algorithm N-Queens(k, n)
// Using backtracking, the algorithm prints all possible solutions.
for i ← 1 to n do
   if Place(k, i) then
      x[k] ← i
      if k = n then
         print x[1...n]
      else
         N-Queens(k + 1, n)
```

Algorithm: Place(k, i)

```
Algorithm Place(k, i)
// Returns True if a queen can be placed at the k-th row and i-th column.
// Otherwise, returns False.
for j ← 1 to (k − 1) do
   if (x[j] = i) OR (abs(x[j] − i) = abs(j − k)) then
      return False
return True
```

Row Attack, Column Attack, and Diagonal Attack

N-Queens Array Representation

Solution 1

Index (Row) 1 2 3 4

x (Column) 3 1 4 2

i.e.,

x = [3, 1, 4, 2]

Solution 2

Index (Row) 1 2 3 4

x (Column) 2 4 1 3

i.e.,

x = [2, 4, 1, 3]

Notes

- **Index of the array x represents the row.**
- **Value of the array x represents the column.**
- **The challenging part is detecting diagonal attacks.**

Diagonal Attack Test

Case 1

Consider any two queens:

(3,2) → (i, j)

(4,3) → (k, l)

Then,

i − j = k − l

3 − 2 = 4 − 3

1 = 1

Hence,

(i − j) = (k − l)

or equivalently,

abs(i − k) = abs(j − l)

Case 2

Consider another pair:

(3,3) → (i, j)

(4,2) → (k, l)

Then,

i + j = k + l

3 + 3 = 4 + 2

6 = 6

Hence,

(i + j) = (k + l)

which is also equivalent to the diagonal condition

abs(i − k) = abs(j − l)

Diagonal Attack Condition

Two queens placed at positions

(i, j) and (k, l)

attack each other diagonally **if and only if**

|i-k| = |j-l|

This is the condition used in the **Place(k, i)** function to determine whether a queen can be safely placed.

Q.10.a. Explain how knapsack problem can be solved using branch and bound technique, with example.

Scheme:

Explanation – 2 M

Example – 8 M

Answer:

- Given (n) items of known weights (w_i) and values (v_i), where

i = 1,2,…..,n,
and a knapsack of capacity (W), find the most valuable subset of items that fits into the knapsack.

- Order the items of a given instance in descending order according to their value-to-weight ratios.

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$$

- The first item gives the best payoff per unit weight, while the last item gives the worst payoff per unit weight.

- Construct the state-space tree as a binary tree.

- Each node at the (i^{th}) level of the tree ((0 \le i \le n)) represents all subsets of the (n) items that include a particular selection made from the first (i) ordered items.

- A particular selection is uniquely determined by the path from the root to the node.
    - A left branch indicates inclusion of the next item.
    - A right branch indicates exclusion of the next item.
- Record the total weight (w) and the total value (v) of the current selection at each node, along with an upper bound on the value obtainable from that node.

Upper Bound (UB)

The **upper bound (UB)** on the value of any subset is obtained by adding zero or more items to the current selection.

Compute the upper bound using:

$$\boxed{\text{UB} = v + (W - w)\left(\frac{v_{i+1}}{w_{i+1}}\right)}$$

where:

- (v) = current total value
- (w) = current total weight
- (W) = knapsack capacity
- $\overline{\left(\frac{v_{i+1}}{w_{i+1}}\right)}$= value-to-weight ratio of the next item

Example

Knapsack capacity:

W=10

Item	Weight	Value	Value/Weight
1	4	$40	10 = (40/4) = (v_1/w_1)
2	7	$42	6 = (42/7) = (v_2/w_2)
3	5	$25	5 = (25/5) = (v_3/w_3)
4	3	$12	4 = (12/3) = (v_4/w_4)

- **Calculate upper bound (UB):**
  - **Initially,** $v = 0$, $w = 0$, $v_1/w_1 = 10$

$$UB = v + (W - w)\left(\frac{v_{i+1}}{w_{i+1}}\right) \Rightarrow UB = 0 + (10 - 0)\left(\frac{v_1}{w_1}\right)$$

$$= 0 + 10 \times 10$$

UB = 100

  - $w = 4$, $v = 40$, $v_2/w_2 = 6$ **(with Item 1)**

$$UB = 40 + (10 - 4)\left(\frac{v_2}{w_2}\right) \Rightarrow UB = 40 + (10 - 4)\left(\frac{v_2}{w_2}\right)$$

$$= 40 + 6 \times 6$$

UB = 76

  - $w = 0$, $v = 0$, $v_2/w_2 = 6$ **(without Item 1)**

$$UB = 0 + (10 - 0)\left(\frac{v_2}{w_2}\right) \Rightarrow UB = 0 + (10 - 0) \times 6 = 10 \times 6 = 60$$

UB = 60

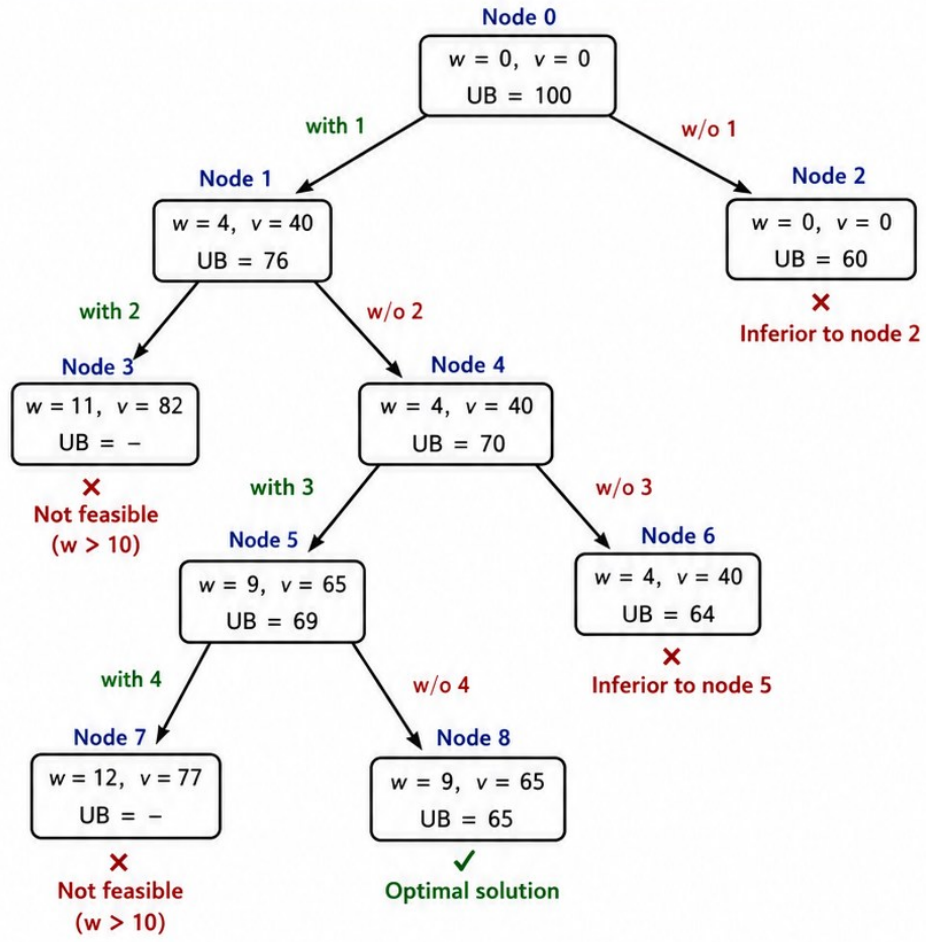


Figure: State-space tree of the best-first B & B algorithm for the instance of the knapsack problem.

- w = 4, v = 40 , v_3/w_3 (without 2)

  UB = 40 + (10 – 4) 5

  = 40 + 6 × 5

  UB = 70

- w = 9, v = 65 , v_4/w_4 (with 3)

  UB = 65 + (10 – 9) 4 = 65 + 4 = 69 **UB = 69**

- w = 4, v = 40 , v_4/w_4 (without 3)

  UB = 40 + (10 – 4) 4 = 40 + 6 × 4 = 64 **UB = 64**

- w = 9, v = 65 , (without 4), no next item, $v_{i+1}/w_{i+1} = 1$

  UB = 65 + (10 – 9) · 1 = 65 + 1 = 65 **UB = 65**

- Objects/items to be placed in knapsack are 1^{st} & 3^{rd}.

  ∴ Max profit = **65**

Q.10.b. Discuss Decision tree for three element selection sort and binary search in a four element array.

Scheme:

Selection sort – 5 M (Explanation – 1M, Decision tree – 3M, Time complexity – 1M)

Binary search – 5 M (Explanation – 1M, Decision tree – 3M, Time complexity – 1M)

Answer:

Selection Sort for Three Elements

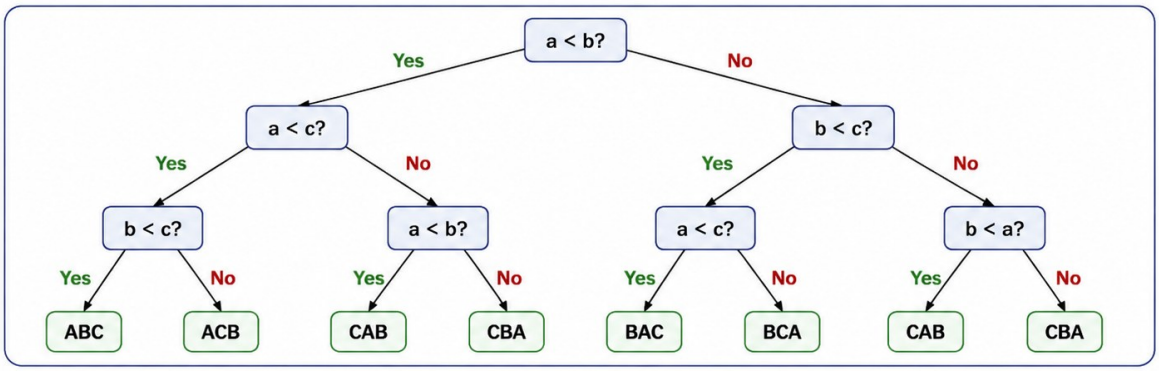
Let the three elements be:

a, b, c

Selection sort first finds the smallest element among the three, places it in the first position, and then compares the remaining two elements. Since there are 3!=63! = 63!=6 possible input orders, the decision tree has 6 leaf nodes.

Steps:

1. Compare a and b.
2. Compare the smaller element with c to determine the minimum.
3. After placing the smallest element in the first position, compare the remaining two elements.
4. The leaf node gives the final sorted order.



All Possible Outcomes (Sorted Orders)

Leaf	1	2	3	4	5	6
Sorted Order	ABC	ACB	BAC	BCA	CAB	CBA

Number of Comparisons

- Best Case : 2 comparisons
- Worst Case : 3 comparisons

Time Complexity

For n elements,

$T(n) = \frac{n(n-1)}{2}$ comparisons

For $n = 3$, $T(3) = 3$ comparisons

$T(n) = O(n^2)$

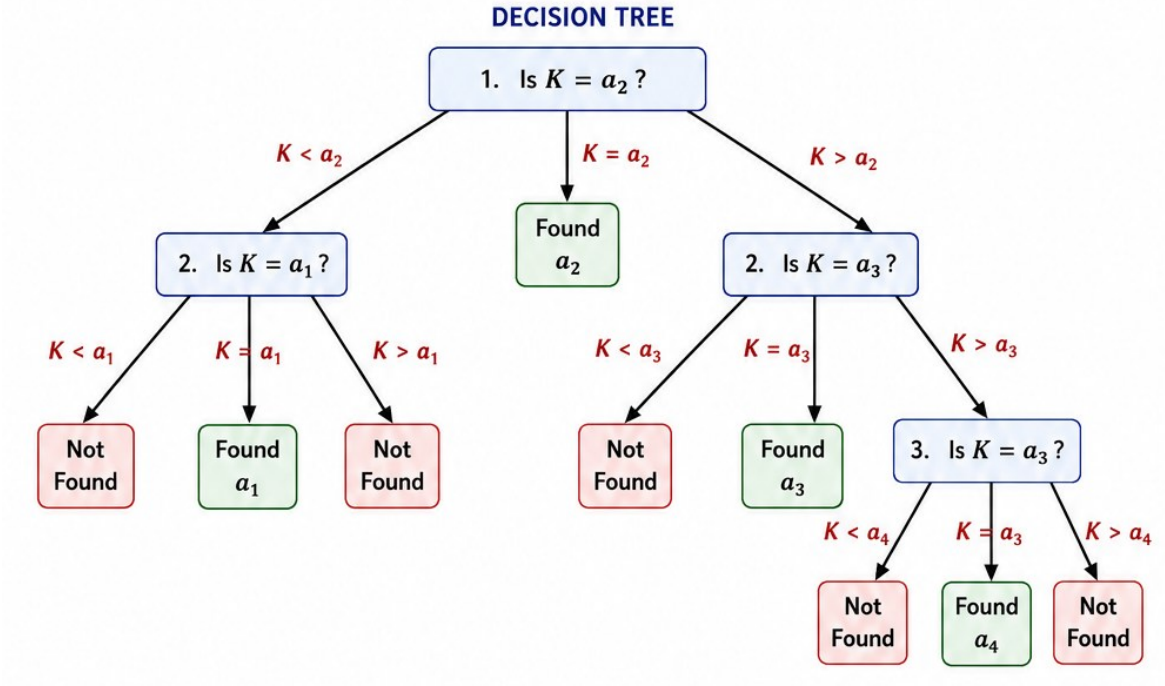
Note

The decision tree shows all possible comparison sequences and the 6 possible sorted permutations.

Decision Tree for Binary Search in a Four-Element Array

Let the sorted array be $A = [a_1, a_2, a_3, a_4]$ where $a_1 < a_2 < a_3 < a_4$

Assumption: For even number of elements, we choose the second middle element (a_2) as the first comparison.



EXPLANATION

Step 1

Compare the key K with the middle element a_2.

- If $K = a_2$ → Found.
- If $K < a_2$ → Search in the left half $[a_1]$.
- If $K > a_2$ → Search in the right half $[a_3, a_4]$.

Step 2

Left Half: Compare with a_1.

- $K = a_1$ → Found
- $K < a_1$ → Not Found
- $K > a_1$ → Not Found

Right Half: Compare with a_3.

- $K = a_3$ → Found
- $K < a_3$ → Not Found
- $K > a_3$ → Compare with a_4

Step 3

Compare with a_4.

- $K = a_4$ → Found
- $K < a_4$ → Not Found
- $K > a_4$ → Not Found

LEVELS OF DECISION TREE

Level	Element Compared	Subarray Considered
0 (Root)	a_2	$[a_1, a_2, a_3, a_4]$
1	a_1 or a_3	Left half $[a_1]$ or Right half $[a_3, a_4]$
2	a_4	Right half's remaining part $[a_4]$

NUMBER OF COMPARISONS

- **Best Case** : 1 comparison (Key is a_2)
- **Worst Case** : 3 comparisons (Key is a_4 or key not present)

TIME COMPLEXITY

For binary search,

$T(n) = O(\log_2 n)$

For $n = 4$, $\log_2 4 = 2$

- **Best Case** : $O(1)$
- **Worst Case** : $O(\log n)$

SUMMARY

The decision tree shows all possible comparisons while searching a key in a four-element sorted array using binary search.