AOA theory

Computer Engineering (University of Mumbai)

Module 1

In computer science, P, NP, NP-Hard, and NP-Complete are complexity classes used to classify problems based on their computational difficulty. Here's a brief definition of each:

1. **P**: P stands for "polynomial time" and refers to the set of decision problems that can be solved by a deterministic Turing machine in polynomial time. In other words, a problem is in P if there exists an algorithm that solves it in a reasonable amount of time, where the time it takes to solve the problem is proportional to a polynomial function of the problem size.

2. **NP**: NP stands for "nondeterministic polynomial time" and refers to the set of decision problems that can be verified by a deterministic Turing machine in polynomial time. In other words, a problem is in NP if there exists a polynomial-time algorithm that can verify whether a given solution to the problem is correct or not. However, it's currently unknown whether all problems in NP can also be solved in polynomial time.

3. **NP-Hard**: A problem is NP-Hard if every problem in NP can be reduced to it in polynomial time. In other words, if there exists a polynomial-time algorithm that can transform any instance of an NP problem into an instance of the NP-Hard problem, then the NP-Hard problem is at least as hard as any problem in NP.

4. **NP-Complete**: A problem is NP-Complete if it is both in NP and NP-Hard. In other words, it's a problem that can be verified in polynomial time and every problem in NP can be reduced to it in polynomial time. NP-Complete problems are considered to be among the most difficult problems in computer science, and no polynomial-time algorithm is currently known for solving them.

Overall, these complexity classes provide a useful way to understand the relative difficulty of computational problems and to identify which problems are likely to be tractable with efficient algorithms.

Asymptotic notation is a mathematical notation used to describe the behavior of a function as the input size tends towards infinity. It is commonly used in computer science to analyze the time and space complexity of algorithms. There are three main types of asymptotic notation:

- Big O notation (O): This notation is used to describe the upper bound of a function's growth rate. For example, if an algorithm has a time complexity of O(n^2), it means that the worst-case time required by the algorithm grows no faster than a quadratic function of the input size n.

- Omega notation (Ω): This notation is used to describe the lower bound of a function's growth rate. For example, if an algorithm has a time complexity of Ω(n^2), it means that the best-case time required by the algorithm grows no slower than a quadratic function of the input size n.

- Theta notation (Θ): This notation is used to describe the tight bound of a function's growth rate. For example, if an algorithm has a time complexity of Θ(n^2), it means that the worst-case and best-case time required by the algorithm both grow at a quadratic rate of the input size n.

Asymptotic notation allows us to compare the efficiency of different algorithms and to predict how they will perform as the input size grows larger. It is a powerful tool for analyzing the time and space complexity of algorithms, and is used extensively in algorithm design and analysis.

Module 2

Selection Sort:

Repeat steps 2-4 for i = 0 to n-1, where n is the length of the array.
Find the index of the smallest element in the unsorted part of the array starting from i.
Swap the smallest element with the first element in the unsorted part of the array.
Continue until all elements are sorted.

Insertion Sort:

Repeat steps 2-4 for i = 1 to n-1, where n is the length of the array.
Pick the ith element and insert it into its correct position in the sorted part of the array (i.e., all elements to the left of the ith element).
Continue until all elements are sorted.

Both have time complexity = **O(n^2)**

Merge Sort:

Divide the array into two halves.
Recursively sort the two halves using merge sort.
Merge the two sorted halves into a single sorted array.

Quick Sort:

Choose a pivot element (usually the last element in the array).
Partition the array into two sub-arrays, one with elements less than the pivot, and one with elements greater than the pivot.
Recursively apply quick sort to the two sub-arrays.
Combine the two sorted sub-arrays.

Both have time complexity = **O(n log n)**

Algorithm to find the minimum and maximum elements in an array:

1. Initialize the minimum and maximum variables to the first element of the array.
2. Traverse the array from the second element to the end.
3. For each element, compare it with the current minimum and maximum values. If it is smaller than the current minimum, update the minimum variable. If it is larger than the current maximum, update the maximum variable.
4. Once the entire array has been traversed, the minimum and maximum variables will hold the smallest and largest values in the array.

```
min = arr[0]
max = arr[0]

for i = 1 to n-1 do
   if arr[i] < min then
      min = arr[i]
   else if arr[i] > max then
      max = arr[i]
```

return min, max

Module 3

Dijkstra's algorithm is a shortest-path algorithm used to find the shortest path between two nodes in a weighted graph. It works by iteratively selecting the node with the lowest distance estimate and updating the distance estimates of its neighbors. The algorithm terminates when the destination node has been marked as visited, or when all nodes have been visited.

Here's the algorithm in more detail:

1. Create a set of unvisited nodes and set the distance of all nodes to infinity, except the source node, which is set to 0.
2. While the set of unvisited nodes is not empty:
    a. Select the node with the smallest distance estimate and mark it as visited.
    b. For each neighbor of the selected node that is still unvisited:
        i. Calculate the tentative distance to that neighbor by adding the weight of the edge connecting the nodes to the current distance estimate of the selected node.
        ii. If the tentative distance is less than the current distance estimate of the neighbor, update the neighbor's distance estimate to the tentative distance.
3. Once the destination node has been marked as visited, the algorithm can terminate. The shortest path can be found by tracing back from the destination node to the source node, using the shortest distance estimates and the parent nodes that were updated during the algorithm.

Dijkstra's algorithm has a time complexity of **O((V+E)logV)** using a binary heap, where V is the number of nodes and E is the number of edges in the graph.

Formula for relaxation:-

```
if distance[u] + weight(u, v) < distance[v]:
    distance[v] = distance[u] + weight(u, v)
```

The **fractional knapsack problem** is a classic optimization problem that involves selecting items with the highest value to weight ratio to fill a knapsack of limited capacity. Unlike the 0/1 knapsack problem, which only allows for selecting or rejecting an item entirely, the fractional knapsack problem allows for selecting a fraction of an item.

Here's the algorithm for solving the fractional knapsack problem:

1. Calculate the value-to-weight ratio of each item.
2. Sort the items in descending order of their value-to-weight ratio.
3. Initialize the maximum value and the remaining capacity of the knapsack to 0 and the total capacity of the knapsack, respectively.
4. For each item, starting from the highest value-to-weight ratio:
   - If the item can fit entirely in the remaining capacity of the knapsack, add its value to the maximum value and subtract its weight from the remaining capacity of the knapsack.
   - Otherwise, add a fraction of the item that can fit into the remaining capacity of the knapsack and adjust the maximum value and remaining capacity of the knapsack accordingly.
   - If the knapsack is full, terminate the loop.
5. Return the maximum value.

The time complexity of the fractional knapsack algorithm is **O(n log n)**, where n is the number of items, due to the sorting step.

Job sequencing with deadlines is a problem where we have `n` jobs, each with a certain profit and a deadline by which it needs to be completed. The goal is to find a sequence of jobs to maximize the total profit while meeting all the deadlines. The algorithm to solve this problem can be described as follows:

1. Sort the jobs in decreasing order of their profit.
2. Initialize an array `slot` of size `n` with all elements set to `False`. This array will keep track of which time slots are available for scheduling the jobs.
3. For each job `j` in the sorted list of jobs:
   - Starting from the deadline of job `j`, find the first available time slot in the `slot` array by traversing the array from right to left.
   - If a time slot is found, mark it as occupied by setting the corresponding element in the `slot` array to `True`. Otherwise, skip the job `j`.
4. Calculate the total profit of all the scheduled jobs and return the sequence of scheduled jobs.

The time complexity of this algorithm is O(n^2), which is dominated by the time it takes to find an available time slot for each job. This can be improved to **O(n log n)** by using a data structure such as a priority queue or a binary search tree to store the available time slots.

1. **Kruskal's Algorithm:**
Kruskal's algorithm is a greedy algorithm that works by iteratively adding the smallest edge that doesn't create a cycle until all nodes are connected. It has the time complexity of **O(E log E)**, where E is the number of edges in the graph.

Here are the steps of the algorithm:

- Sort all the edges of the graph in non-decreasing order of their weights.
- Initialize an empty set for the minimum spanning tree and a set for keeping track of the connected components of the graph.
- For each edge in the sorted list of edges:
    - If the edge connects two different connected components, add it to the minimum spanning tree and merge the two components.
    - If the edge connects two nodes in the same connected component, skip it.
- Once all edges have been processed, the minimum spanning tree will be the set of edges that were added to it.

2. **Prim's Algorithm:**

Prim's algorithm is also a greedy algorithm that works by iteratively adding the smallest edge that connects a visited node to an unvisited node until all nodes are connected. It has the time complexity of **O(E log V)**, where E is the number of edges in the graph and V is the number of nodes.

Here are the steps of the algorithm:

- Initialize an empty set for the minimum spanning tree, a set for visited nodes, and a priority queue for edges sorted by weight.
- Select an arbitrary node to start from and add it to the visited set.
- Add all edges connected to the selected node to the priority queue.
- While the priority queue is not empty:
    - Pop the smallest edge from the priority queue.
    - If the edge connects a visited node to an unvisited node, add it to the minimum spanning tree and mark the unvisited node as visited.
    - Add all edges connected to the newly visited node to the priority queue.
- Once all nodes have been visited, the minimum spanning tree will be the set of edges that were added to it.

Module 4

What is DP?

Dynamic programming is an algorithmic technique used to solve problems by breaking them down into smaller subproblems and solving each subproblem only once, storing the solution in memory so that it can be reused later. This can lead to significant improvements in time complexity over naive approaches that solve the same subproblems multiple times.

The key idea behind dynamic programming is to avoid redundant computations by storing the solutions to subproblems in memory and reusing them later. This is often done using a table or array, where the values in the table correspond to the solutions to subproblems.

There are two main approaches to dynamic programming: top-down and bottom-up. Top-down dynamic programming, also known as memoization, starts by breaking down the original problem into smaller subproblems and then recursively solving them. The solutions to the subproblems are stored in memory so that they can be reused later. Bottom-up dynamic programming, also known as tabulation, starts by solving the smallest subproblems first and then building up to the larger subproblems. The solutions to the subproblems are stored in a table or array so that they can be reused later.

Dynamic programming can be used to solve a wide range of problems, including optimization problems, pathfinding problems, and combinatorial problems. Some classic examples of problems that can be solved using dynamic programming include the Fibonacci sequence, the longest common subsequence, and the knapsack problem.

Overall, dynamic programming is a powerful algorithmic technique that can be used to solve complex problems efficiently by breaking them down into smaller subproblems and solving each subproblem only once.

Multistage graph is a directed acyclic graph (DAG) in which nodes are arranged in layers or stages, and edges only connect nodes between different stages. In other words, it's a graph with a natural partition of nodes into stages, such that all edges go from nodes in one stage to nodes in the next stage.

A multistage graph is often used to represent a process that can be broken down into a sequence of stages, where each stage represents a set of tasks that can be completed in parallel. For example, a manufacturing process, a software development process, or a transportation network can be modeled as a multistage graph.

The multistage graph can be solved using dynamic programming algorithms such as the Bellman-Ford algorithm or the Floyd-Warshall algorithm. These algorithms can find the optimal solution of the multistage graph by recursively solving subproblems from the last stage to the first stage, and storing the solutions in a table or array. The time complexity of solving a multistage graph using dynamic programming is usually **O(n^2)**, where n is the number of nodes in the graph.

Bellman-Ford algorithm is used to find the shortest path from a source vertex to all other vertices in a weighted graph, including graphs with negative weight edges. The algorithm works by repeatedly relaxing all edges in the graph, i.e., checking if the distance to a vertex can be improved by going through another vertex, and updating the distance accordingly.

Here is the algorithm for the Bellman-Ford algorithm:

1. Initialize distance to all vertices as infinity except the source vertex, which is set to 0.
2. Repeat the following for V-1 times, where V is the total number of vertices in the graph:
   a. For each edge (u, v) with weight w, if the distance to u plus the weight of the edge is less than the current distance to v, update the distance to v with the new distance.
3. Check for negative weight cycles. To do this, repeat the following for all edges (u, v) with weight w:
   a. If the distance to u plus the weight of the edge is less than the current distance to v, there is a negative weight cycle in the graph.

The time complexity of the Bellman-Ford algorithm is **O(VE)**, where V is the number of vertices and E is the number of edges in the graph. The algorithm needs to relax all edges in the graph V-1 times, and each relaxation takes O(E) time.

Note that the Bellman-Ford algorithm can be optimized by stopping the algorithm early if no distances are updated during an iteration. This can be done by keeping track of a flag that is set to true if any distance is updated during an iteration, and false otherwise. If the flag is false after an iteration, we can stop the algorithm early, as no more updates can be made.

Module 5

What is Backtracking & Branch-Bound approach ?

Backtracking and branch and bound are two popular techniques used to solve combinatorial optimization problems.

Backtracking is a brute force algorithmic technique that involves exploring all possible solutions to a problem by systematically building candidates, and backing up when a solution cannot be completed. Backtracking algorithms are based on recursive depth-first search techniques and can be very efficient for problems with small solution spaces. However, they can become inefficient for large problems with many possible solutions.

On the other hand, branch and bound is a more advanced algorithmic technique that improves on the efficiency of backtracking by reducing the size of the search space. Branch and bound algorithms systematically divide the problem into smaller sub-problems, or branches, and use a bounding function to eliminate portions of the search space that cannot contain the optimal solution. By pruning the search tree, branch and bound algorithms can be more efficient than backtracking for problems with large solution spaces.

Longest Common Subsequence (LCS) problem is a classic problem in computer science that involves finding the longest subsequence that is common to two given sequences. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Here is the algorithm for solving the LCS problem:

1. Initialize a 2D array, `dp`, with dimensions `m+1` x `n+1`, where `m` and `n` are the lengths of the two sequences. `dp[i][j]` will represent the length of the LCS of the first `i` elements of the first sequence and the first `j` elements of the second sequence.

2. Set `dp[0][j]` and `dp[i][0]` to 0 for all `i` in the range [0, m] and `j` in the range [0, n], since there is no common subsequence between an empty sequence and any other sequence.

3. For each element `i` in the range [1, m]:

   a. For each element `j` in the range [1, n]:

      i. If the `i`-th element of the first sequence is equal to the `j`-th element of the second sequence, set `dp[i][j]` to `dp[i-1][j-1] + 1`, since the current element can be included in the LCS.

      ii. Otherwise, set `dp[i][j]` to the maximum value of `dp[i-1][j]` and `dp[i][j-1]`, since the current element cannot be included in the LCS.

4. The length of the LCS is `dp[m][n]`.

5. To reconstruct the LCS, start from the bottom-right corner of `dp` and work backwards. If the current element is equal to the element above and to the left, it is part of the LCS. Add it to the LCS and move diagonally to the next element. Otherwise, move in the direction of the greater value in `dp`.

The time complexity for the Longest Common Subsequence (LCS) algorithm is **O(mn)**, where m and n are the lengths of the two input sequences.

N-Queens problem is a classic problem in computer science and combinatorial optimization. It involves placing N queens on an N x N chessboard so that no two queens threaten each other. In other words, no two queens are allowed to be placed in the same row, column, or diagonal.

Here is the algorithm for solving the N-Queens problem:

1. Initialize an empty N x N chessboard.
2. Place a queen in the first column of the first row.
3. For each subsequent column:
   a. For each row in the column:
   i. If the queen can be placed in the current row without threatening any other queens, place it and move to the next column.
   ii. If there are no valid rows for the queen in the current column, backtrack to the previous column and try the next row.
4. If a solution has been found, return the chessboard. Otherwise, backtrack until all possible configurations have been explored

The backtracking algorithm, which is the most common approach for solving the N-Queens problem, has a time complexity of **O(N!)**, where N is the size of the chessboard.

Module 6

- **Naïve string-matching algorithm:** This algorithm checks every possible position of the pattern in the text by comparing each character of the pattern with the corresponding character in the text. It has a time complexity of **O(mn)**, where m is the length of the pattern and n is the length of the text.

- **The Rabin-Karp algorithm:** This algorithm uses hash values to efficiently compare the pattern with the text. It computes hash values for the pattern and all possible substrings of the text and compares them to find matches. It has a time complexity of **O(m + n)**, where m is the length of the pattern and n is the length of the text.

- **The Knuth-Morris-Pratt algorithm:** This algorithm preprocesses the pattern to find the longest proper prefix that is also a suffix for each prefix of the pattern. It then uses this information to efficiently skip over unnecessary character comparisons in the text. It has a time complexity of **O(m + n)**, where m is the length of the pattern and n is the length of the text.