Q1)

A)

The algorithm is designed to traverse a graph in a depth-first manner. The graph is represented as a dictionary in which the keys are nodes and the values are lists of nodes that are connected to the key node. The algorithm uses several helper functions to traverse the graph. The Fun function is the main part of the algorithm, and it calls the other functions as needed.

The **Fun** function takes a **node** and the **graph** as arguments, and traverses the graph starting from the given node. It first checks if the node has any **non-visited parents**. If it does, it calls itself recursively with the first non-visited parent as the argument. If the node has no non-visited parents, it adds the node to the visited list and then checks if the node has any children. If it does, it calls itself recursively with the first child as the argument. If the node has no children, the function returns.

The other functions in the algorithm are used to support the Fun function. The **IsItVisited** function takes a node as an argument and checks whether that node has been visited by the algorithm before. The **GetParents** function takes a child node and the graph as arguments, and returns a list of parent nodes for the given child node. The **DoYouHaveNonVisitedParent** function takes a node and the graph as arguments, and checks whether the node has any parents that have not been visited yet. The **GetChild** function takes a parent node and the graph as arguments, and returns the first child node of the parent. The **FindStartPoint** function takes the graph as an argument, and returns a list of nodes that have no parents.

The time complexity of the algorithm is **O(n),** where n is the number of nodes in the graph, because the Fun function traverses the graph once and each of the other functions has a time complexity of O(n).

B)

This code is an implementation of a topological sort algorithm using a breadth-first search approach. It takes a graph as input, represented as a dictionary in which the keys are nodes and the values are lists of nodes that are connected to the key node.

The algorithm first creates a list of in-degree counts for each node in the graph. It then uses this list to create a queue of nodes with an in-degree count of 0. It then performs a breadth-first search on the graph, starting with the nodes in the queue, and appending each visited node to a list of sorted nodes.

If the algorithm is able to visit all nodes in the graph, it returns the sorted list of nodes. If it is unable to visit all nodes, it returns None, indicating that the graph contains a cycle and cannot be topologically sorted.

The time complexity of this algorithm is O(|V| + |E|), where |V| is the number of nodes in the graph and |E| is the number of edges. This is because the algorithm performs a breadth-first search on the graph, which has a time complexity of O(|V| + |E|).

Q2)

If n = 0, return 1 as the result.

If n is even, set x = a^(n/2) and return x * x.

If n is odd, set x = a^((n-1)/2) and return a * x * x.

This algorithm has a time complexity of **O(log n)** because in each iteration, the value of n is halved until it reaches 0. This means that the number of iterations required to reach the base case is logarithmic in the value of n.

Q3)

The SolveSudoku() function is the main function for solving a Sudoku puzzle. It uses a recursive backtracking approach and calls the CanPlace() and solve() functions to determine whether a given number can be placed at a given position in the grid. The printGrid() function is used to print the grid, and the test() function is used to test the correctness of the SolveSudoku() function. This solution has a time complexity of O(9^n), where n is the number of empty cells in the grid. This is because the function must try every possible combination of numbers for the empty cells in order to find a solution.

Q4)

Insertion sort:

The first element, 6, is considered to be sorted.

The second element, 8, is inserted into the sorted array by shifting 6 to the right and placing 8 in the first position. The array is now {8, 6, 9, 8, 3, 3, 12}.

The third element, 9, is inserted into the sorted array by shifting 8 and 6 to the right and placing 9 in the second position. The array is now {8, 9, 6, 8, 3, 3, 12}.

The fourth element, 8, is inserted into the sorted array by shifting 9 and 6 to the right and placing 8 in the third position. The array is now {8, 9, 8, 6, 3, 3, 12}.

The fifth element, 3, is inserted into the sorted array by shifting 8, 9, and 6 to the right and placing 3 in the first position. The array is now {3, 8, 9, 8, 6, 3, 12}.

The sixth element, 3, is inserted into the sorted array by shifting 8, 9, and 8 to the right and placing 3 in the second position. The array is now {3, 3, 8, 9, 8, 6, 12}.

The seventh element, 12, is inserted into the sorted array by shifting 8, 9, and 8 to the right and placing 12 in the seventh position. The array is now {3, 3, 8, 9, 8, 6, 12}.

Quick sort:

The pivot element is chosen to be the first element, 6.

The array is partitioned into two subarrays, one containing elements less than or equal to 6, and the other containing elements greater than 6. The subarrays are {3, 3} and {8, 9, 8, 12}, respectively.

The two subarrays are sorted recursively using the same process.

The sorted subarrays are combined to form the final sorted array: {3, 3, 6, 8, 8, 9, 12}.

Bubble sort:

The first and second elements, 6 and 8, are compared. Since 6 is less than 8, they remain in the same order. The array is now {6, 8, 9, 8, 3, 3, 12}.

The second and third elements, 8 and 9, are compared. Since 8 is less than 9, they remain in the same order. The array is now {6, 8, 9, 8, 3, 3, 12}.

The third and fourth elements, 9 and 8, are compared. Since 9 is greater than 8, they are swapped. The array is now {6, 8, 8, 9, 3, 3, 12}.

The fourth and fifth elements, 8 and 3, are compared. Since 8 is greater than 3, they are swapped. The array is now {6, 8, 3, 9, 8, 3, 12}.

The fifth and sixth elements, 9 and 8, are compared. Since 9 is greater than 8, they are swapped. The array is now {6, 8, 3, 8, 9, 3, 12}.

The sixth and seventh elements, 8 and 3, are compared. Since 8 is greater than 3, they are swapped. The array is now {6, 8, 3, 3, 9, 8, 12}.

The seventh and eighth elements, 9 and 8, are compared. Since 9 is greater than 8,

Insertion sort: -> stable

Quick sort: -> not stable

Bubble sort: -> stable

Q5

A) Exhaustive search, also known as a brute force approach, is a method used to find the best solution to a problem by trying all possible combinations. This algorithm is commonly used for problems such as the Knapsack and traveling salesperson problem. While it is effective, it can be time-consuming compared to other search methods. Brute force search is a non-uniform search that involves trying every possible option, even if it is unlikely to be the correct solution. It can also be used for finding a substring within a larger piece of text.

B) Caesar's Cipher and AES are both encryption techniques used to protect sensitive information. Caesar's Cipher is a simpler method, but it is vulnerable to brute force attacks. This means that, in the worst case scenario, the password can be decrypted by trying every possible combination of letters in the alphabet. In contrast, AES is much more secure against brute force attacks. It uses a variety of encryption keys, such as 256-

bit and 128-bit keys, to protect data. For this reason, AES is generally considered a better option than Caesar's Cipher for secure systems.

C) The naive primality test is an algorithm that is exponential in size when compared to the input. The size of the input is typically measured in bits or bytes. In the naive solution, the number of bits required to store the number n grows exponentially with n. This results in an exponential growth rate for the algorithm as a whole. Because of this, the naive primality test can be inefficient for large inputs.

HACI HASAN SAVAN

1901042704