**CS 3401- ALGORITHMS**

**UNIT I INTRODUCTION**

**1.Define the asymptotic notations used for best case average case and worst case analysis?**

Asymptotic notations are mathematical tools used to describe the behavior of algorithms in terms of time and space complexity, especially as the input size (denoted as **n**) becomes large. These notations are essential for analyzing the efficiency of algorithms in various scenarios, such as the best case, average case, and worst case. Below are the key asymptotic notations used in this context:

**1. Best Case (Ω-notation):**

* The **Ω (Big-Omega)** notation is used to describe the lower bound of an algorithm's performance. It indicates the minimum time (or space) that an algorithm will take on an input of size **n**, regardless of how the input is structured. The best case occurs when the algorithm performs at its most efficient (quickest or least resource-consuming).
* Mathematically, for a given algorithm, **Ω(f(n))** means that the algorithm's running time is at least **f(n)** for sufficiently large **n**.
* Example: If an algorithm searches for a specific item in a list and the item is at the beginning, the best case performance might be **Ω(1)** (constant time), since the search completes immediately.

**Best Case Example**:

* Linear search in an unsorted array: If the element is found at the first position, the best case time complexity is **Ω(1)** (constant time).

**2. Average Case (Θ-notation):**

* The **Θ (Big-Theta)** notation represents the tight bound of the algorithm's performance. It is used to describe the expected performance of the algorithm over all possible inputs, assuming a uniform distribution of inputs or a specific probability distribution.
* Mathematically, for an algorithm, **Θ(f(n))** indicates that the algorithm’s running time (or space complexity) is both upper and lower bounded by **f(n)** for large **n**, i.e., the performance of the algorithm is typically on the order of **f(n)** in most cases.
* This notation is useful when considering an "average" input set and is often used when analyzing algorithms like sorting, where the input is neither best nor worst but represents typical cases.

**Average Case Example**:

* QuickSort algorithm: If the pivot divides the array roughly into two equal parts, the average case time complexity is **Θ(n log n)**.

**3. Worst Case (O-notation):**

* The **O (Big-O)** notation is the most commonly used and represents the upper bound of an algorithm’s performance. It describes the maximum amount of time (or space) the algorithm could possibly take, considering the worst possible input.
* Mathematically, **O(f(n))** means that the algorithm’s running time will not exceed **f(n)** for sufficiently large **n**. It provides an upper limit on the performance, ensuring that the algorithm will never take longer than this.
* The worst case is often used as the basis for comparing algorithms, as it gives a guarantee that the algorithm will not exceed the specified time (or space) under any circumstance.

**Worst Case Example**:

* QuickSort algorithm: In the worst case, when the pivot is always the smallest or largest element, the time complexity becomes **O(n²)** due to poor partitioning.

**2.Prove that if g(n) is Ω(f(n)) then f(n) is O(g(n)). (5) (NOV/DEC 2018)**

**(ii) Discuss vrious methods used for mathematical analysis of recursive algorithms.(8)**

**(i) Proof that if g(n)g(n)g(n) is Ω(f(n)f(n)f(n)), then f(n)f(n)f(n) is O(g(n)g(n)g(n))**

**To prove:** If g(n)g(n)g(n) is Ω(f(n)f(n)f(n)), then f(n)f(n)f(n) is O(g(n)g(n)g(n)).

**Definitions:**

1. **Ω-notation**:  
   We say g(n)g(n)g(n) is Ω(f(n)f(n)f(n)) if there exist positive constants ccc and n0n\_0n0​ such that for all n≥n0n \geq n\_0n≥n0​,

g(n)≥c⋅f(n)g(n) \geq c \cdot f(n)g(n)≥c⋅f(n)

This means that g(n)g(n)g(n) grows at least as fast as f(n)f(n)f(n) for sufficiently large nnn.

1. **O-notation**:  
   We say f(n)f(n)f(n) is O(g(n)g(n)g(n)) if there exist positive constants c′c'c′ and n0′n'\_0n0′​ such that for all n≥n0′n \geq n'\_0n≥n0′​,

f(n)≤c′⋅g(n)f(n) \leq c' \cdot g(n)f(n)≤c′⋅g(n)

This means that f(n)f(n)f(n) grows at most as fast as g(n)g(n)g(n) for sufficiently large nnn.

**Proof:**

Assume that g(n)g(n)g(n) is Ω(f(n)f(n)f(n)), which means there exist constants c>0c > 0c>0 and n0n\_0n0​ such that for all n≥n0n \geq n\_0n≥n0​,

g(n)≥c⋅f(n)g(n) \geq c \cdot f(n)g(n)≥c⋅f(n)

Now, we need to prove that f(n)f(n)f(n) is O(g(n)g(n)g(n)).

By the definition of O-notation, we need to show that there exist constants c′>0c' > 0c′>0 and n0′n'\_0n0′​ such that for all n≥n0′n \geq n'\_0n≥n0′​,

f(n)≤c′⋅g(n)f(n) \leq c' \cdot g(n)f(n)≤c′⋅g(n)

From the earlier assumption, we know that

g(n)≥c⋅f(n)for alln≥n0g(n) \geq c \cdot f(n) \quad \text{for all} \quad n \geq n\_0g(n)≥c⋅f(n)for alln≥n0​

By rearranging this inequality, we get:

f(n)≤1c⋅g(n)f(n) \leq \frac{1}{c} \cdot g(n)f(n)≤c1​⋅g(n)

This is exactly in the form required for f(n)f(n)f(n) to be O(g(n)g(n)g(n)), with c′=1cc' = \frac{1}{c}c′=c1​ and n0′=n0n'\_0 = n\_0n0′​=n0​.

Thus, we have shown that if g(n)g(n)g(n) is Ω(f(n)f(n)f(n)), then f(n)f(n)f(n) is O(g(n)g(n)g(n)).

**(ii) Methods Used for Mathematical Analysis of Recursive Algorithms**

The analysis of recursive algorithms can be challenging because their time complexity depends not only on the work done at the current step but also on the work done in recursive calls. There are several common methods used for analyzing recursive algorithms:

**1. Recurrence Relations**

* **Definition**: A recurrence relation is an equation or inequality that describes a function in terms of its value at smaller inputs.
* **Use**: Recursive algorithms can often be represented by a recurrence relation that expresses the time complexity of the algorithm in terms of the size of the input and the time complexity of recursive subproblems.
* **Example**: For a divide-and-conquer algorithm like Merge Sort, the recurrence relation is: T(n)=2T(n/2)+O(n)T(n) = 2T(n/2) + O(n)T(n)=2T(n/2)+O(n)
* **Solving the recurrence** can give the time complexity of the algorithm.

**2. The Substitution Method**

* **Definition**: The substitution method involves guessing the solution to the recurrence relation and then proving it by induction.
* **Steps**:
  1. Make an educated guess for the solution.
  2. Use induction to prove that the guess is correct.
* **Example**: For Merge Sort, you might guess T(n)=O(nlog⁡n)T(n) = O(n \log n)T(n)=O(nlogn), and then prove it by induction.

**3. The Recursion Tree Method**

* **Definition**: The recursion tree method visualizes the recursive calls and the work done at each level of recursion.
* **Use**: A tree is drawn where each node represents a recursive call, and the edges represent the recursive calls themselves. The work done at each level is added together, and the total work is calculated by summing the work at all levels.
* **Example**: For Merge Sort, you would draw a tree with nnn at the root, dividing into two subproblems of size n/2n/2n/2, and continue recursively until the subproblem size reaches 1. You then compute the total work by summing the work at each level.
* **Work Calculation**: If at each level the total work is proportional to nnn, and the depth of the tree is log⁡n\log nlogn, the total time complexity would be O(nlog⁡n)O(n \log n)O(nlogn).

**4. The Master Theorem**

* **Definition**: The Master Theorem provides a direct way to solve recurrences of the form: T(n)=aT(n/b)+O(nd)T(n) = aT(n/b) + O(n^d)T(n)=aT(n/b)+O(nd) where a≥1a \geq 1a≥1, b>1b > 1b>1, and d≥0d \geq 0d≥0.
* **Use**: It allows you to find the time complexity of divide-and-conquer recurrences without needing to manually solve them.
* **Cases**:
  + **Case 1**: If a>bda > b^da>bd, the time complexity is O(nlog⁡ba)O(n^{\log\_b a})O(nlogb​a).
  + **Case 2**: If a=bda = b^da=bd, the time complexity is O(ndlog⁡n)O(n^d \log n)O(ndlogn).
  + **Case 3**: If a<bda < b^da<bd, the time complexity is O(nd)O(n^d)O(nd).
* **Example**: For Merge Sort with T(n)=2T(n/2)+O(n)T(n) = 2T(n/2) + O(n)T(n)=2T(n/2)+O(n), using the Master Theorem, we get T(n)=O(nlog⁡n)T(n) = O(n \log n)T(n)=O(nlogn).

**5. Iteration Method**

* **Definition**: This method involves "unfolding" the recurrence by expanding it iteratively to express it in terms of simpler problems.
* **Steps**: Expand the recurrence multiple times until a pattern emerges, then sum the work done at each level of recursion.
* **Example**: For the recurrence T(n)=2T(n/2)+O(n)T(n) = 2T(n/2) + O(n)T(n)=2T(n/2)+O(n), by unfolding it we get: T(n)=2T(n/2)+O(n)T(n) = 2T(n/2) + O(n)T(n)=2T(n/2)+O(n) T(n/2)=2T(n/4)+O(n/2)T(n/2) = 2T(n/4) + O(n/2)T(n/2)=2T(n/4)+O(n/2) and so on, leading to a summation that gives the overall complexity O(nlog⁡n)O(n \log n)O(nlogn).

1. **Explain in detail about Rabin-Karp algorithm.**

**Rabin-Karp Algorithm**

The **Rabin-Karp algorithm** is a string searching (or string matching) algorithm that uses hashing to find patterns within a text efficiently. It was developed by Michael O. Rabin and Richard M. Karp in 1987. The algorithm is particularly useful when we have multiple patterns to search for in a given text, as it can handle multiple patterns in a single scan of the text.

**Overview:**

The key idea behind the Rabin-Karp algorithm is to use a *hash function* to represent the pattern and substrings of the text. The algorithm calculates the hash value of the pattern and the hash values of all substrings of the text with the same length as the pattern. It then compares these hash values to identify matching substrings.

**Steps of the Rabin-Karp Algorithm:**

1. **Precompute the Hash of the Pattern:**
   * Choose a hash function to calculate the hash value of the pattern. A common hash function used is the **rolling hash** function, which helps efficiently update hash values as we slide the window over the text.
   * The hash value for a string SSS of length mmm is typically calculated as: H(S)=(S0⋅pm−1+S1⋅pm−2+⋯+Sm−1⋅p0)mod  qH(S) = (S\_0 \cdot p^{m-1} + S\_1 \cdot p^{m-2} + \dots + S\_{m-1} \cdot p^0) \mod qH(S)=(S0​⋅pm−1+S1​⋅pm−2+⋯+Sm−1​⋅p0)modq where:
     + ppp is a constant (a small prime number),
     + qqq is a large prime number used to reduce hash values and avoid collisions,
     + S0,S1,…,Sm−1S\_0, S\_1, \dots, S\_{m-1}S0​,S1​,…,Sm−1​ are the characters in the string.
2. **Calculate Hash of Substrings of the Text:**
   * Slide a window of the same length as the pattern over the text.
   * For each window, calculate the hash value of the substring. This is done using a rolling hash to update the hash efficiently rather than recalculating the hash from scratch for every substring.
3. **Compare Hash Values:**
   * Once the hash value of a substring is calculated, compare it with the hash of the pattern.
   * If the hash values match, there is a potential match, but you need to confirm by checking the actual substring in the text.
   * If the hash values do not match, continue moving the window and calculating the next hash.
4. **Final Verification (if necessary):**
   * If two hash values match, a final character-by-character comparison of the substring and the pattern is required to handle potential hash collisions.

**Time Complexity:**

* The average time complexity of the Rabin-Karp algorithm is **O(n + m)**, where:
  + nnn is the length of the text,
  + mmm is the length of the pattern.
* In the worst case, when there are many hash collisions, the time complexity can degrade to **O(nm)**, where you need to check each character of the substring against the pattern.

Thus, in practice, Rabin-Karp is very efficient when searching for a single pattern in a large text, but its performance can deteriorate if hash collisions are frequent.

1. **Explain in detail about Knuth-Morris-Pratt algorithm**

The **Knuth-Morris-Pratt (KMP) algorithm** is another efficient string matching algorithm that improves upon the naive approach by eliminating unnecessary re-checking of characters. It was developed by Donald Knuth, Vaughan Pratt, and James Morris in 1977.

**Overview:**

The KMP algorithm improves string matching by utilizing previously gathered information about the pattern itself. Instead of starting over with each mismatch, it uses a *partial match table* (also known as the **prefix function** or **failure function**) to skip over unnecessary comparisons.

**Key Concepts of KMP:**

1. **Prefix Function:**
   * The prefix function for a pattern is an array where each element π[i]\pi[i]π[i] represents the length of the longest proper prefix of the substring pattern[0…i]pattern[0 \dots i]pattern[0…i] that is also a suffix of the same substring. A *proper prefix* is one that is not equal to the entire string.
   * For example, if the pattern is "ABAB", the prefix function array would be: π=[0,0,1,2]\pi = [0, 0, 1, 2]π=[0,0,1,2] This means:
     + For the substring "A", there is no proper prefix that is also a suffix.
     + For the substring "AB", there is no proper prefix that is a suffix.
     + For "ABA", the proper prefix "A" is also a suffix.
     + For "ABAB", the proper prefix "AB" is also a suffix.
2. **Using the Prefix Function:**
   * The prefix function helps skip re-checking of characters by telling us how much we can shift the pattern when a mismatch occurs. Specifically, if a mismatch happens at position jjj in the pattern, we can shift the pattern to align the longest proper prefix (as indicated by the prefix function) with the substring of the text.
3. **Search Algorithm:**
   * First, the prefix function for the pattern is computed.
   * Then, we scan the text and the pattern simultaneously. If a mismatch occurs, we use the prefix function to determine how much to shift the pattern. If a match is found, the algorithm proceeds to the next position in the text.
   * This avoids unnecessary comparisons by using the partial information already gathered.

**Steps of the KMP Algorithm:**

1. **Preprocess the Pattern:**
   * Compute the prefix function for the pattern, which gives us a table indicating the length of the longest proper prefix of any substring of the pattern that matches its suffix.
2. **Scan the Text:**
   * Begin scanning the text from left to right, matching characters of the text with the pattern.
   * If a mismatch occurs at position jjj in the pattern, use the prefix table to skip over the first jjj characters and continue searching.
   * If a full match is found, return the index of the match in the text.
3. **Efficient Shifting:**
   * The key idea is that the pattern can be shifted by more than one character after a mismatch, as determined by the prefix function, thus avoiding unnecessary re-checks.

**Time Complexity:**

* The **time complexity** of the Knuth-Morris-Pratt algorithm is **O(n + m)**, where:
  + nnn is the length of the text,
  + mmm is the length of the pattern.
* The KMP algorithm ensures that every character in the text and pattern is compared only once, making it more efficient than the naive string matching algorithm, which has a worst-case time complexity of **O(nm)**.

**Example of KMP:**

Given the pattern "ABAB" and text "ABABABAB", the KMP algorithm would:

1. Preprocess the pattern to generate the prefix function: π=[0,0,1,2]\pi = [0, 0, 1, 2]π=[0,0,1,2]
2. Compare characters in the text with the pattern, and when a mismatch occurs, use the prefix function to shift the pattern and continue the comparison efficiently without re-checking characters.

**Comparison of Rabin-Karp and KMP:**

* **Rabin-Karp Algorithm:**
  + Best for multiple pattern searches.
  + Works by hashing substrings and comparing hash values.
  + Time complexity is **O(n + m)** on average but can degrade to **O(nm)** in the worst case due to hash collisions.
* **Knuth-Morris-Pratt (KMP) Algorithm:**
  + Best for a single pattern search.
  + Uses a preprocessed prefix function to avoid unnecessary re-checking of characters.
  + Time complexity is **O(n + m)**, which is more efficient for single pattern searches compared to the worst case of Rabin-Karp.

**UNIT II GRAPH ALGORITHMS**

1. **Write an algorithm for all pairs shortest path algorithm and what are the time and space complexity of the algorithm.**

The **Floyd-Warshall algorithm** is an algorithm for finding the shortest paths between all pairs of vertices in a weighted graph. It is a dynamic programming algorithm that computes the shortest paths in O(V3)O(V^3)O(V3) time, where VVV is the number of vertices in the graph. It is particularly useful for dense graphs or when we need to compute the shortest paths between all pairs of vertices.

**Algorithm (Floyd-Warshall):**

Let the graph be represented as an adjacency matrix dist[][], where dist[i][j] is the weight of the edge from vertex iii to vertex jjj. If there is no edge between vertices iii and jjj, dist[i][j] is set to infinity (∞\infty∞).

**Steps of the Algorithm:**

1. **Initialization:**
   * Initialize the distance matrix dist[i][j] as follows:
     + dist[i][i] = 0 for all vertices iii (the distance from a vertex to itself is zero),
     + dist[i][j] = weight(i, j) if there is an edge from vertex iii to vertex jjj,
     + dist[i][j] = ∞ if there is no edge between vertex iii and vertex jjj.
2. **Main Loop:**
   * For each vertex kkk, iterate over all pairs of vertices (i,j)(i, j)(i,j) and update the distance matrix as follows: dist[i][j]=min⁡(dist[i][j],dist[i][k]+dist[k][j])dist[i][j] = \min(dist[i][j], dist[i][k] + dist[k][j])dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j]) This step considers the possibility that the shortest path from vertex iii to vertex jjj may pass through vertex kkk.
3. **Termination:**
   * After all the vertices have been considered as intermediate vertices, the matrix dist[][] will contain the shortest path distances between all pairs of vertices.

**Pseudocode:**

python

Copy

def floyd\_warshall(graph):

# Step 1: Initialize the distance matrix

n = len(graph)

dist = [[float('inf')] \* n for \_ in range(n)]

# Set the diagonal to 0 (distance to itself)

for i in range(n):

dist[i][i] = 0

# Copy the graph weights to dist[][] for direct edges

for i in range(n):

for j in range(n):

if graph[i][j] != 0:

dist[i][j] = graph[i][j]

# Step 2: Floyd-Warshall main loop

for k in range(n): # Iterate through each vertex k as an intermediate vertex

for i in range(n): # Iterate through each source vertex i

for j in range(n): # Iterate through each destination vertex j

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

return dist

**Explanation of the Algorithm:**

* **Initialization:** We first initialize the distance matrix dist by setting the diagonal elements to 0 and the other elements to the weights of the edges between vertices.
* **Main Loop:** The main loop iterates over all possible intermediate vertices (denoted by kkk), and for each intermediate vertex, it iterates over all pairs of vertices (i,j)(i, j)(i,j) to update the shortest distance. The update rule is: dist[i][j]=min⁡(dist[i][j],dist[i][k]+dist[k][j])dist[i][j] = \min(dist[i][j], dist[i][k] + dist[k][j])dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j]) This checks if the path from iii to jjj via kkk is shorter than the direct path from iii to jjj.
* **Termination:** Once all intermediate vertices have been considered, the matrix dist contains the shortest distances between all pairs of vertices.

**Time Complexity:**

The time complexity of the Floyd-Warshall algorithm is determined by the three nested loops that iterate over all pairs of vertices and intermediate vertices. Specifically:

* The outer loop runs nnn times (for each vertex kkk).
* The two inner loops each run nnn times (for each pair of vertices iii and jjj).

Thus, the time complexity is:

O(n3)O(n^3)O(n3)

where nnn is the number of vertices in the graph.

**Space Complexity:**

The space complexity of the algorithm is mainly determined by the space required to store the distance matrix dist[][], which has dimensions n×nn \times nn×n. Therefore, the space complexity is:

O(n2)O(n^2)O(n2)

where nnn is the number of vertices in the graph.

1. **Explain Dijkstra’s algorithm using the following graph. Find the shortest path between v1, v2, v3, v4, v5, v6 & v7**

**V1**

**V2**

4

1

3

10

2

**V4**

7

**V3**

**V5**

5

8

4

6

**V6**

**V7**

To explain **Dijkstra's Algorithm** and find the shortest path between the vertices in the graph, let's break the problem into two parts:

1. **Understanding the Graph**: You have provided the graph with weights and vertices v1v\_1v1​ through v7v\_7v7​. However, the exact structure of the graph is not fully clear in the text, so I will assume a possible configuration based on the information you've provided. Here's how I interpret the graph:

**Graph Assumption:**

Let the graph have the following vertices and edges, with the weights as provided:

* v1v\_1v1​ — 4, 2 — v2v\_2v2​
* v1v\_1v1​ — 1, 7 — v3v\_3v3​
* v2v\_2v2​ — 3 — v4v\_4v4​
* v3v\_3v3​ — 10 — v4v\_4v4​
* v4v\_4v4​ — 5 — v5v\_5v5​
* v5v\_5v5​ — 8 — v6v\_6v6​
* v5v\_5v5​ — 4 — v7v\_7v7​
* v6v\_6v6​ — 6 — v7v\_7v7​

This gives the following graph, with weights on the edges:

lua

Copy

v1 --4-- v2 --3-- v4 --5-- v5 --8-- v6

| | |

1 7 4

| | |

v3 --------10----------' v7

**Dijkstra's Algorithm Overview:**

**Dijkstra's algorithm** is used to find the shortest path from a starting vertex to all other vertices in a weighted graph with non-negative edge weights. It works by repeatedly selecting the vertex with the smallest tentative distance and updating the distances of its neighbors.

**Steps of Dijkstra's Algorithm:**

1. **Initialization**:
   * Set the distance to the source vertex as 0 (i.e., dist[source] = 0).
   * Set the distance to all other vertices as infinity (dist[v] = ∞).
   * Mark all vertices as unvisited.
   * Set the previous vertex for each vertex as None.
2. **Visit the current vertex**:
   * Mark the current vertex as visited.
   * Update the tentative distances for its unvisited neighbors. For each neighbor vvv of the current vertex uuu, if the distance from the source to vvv through uuu is shorter than the current known distance to vvv, update the distance: dist[v]=min⁡(dist[v],dist[u]+weight(u,v))\text{dist}[v] = \min(\text{dist}[v], \text{dist}[u] + \text{weight}(u, v))dist[v]=min(dist[v],dist[u]+weight(u,v))
3. **Select the next vertex**:
   * After visiting all neighbors of the current vertex, select the unvisited vertex with the smallest tentative distance as the next current vertex.
4. **Repeat** the process until all vertices are visited or the smallest distance is infinity (indicating that remaining vertices are unreachable).

**Execution of Dijkstra's Algorithm on the Graph:**

Let’s assume we start at vertex v1v\_1v1​ and find the shortest path to all other vertices. Below are the steps with details.

**Initialization:**

* Distances from v1v\_1v1​:  
  dist[v1]=0\text{dist}[v\_1] = 0dist[v1​]=0 (starting vertex)  
  dist[v2]=∞\text{dist}[v\_2] = \inftydist[v2​]=∞  
  dist[v3]=∞\text{dist}[v\_3] = \inftydist[v3​]=∞  
  dist[v4]=∞\text{dist}[v\_4] = \inftydist[v4​]=∞  
  dist[v5]=∞\text{dist}[v\_5] = \inftydist[v5​]=∞  
  dist[v6]=∞\text{dist}[v\_6] = \inftydist[v6​]=∞  
  dist[v7]=∞\text{dist}[v\_7] = \inftydist[v7​]=∞

**Step 1: Visit v1v\_1v1​ (current vertex).**

* Update distances of neighbors of v1v\_1v1​:
  + v2:dist[v2]=min⁡(∞,0+4)=4v\_2: \text{dist}[v\_2] = \min(\infty, 0 + 4) = 4v2​:dist[v2​]=min(∞,0+4)=4
  + v3:dist[v3]=min⁡(∞,0+1)=1v\_3: \text{dist}[v\_3] = \min(\infty, 0 + 1) = 1v3​:dist[v3​]=min(∞,0+1)=1
* The distances now are:  
  dist[v1]=0\text{dist}[v\_1] = 0dist[v1​]=0, dist[v2]=4\text{dist}[v\_2] = 4dist[v2​]=4, dist[v3]=1\text{dist}[v\_3] = 1dist[v3​]=1, dist[v4]=∞\text{dist}[v\_4] = \inftydist[v4​]=∞, dist[v5]=∞\text{dist}[v\_5] = \inftydist[v5​]=∞, dist[v6]=∞\text{dist}[v\_6] = \inftydist[v6​]=∞, dist[v7]=∞\text{dist}[v\_7] = \inftydist[v7​]=∞.

**Step 2: Visit v3v\_3v3​ (next smallest distance).**

* Update distances of neighbors of v3v\_3v3​:
  + v4:dist[v4]=min⁡(∞,1+10)=11v\_4: \text{dist}[v\_4] = \min(\infty, 1 + 10) = 11v4​:dist[v4​]=min(∞,1+10)=11
* The distances now are:  
  dist[v1]=0\text{dist}[v\_1] = 0dist[v1​]=0, dist[v2]=4\text{dist}[v\_2] = 4dist[v2​]=4, dist[v3]=1\text{dist}[v\_3] = 1dist[v3​]=1, dist[v4]=11\text{dist}[v\_4] = 11dist[v4​]=11, dist[v5]=∞\text{dist}[v\_5] = \inftydist[v5​]=∞, dist[v6]=∞\text{dist}[v\_6] = \inftydist[v6​]=∞, dist[v7]=∞\text{dist}[v\_7] = \inftydist[v7​]=∞.

**Step 3: Visit v2v\_2v2​ (next smallest distance).**

* Update distances of neighbors of v2v\_2v2​:
  + v4:dist[v4]=min⁡(11,4+3)=7v\_4: \text{dist}[v\_4] = \min(11, 4 + 3) = 7v4​:dist[v4​]=min(11,4+3)=7
* The distances now are:  
  dist[v1]=0\text{dist}[v\_1] = 0dist[v1​]=0, dist[v2]=4\text{dist}[v\_2] = 4dist[v2​]=4, dist[v3]=1\text{dist}[v\_3] = 1dist[v3​]=1, dist[v4]=7\text{dist}[v\_4] = 7dist[v4​]=7, dist[v5]=∞\text{dist}[v\_5] = \inftydist[v5​]=∞, dist[v6]=∞\text{dist}[v\_6] = \inftydist[v6​]=∞, dist[v7]=∞\text{dist}[v\_7] = \inftydist[v7​]=∞.

**Step 4: Visit v4v\_4v4​ (next smallest distance).**

* Update distances of neighbors of v4v\_4v4​:
  + v5:dist[v5]=min⁡(∞,7+5)=12v\_5: \text{dist}[v\_5] = \min(\infty, 7 + 5) = 12v5​:dist[v5​]=min(∞,7+5)=12
* The distances now are:  
  dist[v1]=0\text{dist}[v\_1] = 0dist[v1​]=0, dist[v2]=4\text{dist}[v\_2] = 4dist[v2​]=4, dist[v3]=1\text{dist}[v\_3] = 1dist[v3​]=1, dist[v4]=7\text{dist}[v\_4] = 7dist[v4​]=7, dist[v5]=12\text{dist}[v\_5] = 12dist[v5​]=12, dist[v6]=∞\text{dist}[v\_6] = \inftydist[v6​]=∞, dist[v7]=∞\text{dist}[v\_7] = \inftydist[v7​]=∞.

**Step 5: Visit v3v\_3v3​ (next smallest distance).**

* Update distances of neighbors of v5v\_5v5​:
  + v6:dist[v6]=min⁡(∞,12+8)=20v\_6: \text{dist}[v\_6] = \min(\infty, 12 + 8) = 20v6​:dist[v6​]=min(∞,12+8)=20
  + v7:dist[v7]=min⁡(∞,12+4)=16v\_7: \text{dist}[v\_7] = \min(\infty, 12 + 4) = 16v7​:dist[v7​]=min(∞,12+4)=16
* The distances now are:  
  dist[v1]=0\text{dist}[v\_1] = 0dist[v1​]=0, dist[v2]=4\text{dist}[v\_2] = 4dist[v2​]=4, dist[v3]=1\text{dist}[v\_3] = 1dist[v3​]=1, dist[v4]=7\text{dist}[v\_4] = 7dist[v4​]=7, dist[v5]=12\text{dist}[v\_5] = 12dist[v5​]=12, dist[v6]=20\text{dist}[v\_6] = 20dist[v6​]=20, dist[v7]=16\text{dist}[v\_7] = 16dist[v7​]=16.

**Final Shortest Distances from v1v\_1v1​:**

* dist[v1]=0\text{dist}[v\_1] = 0dist[v1​]=0
* dist[v2]=4\text{dist}[v\_2] = 4dist[v2​]=4
* dist[v3]=1\text{dist}[v\_3] = 1dist[v3​]=1
* dist[v4]=7\text{dist}[v\_4] = 7dist[v4​]=7
* dist[v5]=12\text{dist}[v\_5] = 12dist[v5​]=12
* dist[v6]=20\text{dist}[v\_6] = 20dist[v6​]=20
* dist[v7]=16\text{dist}[v\_7] = 16dist[v7​]=16

These are the shortest distances from vertex v1v\_1v1​ to all other vertices.

**Time and Space Complexity of Dijkstra’s Algorithm:**

* **Time Complexity**:
  + With a **simple array**, the time complexity is **O(V^2)**, where VVV is the number of vertices.
  + With a **min-heap** (priority queue), the time complexity is **O((V + E) \log V)**, where EEE is the number of edges.
* **Space Complexity**:
  + The space complexity is **O(V + E)**, as we need to store the graph, the distances array, and the priority queue (if used).

1. **State and Prove Maximum Flow Min cut Theorem.**

**Theorem Statement:**

The **Maximum Flow Min-Cut Theorem** states that in a flow network, the maximum value of flow that can be sent from the source sss to the sink ttt is equal to the minimum capacity of a cut that separates the source from the sink.

Mathematically:

Maximum Flow=Minimum Cut\text{Maximum Flow} = \text{Minimum Cut}Maximum Flow=Minimum Cut

Where:

* **Maximum Flow** is the maximum flow value that can be sent from the source sss to the sink ttt in the network.
* **Minimum Cut** is the smallest total capacity of edges that, if removed, would disconnect the source sss from the sink ttt.

**Definitions:**

* **Flow Network:** A directed graph where each edge has a capacity (maximum flow that can pass through it) and there are two special vertices: a source sss and a sink ttt.
* **Flow:** A function that assigns a flow value to each edge, subject to the constraint that the flow on each edge cannot exceed its capacity.
* **Cut:** A partition of the set of vertices into two disjoint sets, one containing the source sss and the other containing the sink ttt. The cut is the set of edges going from the set containing sss to the set containing ttt.

**Proof of the Maximum Flow Min-Cut Theorem:**

The proof of the Maximum Flow Min-Cut Theorem is typically carried out using **two key results** from flow theory: the **Max-Flow Min-Cut Theorem** itself, and the **Ford-Fulkerson Algorithm** which iteratively finds augmenting paths to increase the flow in the network.

Here is an outline of the proof:

**Step 1: Define Flow and Cut**

Given a flow network G=(V,E)G = (V, E)G=(V,E) with source sss and sink ttt, we define:

* A **flow** fff as a function f:E→Rf : E \rightarrow \mathbb{R}f:E→R where the flow on an edge (u,v)(u, v)(u,v) satisfies:

0≤f(u,v)≤c(u,v)0 \leq f(u, v) \leq c(u, v)0≤f(u,v)≤c(u,v)

where c(u,v)c(u, v)c(u,v) is the capacity of the edge (u,v)(u, v)(u,v), and the total flow leaving any vertex (except sss) equals the total flow entering it (flow conservation).

* A **cut** (S,T)(S, T)(S,T) is a partition of the vertices into two sets SSS and TTT, where s∈Ss \in Ss∈S and t∈Tt \in Tt∈T, and the cut-set is the set of edges that go from SSS to TTT. The capacity of the cut is the sum of the capacities of edges from SSS to TTT:

Capacity of cut(S,T)=∑u∈S,v∈Tc(u,v)\text{Capacity of cut} (S, T) = \sum\_{u \in S, v \in T} c(u, v)Capacity of cut(S,T)=u∈S,v∈T∑​c(u,v)

The **maximum flow** is the largest possible flow from sss to ttt, and the **minimum cut** is the smallest possible capacity of a cut separating sss and ttt.

**Step 2: Max-Flow-Min-Cut Inequality**

First, we will prove that the **maximum flow** is less than or equal to the **minimum cut**.

* Consider a maximum flow fff from sss to ttt in the flow network.
* Let (S,T)(S, T)(S,T) be any cut separating sss and ttt. The flow fff passing through the cut can be split into two parts: flow going from SSS to TTT, and flow going from TTT to SSS.
* By the **flow conservation property**, the total flow from SSS to TTT is exactly the total flow entering TTT from SSS, which is equal to the total flow leaving SSS and entering TTT.

Thus, the total flow passing through the cut is bounded by the capacity of the cut:

Flow through the cut≤Capacity of the cut\text{Flow through the cut} \leq \text{Capacity of the cut}Flow through the cut≤Capacity of the cut

Since the cut separates sss and ttt, and the flow is maximized, the flow through any cut is at most the capacity of the cut.

Therefore:

Maximum Flow≤Minimum Cut\text{Maximum Flow} \leq \text{Minimum Cut}Maximum Flow≤Minimum Cut

**Step 3: Max-Flow = Min-Cut (Achieving Equality)**

Next, we show that the **maximum flow** is equal to the **minimum cut**.

* **Ford-Fulkerson Algorithm**: This algorithm iteratively increases the flow in the network by finding augmenting paths (paths where additional flow can be pushed from sss to ttt). This continues until no augmenting paths can be found, meaning the flow is maximized.
* When the algorithm terminates, the flow in the network is a **maximum flow**, and the residual graph (a graph where edges with remaining capacity are still available) will have no augmenting paths. This implies that there is a set of vertices reachable from sss using only residual edges in the residual graph. This set of reachable vertices is denoted as SSS, and its complement is TTT.
* The set (S,T)(S, T)(S,T) is a cut that separates sss from ttt. The total flow sent from SSS to TTT is equal to the total flow in the network, and by the properties of residual graphs, the capacity of the cut (S,T)(S, T)(S,T) is exactly equal to the maximum flow in the network.

Thus, the **maximum flow** achieved by the Ford-Fulkerson algorithm is equal to the **capacity of the minimum cut** (S,T)(S, T)(S,T), which proves the theorem:

Maximum Flow=Minimum Cut\text{Maximum Flow} = \text{Minimum Cut}Maximum Flow=Minimum Cut

1. **How do you construct a minimum spanning tree using Kruskals algorithm? Explain**?(4)

Kruskal's algorithm is an efficient algorithm used to find a **Minimum Spanning Tree (MST)** for a connected, weighted, undirected graph. The MST is a subgraph that connects all the vertices together with the minimum possible total edge weight, without forming any cycles.

**Steps to Construct MST Using Kruskal's Algorithm:**

1. **Sort all the edges** in the graph by their weights in **non-decreasing order**.
2. **Initialize** the MST as an empty set.
3. **Iterate** through the sorted edges and add each edge to the MST, provided that adding the edge does not form a cycle.
   * To check if adding an edge forms a cycle, use a **disjoint-set data structure** (also known as Union-Find) to keep track of which vertices are connected.
   * If the two vertices of an edge belong to the same set (i.e., adding the edge would form a cycle), skip that edge.
   * Otherwise, add the edge to the MST and **union** the sets of the two vertices.
4. **Repeat** the above process until the MST contains V−1V-1V−1 edges (where VVV is the number of vertices in the graph). Once this condition is met, the MST is complete.

**Step-by-Step Procedure:**

1. **Sort the Edges:**
   * First, list all the edges of the graph and sort them in increasing order of their weights.
2. **Initialize Disjoint Sets:**
   * Each vertex is initially its own parent (or set).
   * Use a disjoint-set (union-find) data structure to efficiently manage which vertices belong to which sets and to check if adding an edge forms a cycle.
   * The two main operations in this structure are:
     + **Find**: Determines which set a particular element belongs to.
     + **Union**: Merges two sets if they are not already connected.
3. **Iterate Over the Sorted Edges:**
   * Consider the edges one by one, starting with the smallest weight.
   * For each edge, check if the two vertices are in the same set:
     + If they are in different sets, add the edge to the MST and **union** the sets of the two vertices.
     + If they are already in the same set, skip the edge to avoid creating a cycle.
4. **Stop When MST Contains V−1V-1V−1 Edges:**
   * Once the MST has V−1V-1V−1 edges, where VVV is the number of vertices in the graph, the algorithm stops.

**Example:**

Consider the following weighted graph:

less

Copy

1 3

A ------- B

| |

| 4 | 2

| |

D ------- C

5

**Edges with weights:**

* A-B: 1
* A-D: 4
* B-C: 2
* C-D: 5

**Step 1: Sort the edges by weights:**

* A-B: 1
* B-C: 2
* A-D: 4
* C-D: 5

**Step 2: Initialize disjoint sets:**

* A, B, C, D each belong to their own set initially.

**Step 3: Start adding edges:**

1. **Add A-B (weight 1)**: A and B are in different sets, so add A-B to the MST and union A and B.

Sets: {A, B}, {C}, {D}

1. **Add B-C (weight 2)**: B and C are in different sets, so add B-C to the MST and union B and C.

Sets: {A, B, C}, {D}

1. **Add A-D (weight 4)**: A and D are in different sets, so add A-D to the MST and union A, B, C, and D.

Sets: {A, B, C, D}

At this point, the MST contains 3 edges (A-B, B-C, A-D) and all vertices are connected. We have added V−1=4−1=3V-1 = 4-1 = 3V−1=4−1=3 edges, so the algorithm stops.

**Resulting MST:**

The edges in the MST are:

* A-B: 1
* B-C: 2
* A-D: 4

**Time Complexity:**

* Sorting the edges takes O(Elog⁡E)O(E \log E)O(ElogE), where EEE is the number of edges.
* Each union-find operation (find and union) takes nearly constant time, O(α(V))O(\alpha(V))O(α(V)), where α\alphaα is the inverse Ackermann function, which grows very slowly and is considered constant for all practical purposes.
* Therefore, the overall time complexity of Kruskal's algorithm is O(Elog⁡E)O(E \log E)O(ElogE), which can also be expressed as O(Elog⁡V)O(E \log V)O(ElogV) because E≥VE \geq VE≥V.

**Space Complexity:**

* Kruskal's algorithm uses a disjoint-set data structure, which requires O(V)O(V)O(V) space for the parent and rank arrays.
* The space complexity is therefore O(V+E)O(V + E)O(V+E) because we also store the edges.

**UNIT III ALGORITHM DESIGN TECHNIQUES**

1. Define divide and conquer to apply the technique in binary search algorithm and to analysis it

**Divide and Conquer Technique:**

**Divide and Conquer** is a powerful algorithmic technique used for solving problems by breaking them down into smaller sub-problems, solving each sub-problem independently, and combining the results of the sub-problems to form the final solution. The technique typically follows three steps:

1. **Divide**: Break the problem into smaller sub-problems.
2. **Conquer**: Solve each sub-problem recursively. If the sub-problem is small enough, solve it directly.
3. **Combine**: Merge the solutions of the sub-problems to get the final result.

**Binary Search Algorithm Using Divide and Conquer:**

**Binary Search** is a classic example of a Divide and Conquer algorithm. It is used to search for a target element in a **sorted** array or list. The basic idea is to divide the array into two halves and check if the target element lies in the left or right half. If it lies in the left half, the search continues in that half; if it lies in the right half, the search continues there.

**Steps of Binary Search:**

1. **Divide**: Divide the array into two halves. Find the middle element.
2. **Conquer**: If the target element is equal to the middle element, return the index of the middle element. If the target is smaller, recursively search the left half of the array. If the target is larger, recursively search the right half of the array.
3. **Combine**: Since binary search either finds the element or reduces the problem size to half at each step, no explicit combining step is needed other than returning the result.

**Binary Search Algorithm:**

python

Copy

def binary\_search(arr, target):

low = 0

high = len(arr) - 1

while low <= high:

mid = (low + high) // 2 # Find the middle index

if arr[mid] == target:

return mid # Target found, return the index

elif arr[mid] < target:

low = mid + 1 # Target is in the right half, update low

else:

high = mid - 1 # Target is in the left half, update high

return -1 # Target is not found

**Explanation of the Algorithm:**

1. **Divide**: In each iteration of the while loop, we calculate the middle index (mid) by dividing the current search range (low to high) into two halves.
2. **Conquer**: We then compare the middle element (arr[mid]) with the target:
   * If the target is found, we return the index of the middle element.
   * If the target is smaller than arr[mid], we narrow the search to the left half by updating the high pointer (high = mid - 1).
   * If the target is larger than arr[mid], we narrow the search to the right half by updating the low pointer (low = mid + 1).
3. **Combine**: There is no explicit combine step, as we directly return the result when the target is found, or the search range becomes empty (i.e., low > high).

**Time Complexity Analysis:**

The time complexity of binary search is determined by how many times the search space is halved.

* In each step, the search range is reduced by half (either the left half or the right half).
* The number of steps required to reduce the search space to 1 element is the logarithm (base 2) of the number of elements, which is O(log⁡n)O(\log n)O(logn), where nnn is the number of elements in the array.

Thus, the **time complexity** of binary search is:

T(n)=O(log⁡n)T(n) = O(\log n)T(n)=O(logn)

where nnn is the number of elements in the array.

**Space Complexity Analysis:**

Binary search uses a constant amount of space to store the pointers (low, high, mid), regardless of the size of the input array. Therefore, the **space complexity** of binary search is:

S(n)=O(1)S(n) = O(1)S(n)=O(1)

1. **Explain in detail in merge sort give an example**

**Merge Sort** is a classic example of the **Divide and Conquer** technique. It is an efficient, stable, and comparison-based sorting algorithm. Merge Sort divides the input array into two halves, recursively sorts each half, and then merges the two sorted halves to produce the sorted output.

**Steps of Merge Sort:**

1. **Divide**: If the array has more than one element, split the array into two halves.
2. **Conquer**: Recursively sort each half of the array. If the array has only one element, it is already sorted.
3. **Combine**: Merge the two sorted halves back together to form a single sorted array.

**Merge Sort Algorithm (Pseudocode):**

python

Copy

def merge\_sort(arr):

# Base case: if the array has only one element, it is already sorted

if len(arr) <= 1:

return arr

# Divide the array into two halves

mid = len(arr) // 2

left\_half = arr[:mid]

right\_half = arr[mid:]

# Recursively sort both halves

left\_half = merge\_sort(left\_half)

right\_half = merge\_sort(right\_half)

# Combine the two sorted halves

return merge(left\_half, right\_half)

def merge(left, right):

sorted\_array = []

i = j = 0

# Merge the two sorted arrays

while i < len(left) and j < len(right):

if left[i] < right[j]:

sorted\_array.append(left[i])

i += 1

else:

sorted\_array.append(right[j])

j += 1

# Add any remaining elements from both arrays

sorted\_array.extend(left[i:])

sorted\_array.extend(right[j:])

return sorted\_array

**Explanation of Merge Sort:**

**Step 1: Divide**

* The array is divided into two halves using the middle index (mid = len(arr) // 2).
* This process is recursively repeated for each half of the array until all sub-arrays have a length of 1 (i.e., they are trivially sorted).

**Step 2: Conquer**

* The recursive calls sort the left and right halves. Once an array has been divided down to a length of 1, it is considered sorted, and the algorithm begins merging the sorted sub-arrays back together.

**Step 3: Combine (Merging)**

* The **merge function** takes two sorted arrays and merges them into a single sorted array. It does this by:
  + Comparing the elements of both arrays.
  + Appending the smaller element to the result array.
  + Once one array is exhausted, it appends all remaining elements from the other array.

**Example:**

Let's see how Merge Sort works on an example array:

**Example Array:**

38,27,43,3,9,82,1038, 27, 43, 3, 9, 82, 1038,27,43,3,9,82,10

**Step 1: Divide**

* Split the array into two halves:
  + Left half: 38,27,4338, 27, 4338,27,43
  + Right half: 3,9,82,103, 9, 82, 103,9,82,10

**Step 2: Recursively Sort Both Halves**

1. **Sort Left Half 38,27,4338, 27, 4338,27,43:**
   * Split into 383838 and 27,4327, 4327,43.
   * 383838 is already sorted.
   * Sort 27,4327, 4327,43 by splitting into 272727 and 434343, both already sorted.
   * Merge 272727 and 434343 to form 27,4327, 4327,43.
   * Merge 383838 and 27,4327, 4327,43 to form 27,38,4327, 38, 4327,38,43.
2. **Sort Right Half 3,9,82,103, 9, 82, 103,9,82,10:**
   * Split into 3,93, 93,9 and 82,1082, 1082,10.
   * 3,93, 93,9 is sorted by merging into 3,93, 93,9.
   * 82,1082, 1082,10 is sorted by splitting into 828282 and 101010, then merging into 10,8210, 8210,82.
   * Merge 3,93, 93,9 and 10,8210, 8210,82 to form 3,9,10,823, 9, 10, 823,9,10,82.

**Step 3: Combine**

* Now we have two sorted halves:
  + Left half: 27,38,4327, 38, 4327,38,43
  + Right half: 3,9,10,823, 9, 10, 823,9,10,82
* Merge 27,38,4327, 38, 4327,38,43 and 3,9,10,823, 9, 10, 823,9,10,82:
  + Compare 27 and 3 → 3 is smaller, so add 3 to the sorted array.
  + Compare 27 and 9 → 9 is smaller, so add 9 to the sorted array.
  + Compare 27 and 10 → 10 is smaller, so add 10 to the sorted array.
  + Compare 27 and 82 → 27 is smaller, so add 27 to the sorted array.
  + Compare 38 and 82 → 38 is smaller, so add 38 to the sorted array.
  + Compare 43 and 82 → 43 is smaller, so add 43 to the sorted array.
  + Finally, add the remaining 82 to the sorted array.
* Final sorted array: 3,9,10,27,38,43,823, 9, 10, 27, 38, 43, 823,9,10,27,38,43,82

**Time Complexity of Merge Sort:**

* **Divide Step**: The array is split into two halves at each recursive step. This happens log⁡n\log nlogn times, where nnn is the number of elements.
* **Conquer Step**: At each level of recursion, we perform nnn comparisons and operations to merge the two halves.
* Therefore, the total time complexity is O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the number of elements in the array.

**Space Complexity of Merge Sort:**

* The algorithm requires additional space to store the sub-arrays during the merging process. The total space complexity is O(n)O(n)O(n), where nnn is the number of elements in the array, because we are using extra space to store the merged arrays.

**Advantages of Merge Sort:**

1. **Efficient**: Merge Sort has a time complexity of O(nlog⁡n)O(n \log n)O(nlogn), which is better than other algorithms like Bubble Sort and Insertion Sort that have a time complexity of O(n2)O(n^2)O(n2).
2. **Stable**: Merge Sort is a stable sort, meaning that if two elements have equal values, their relative order remains the same in the sorted array.
3. **Predictable Performance**: Merge Sort has consistent time complexity regardless of the input array's initial order.

**Disadvantages of Merge Sort:**

1. **Space Complexity**: Merge Sort requires O(n)O(n)O(n) extra space for the merge process, making it less space-efficient compared to in-place algorithms like Quick Sort.
2. **Slower for Small Arrays**: For very small arrays, simpler algorithms like Insertion Sort may outperform Merge Sort due to lower constant factors.
3. **Explain in detail quick sorting method. Provide a complete analysis of quick sort**.

**Quick Sort** is a **divide and conquer** sorting algorithm that is very efficient for large datasets. It was developed by Tony Hoare in 1960 and is widely used because of its **average-case performance** and its ability to perform **in-place sorting** (i.e., it doesn’t require additional storage space for the sorted array).

The basic idea behind Quick Sort is to **partition** the array into two smaller sub-arrays, **recursively sort** those sub-arrays, and **combine** them to get the sorted array. The partitioning process is the key to Quick Sort.

### ****Steps of Quick Sort:****

1. **Pick a pivot**: Choose an element from the array (this element is called the "pivot"). The choice of pivot can vary, and we'll discuss different strategies later.
2. **Partition the array**: Rearrange the elements in the array so that:
   * All elements less than the pivot go to the left of the pivot.
   * All elements greater than the pivot go to the right of the pivot.
   * The pivot is now in its correct sorted position in the array.
3. **Recursively sort** the left and right sub-arrays formed by partitioning. Repeat the process on each sub-array until the base case is reached (sub-array has 1 or 0 elements, which are trivially sorted).

### ****Partitioning Process:****

The partitioning process is the key part of Quick Sort. In the typical implementation, we do the following:

1. **Choose a pivot** (e.g., the last element).
2. **Rearrange** the array so that all elements smaller than the pivot are on the left side, and all elements greater than the pivot are on the right side.
3. The pivot is then placed at its correct position, and the array is split into two sub-arrays that need to be sorted.

### ****Quick Sort Algorithm (Pseudocode):****

python

Copy

def quick\_sort(arr, low, high):

if low < high:

# Partition the array

pi = partition(arr, low, high)

# Recursively sort the left and right sub-arrays

quick\_sort(arr, low, pi - 1)

quick\_sort(arr, pi + 1, high)

def partition(arr, low, high):

# Choose the pivot (typically the last element)

pivot = arr[high]

i = low - 1 # pointer for the smaller element

# Rearrange elements based on pivot

for j in range(low, high):

if arr[j] < pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i] # Swap elements

# Place the pivot in its correct position

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

return i + 1 # Return the index of the pivot

### ****Explanation of the Algorithm:****

1. **Base Case**: The quick\_sort function terminates when the sub-array has only one element or is empty (low >= high), meaning it's already sorted.
2. **Partitioning**: The partition function is where the magic happens. It selects a pivot element (typically the last element) and rearranges the array by placing all elements smaller than the pivot to its left and all elements greater than the pivot to its right. It then places the pivot in its correct position (where it will not move anymore) and returns the index of the pivot.
3. **Recursive Sorting**: After partitioning, the pivot is at its correct position. The quick\_sort function is called recursively to sort the left and right sub-arrays.

### ****Example:****

Consider the array:

10,80,30,90,40,50,7010, 80, 30, 90, 40, 50, 7010,80,30,90,40,50,70

1. **Choose a Pivot**: Select the last element as the pivot. In this case, 707070 is the pivot.
2. **Partitioning**:
   * The array is rearranged so that all elements smaller than 707070 are on the left and all elements larger are on the right:
     + After partitioning, the array becomes 10,30,40,50,70,90,8010, 30, 40, 50, 70, 90, 8010,30,40,50,70,90,80, and the pivot 707070 is placed in its correct position at index 4.
3. **Recursively Sort Left and Right Sub-arrays**:
   * Left sub-array: 10,30,40,5010, 30, 40, 5010,30,40,50
     + Choose pivot 505050, partition, and sort recursively.
     + Left sub-array: 10,30,4010, 30, 4010,30,40
     + Choose pivot 404040, partition, and sort.
     + The final sorted array after sorting both sub-arrays is 10,30,40,5010, 30, 40, 5010,30,40,50.
   * Right sub-array: 90,8090, 8090,80
     + Choose pivot 808080, partition, and sort.
4. **Final Sorted Array**: 10,30,40,50,70,80,9010, 30, 40, 50, 70, 80, 9010,30,40,50,70,80,90

### ****Time Complexity Analysis:****

Quick Sort has different time complexities depending on the choice of pivot and the distribution of elements in the array.

1. **Best Case**:
   * The best case occurs when the pivot divides the array into two equal halves. In this case, the array is divided in a balanced way at each recursive step, and the time complexity is:

T(n)=O(nlog⁡n)T(n) = O(n \log n)T(n)=O(nlogn)

where nnn is the number of elements in the array.

1. **Average Case**:
   * In the average case, Quick Sort also behaves like the best case where the pivot splits the array into approximately equal parts, leading to a time complexity of:

T(n)=O(nlog⁡n)T(n) = O(n \log n)T(n)=O(nlogn)

1. **Worst Case**:
   * The worst case occurs when the pivot is the smallest or largest element in the array, which results in highly unbalanced partitioning. In this case, the algorithm has to perform n−1n-1n−1 comparisons at each step, leading to a time complexity of:

T(n)=O(n2)T(n) = O(n^2)T(n)=O(n2)

This happens if the array is already sorted or nearly sorted, or if a poor pivot choice is made.

1. **Space Complexity**:
   * The space complexity of Quick Sort is O(log⁡n)O(\log n)O(logn) due to the recursive calls. The recursion depth is proportional to the height of the recursive tree, which is log⁡n\log nlogn in the best/average case and nnn in the worst case.
   * **In-place Sorting**: Since Quick Sort is an in-place sorting algorithm (it doesn't require additional space for a new array), the space complexity for the algorithm is minimal.

### ****Comparison of Quick Sort with Other Sorting Algorithms:****

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Space Complexity** |
| --- | --- | --- | --- | --- |
| Quick Sort | O(nlog⁡n)O(n \log n)O(nlogn) | O(nlog⁡n)O(n \log n)O(nlogn) | O(n2)O(n^2)O(n2) | O(log⁡n)O(\log n)O(logn) |
| Merge Sort | O(nlog⁡n)O(n \log n)O(nlogn) | O(nlog⁡n)O(n \log n)O(nlogn) | O(nlog⁡n)O(n \log n)O(nlogn) | O(n)O(n)O(n) |
| Bubble Sort | O(n)O(n)O(n) | O(n2)O(n^2)O(n2) | O(n2)O(n^2)O(n2) | O(1)O(1)O(1) |
| Insertion Sort | O(n)O(n)O(n) | O(n2)O(n^2)O(n2) | O(n2)O(n^2)O(n2) | O(1)O(1)O(1) |

### ****Advantages of Quick Sort:****

1. **Efficient**: In the average case, Quick Sort is very fast, with a time complexity of O(nlog⁡n)O(n \log n)O(nlogn).
2. **In-place Sorting**: Quick Sort does not require extra space, making it space-efficient compared to algorithms like Merge Sort, which requires O(n)O(n)O(n) extra space.
3. **Good for Large Datasets**: Quick Sort is particularly efficient for large datasets due to its O(nlog⁡n)O(n \log n)O(nlogn) average time complexity.

### ****Disadvantages of Quick Sort:****

1. **Worst Case Performance**: The worst-case time complexity is O(n2)O(n^2)O(n2), which can occur if the pivot selection is poor (e.g., the pivot is always the smallest or largest element).
2. **Not Stable**: Quick Sort is not a stable sort. It may change the relative order of equal elements in the array.

**4.Write the Huffman code algorithm and derive its time complexity.** **) Generate the Huffman code for the following data comprising of alphabet and their frequency.(6)**

**a:1,b:1,c:2,d:3,e:5,f:8,g:13,h:21**

Huffman coding is a popular algorithm used for lossless data compression. It assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters and longer codes assigned to less frequent characters.

**Steps of Huffman Coding Algorithm:**

1. **Build a Priority Queue (Min-Heap)**:
   * Create a leaf node for each unique character and its frequency.
   * Insert all the nodes into a min-heap, where each node represents a character and its frequency.
2. **Build the Huffman Tree**:
   * While there is more than one node in the heap:
     + Remove the two nodes with the lowest frequencies.
     + Create a new internal node with a frequency equal to the sum of the two nodes’ frequencies.
     + Assign the two nodes as children of this new node.
     + Insert the new node back into the heap.
3. **Generate Huffman Codes**:
   * Once the tree is built, traverse it starting from the root.
   * Assign '0' for left edges and '1' for right edges.
   * The Huffman code for a character is the sequence of bits from the root to the character’s leaf node.

**Pseudocode for Huffman Coding:**

python

Copy

import heapq

class Node:

def \_\_init\_\_(self, char, freq):

self.char = char

self.freq = freq

self.left = None

self.right = None

def \_\_lt\_\_(self, other):

return self.freq < other.freq

def build\_huffman\_tree(freq\_dict):

heap = []

# Step 1: Build a min-heap from the frequency dictionary

for char, freq in freq\_dict.items():

heapq.heappush(heap, Node(char, freq))

# Step 2: Build the Huffman Tree

while len(heap) > 1:

left = heapq.heappop(heap)

right = heapq.heappop(heap)

merged\_node = Node(None, left.freq + right.freq)

merged\_node.left = left

merged\_node.right = right

heapq.heappush(heap, merged\_node)

return heap[0]

def generate\_codes(node, current\_code="", codebook={}):

if node is not None:

if node.char is not None:

codebook[node.char] = current\_code

generate\_codes(node.left, current\_code + "0", codebook)

generate\_codes(node.right, current\_code + "1", codebook)

return codebook

# Example to build the Huffman tree and generate the codes

freq\_dict = {'a': 1, 'b': 1, 'c': 2, 'd': 3, 'e': 5, 'f': 8, 'g': 13, 'h': 21}

root = build\_huffman\_tree(freq\_dict)

huffman\_codes = generate\_codes(root)

print("Huffman Codes:", huffman\_codes)

**Time Complexity of Huffman Coding:**

The time complexity of the Huffman coding algorithm is determined by the following steps:

1. **Building the Min-Heap**:
   * Inserting nnn elements into the heap takes O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the number of unique characters in the input.
2. **Building the Huffman Tree**:
   * The tree-building process involves merging nodes. Each merge operation involves removing the two smallest elements from the heap and inserting a new merged node. Since there are n−1n-1n−1 merges and each heap operation (pop and push) takes O(log⁡n)O(\log n)O(logn), the total time for building the tree is O(nlog⁡n)O(n \log n)O(nlogn).
3. **Generating Huffman Codes**:
   * The generation of Huffman codes involves a traversal of the Huffman tree, which takes O(n)O(n)O(n) time, as we visit every node once.

Thus, the overall time complexity of the algorithm is:

O(nlog⁡n)O(n \log n)O(nlogn)

where nnn is the number of characters in the input.

**Example of Huffman Coding for the Given Data:**

Given the following characters and their frequencies:

yaml

Copy

a: 1, b: 1, c: 2, d: 3, e: 5, f: 8, g: 13, h: 21

**Step 1: Build the Min-Heap**

The nodes for each character are:

css

Copy

[(a, 1), (b, 1), (c, 2), (d, 3), (e, 5), (f, 8), (g, 13), (h, 21)]

**Step 2: Build the Huffman Tree**

1. Combine (a, 1) and (b, 1) → New node with frequency 2.
2. Combine (c, 2) and the node with frequency 2 → New node with frequency 4.
3. Combine (d, 3) and (e, 5) → New node with frequency 8.
4. Combine (f, 8) and the node with frequency 4 → New node with frequency 12.
5. Combine (g, 13) and (h, 21) → New node with frequency 34.
6. Combine the two nodes with frequencies 12 and 8 → New node with frequency 20.
7. Combine the two nodes with frequencies 20 and 34 → New node with frequency 54.
8. Combine the two nodes with frequencies 12 and 8 → New node with frequency 20.

**Step 3: Generate Huffman Codes**

The final Huffman codes are:

* **h**: 0
* **g**: 10
* **f**: 11
* **e**: 100
* **d**: 101
* **c**: 1100
* **b**: 1101
* **a**: 111

**UNIT IV STATE SPACE SEARCH ALGORITHMS**

1. **Explain subset sum problem & discuss the possible solution strategies using backtracking.**

The **Subset Sum Problem** is a classic problem in computer science and mathematics. It is defined as follows:

**Given**: A set of integers S={s1,s2,…,sn}S = \{s\_1, s\_2, \dots, s\_n\}S={s1​,s2​,…,sn​} and a target sum TTT, the task is to determine if there is a subset of SSS whose sum is equal to TTT.

**Problem Definition**:

* Find whether there exists a subset of the given set SSS such that the sum of the elements of the subset equals the target sum TTT.
* If such a subset exists, return **True** (indicating that the subset exists).
* Otherwise, return **False**.

**Example:**

* **Input**:  
  S={3,34,4,12,5,2}S = \{3, 34, 4, 12, 5, 2\}S={3,34,4,12,5,2}, T=9T = 9T=9
* **Output**:  
  **True**, because the subset {4,5}\{4, 5\}{4,5} adds up to 999.

**Solution Strategies**

There are multiple approaches to solving the Subset Sum problem, including:

1. **Dynamic Programming**
2. **Backtracking**
3. **Brute Force**

We will focus on the **backtracking approach** in detail.

**Backtracking Solution for Subset Sum Problem**

Backtracking is a general algorithmic technique that involves building solutions incrementally, one step at a time, and discarding solutions that fail to meet the criteria at any point during the process. In the case of the Subset Sum Problem, backtracking involves trying each possible subset and exploring whether it sums up to the target value. If it doesn't, we backtrack and try another subset.

**Steps for Backtracking:**

1. **Start at the first element** of the set.
2. **Choose to include or exclude** the element in the subset.
3. **Include the current element**: If we include the element, subtract its value from the target sum.
4. **Exclude the current element**: If we exclude the element, continue exploring without changing the target sum.
5. **Base Case**: If the target sum becomes 0, return True (because we found a subset with the required sum).
6. **Pruning**: If the sum becomes negative, return False because it's impossible to reach the target sum.

**Backtracking Algorithm for Subset Sum:**

python

Copy

def is\_subset\_sum(arr, n, target):

# Base case: If target becomes 0, we found a subset

if target == 0:

return True

# If no elements left or target becomes negative, no solution

if n == 0 or target < 0:

return False

# Include the current element and move to the next

include = is\_subset\_sum(arr, n-1, target-arr[n-1])

# Exclude the current element and move to the next

exclude = is\_subset\_sum(arr, n-1, target)

# Return true if either including or excluding works

return include or exclude

# Example usage

arr = [3, 34, 4, 12, 5, 2]

target = 9

n = len(arr)

print(is\_subset\_sum(arr, n, target)) # Output: True

**Explanation of the Backtracking Algorithm:**

* **Base Case 1**: If the target sum TTT is 0, return True because we have found a subset that sums to the target.
* **Base Case 2**: If we have exhausted all elements in the array (i.e., n=0n = 0n=0), or if the target becomes negative, return False because it’s not possible to find a valid subset.
* For each element:
  + First, attempt to **include** the current element in the subset and reduce the target sum accordingly.
  + Second, attempt to **exclude** the current element and keep the target sum unchanged.
* **Recursive calls** are made for both include and exclude options. The algorithm returns True if either approach leads to a valid solution.

**Time Complexity of Backtracking Solution:**

The time complexity of the backtracking solution can be analyzed as follows:

* The algorithm explores every possible subset of the given set. Since the set has nnn elements, there are 2n2^n2n possible subsets (because each element can either be included or excluded).
* In the worst case, we explore all subsets. For each subset, the algorithm performs a constant amount of work (subtracting the current element from the target, making recursive calls).

Thus, the time complexity is O(2n)O(2^n)O(2n), where nnn is the number of elements in the input array. This exponential complexity is typical for backtracking algorithms.

**Space Complexity:**

* The space complexity is mainly due to the recursion stack. In the worst case, the recursion depth will be O(n)O(n)O(n) (in cases where every element is considered either for inclusion or exclusion).
* Thus, the space complexity is O(n)O(n)O(n).

1. **What is Branch and bound? Explain in detail**

**Branch and Bound (B&B)** is a general algorithmic technique for solving optimization problems, particularly those that are combinatorial in nature. It is used to find the optimal solution to problems like the **Travelling Salesman Problem (TSP)**, **0/1 Knapsack Problem**, and various other NP-hard problems. The method systematically explores and prunes the solution space by breaking it down into smaller subproblems and discarding suboptimal solutions. Here's an in-depth explanation of the Branch and Bound technique:

**Key Concepts of Branch and Bound**

1. **Branching**:
   * The "branch" part refers to dividing the problem into smaller subproblems (branches). This step is often done by considering different possibilities (e.g., decision variables or choices) and breaking the problem into subproblems based on these possibilities.
   * For example, in a **0/1 Knapsack Problem**, the branch could involve deciding whether to include or exclude an item from the knapsack.
2. **Bounding**:
   * The "bound" refers to computing upper and lower bounds for the optimal solution in each subproblem. These bounds help to evaluate whether the current subproblem can potentially lead to a better solution than the best one found so far.
   * The upper bound provides an estimate of the best possible solution in the subproblem, while the lower bound provides an estimate of the worst-case solution.
   * For example, in **minimization problems**, an upper bound is a potential solution that is worse than or equal to the current best, and a lower bound is the best achievable solution in the subproblem.
3. **Pruning**:
   * Pruning involves discarding or eliminating subproblems that are not promising. This is done based on the bounds, ensuring that we do not explore branches that cannot lead to a better solution than what has already been found.
   * If the lower bound of a subproblem is worse than the current best solution, that subproblem is pruned because it cannot improve the optimal solution.
4. **Optimality**:
   * The goal of branch and bound is to explore enough of the solution space to guarantee finding the **optimal solution** while avoiding the full enumeration of all possibilities.
   * At the end of the process, the algorithm returns the optimal solution found during the search.

**Steps in Branch and Bound Algorithm**

1. **Initialization**:
   * Start with the entire problem and compute an initial bound (usually based on a relaxation of the problem). This bound gives an estimate of the best possible solution.
   * If the problem has a feasible solution space, initialize the best solution found so far.
2. **Branching**:
   * Choose a subproblem to explore based on a branching rule. This may involve making decisions (such as dividing the problem into subproblems or making a choice for a variable).
   * For example, in a **0/1 knapsack problem**, a branch might involve either including or excluding an item.
3. **Bounding**:
   * Calculate the bound for the current subproblem.
   * If the bound is worse than the current best solution, prune the subproblem (i.e., discard it and do not explore further).
   * If the bound is better, continue exploring this branch.
4. **Pruning**:
   * If the current solution is promising (i.e., it has a lower bound better than the current best solution), further branches are explored.
   * If the current subproblem cannot lead to a better solution than the best one found, it is pruned.
5. **Repeat**:
   * This process is repeated until all relevant subproblems have been either solved or pruned.
6. **Final Solution**:
   * The process ends when no subproblems are left to explore, and the optimal solution is returned.

**Example: Solving the 0/1 Knapsack Problem using Branch and Bound**

In the **0/1 Knapsack Problem**, you are given a set of items, each with a weight and a value, and a knapsack with a limited capacity. The objective is to maximize the total value of items in the knapsack without exceeding its capacity.

**Steps:**

1. **Branching**:
   * At each node, decide whether to include or exclude each item.
2. **Bounding**:
   * Calculate the upper bound for each branch. The upper bound can be the total value of the items that can fit into the knapsack, assuming fractional items can be included (relaxation).
3. **Pruning**:
   * If the upper bound of a branch is less than the best solution found so far, prune that branch and do not explore further.
4. **Repeat**:
   * Continue branching and bounding until all subproblems are either solved or pruned.

**Advantages of Branch and Bound:**

* **Guaranteed Optimal Solution**: Since the method systematically explores the solution space and uses bounds to prune unnecessary paths, it guarantees finding the optimal solution.
* **Efficient Search**: It avoids evaluating every possible solution by pruning branches that cannot lead to a better solution.
* **Flexibility**: It can be adapted to a wide range of optimization problems.

**Disadvantages of Branch and Bound:**

* **Computational Complexity**: Although it avoids exploring all possible solutions, the method can still be very computationally expensive for large problems (especially NP-hard ones).
* **Memory Usage**: Branch and bound algorithms can require a lot of memory to store the subproblems, especially for complex problems with a large solution space.

**Use Cases:**

* **Travelling Salesman Problem (TSP)**: Branch and bound is often used to find the shortest possible route that visits all cities once and returns to the starting point.
* **0/1 Knapsack Problem**: Optimizing the selection of items that maximize the total value while staying within a weight limit.
* **Integer Programming Problems**: Where variables are restricted to integer values.

1. (i)Suggest an approximation algorithm for travelling salesperson problem. Assume that the cost function satisfies the triangle inequality. **(MAY 2015) (R)**

(ii) Explain how job assignment problem could be solved, given n tasks and n agents where each agent has a cost to complete each task, using branch and bound.

**(i) Approximation Algorithm for the Travelling Salesperson Problem (TSP) - Triangle Inequality Assumed**

The **Travelling Salesperson Problem (TSP)** is a classical problem in optimization where a salesman must visit each city exactly once and return to the starting city, with the goal of minimizing the total distance or cost of travel. When the **triangle inequality** holds (i.e., the direct distance between two cities is always shorter than or equal to the distance of any indirect path), an approximation algorithm can be used to find a near-optimal solution efficiently.

**2-Approximation Algorithm (Christofides' Algorithm)**

One of the well-known approximation algorithms for TSP, when the triangle inequality is satisfied, is the **2-Approximation Algorithm**. This algorithm guarantees that the solution will be at most **twice** the length of the optimal solution.

The steps for the 2-approximation algorithm are as follows:

1. **Construct a Minimum Spanning Tree (MST)**:
   * First, construct a **Minimum Spanning Tree (MST)** of the cities using any algorithm, such as **Kruskal's Algorithm** or **Prim's Algorithm**. The MST connects all cities in such a way that the total distance of the edges is minimized, and no cycles are formed.
2. **Find the Minimum Weight Perfect Matching (MWPM)**:
   * From the MST, identify the set of vertices that have an odd degree. A minimum weight perfect matching (MWPM) is created by pairing up the vertices with odd degrees, ensuring that the total edge weight of this matching is minimized. This step ensures that all vertices in the MST will have an even degree (a property that is essential for the Eulerian circuit).
3. **Combine the MST and MWPM**:
   * Add the edges of the MWPM to the MST, forming a multigraph where all vertices have an even degree. This guarantees that the graph has an Eulerian circuit, i.e., a circuit where each edge is visited exactly once.
4. **Find an Eulerian Circuit**:
   * Traverse the graph (MST + MWPM) to find an Eulerian circuit. This circuit can be done by a **Depth-First Search (DFS)** or a **Hierholzer's algorithm** to find the Eulerian path, which visits all edges exactly once.
5. **Convert the Eulerian Circuit to a Hamiltonian Circuit**:
   * The Eulerian circuit will visit some vertices multiple times, so we need to transform it into a Hamiltonian circuit (i.e., visit each vertex exactly once). This can be done by skipping the repeated vertices in the Eulerian path.

**Performance Guarantee:**

* The 2-approximation algorithm for TSP ensures that the solution will be no more than twice the length of the optimal solution. This approximation is very useful when exact solutions are computationally expensive or intractable.

**(ii) Solving the Job Assignment Problem using Branch and Bound**

The **Job Assignment Problem** is a classic optimization problem where you have **n agents** and **n tasks**, and the goal is to assign tasks to agents such that the total cost of completing all the tasks is minimized. Each agent has a different cost for completing each task, and the objective is to minimize the total assignment cost.

The **Branch and Bound** algorithm can be used to solve this problem by systematically exploring the possible assignments and pruning non-promising ones. Below are the steps to solve this problem using Branch and Bound:

**Steps for Solving the Job Assignment Problem Using Branch and Bound:**

1. **Problem Representation**:
   * Represent the problem as a matrix of size n×nn \times nn×n, where the element C(i,j)C(i, j)C(i,j) represents the cost of assigning the ithi^{th}ith agent to the jthj^{th}jth task.
2. **Initial Bound Calculation**:
   * Start with an initial **lower bound** for the cost. This can be done using a relaxation technique such as:
     + **Row reduction**: Subtract the minimum value in each row from all the elements in that row.
     + **Column reduction**: Subtract the minimum value in each column from all the elements in that column.
     + The sum of all these subtractions forms a lower bound for the assignment problem.
3. **Branching**:
   * At each node, decide which agent will be assigned which task. The branching process will systematically assign each agent to a task.
   * For each agent, you make a binary decision: either assign them to a particular task or leave that task unassigned. This generates subproblems (branches).
4. **Bounding**:
   * For each subproblem, calculate a lower bound based on the partial assignments made so far. This is typically done by updating the lower bound after each assignment.
   * If the lower bound of a subproblem exceeds the current best known solution (i.e., the current best assignment), prune that branch (i.e., do not explore it further).
   * The lower bound is updated based on how many tasks have already been assigned and what the remaining costs will be for the unassigned tasks.
5. **Pruning**:
   * If a branch has a lower bound that is greater than or equal to the best solution found so far, prune it. This is because no further exploration in this branch will lead to a better solution than what has already been found.
   * If a complete assignment is made (i.e., all agents are assigned tasks), then check if the cost is less than the current best solution. If it is, update the best solution.
6. **Repeat**:
   * Continue branching and bounding until all possible assignments have been either explored or pruned. At the end, the best solution found is the optimal assignment with the minimum cost.

**Performance Considerations:**

* **Branch and Bound** guarantees that you will find the optimal solution to the Job Assignment Problem.
* However, the time complexity can be quite high for large problems since the algorithm explores a large number of potential assignments.
* The number of possible assignments is n!n!n!, but the pruning process can significantly reduce the number of subproblems to explore.

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

1. **Discuss the approximation algorithm for NP hard problem?**

**NP-hard problems** are computationally difficult problems for which no known polynomial-time algorithm can guarantee finding an optimal solution in a reasonable amount of time. Since solving NP-hard problems optimally in polynomial time is unlikely (as no polynomial-time algorithms have been found and it is assumed P ≠ NP), **approximation algorithms** are used as practical alternatives. These algorithms do not guarantee an optimal solution, but they can find a solution that is **close enough** to the optimal one within a provable bound.

**Key Features of Approximation Algorithms:**

1. **Polynomial Time Complexity**: Approximation algorithms run in polynomial time, making them feasible for large problem instances.
2. **Performance Guarantee**: These algorithms provide a bound on the quality of the solution they produce. For example, an approximation algorithm might guarantee that the solution is within a certain factor of the optimal solution (often called the approximation ratio).
3. **Trade-off Between Accuracy and Time**: Approximation algorithms may not give the optimal solution but are designed to find a "good enough" solution in much less time than exact algorithms.

**Types of Approximation Algorithms:**

1. **Constant Factor Approximation**: These algorithms guarantee that the solution is within a fixed constant factor of the optimal solution.
2. **Polynomial Factor Approximation**: These algorithms provide a performance guarantee that is polynomial in terms of the input size.
3. **Randomized Approximation**: These algorithms incorporate randomness to achieve good solutions with high probability.

**General Steps in Approximation Algorithms:**

1. **Relaxation**: Involves relaxing some constraints of the problem to make it easier to solve. For example, in integer programming, the problem might be relaxed to a linear programming problem where variables are allowed to take fractional values.
2. **Greedy Techniques**: Many approximation algorithms rely on greedy strategies, where you make the locally optimal choice at each step with the hope that it leads to a globally good solution.
3. **Randomization**: Some approximation algorithms employ randomization, making probabilistic decisions that lead to high-quality solutions with high probability.

**Examples of Approximation Algorithms for NP-Hard Problems**

1. **Travelling Salesperson Problem (TSP)**:
   * **Problem**: Given a set of cities, find the shortest possible tour that visits each city exactly once and returns to the starting city.
   * **Approximation Algorithm (2-Approximation Algorithm)**:
     + If the cost function satisfies the **triangle inequality**, the **2-approximation algorithm** can be used.
     + **Steps**:
       1. Construct a **Minimum Spanning Tree (MST)** of the cities.
       2. Find a **Minimum Weight Perfect Matching (MWPM)** for cities with odd degree in the MST.
       3. Combine the MST and MWPM to form an Eulerian circuit.
       4. Convert the Eulerian circuit to a Hamiltonian circuit (visiting each city exactly once).
     + **Performance**: This algorithm guarantees that the tour length is at most **twice** the optimal length (a 2-approximation).
2. **Vertex Cover Problem**:
   * **Problem**: Given a graph, find the smallest set of vertices such that every edge is incident to at least one vertex in the set.
   * **Approximation Algorithm (Greedy)**:
     + Start with an empty vertex cover. Repeatedly pick an uncovered edge and add both its endpoints to the vertex cover.
     + **Performance**: This greedy algorithm guarantees a solution that is at most **2 times** the size of the optimal vertex cover (a 2-approximation).
3. **Set Cover Problem**:
   * **Problem**: Given a collection of sets and a universal set, find the smallest sub-collection of sets that covers all elements in the universal set.
   * **Approximation Algorithm (Greedy)**:
     + Select the set that covers the most number of uncovered elements. Repeat this process until all elements are covered.
     + **Performance**: This greedy algorithm provides a solution within a **logarithmic factor** of the optimal solution, specifically **O(log n)**, where **n** is the number of elements to cover.
4. **Knapsack Problem (0/1 Knapsack)**:
   * **Problem**: Given a set of items with weights and values, and a knapsack with a weight limit, find the set of items to include in the knapsack that maximizes the total value without exceeding the weight limit.
   * **Approximation Algorithm (Greedy based on value-to-weight ratio)**:
     + Sort the items by their **value-to-weight ratio** and take the items with the highest ratio until the knapsack is full.
     + **Performance**: This is not a fully optimal solution but can provide a good solution depending on the problem instance.

**Types of Approximation Guarantees:**

1. **Constant Factor Approximation**:
   * A **c-approximation algorithm** guarantees that the solution will be within a factor of **c** of the optimal solution, where **c** is a constant.
   * Example: The **2-approximation algorithm** for the **TSP** guarantees that the solution is at most 2 times the optimal solution.
2. **Logarithmic Approximation**:
   * In some problems, a **logarithmic approximation** is possible, which means the approximation ratio is related to the logarithm of the input size.
   * Example: The greedy algorithm for the **Set Cover Problem** has an approximation ratio of **O(log n)**, where **n** is the number of elements to be covered.
3. **Probabilistic Approximation**:
   * These algorithms employ randomness to find good solutions with high probability.
   * Example: **Randomized Algorithms** for the **Max-Cut Problem** can approximate the maximum cut in a graph within a factor of **0.878** with high probability.
4. **Write an algorithm to solve the travelling salesman problem and prove that it is a 2 time approximation algorithm.**

**Travelling Salesman Problem (TSP) and 2-Approximation Algorithm**

The **Travelling Salesman Problem (TSP)** is a classical optimization problem where you are given a set of cities, and the goal is to find the shortest possible route that visits each city exactly once and returns to the starting city. The problem is known to be NP-hard, meaning there is no known polynomial-time algorithm that guarantees an optimal solution for all instances of the problem.

However, when the **triangle inequality** holds (i.e., the direct path between two cities is always shorter than or equal to any indirect path), we can use an approximation algorithm that guarantees a solution that is at most **twice** the length of the optimal solution. This is known as the **2-Approximation Algorithm** for TSP.

**2-Approximation Algorithm for TSP**

The **2-Approximation Algorithm** works in two main steps: constructing a Minimum Spanning Tree (MST) and then finding an Eulerian circuit.

**Steps of the 2-Approximation Algorithm for TSP:**

1. **Construct the Minimum Spanning Tree (MST)**:
   * Use an algorithm such as **Prim’s Algorithm** or **Kruskal’s Algorithm** to construct an MST of the cities. An MST is a tree that connects all cities with the minimum possible total edge weight, and no cycles are allowed. The MST guarantees that the total distance to visit all cities will be minimized.
2. **Create an Eulerian Circuit**:
   * An Eulerian circuit is a circuit that visits every edge exactly once and returns to the starting point. However, in the MST, some cities may have an odd degree (number of edges incident to the city), which prevents an Eulerian circuit. To ensure an Eulerian circuit, you need to:
     1. Find the vertices with an **odd degree** (cities with an odd number of connections in the MST).
     2. **Pair up** these vertices using a **minimum-weight perfect matching (MWPM)**. This is a step where you connect the odd-degree vertices with edges that have the minimum cost, which makes sure that every vertex has an even degree.
3. **Form the Eulerian Circuit**:
   * After adding the matching edges, all vertices will have even degrees, making the graph Eulerian. Now, find an Eulerian circuit in the graph. This can be done using a **Depth-First Search (DFS)** or **Hierholzer’s algorithm**, which will ensure you visit every edge exactly once.
4. **Convert Eulerian Circuit to a Hamiltonian Circuit**:
   * The Eulerian circuit might visit some cities more than once. You need to remove the repeated cities to form a Hamiltonian circuit (a path that visits every city exactly once). This can be done by skipping cities that have already been visited in the Eulerian circuit.
5. **Return the Approximate Solution**:
   * The resulting path will be a **Hamiltonian circuit** that visits each city exactly once and returns to the starting city. This circuit is guaranteed to be no worse than twice the length of the optimal TSP solution.

**2-Approximation Theorem Proof**

**Theorem: The 2-Approximation Algorithm for TSP provides a solution that is at most twice the length of the optimal solution.**

**Proof:**

1. **Let the optimal solution (OPT)** represent the shortest possible tour that visits all cities exactly once and returns to the start.
2. **Let the MST (MST\_cost)** represent the cost of the Minimum Spanning Tree.
3. **Step 1**: The MST connects all cities with the minimum possible edge weight. The cost of the MST, **MST\_cost**, is a lower bound for the cost of the optimal TSP solution. This is because the MST has no cycles, and the shortest possible tour must connect the cities in a way that respects the distances between them.

Thus, we have:

\text{MST\_cost} \leq \text{OPT}

1. **Step 2**: The 2-Approximation Algorithm adds edges to the MST in the form of a **minimum-weight perfect matching**. This matching ensures that every vertex has an even degree, allowing the creation of an Eulerian circuit. The cost of these additional edges is at most **MST\_cost** because we are simply adding edges that connect odd-degree vertices in the MST.
2. **Step 3**: The Eulerian circuit, created by the MST and the matching, visits each edge in the graph exactly once. The total cost of this Eulerian circuit is:

\text{Eulerian Circuit Cost} = 2 \times \text{MST\_cost}

This is because the circuit traverses every edge in the MST twice: once in each direction.

1. **Step 4**: After the Eulerian circuit is formed, we remove the repeated cities to create a Hamiltonian circuit. The total length of the Hamiltonian circuit is at most **twice the length of the optimal solution** because we are simply skipping repeated cities in the Eulerian circuit, which only reduces the length of the tour. Thus, the cost of the final Hamiltonian circuit (the approximation) is at most:

\text{Approximate Cost} = 2 \times \text{MST\_cost} \leq 2 \times \text{OPT}

1. **Conclusion**: Therefore, the 2-Approximation Algorithm provides a solution whose cost is at most **twice** the optimal cost. This guarantees that the solution is within a factor of **2** of the optimal solution.

**Algorithm:**

python

Copy

# 2-Approximation Algorithm for TSP

def prim\_mst(graph):

# Use Prim's Algorithm to construct MST

n = len(graph)

mst = [False] \* n

min\_edge = [float('inf')] \* n

parent = [-1] \* n

min\_edge[0] = 0

for \_ in range(n):

u = min((i for i in range(n) if not mst[i]), key=lambda x: min\_edge[x])

mst[u] = True

for v in range(n):

if graph[u][v] != 0 and not mst[v] and graph[u][v] < min\_edge[v]:

min\_edge[v] = graph[u][v]

parent[v] = u

return parent, min\_edge

def create\_eulerian\_circuit(mst\_parent, graph):

# Create Eulerian Circuit by adding a minimum-weight perfect matching

# This part can be simplified in the context of the 2-approximation description

pass # Implement finding and adding perfect matching edges

def remove\_repeated\_cities(eulerian\_circuit):

# Remove repeated cities to form a Hamiltonian circuit

return list(dict.fromkeys(eulerian\_circuit)) # This will remove duplicates

def tsp\_2\_approximation(graph):

# Step 1: Construct the MST

mst\_parent, mst\_cost = prim\_mst(graph)

# Step 2: Create the Eulerian Circuit (using MST and perfect matching)

eulerian\_circuit = create\_eulerian\_circuit(mst\_parent, graph)

# Step 3: Remove repeated cities to get the Hamiltonian circuit

hamiltonian\_circuit = remove\_repeated\_cities(eulerian\_circuit)

return hamiltonian\_circuit