1.Discuss different type of asymptoic notation.

1. Big O Notation – O(f(n))

Definition: It gives the upper bound of an algorithm's running time.

Purpose: Describes the worst-case scenario.

Usage: Guarantees that the algorithm will not take more than this time.

Example:

If an algorithm takes at most $3n^2 + 5n + 2$ steps, we write:

$O(n^2)$

2. Omega Notation – Ω(f(n))

Definition: It gives the lower bound of an algorithm's running time.

Purpose: Describes the best-case scenario.

Usage: Guarantees that the algorithm will take at least this much time.

Example:

If an algorithm takes at least `n` steps in the best case, we write:

$\Omega(n)$

3. Theta Notation – Θ(f(n))

Definition: It gives both the upper and lower bound of an algorithm's running time.

Purpose: Describes the exact asymptotic behavior.

Usage: Useful when best-case and worst-case are the same.

Example:

If an algorithm takes $5n + 3$ steps, we write:

$\Theta(n)$

2.State the 0/1 knapsack problem.

The 0/1 Knapsack Problem is a classic optimization problem in computer science and mathematics. It is defined as follows:

Problem Statement:

Given:

A set of `n` items, each with:

A weight `w[i]`

A value `v[i]`

A knapsack with a maximum weight capacity `W`

Objective: * Select items to include in the knapsack such that:

*The total weight does not exceed the capacity `W`

The total value is maximized

Each item can be included only once (either 0 or 1 time – hence the name 0/1)

Mathematical Formulation:

Let: `x[i]` = 1 if item `i` is included, 0 otherwise

Maximize:

$$\sum_{i=1}^{n} v[i] \cdot x[i]$$

Subject to:

$$\sum_{i=1}^{n} w[i] \cdot x[i] \le W \quad \text{and} \quad x[i] \in \{0,1\}$$

Example:

Suppose you have:

Items:

Item 1: value = 60, weight = 10
Item 2: value = 100, weight = 20
Item 3: value = 120, weight = 30
Knapsack capacity `W = 50`
You must decide which items to include to maximize total value without exceeding 50 weight.

3.Write an algorithm of matrix chain multiplication.
Input:
`p[0..n]`: Dimensions of `n` matrices where matrix `Ai` has size `p[i-1] × p[i]`.
Output:
Minimum number of scalar multiplications to multiply the chain.

◈ Algorithm:

```
MatrixChainOrder(p, n):
1. Create m[1..n][1..n] and set m[i][i] = 0 for all i

2. for L = 2 to n:               // L = chain length
3.   for i = 1 to n - L + 1:
4.     j = i + L - 1
5.     m[i][j] = ∞
6.     for k = i to j - 1:
7.       cost = m[i][k] + m[k+1][j] + p[i-1] * p[k] * p[j]
8.       if cost < m[i][j]:
9.         m[i][j] = cost

10. return m[1][n]
```

4.Queen problem using backtracking approach .

N-Queens Problem – Backtracking Algorithm (Short)
text
Copy code

```
function solveNQueens(row):
    if row == N:
        print board
        return

    for col = 0 to N-1:
        if isSafe(row, col):
            place queen at (row, col)
            solveNQueens(row + 1)
            remove queen from (row, col)  // backtrack

function isSafe(row, col):
    check column, upper-left diagonal, and upper-right diagonal
    return True if no queen attacks
```

How it works:
Start placing queens row by row.
For each row, try placing the queen in all columns.
Use `isSafe()` to check if a position is valid.
If valid, place the queen and move to the next row.
If a placement leads to no soluti
5.Difference Between divide and conquer and dynamic programming.

Aspect	Divide and Conquer	Dynamic Programming

Aspect	Divide and Conquer	Dynamic Programming
Problem Type	Breaks into independent subproblems	Breaks into overlapping subproblems
Subproblem Reuse	Solves each subproblem separately	Reuses results of subproblems
Storage	No storage of intermediate results	Yes, stores results (memoization/tabulation)
Efficiency	May repeat work	Avoids repetition, hence more efficient
Examples	Merge Sort, Quick Sort, Binary Search	Fibonacci, 0/1 Knapsack, Matrix Chain Multiplication

5.What do you mean by divide and conquer strategy?
Divide and Conquer Strategy – Definition
Divide and Conquer is a problem-solving strategy where:
Divide: The main problem is divided into smaller **subproblems** of the same type.
Conquer: Each subproblem is **solved recursively**.
Combine: The solutions of subproblems are **combined** to get the final result.
Key Idea:
Break the problem down, solve the smaller parts independently, and combine them.
📌 **Examples:**
Merge Sort: Divide the array, sort each half, and merge.
Quick Sort: Partition the array, recursively sort subarrays.
Binary Search: Repeatedly divide the search interval in half.
◈ **Advantages:**
Reduces complex problems into manageable ones.
Often leads to efficient algorithms.

6.What is the dynamic programming?
Dynamic Programming (DP) is a method used to **solve complex problems by breaking them into smaller overlapping subproblems**, solving each subproblem **only once**, and **storing their results** to avoid repeated work.
Key Concepts:
Overlapping Subproblems – Same subproblems appear multiple times.
Optimal Substructure – The solution to the problem can be built from the solutions of its subproblems.
Memoization (Top-Down) – Store results of recursive calls.
Tabulation (Bottom-Up) – Build solution iteratively using a table.
📌 **Examples:**
Fibonacci sequence
0/1 Knapsack problem
Longest Common Subsequence (LCS)
Matrix Chain Multiplication

7.Discuss activity selection problem for job sequencing .
Activity Selection Problem for Job Sequencing
The **Activity Selection Problem** (also known as **Job Sequencing with Deadlines**) is a classic **greedy algorithm problem**. It focuses on selecting the **maximum number of activities** or **jobs** that can be done by a person/machine **without overlapping**.
Problem Statement:
Given `n` activities with:
Start time: `start[i]`
Finish time: `finish[i]`

Objective:
Select the **maximum number of non-overlapping activities** that can be performed by a single person, one at a time.

Greedy Strategy:

Sort all activities by their **finish time**.
Select the first activity.
For each next activity, **select it only if** its **start time ≥ finish time** of the previously selected activity.

Algorithm:

text
Copy code

```
ActivitySelection(start[], finish[], n):
1. Sort activities by finish time
2. Select the first activity and set `last_finish = finish[0]`
3. for i = 1 to n-1:
       if start[i] ≥ last_finish:
           select activity i
           last_finish = finish[i]
```

📌 **Example:**

Activity	Start	Finish
A1	1	3
A2	2	4
A3	3	5
A4	0	6
A5	5	7

After sorting and applying the algorithm, selected activities could be: **A1, A3, A5**.

8.Difference between dynamic programming and greedy method.

Feature	Dynamic Programming	Greedy Method
Approach	Solves all subproblems and combines solutions	Makes locally optimal choices at each step
Subproblem Reuse	Yes (uses overlapping subproblems)	No reuse
Optimal Guarantee	Always gives optimal solution	May or may not give optimal solution
Used When	Problem has overlapping subproblems + optimal substructure	Problem has greedy-choice property
Examples	Fibonacci, 0/1 Knapsack, LCS	Activity Selection, Kruskal's, Prim's

9.What is the heap property? Write algorithm of heap sort? Find the running time of this algorithm.

A heap is a special binary tree that satisfies the heap property:
Max-Heap: For every node `i`, the value of `i` ≥ its children. (Root is the largest)
Min-Heap: For every node `i`, the value of `i` ≤ its children. (Root is the smallest)
Also, a heap is always a complete binary tree, meaning all levels are fully filled except possibly the last, which is filled left to right.

🔧 Heap Sort Algorithm

Input: An array `A[1..n]`
Output: Sorted array in ascending order

📃 Algorithm (Heap Sort in Ascending Order):

```
text
Copy code
HeapSort(A):
1. BuildMaxHeap(A)
2. for i = length(A) down to 2:
     swap A[1] with A[i]
     heap_size = heap_size - 1
     MaxHeapify(A, 1)

BuildMaxHeap(A):
1. heap_size = length(A)
2. for i = floor(n/2) down to 1:
     MaxHeapify(A, i)

MaxHeapify(A, i):
1. left = 2 * i
2. right = 2 * i + 1
3. largest = i
4. if left ≤ heap_size and A[left] > A[largest]:
       largest = left
5. if right ≤ heap_size and A[right] > A[largest]:
       largest = right
6. if largest ≠ i:
       swap A[i] with A[largest]
       MaxHeapify(A, largest)
```

□ Time Complexity

Step	Time
`BuildMaxHeap()`	O(n)
`MaxHeapify()`	O(log n) per call
Heap Sort loop	O(n log n)

Total Time Complexity:
? O(n log n) (in all cases: best, average, worst)

10.Write short notes on the following: a)8 queens problem b)External sorting c)Heap creation technique d)Hamiltonian cycle

a) 8 Queens Problem

The **8 Queens Problem** involves placing 8 queens on a chessboard so that no two queens threaten each other. This means no two queens can be in the same row, column, or diagonal. The problem is typically solved using **backtracking**, where queens are placed row by row, and the algorithm backtracks if a conflict is found, trying different positions until all 8 queens are safely placed.

b) External Sorting

External Sorting is used when the data size is too large to fit into the computer's main memory (RAM). The data is divided into smaller chunks that fit into memory, each chunk is sorted individually, and then the sorted chunks are merged together. This method minimizes disk I/O operations and is often implemented using **external merge sort**.

c) Heap Creation Technique

Heap creation builds a heap (max-heap or min-heap) from an unsorted array. The efficient way is the **bottom-up method**, where starting from the last non-leaf node, the `heapify` process is applied upward to the root. This ensures the heap property is maintained throughout the tree. This technique runs in **linear time O(n)**, which is faster than building the heap by inserting elements one by one.

d) Hamiltonian Cycle

A **Hamiltonian cycle** is a path in a graph that visits every vertex exactly once and returns to the starting vertex, forming a cycle. Finding whether such a cycle exists in a graph is a difficult problem classified as **NP-complete**, meaning no known efficient algorithm solves all cases quickly. It is used in routing, scheduling, and optimization problems.

11.Bellman ford algorithm

Initialize distances:

Set the distance to the **source vertex** as `0`.+

Set the distance to all other vertices as **infinity** (∞).

Relax edges repeatedly:

Repeat for `(V - 1)` times (where V = number of vertices):

For **each edge (u, v)** with weight `w`:

If `distance[u] + w < distance[v]`, then update `distance[v] = distance[u] + w`.

Check for negative weight cycles:

For each edge `(u, v)` with weight `w`:

If `distance[u] + w < distance[v]`, then **a negative weight cycle exists** in the graph.

Output:

If no negative weight cycle is found, the `distance[]` array contains the shortest distances from the source to every vertex.

12.Write down Floyd's algorithm to find all paired shortest paths of a graph.

Floyd's Algorithm (Step-by-Step)

Initialize Distance Matrix

`dist[i][j] = 0` if `i == j`

`dist[i][j] = weight(i→j)` if edge exists

`dist[i][j] = ∞` if no edge

Iterate Over Intermediate Vertices (k)

For `k = 1 to n`:

For `i = 1 to n`:

For `j = 1 to n`:

If `dist[i][j] > dist[i][k] + dist[k][j]`

→ update `dist[i][j] = dist[i][k] + dist[k][j]`

Output Final Distance Matrix

Contains shortest paths between all pairs of vertices.

▯ Time Complexity:

O(n^3)

13..Write short notes on the following:

a)approximation algorithm b) approximation schemey c)Vertex cover algorithm. d)FFT e)strassen's matrix multiplication f)Amortized analysis g)15 puzzle problem

a) **Approximation Algorithm**

An approximation algorithm is used for **NP-hard problems** where finding the exact optimal solution is difficult or time-consuming. It provides a solution that is **close to optimal** within a guaranteed **approximation ratio**. Example: Vertex Cover Approximation.

b) **Approximation Scheme**

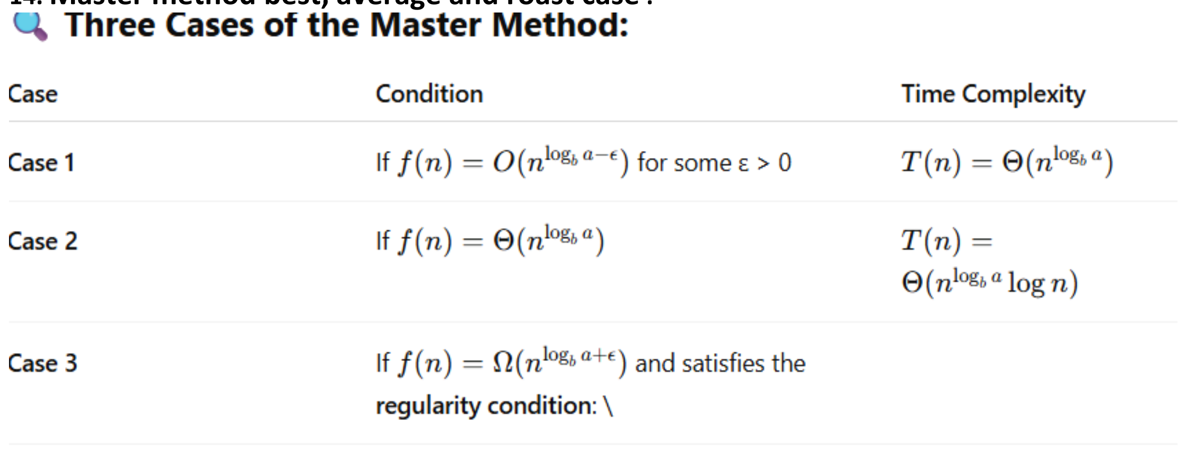
An approximation scheme is an algorithm that takes an input parameter **ε (epsilon)** and produces a solution that is within a factor of **(1 + ε)** of the optimal.

If it runs in polynomial time for **fixed ε**, it's called a **PTAS** (Polynomial-Time Approximation Scheme).

If polynomial for both `n` and `1/ε`, it's an **FPTAS**.

c) **Vertex Cover Algorithm**
The **Vertex Cover Problem** asks for the smallest set of vertices such that every edge in the graph is incident to at least one selected vertex.
A **2-approximation algorithm** picks both endpoints of each uncovered edge.
Time Complexity: **O(E)** (for the greedy version).
d) **FFT (Fast Fourier Transform)**
FFT is an efficient algorithm to compute the **Discrete Fourier Transform (DFT)** in **O(n log n)** time.
Used in **signal processing**, **polynomial multiplication**, and **image compression**.
Reduces time complexity from O(n^2) to O(n log n).
e) **Strassen's Matrix Multiplication**
Strassen's algorithm multiplies two matrices faster than the standard O(n^3) method.
It uses **divide and conquer** to reduce the number of multiplications.
Time Complexity: **O(n^2.81)**
Useful for large matrices.
f) **Amortized Analysis**
It's a method to analyze the average performance of an algorithm over a sequence of operations, **even if a single operation is expensive**.
Types: **Aggregate**, **Accounting**, and **Potential** methods.
Example: Inserting into a dynamic array.
g) **15 Puzzle Problem**
It's a **sliding puzzle** with 15 numbered tiles in a 4×4 grid and one blank space.
The goal is to arrange the tiles in order.
Solved using **A*** or other heuristic search algorithms.
It is **NP-hard** in its generalized form.

14. Master method best, average and roast case .



🔍 **Three Cases of the Master Method:**

Case	Condition	Time Complexity
Case 1	If $f(n) = O(n^{\log_b a - \epsilon})$ for some ε > 0	$T(n) = \Theta(n^{\log_b a})$
Case 2	If $f(n) = \Theta(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
Case 3	If $f(n) = \Omega(n^{\log_b a + \epsilon})$ and satisfies the regularity condition: \	

15.Write a quick sort algorithms step by step and find the best, roast and average case time complexity.

Step 1:

```
If low < high then:
pivotIndex ← Partition(A, low, high)
QuickSort(A, low, pivotIndex - 1)
QuickSort(A, pivotIndex + 1, high)
```

Sub-Algorithm: `Partition(A, low, high)`

```
pivot ← A[high]
i ← low - 1
For j ← low to high - 1 do:
a. If A[j] ≤ pivot then:
  i ← i + 1
  Swap A[i] with A[j]
Swap A[i + 1] with A[high]
Return i + 1 (pivotIndex)
```

Time Complexities:

Case	Time Complexity
Best Case (Balanced partition)	**O(n log n)**
Average Case	**O(n log n)**
Worst Case (Unbalanced, already sorted)	**O(n²)**

16.Write a algorithms step by step merge short. Find the best ,average, roast case time complexity of Merge sort .

```
If low < high then:
Find the middle point:
mid=low+high/2
Recursively sort the first half:
MergeSort(A, low, mid)
Recursively sort the second half:
MergeSort(A, mid + 1, high)
Merge the two sorted halves by calling:
Merge(A, low, mid, high)
```

Merge Sort Time Complexity

Case	Time Complexity	Explanation
Best Case	**O(n log n)**	Even if the array is already sorted, Merge Sort divides and merges recursively, so time remains the same.
Average Case	**O(n log n)**	Average case follows the same divide-and-conquer steps.
Worst Case	**O(n log n)**	Occurs when array is in reverse order or any arbitrary order; merge operations remain the same.

17.Write a algorithms step by step Bubble short. Best ,average and worst case time complexity.

Algorithm: BubbleSort(A, n)
Input: An array A[] of size n
Output: Sorted array A[] in ascending order
Steps:

```
For i = 0 to n - 1 do:
(This is for each pass through the array)
  For j = 0 to n - i - 2 do:
Compare A[j] and A[j + 1]
    If A[j] > A[j + 1] then:
Swap A[j] and A[j + 1]
End
```

Explanation:
The largest element "bubbles up" to the end in each pass.
In each outer loop, one less element needs to be checked.

Time Complexity:

Case	Time	Reason
Best Case (already sorted)	**O(n)**	One pass with no swaps (optimized version)
Average Case	**O(n²)**	Repeated comparisons and swaps
Worst Case (reverse order)	**O(n²)**	Maximum number of swaps needed

18.Write a algorithms step by step Insertion short. Beast, wrost and average case time complexity.

Algorithm: `InsertionSort(A, n)`

Input: An array `A[]` of size `n`

Output: Sorted array `A[]` in ascending order

Steps:

```
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
End For
```

▯ Time Complexity:

Case	Time	Reason
Best Case (already sorted)	**O(n)**	Only one comparison per element
Average Case	**O(n^2)**	Comparisons and shifts needed
Worst Case (reverse order)	**O(n^2)**	Maximum shifts required

19. Write a algorithms step by step Selection short. Best ,average and worst case time complexity.

Algorithm: `InsertionSort(A, n)`

Input: An array `A[]` of size `n`

Output: Sorted array `A[]` in ascending order

Steps:

```
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
End For
```

▯ Time Complexity:

Case	Time	Reason
Best Case (already sorted)	O(n)	Only one comparison per element
Average Case	O(n^2)	Comparisons and shifts needed
Worst Case (reverse order)	O(n^2)	Maximum shifts required

20.Write a algorithms step by step Binary search. Best ,average and worst case time complexity.

Algorithm: `BinarySearch(A, n, key)`

Input:

A sorted array `A[0..n-1]`

A target value `key`

Output:

The index of `key` if found, otherwise `-1`

Steps:

Set `low ← 0`, `high ← n - 1`

While `low ≤ high` do:
`mid ← (low + high) // 2`
If `A[mid] == key` then
→ Return `mid`
Else if `A[mid] < key` then
→ `low ← mid + 1`
Else
→ `high ← mid - 1`
If not found, return `-1`
TTime Complexity:

Case	Time
Best Case	O(1) (found in first guess)
Average Case	O(log n)
Worst Case	O(log n)

21.Write a algorithms step by step heap sort. Best ,average and worst case time complexity.

1. **Max Heap(Ascending order)**
Property: Parent ≥ Children
Max Element: Always at the **root** (`A[0]`)
Used to **quickly find the maximum** value.
2. **Min Heap(Descending order)**
Property: Parent ≤ Children
Min Element: Always at the **root** (`A[0]`)
Used to **quickly find the minimum** value.

Extract Max (Max Heap)	O(log n)	O(log n)	O(log n)	After removal, heapify down costs O(log n)
Extract Min (Min Heap)	O(log n)	O(log n)	O(log n)	Same as extract max but for min heap

22.Write a algorithms step by step Huffman. Best ,average and worst case time complexity.

Build a min-heap with all characters and their frequencies.
Repeat until one node remains:
Extract two nodes with smallest frequencies.
Create a new internal node with frequency = sum of extracted nodes.
Insert the new node back into the min-heap.
The last node is the root of the Huffman tree.
Traverse the tree to assign binary codes:
Left edge = 0
Right edge = 1
Time Complexity
Best, Average, Worst: O(n log n)
(where n = number of characters)

23.Write a algorithms step by step Dijkstra. Best ,average and worst case time complexity.

Dijkstra's Algorithm (Short)
Initialize distances: `dist[] = ∞`, `dist[src] = 0`
Use a min-heap (priority queue) and insert `(src, 0)`
While heap not empty:
Extract vertex `u` with smallest distance

For each neighbor `v` of `u`, update `dist[v]` if shorter path found
Repeat until all vertices processed
Time Complexity:
Best, Average, Worst: O((V + E) log V)
(where V = number of vertices, E = number of edges)

24.Write a algorithms step by step krusal. Best ,average and worst case time complexity.
1. Sort all edges by increasing weight
2. Initialize each vertex as its own set (Union-Find)
3. For each edge (u, v):

If u and v are in different sets:
Add edge to MST
Union the sets
Stop when MST has (V - 1) edges
Time Complexity:
Best / Average / Worst: O(E log E)
(E = number of edges)

25.Write a algorithms step by step prims. Best ,average and worst case time complexity.
Start with any vertex; set its key = 0, others = ∞
Use a min-heap to pick the vertex with the smallest key
Add the vertex to the MST
Update keys of its adjacent vertices if a shorter edge is found
Repeat until all vertices are in the MST
Time Complexity:
Best / Average / Worst:
🔹 **O((V + E) log V)** with Min-Heap
🔹 **O(V^2)** with Adjacency Matrix

26. DFS,BFS algorithm step by step. best ,average and wrost case time complexity.
1. DFS Algorithm (Depth-First Search)
Steps:
Start from the source node and **mark it as visited**.
Visit all **unvisited adjacent vertices** (recursive or use stack).
For each adjacent node, repeat the process.
Backtrack when no unvisited neighbors are left.
Time Complexity (DFS):

Case	Time Complexity
Best	O(V + E)
Average	O(V + E)
Worst	O(V + E)

V = number of vertices, E = number of edges
Applies to both adjacency list and matrix (with slight variation in performance)
. BFS Algorithm (Breadth-First Search)
Steps:
Start from the source node and **enqueue it**.
Mark it as visited.
While the queue is not empty:
Dequeue a node `u`.

Visit all **unvisited adjacent nodes** of `u`, mark them visited, and enqueue them.

Time Complexity (BFS):

Case	Time Complexity
Best	O(V + E)
Average	O(V + E)
Worst	O(V + E)