**PART-C (15 MARK)**

**UNIT I INTRODUCTION**

1. **Explain how analysis of linear search is done with a suitable illustration. (10)**

**(ii) Define recurrence equation and explain how solving recurrence equations are done.(6) (NOV/DEC 2011) (R)**

**(i) Analysis of Linear Search**

**Linear Search** is a simple searching algorithm that checks each element of the list or array one by one until the desired element is found or the entire list is traversed. The key advantage of Linear Search is its simplicity, but it is not the most efficient when dealing with large datasets.

**Linear Search Algorithm:**

python

Copy

def linear\_search(arr, key):

for i in range(len(arr)):

if arr[i] == key:

return i # Return index if found

return -1 # Return -1 if key is not found

**Time Complexity Analysis:**

* **Best Case:** The best case occurs when the element is found in the first position. In this case, the algorithm makes only **1 comparison**.
  + **Best case time complexity:** O(1)O(1)O(1) (constant time)
* **Worst Case:** The worst case occurs when the element is not found in the array, or the element is at the very last position. The algorithm will have to examine every element of the list to conclude that the element is not present.
  + **Worst case time complexity:** O(n)O(n)O(n), where nnn is the size of the list, because the algorithm makes nnn comparisons in the worst case.
* **Average Case:** In the average case, the algorithm will have to search through approximately half of the list before finding the target (if it exists).
  + **Average case time complexity:** O(n)O(n)O(n)

**Illustration of Linear Search:**

Let's consider an array: **arr = [4, 2, 7, 1, 9]**, and we want to search for the element **7**.

1. Start at the first element: arr[0] = 4. This is not equal to 7.
2. Move to the next element: arr[1] = 2. This is also not equal to 7.
3. Move to the next element: arr[2] = 7. We find that this is equal to 7, so we return the index 2.

Thus, the Linear Search would find the element 7 at index **2** after checking 3 elements.

**Time complexity for this search is O(3)O(3)O(3),** which is effectively **O(n)**.

**Conclusion:**

* **Best Case:** O(1)O(1)O(1)
* **Worst Case:** O(n)O(n)O(n)
* **Average Case:** O(n)O(n)O(n)

Linear Search is simple but inefficient for large datasets, especially when the desired element is near the end or not present at all.

**(ii) Recurrence Equation and Solving Recurrence Equations**

A **recurrence equation** is an equation that defines a function in terms of itself. In computer science, recurrence equations are commonly used to describe the running time of recursive algorithms. These equations capture the problem's behavior as it is broken down into smaller subproblems.

**Definition of Recurrence Equation:**

A recurrence equation defines a sequence where each term is expressed as a function of its preceding terms. In the context of algorithms, recurrence equations are used to describe the running time of recursive algorithms.

For example, the recurrence relation for the Merge Sort algorithm is:

T(n)=2T(n2)+O(n)T(n) = 2T\left(\frac{n}{2}\right) + O(n)T(n)=2T(2n​)+O(n)

* T(n)T(n)T(n) represents the time taken for a problem of size nnn.
* 2T(n2)2T\left(\frac{n}{2}\right)2T(2n​) represents the time to recursively solve two subproblems, each of size n/2n/2n/2.
* O(n)O(n)O(n) represents the linear time to merge the two halves together.

**Solving Recurrence Equations:**

There are several methods to solve recurrence equations. Common methods include:

1. **Substitution Method (Induction)**:
   * This method involves making a guess about the solution and proving it by induction.
   * You assume a form for the solution, then prove it works through mathematical induction.

**Example:**

Consider the recurrence relation for a simple recursive algorithm:

T(n)=T(n−1)+O(1)T(n) = T(n - 1) + O(1)T(n)=T(n−1)+O(1)

* + **Base Case:** T(1)=O(1)T(1) = O(1)T(1)=O(1).
  + Assume T(k)=O(k)T(k) = O(k)T(k)=O(k) for some kkk.
  + Prove that T(k+1)=O(k+1)T(k + 1) = O(k + 1)T(k+1)=O(k+1).
  + By solving, we see that T(n)=O(n)T(n) = O(n)T(n)=O(n).

1. **Recursion Tree Method**:
   * This method visualizes the recurrence as a tree, where each level of the tree represents a recursive call, and you sum the work done at each level.
   * This method is particularly useful for recurrences that split into multiple subproblems.

**Example:**

Consider the recurrence for Merge Sort:

T(n)=2T(n2)+O(n)T(n) = 2T\left(\frac{n}{2}\right) + O(n)T(n)=2T(2n​)+O(n)

The recursion tree for this would look like:

* + At level 0 (root), the work is O(n)O(n)O(n).
  + At level 1, we have two subproblems, each of size n/2n/2n/2, and the total work at this level is 2×O(n/2)=O(n)2 \times O(n/2) = O(n)2×O(n/2)=O(n).
  + At level 2, we have four subproblems, each of size n/4n/4n/4, and the total work at this level is 4×O(n/4)=O(n)4 \times O(n/4) = O(n)4×O(n/4)=O(n).
  + This continues until the problem size reaches 1.

By summing up the work at all levels, we get:

T(n)=O(n)+O(n)+O(n)+⋯=O(nlog⁡n)T(n) = O(n) + O(n) + O(n) + \dots = O(n \log n)T(n)=O(n)+O(n)+O(n)+⋯=O(nlogn)

1. **Master Theorem**:
   * The Master Theorem provides a straightforward way to solve recurrences of the form:

T(n)=aT(nb)+O(nd)T(n) = aT\left(\frac{n}{b}\right) + O(n^d)T(n)=aT(bn​)+O(nd)

where a≥1a \geq 1a≥1, b>1b > 1b>1, and d≥0d \geq 0d≥0.

* + The Master Theorem has three cases:
    1. If a>bda > b^da>bd, then T(n)=O(nlog⁡ba)T(n) = O(n^{\log\_b a})T(n)=O(nlogb​a).
    2. If a=bda = b^da=bd, then T(n)=O(ndlog⁡n)T(n) = O(n^d \log n)T(n)=O(ndlogn).
    3. If a<bda < b^da<bd, then T(n)=O(nd)T(n) = O(n^d)T(n)=O(nd).

**Example:**

For the recurrence relation:

T(n)=2T(n2)+O(n)T(n) = 2T\left(\frac{n}{2}\right) + O(n)T(n)=2T(2n​)+O(n)

We can apply the Master Theorem:

* + a=2a = 2a=2, b=2b = 2b=2, and d=1d = 1d=1.
  + Since a=bda = b^da=bd, the solution is T(n)=O(nlog⁡n)T(n) = O(n \log n)T(n)=O(nlogn).

1. **Give the recursive algorithm which finds the number of binary digits in the binary representation of a positive decimal integer. Find the recurrence relation and complexity.**

To find the number of binary digits (or bits) in the binary representation of a positive decimal integer, we can repeatedly divide the number by 2 (using integer division) and count how many times we need to divide until the number becomes 0.

A recursive approach to solve this problem would look like this:

**Recursive Algorithm:**

python

Copy

def count\_binary\_digits(n):

# Base case: if n is 0, the number of binary digits is 0

if n == 0:

return 0

# Recursive case: reduce n by half and add 1 to the count

return 1 + count\_binary\_digits(n // 2)

**Explanation of the Algorithm:**

1. **Base Case:** If n == 0, the function returns 0 because the number 0 has no binary digits in this context (or equivalently, we can consider 0 to have 1 digit, but we'll assume the former here).
2. **Recursive Case:** For any other number n > 0, the function divides the number by 2 (n // 2), effectively removing the least significant binary digit (rightmost bit), and recursively counts the remaining binary digits, adding 1 for the current binary digit.

For example:

* To find the binary digits of 13:
  + The binary representation of 13 is 1101 (which has 4 binary digits).
  + The algorithm works as follows:
    - count\_binary\_digits(13) → 1 + count\_binary\_digits(6)
    - count\_binary\_digits(6) → 1 + count\_binary\_digits(3)
    - count\_binary\_digits(3) → 1 + count\_binary\_digits(1)
    - count\_binary\_digits(1) → 1 + count\_binary\_digits(0)
    - count\_binary\_digits(0) → 0 (base case)
  + The total count of binary digits is 1 + 1 + 1 + 1 = 4.

**Recurrence Relation:**

Let T(n)T(n)T(n) represent the number of binary digits in the binary representation of a positive integer nnn.

The recurrence relation can be written as:

T(n)=1+T(n2)forn>0T(n) = 1 + T\left(\frac{n}{2}\right) \quad \text{for} \quad n > 0T(n)=1+T(2n​)forn>0

Where:

* 111 accounts for the current binary digit (bit),
* T(n2)T\left(\frac{n}{2}\right)T(2n​) represents the recursive call on the integer obtained by dividing nnn by 2.

**Base Case:**

T(0)=0T(0) = 0T(0)=0

**Time Complexity:**

The number of recursive calls is determined by how many times we can divide nnn by 2 until nnn becomes 0. This is equivalent to finding the number of bits required to represent nnn in binary, which is the logarithm (base 2) of nnn.

Thus, the number of recursive calls is proportional to log⁡2n\log\_2 nlog2​n.

* **Time Complexity:** The time complexity is O(log⁡n)O(\log n)O(logn), since the algorithm reduces the problem size by half in each recursive step.

**Conclusion:**

* **Recursive Algorithm**: A recursive approach to find the number of binary digits in a decimal integer involves dividing the number by 2 and counting the number of divisions (recursive calls).
* **Recurrence Relation**:T(n)=1+T(n2),T(0)=0T(n) = 1 + T\left(\frac{n}{2}\right), \quad T(0) = 0T(n)=1+T(2n​),T(0)=0
* **Time Complexity**: The time complexity of this algorithm is O(log⁡n)O(\log n)O(logn), where nnn is the input number.

**UNIT II**

# GRAPH ALGORITHMS

1. **Discuss about the algorithm and pseudocode to find the Minimum Spanning Tree using Prim’s Algorithm. Find the Minimum Spanning Tree for the graph. Discuss about the efficiency of the algorithm.**

**Prim's Algorithm for Minimum Spanning Tree (MST)**

Prim's algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a connected, undirected graph. A Minimum Spanning Tree is a subset of the edges of the graph that connects all the vertices without any cycles and with the minimum possible total edge weight.

Prim's algorithm works by building the MST one vertex at a time, starting from an arbitrary vertex and gradually adding the smallest edge that connects a vertex in the MST to a vertex outside the MST.

**Steps of Prim's Algorithm:**

1. **Start from an arbitrary vertex:** Choose any vertex as the starting point.
2. **Edge selection:** Look at all edges that connect a vertex in the MST to a vertex outside the MST. Select the edge with the smallest weight.
3. **Add the edge to the MST:** Add the selected edge and its corresponding vertex to the MST.
4. **Repeat the process:** Repeat steps 2 and 3 until all vertices are included in the MST.

**Pseudocode for Prim's Algorithm:**

text

Copy

Prim(Graph G, start\_vertex):

Initialize min\_heap (priority queue) to keep track of edges with minimum weights

Initialize the `key` array to store the minimum edge weight for each vertex

Initialize the `parent` array to store the parent vertex of each vertex in the MST

Initialize a `visited` array to keep track of the vertices included in the MST

For each vertex v in G:

key[v] = ∞ (except for the start\_vertex, which is 0)

parent[v] = NIL

key[start\_vertex] = 0

Push the start\_vertex into the priority queue with priority 0

while the priority queue is not empty:

u = ExtractMin(min\_heap) // Get the vertex with the smallest key

visited[u] = true

For each neighbor v of u:

if v is not in MST and the weight of edge (u, v) is smaller than key[v]:

key[v] = weight(u, v)

parent[v] = u

Push v into the priority queue with updated key[v]

// The `parent` array contains the MST structure

// Return the MST formed from the `parent` and `key` arrays

**Explanation of the Pseudocode:**

1. **Initialization:**
   * min\_heap: A priority queue (min-heap) is used to always fetch the vertex with the minimum edge weight.
   * key: This array keeps track of the minimum weight edge that connects a vertex to the MST.
   * parent: This array stores the parent of each vertex, which will form the MST structure.
   * visited: This array keeps track of whether a vertex is included in the MST.
2. **Prim's Main Loop:**
   * The algorithm repeatedly extracts the vertex with the minimum edge weight from the priority queue.
   * For each unvisited neighbor of this vertex, the algorithm updates the key and parent arrays if the weight of the edge to the neighbor is smaller than the current key for that neighbor.
   * The neighbor is then pushed back into the priority queue with the updated key value.
3. **Termination:**
   * The algorithm terminates when all vertices have been added to the MST.

**Example of Prim’s Algorithm:**

Let's consider the following graph:

lua

Copy

(1)---(3)---(5)

| / |

(2) (4) (6)

| / |

(7)---(8)---(9)

Edges and their weights:

* (1, 2) = 4, (1, 3) = 1, (2, 4) = 3, (2, 7) = 5, (3, 4) = 6, (3, 5) = 2, (4, 7) = 7, (5, 6) = 8, (7, 8) = 9, (8, 9) = 10.

We start from vertex 1. Prim's algorithm will proceed as follows:

1. Start with vertex 1, select the minimum edge (1, 3) with weight 1.
2. Add vertex 3, now the edges available are (3, 5) with weight 2, (3, 4) with weight 6. Select (3, 5) with weight 2.
3. Add vertex 5, now the edges available are (5, 6) with weight 8. Select (5, 6) with weight 8.
4. Add vertex 6, now the edges available are (6, 9) with weight 10. Select (6, 9) with weight 10.
5. Add vertex 9, now the edges available are (9, 8) with weight 9. Select (9, 8) with weight 9.
6. Add vertex 8, now the edges available are (8, 7) with weight 5. Select (8, 7) with weight 5.
7. Add vertex 7, now the edges available are (7, 4) with weight 7. Select (7, 4) with weight 7.
8. Add vertex 4, the MST is complete.

**MST Result:**

The MST contains the edges:  
(1, 3), (3, 5), (5, 6), (6, 9), (9, 8), (8, 7), (7, 4).

**Efficiency of Prim’s Algorithm:**

The time complexity of Prim’s algorithm depends on the data structures used to implement it:

1. **Using a simple array or list for the priority queue:**
   * **Time Complexity:** O(V2)O(V^2)O(V2), where VVV is the number of vertices in the graph. This is because for each vertex, we might have to scan through all other vertices to find the minimum weight edge.
2. **Using a binary heap (priority queue):**
   * **Time Complexity:** O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV), where EEE is the number of edges. This is more efficient than the simple array approach because it allows us to extract the minimum key and update the key values in logarithmic time.
3. **Using a Fibonacci heap:**
   * **Time Complexity:** O(E+Vlog⁡V)O(E + V \log V)O(E+VlogV). This is the most efficient version in theory, but the constant factors can make it slower in practice compared to the binary heap implementation.
4. **Distinguish between breadth first search and depth first search with example**

**Breadth-First Search (BFS) vs Depth-First Search (DFS)**

Breadth-First Search (BFS) and Depth-First Search (DFS) are two fundamental graph traversal algorithms used to explore or search through graphs. They differ in the approach they take to visit the vertices of the graph.

Here is a detailed comparison between BFS and DFS:

**1. BFS (Breadth-First Search):**

**Definition:**

* BFS is an algorithm for traversing or searching a graph or tree in a breadthward motion.
* It explores all the vertices at the present depth level before moving on to the vertices at the next depth level.

**Working Principle:**

* BFS starts at the root (or an arbitrary node in the case of a graph) and explores all the neighbors at the present depth level before moving on to nodes at the next depth level.
* It uses a **queue** to keep track of the vertices to be explored.

**Steps:**

1. Start with a chosen node, mark it as visited.
2. Add the starting node to a queue.
3. While the queue is not empty, dequeue a node and explore all its unvisited neighbors.
4. For each unvisited neighbor, mark it as visited and enqueue it.
5. Repeat until all nodes are visited.

**Data Structure Used:**

* **Queue** (FIFO: First In, First Out)

**Time Complexity:**

* **O(V + E)** where VVV is the number of vertices and EEE is the number of edges in the graph.

**Space Complexity:**

* **O(V)**, as we store the visited nodes and the queue.

**Example:** Consider the following graph:

mathematica

Copy

A

/ \

B C

/ \

D E

BFS starting from node A:

1. Start with A, mark A as visited, and enqueue A.
2. Dequeue A, visit its neighbors B and C, mark them as visited, and enqueue them.
3. Dequeue B, visit its neighbor D, mark it as visited, and enqueue it.
4. Dequeue C, visit its neighbor E, mark it as visited, and enqueue it.
5. Dequeue D and E, but they have no unvisited neighbors.

**BFS traversal result:**

* A → B → C → D → E

**2. DFS (Depth-First Search):**

**Definition:**

* DFS is an algorithm for traversing or searching a graph or tree in a depthward motion.
* It explores as far as possible along each branch before backtracking.

**Working Principle:**

* DFS starts at the root (or an arbitrary node in the case of a graph) and explores each branch completely before moving to the next branch.
* It uses a **stack** (or recursion) to keep track of the vertices to be explored.

**Steps:**

1. Start with a chosen node, mark it as visited.
2. Visit an unvisited neighbor of the current node and push it to the stack.
3. Repeat the process for the current node's neighbor until no unvisited neighbors are left.
4. Backtrack (pop the stack) to the previous node and explore the next unvisited neighbor.
5. Continue until all nodes are visited.

**Data Structure Used:**

* **Stack** (LIFO: Last In, First Out) or **Recursion**

**Time Complexity:**

* **O(V + E)** where VVV is the number of vertices and EEE is the number of edges in the graph.

**Space Complexity:**

* **O(V)**, as we store the visited nodes and the stack.

**Example:** Consider the same graph as before:

mathematica

Copy

A

/ \

B C

/ \

D E

DFS starting from node A:

1. Start with A, mark A as visited, and push it onto the stack.
2. Visit neighbor B, mark B as visited, and push it onto the stack.
3. Visit neighbor D, mark D as visited, and push it onto the stack.
4. Since D has no unvisited neighbors, backtrack to B.
5. Since B has no unvisited neighbors, backtrack to A.
6. Visit neighbor C, mark C as visited, and push it onto the stack.
7. Visit neighbor E, mark E as visited, and push it onto the stack.
8. Since E has no unvisited neighbors, backtrack to C.

**DFS traversal result:**

* A → B → D → C → E

**Key Differences Between BFS and DFS:**

| **Criteria** | **Breadth-First Search (BFS)** | **Depth-First Search (DFS)** |
| --- | --- | --- |
| **Traversal Strategy** | Explores level by level, visiting all neighbors at the current level before moving deeper. | Explores as deep as possible along one branch before backtracking. |
| **Data Structure** | Queue (FIFO) | Stack (LIFO) or Recursion |
| **Time Complexity** | O(V + E) | O(V + E) |
| **Space Complexity** | O(V) (due to the queue) | O(V) (due to the stack/recursion) |
| **Complete Search** | Guarantees finding the shortest path in an unweighted graph. | Does not guarantee the shortest path. |
| **Use Case** | Suitable for finding the shortest path, level order traversal, and search in unweighted graphs. | Suitable for tasks like topological sorting, solving puzzles, and exploring deep paths in the graph. |
| **Implementation** | Can be implemented using a queue and iterative method. | Can be implemented using recursion or stack (either explicit or via function call stack). |

**Example with Comparison:**

**Example with Comparison:**

Given the graph:

mathematica

Copy

A

/ \

B C

/ \

D E

* **BFS traversal:** A → B → C → D → E
* **DFS traversal:** A → B → D → C → E

**UNIT III**

**ALGORITHM DESIGN TECHNIQUES**

**1. (i) Algorithm to Construct the Optimal Binary Search Tree (8 Marks)**

An **Optimal Binary Search Tree (OBST)** is a binary search tree that provides the least search cost, given the frequency of searches for each key. The idea is to choose the root in such a way that the total cost of searching all keys is minimized.

**Problem Statement:** Given a sorted array of keys K1 < K2 < ... < Kn, and an array freq[1..n] where freq[i] is the frequency of searches for key Ki, construct a binary search tree with minimal search cost.

**Algorithm Using Dynamic Programming:**

1. Let cost[i][j] be the cost of the optimal BST that can be built from keys Ki to Kj.
2. Initialize cost[i][i] = freq[i] for all i.
3. For lengths from 2 to n:
   * For i = 1 to n - len + 1:
     + j = i + len - 1
     + cost[i][j] = minimum of (cost[i][r-1] + cost[r+1][j] + sum(freq[i..j])) for all r from i to j
4. sum(freq[i..j]) is the sum of frequencies from i to j.
5. Return cost[1][n] as the minimum cost of the optimal BST.

**Pseudocode:**

function optimalBST(freq[], n):

cost[n+1][n+1]

for i from 1 to n:

cost[i][i] = freq[i]

for L = 2 to n:

for i = 1 to n - L + 1:

j = i + L - 1

cost[i][j] = INF

sum = sum of freq[i..j]

for r = i to j:

c = cost[i][r-1] + cost[r+1][j] + sum

if c < cost[i][j]:

cost[i][j] = c

return cost[1][n]

**Time Complexity:** O(n^3) **Space Complexity:** O(n^2)

**1. (ii) Application of Dynamic Programming to Solve Travelling Salesperson Problem (8 Marks)**

**Problem Statement:** Given a set of cities and the cost of travel between each pair of cities, the Travelling Salesperson Problem (TSP) aims to find the shortest possible route that visits every city once and returns to the origin city.

**Dynamic Programming Approach:** Let there be n cities numbered from 0 to n-1. Use bitmasking to represent subsets of visited cities.

**Define:**

* Let dp[mask][i] be the minimum cost to reach city i with mask representing the set of visited cities.
* mask is a binary number of n bits where 1 means the city is visited.

**Steps:**

1. Initialize dp[2^n][n] to INF.
2. Set dp[1 << i][i] = cost[0][i] for all cities i.
3. For all masks and for all cities j in that mask:
   * Try all cities k not in the mask and update dp[mask | (1 << k)][k] = min(dp[mask | (1 << k)][k], dp[mask][j] + cost[j][k])
4. Final answer: min(dp[2^n - 1][i] + cost[i][0]) for all i

**Pseudocode:**

function tsp(cost[][], n):

dp[2^n][n]

for mask in range(2^n):

for i in range(n):

dp[mask][i] = INF

dp[1][0] = 0

for mask in range(2^n):

for u in range(n):

if mask & (1 << u):

for v in range(n):

if not (mask & (1 << v)):

dp[mask | (1 << v)][v] = min(dp[mask | (1 << v)][v], dp[mask][u] + cost[u][v])

return min(dp[2^n - 1][i] + cost[i][0] for i in range(n))

**Time Complexity:** O(n^2 \* 2^n) **Space Complexity:** O(n \* 2^n)

**Conclusion:**

* The OBST algorithm helps in creating efficient binary search trees with minimum search cost using dynamic programming.
* The TSP problem is solved using bitmasking and dynamic programming by storing the subproblem solutions to avoid redundant calculations.

**2. Quick Sort Algorithm, Explanation, and Time Complexity Analysis**

**Quick Sort Algorithm:**

void quicksort(int arr[], int low, int high) {

if (low < high) {

int pivotIndex = partition(arr, low, high);

quicksort(arr, low, pivotIndex - 1);

quicksort(arr, pivotIndex + 1, high);

}

}

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j] <= pivot) {

i++;

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return (i + 1);

}

**Explanation with Example:**

Let us sort the array: arr[] = {10, 7, 8, 9, 1, 5}

1. Choose the last element as the pivot (5).
2. Partition the array:
   * After rearranging: {1, 5, 8, 9, 7, 10} with pivot (5) at index 1
3. Recursively apply quicksort to left sub-array {1} and right sub-array {8, 9, 7, 10}
4. Continue recursively until each sub-array is sorted.

**Final Sorted Array:** {1, 5, 7, 8, 9, 10}

**Time Complexity Derivation:**

Let T(n) be the time taken to sort an array of n elements.

1. **Worst Case (O(n^2))**:
   * Occurs when the pivot is always the smallest or largest element.
   * This leads to unbalanced partitions: T(n) = T(n-1) + O(n)
   * => T(n) = O(n^2)
2. **Best Case (O(n log n))**:
   * Pivot divides the array into two equal halves.
   * T(n) = 2T(n/2) + O(n) => T(n) = O(n log n)
3. **Average Case (O(n log n))**:
   * Assuming all positions of pivot are equally likely.
   * Recurrence: T(n) = T(k) + T(n-k-1) + O(n) => T(n) = O(n log n)

**Summary Table:**

| **Case** | **Time Complexity** |
| --- | --- |
| Best Case | O(n log n) |
| Average Case | O(n log n) |
| Worst Case | O(n^2) |

**Space Complexity:** O(log n) due to recursive stack space (for in-place sorting)

**Advantages of Quick Sort:**

* Faster in practice for large datasets
* In-place sorting (requires no extra memory)
* Divide and conquer strategy

**Disadvantages:**

* Performance degrades in worst case
* Not stable (does not maintain the relative order of equal elements)

**UNIT IV STATE SPACE SEARCH ALGORITHMS**

**1. (i) Hamiltonian Cycles (NOV/DEC 2012) (8 Marks)**

**Definition:** A Hamiltonian cycle (or Hamiltonian circuit) in a graph is a cycle that visits each vertex exactly once and returns to the starting vertex. The graph containing such a cycle is called a Hamiltonian graph.

**Properties:**

1. A Hamiltonian cycle must visit every vertex of the graph exactly once.
2. The cycle should return to the starting vertex.
3. Unlike Eulerian circuits (which focus on edges), Hamiltonian cycles focus on vertices.

**Sufficient Conditions (Dirac's and Ore’s Theorem):**

* **Dirac’s Theorem:** If a graph G with n vertices (n ≥ 3) has the degree of each vertex ≥ n/2, then G is Hamiltonian.
* **Ore’s Theorem:** If for every pair of non-adjacent vertices u and v, deg(u) + deg(v) ≥ n, then G is Hamiltonian.

**Example:** Consider the graph G with vertices A, B, C, D, E where the edges are: (A, B), (B, C), (C, D), (D, E), (E, A), (A, C), (B, D)

Possible Hamiltonian Cycle: A → B → C → D → E → A

**Applications:**

1. Traveling Salesman Problem (TSP)
2. Routing and Network topology
3. Scheduling problems

**1. (ii) Graph Coloring Algorithm (8 Marks)**

**Definition:** Graph coloring is the method of assigning colors to the vertices of a graph such that no two adjacent vertices share the same color. The minimum number of colors required to color a graph is known as its chromatic number.

**Types of Graph Coloring:**

1. **Vertex Coloring** – Coloring the vertices such that adjacent vertices have different colors.
2. **Edge Coloring** – Coloring the edges so that no two adjacent edges share the same color.
3. **Face Coloring** – Coloring faces of a planar graph so that adjacent faces have different colors.

**Graph Coloring Algorithm:** The Greedy Coloring Algorithm is commonly used:

**Steps:**

1. Arrange all the vertices in some order.
2. Assign the first color to the first vertex.
3. For each remaining vertex:
   * Check the colors assigned to its adjacent vertices.
   * Assign the smallest color that has not been used by its adjacent vertices.

**Example:** Given Graph G with vertices: A, B, C, D Edges: (A, B), (A, C), (B, C), (C, D)

Apply Greedy Coloring:

* Color A with Color 1
* B is adjacent to A → Color 2
* C is adjacent to A and B → Color 3
* D is adjacent to C → Color 1

**Coloring Result:** A - 1, B - 2, C - 3, D - 1

**Applications:**

1. Register Allocation in compilers
2. Scheduling of exams and tasks
3. Frequency assignment in mobile networks
4. Map coloring

**Conclusion:** Hamiltonian cycles and graph coloring are crucial topics in graph theory, with numerous practical applications. Hamiltonian cycles help in routing and pathfinding, while graph coloring provides optimal solutions in scheduling and resource allocation problems.

**2.Knapsack Problem using Backtracking – 16 Marks Answer**

**Introduction:** The Knapsack Problem is a combinatorial optimization problem. The goal is to select a subset of items such that the total profit is maximized without exceeding the weight capacity of the knapsack. The 0/1 Knapsack Problem considers each item only once, either included (1) or excluded (0).

**Problem Definition:** Given:

* n items, each with a weight w[i] and profit p[i]
* A knapsack of capacity W Objective:
* Maximize total profit such that total weight ≤ W

**Backtracking Approach:** Backtracking is a systematic way to iterate through all the possible configurations of a search space. In the case of the knapsack problem, we use backtracking to explore all possible subsets of items.

**Steps Involved:**

1. Start with an empty knapsack.
2. At each level of recursion, consider two choices for each item:
   * Include the item in the knapsack.
   * Exclude the item from the knapsack.
3. Keep track of current profit and weight.
4. If at any point the weight exceeds the capacity, backtrack.
5. If a better profit is found under the capacity limit, update the best solution.

**Backtracking Algorithm:**

void knapsack(int i, int profit, int weight) {

if (weight <= W && profit > maxProfit) {

maxProfit = profit;

// Store current solution as best solution

}

if (promising(i, weight, profit)) {

// Include item i+1

knapsack(i + 1, profit + p[i+1], weight + w[i+1]);

// Exclude item i+1

knapsack(i + 1, profit, weight);

}

}

bool promising(int i, int weight, int profit) {

if (weight >= W) return false;

int j = i + 1;

int bound = profit;

int totalWeight = weight;

// Calculate upper bound

while (j < n && totalWeight + w[j] <= W) {

totalWeight += w[j];

bound += p[j];

j++;

}

if (j < n) {

bound += (W - totalWeight) \* p[j] / w[j];

}

return bound > maxProfit;

}

**Example:** Let:

* n = 4, W = 16
* Profits = {40, 30, 50, 10}
* Weights = {2, 5, 10, 5}

Using backtracking, the optimal subset is items {1, 3} with profit 90 and weight 12.

**Advantages of Backtracking:**

* Efficient for small datasets.
* Avoids brute force search.

**Disadvantages:**

* Not efficient for large datasets.
* Exponential time complexity.

**Conclusion:** The backtracking approach provides an elegant way to solve the 0/1 knapsack problem using recursive tree traversal. While not optimal for very large instances, it’s useful for educational and small-sized problems.

**UNIT V NP-COMPLETE AND APPROXIMATION ALGORITHM**

**16-Marks Question and Answer**

**1.**Elaborate on the Nearest-Neighbor Algorithm and Multifragments Heuristic Algorithm for the Travelling Salesman Problem (TSP).

**Answer:**

The Travelling Salesman Problem (TSP) is a classic optimization problem where a salesman must travel through a given list of cities, visiting each exactly once and returning to the origin city, with the objective of minimizing the total distance traveled.

Two prominent heuristic approaches used to solve TSP are:

**1. Nearest-Neighbor Algorithm (NNA):**

The Nearest-Neighbor Algorithm is a greedy algorithm that builds a path by selecting the nearest unvisited city at each step.

**Steps Involved:**

1. Select a starting city.
2. From the current city, find the nearest unvisited city.
3. Move to that city and mark it as visited.
4. Repeat step 2 until all cities have been visited.
5. Return to the starting city to complete the cycle.

**Advantages:**

* Simple and easy to implement.
* Fast and efficient for small datasets.

**Disadvantages:**

* Doesn’t always provide the optimal solution.
* Can produce suboptimal paths due to the greedy nature.
* Highly dependent on the starting point.

**Example:** If the cities are A, B, C, D with distances:

* A to B: 10
* A to C: 15
* A to D: 20
* B to C: 35
* B to D: 25
* C to D: 30

Starting from A:

* A to B (10)
* B to D (25)
* D to C (30)
* C to A (15) Total distance = 10 + 25 + 30 + 15 = **80 units**

**2. Multifragment Heuristic Algorithm (MFA):**

The Multifragment Algorithm is another heuristic that constructs a tour by selecting the shortest available edge that does not form a cycle prematurely or increase the degree of any vertex beyond two.

**Steps Involved:**

1. List all edges and sort them in increasing order of distance.
2. Select the smallest edge that:
   * Does not form a cycle unless all nodes are included.
   * Does not increase the degree of any node to more than two.
3. Repeat until a tour involving all cities is formed.

**Advantages:**

* Often provides better solutions than NNA.
* Considers the global structure rather than local minimum.

**Disadvantages:**

* More complex than NNA.
* Slower for very large datasets.

**Example:** Given cities A, B, C, D with distances:

* AB: 10, AC: 15, AD: 20, BC: 35, BD: 25, CD: 30

Sort edges: AB (10), AC (15), AD (20), BD (25), CD (30), BC (35) Select edges avoiding cycles and vertex degrees > 2:

* AB (10), AC (15), CD (30), DB (25) ← form a valid tour

Total distance = 10 + 15 + 30 + 25 = **80 units**

**Comparison Between NNA and MFA:**

| **Criteria** | **Nearest-Neighbor Algorithm** | **Multifragments Algorithm** |
| --- | --- | --- |
| Type | Greedy | Greedy Heuristic |
| Complexity | O(n^2) | O(n log n) |
| Tour Quality | May be suboptimal | Often better |
| Implementation | Simple | Moderate Complexity |
| Dependence on Start | High | Low |

**Conclusion:**

Both algorithms serve as efficient heuristics for TSP where exact solutions are computationally expensive. NNA is suitable for quick solutions with small datasets, while MFA provides more optimized results for larger datasets with slightly higher complexity. For practical purposes, these algorithms are often used in real-time applications like routing, logistics, and network design.

**Note:** For optimal results in real-world TSP, hybrid approaches or advanced heuristics such as Genetic Algorithms, Simulated Annealing, and Ant Colony Optimization may also be considered.

**2: Suggest an Approximation Algorithm for the Travelling Salesperson Problem (TSP) Assuming Triangle Inequality.**

**Answer:**

The Travelling Salesperson Problem (TSP) is a well-known NP-Hard problem in which a salesperson needs to visit a set of cities exactly once and return to the starting city, minimizing the total travel cost or distance. When the cost function (distance between cities) satisfies the **triangle inequality**, i.e., for any three cities A, B, and C:

d(A,C)≤d(A,B)+d(B,C)d(A, C) \leq d(A, B) + d(B, C)

an efficient approximation algorithm can be used to find a near-optimal solution. One of the most popular such algorithms is the **Christofides' Algorithm**.

**Christofides' Algorithm**

**Objective:** To produce a tour whose total cost is at most 1.5 times the cost of the optimal tour for the TSP with triangle inequality.

**Steps of the Algorithm:**

1. **Minimum Spanning Tree (MST):**
   * Construct a Minimum Spanning Tree (MST) of the graph using Prim's or Kruskal's algorithm.
   * Let the cost of MST be **C(MST)**.
2. **Find Odd Degree Vertices:**
   * Identify all vertices in the MST that have an odd degree. Let the set of these vertices be **O**.
3. **Minimum Weight Perfect Matching:**
   * On the subgraph induced by **O**, compute a **Minimum Weight Perfect Matching**.
   * Add the edges of this matching to the MST. This makes the resulting graph **Eulerian** (each vertex has even degree).
4. **Eulerian Tour:**
   * Find an **Eulerian tour** in the graph. This can be done using Fleury’s or Hierholzer’s algorithm.
5. **Hamiltonian Circuit (Shortcutting):**
   * Convert the Eulerian tour into a **Hamiltonian circuit** by skipping already visited vertices. Due to the triangle inequality, the cost will not increase.

**Time Complexity:**

* Minimum Spanning Tree: O(Elog⁡V)O(E \log V)
* Perfect Matching: O(V3)O(V^3)
* Eulerian Tour and Shortcutting: O(V)O(V)

Hence, the total time complexity is **polynomial**.

**Performance Guarantee:**

* The algorithm guarantees that the total cost of the tour is **at most 1.5 times** the optimal cost: Cost of tour≤1.5×OPT\text{Cost of tour} \leq 1.5 \times \text{OPT}

This is the **best possible approximation ratio** for the general metric TSP known so far.

**Example:**

Suppose we have 4 cities with the following distances:

|  | **A** | **B** | **C** | **D** |
| --- | --- | --- | --- | --- |
| A | 0 | 2 | 9 | 10 |
| B | 2 | 0 | 6 | 4 |
| C | 9 | 6 | 0 | 8 |
| D | 10 | 4 | 8 | 0 |

1. **MST**: A-B, B-D, B-C (cost = 2 + 4 + 6 = 12)
2. **Odd Degree Vertices**: A, C, D
3. **Perfect Matching on Odd Vertices**: Match A-D (cost = 10)
4. Add A-D to MST: Graph becomes Eulerian
5. Euler tour: A-B-D-A-C-B
6. Shortcut to Hamiltonian cycle: A-B-D-C-A (cost = 2 + 4 + 8 + 9 = 23)

**Conclusion:**

Christofides' algorithm provides an efficient and practical solution for the TSP when the cost function satisfies the triangle inequality. It is easy to implement and guarantees a solution close to optimal, making it a suitable choice for many real-world applications.

**Note:** This algorithm is only applicable for **metric TSP** (i.e., satisfying the triangle inequality). For general TSP, no constant-factor approximation algorithm is known.