Merve horuz-1801042651

CSE 321 HW3 report

Q1)

a) This algorithm is an implementation of the depth-first search (DFS) algorithm for traversing a graph. The algorithm begins by initializing an empty set **visited** to keep track of the nodes that have been visited, and an empty list **ans** to store the traversal order of the nodes.

Next, the algorithm defines a helper function **DFSUtil** that takes a node **v** as input and performs the following steps:

1. Adds **v** to the set of visited nodes.
2. For each neighbor **neighbor** of **v**, checks if the neighbor has been visited. If the neighbor has not been visited, calls **DFSUtil** recursively on the neighbor.
3. Adds **v** to the beginning of the list **ans** (i.e., inserts **v** at the 0th index of the list). This step ensures that the nodes are added to the list in the order in which they are traversed by the DFS algorithm.

After defining the **DFSUtil** function, the algorithm iterates over all nodes in the graph, and calls **DFSUtil** on each node that has not been visited yet. This step ensures that all nodes in the graph are traversed, even if the graph is not connected.

Finally, the algorithm returns the list **ans** containing the traversal order of the nodes.

The time complexity of this algorithm is O(V+E), where V is the number of vertices in the graph and E is the number of edges. This is because the algorithm visits each vertex and each edge exactly once.

b) This algorithm uses a modified version of breadth-first search (BFS) to compute a topological sorting of a directed graph. A topological sorting of a directed graph is a linear ordering of the vertices in the graph such that, for every edge uv from vertex u to vertex v, u comes before v in the ordering.

The algorithm starts by initializing a dictionary to store the in-degree of each vertex in the graph. The in-degree of a vertex is the number of incoming edges to that vertex. Initially, all in-degrees are set to zero.

Next, the algorithm iterates over all the vertices and their neighbors in the graph, and increments the in-degree of each neighbor of the vertex. This allows us to compute the in-degree of each vertex in the graph.

Once the in-degrees have been computed, the algorithm initializes a queue with all the vertices that have an in-degree of zero. This is because vertices with no incoming edges can be added to the topological sorting in any order.

The algorithm then repeatedly removes a vertex from the front of the queue, adds it to the topological sorting, and decrements the in-degree of each of its neighbors. If a neighbor's in-degree becomes zero as a result, it is added to the queue. This continues until the queue is empty, at which point a topological sorting of the graph has been computed.

The time complexity of this algorithm is O(V+E), where V is the number of vertices in the graph and E is the number of edges. This is because the algorithm iterates over all the vertices and edges in the graph exactly once.

Q2)

This algorithm calculates the value of a to the power of n, where a is a real number and n is a non-negative integer. The algorithm uses a divide-and-conquer approach to compute the result more efficiently.

First, the algorithm checks if n is zero. If so, it returns 1, since any non-zero number raised to the power of zero is 1.

Next, the algorithm recursively calculates the value of a to the power of n//2 (i.e., n divided by 2 and rounded down to the nearest integer), and assigns the result to the variable half. This is done by calling the pow() function again with the same arguments.

The algorithm then checks if n is even or odd. If n is even, it returns half * half, which is equivalent to (a to the power of n//2) * (a to the power of n//2). This is because, when n is even, a to the power of n is equal to (a to the power of n//2) squared.

If n is odd, on the other hand, the algorithm returns a * half * half, which is equivalent to a * (a to the power of n//2) * (a to the power of n//2). This is because, when n is odd, a to the power of n is equal to a * (a to the power of n//2) squared.

The time complexity of this algorithm is O(log n), since the input value n is halved at each recursive call, and the algorithm makes at most log n calls before n becomes zero. This is more efficient than the naive approach of multiplying a by itself n times, which has a time complexity of O(n).

Q3)

This algorithm solves a Sudoku puzzle by finding a valid assignment of values to the empty cells in the grid that satisfies the constraints of the puzzle.

The Sudoku puzzle is represented as a grid of size 9x9, where each cell can hold a value between 1 and 9. Some of the cells in the grid are initially filled with values, while others are empty. The goal of the puzzle is to fill in the empty cells with values such that:

Each row of the grid contains each of the digits 1 through 9 exactly once.

Each column of the grid contains each of the digits 1 through 9 exactly once.

Each of the nine 3x3 sub-grids (or "boxes") contains each of the digits 1 through 9 exactly once.

The algorithm uses a recursive backtracking approach to find a solution to the puzzle. It starts by calling the findNextEmptyCell() function, which iterates over the cells in the grid and returns the coordinates of the first empty cell it finds (i.e., the cell with a value of 0). If no empty cells are found, the function returns None, indicating that the puzzle has been solved.

If an empty cell is found, the algorithm enters a loop that tries assigning each of the values 1 through 9 to the empty cell. For each value, it first checks if the assignment is valid by calling the isValid() function. This function checks if the value appears in the same row, column, or 3x3 box as the empty cell, and returns False if it does. If the assignment is valid, the algorithm calls itself recursively with the updated grid to continue solving the puzzle.

If the recursive call returns True, indicating that the puzzle has been solved, the algorithm returns True and exits. Otherwise, the algorithm undoes the assignment by setting the value of the empty cell back to 0 and tries the next value. If all values have been tried and none of them leads to a solution, the algorithm returns False, indicating that no solution exists for the given grid.

The time complexity of this algorithm is O(9^n), where n is the number of empty cells in the grid. This is because the algorithm tries at most 9 different values for each empty cell, and the number of empty cells can be up to 81 (the maximum number of cells in a Sudoku grid). This is a relatively high time complexity, but it is still feasible for small Sudoku puzzles with a reasonable number of empty cells.

Q4) $array$ = {6, 8, 9, 8, 3, 3, 12}

-Insertion sort-

Began with the record in index 0 in the sorted portion, and moving the record in index 1 to the left until it is sorted.

{6, 8, 9, 8, 3, 3, 12}

Now in index 2: It looks {6, 8} it is already sorted.

{6, 8, 9, 8, 3, 3, 12}

Now in index 3: It looks {6,8,9}, then swap index 2 and index 3.

{6, 8, 8, 9, 3, 3, 12}

Now in index 4: It looks {6,8,8,9}, then 3 is less than all of them swap index 4 to index 0.

{3, 6, 8, 8, 9, 3, 12}

Now in index 5: It looks {3,6,8,8,9}, then 3 is less than all of them except index 0, then swap index 4 to index 0.

{3, 3, 6, 8, 8, 9, 12}

Now in index 6: It looks {3, 3, 6, 8, 8, 9}, then 12 is bigger than all of them, then no swap.

{3, 3, 6, 8, 8, 9, 12}.

-quick sort-

1. Choose 9 as the pivot and rearrange the array: {6, 3, 3, 8, 8, 9, 12}
2. Recursively apply the above steps to the sub-arrays {6, 3, 3, 8} and {8, 9, 12}:
    a. Choose 3 as the pivot for the first sub-array and rearrange it: {3, 3, 6, 8}
    b. Choose 8 as the pivot for the second sub-array and rearrange it: {8, 9, 12}
3. Recursively apply the above steps to the sub-arrays {3, 3} and {6, 8} in the first sub-array:
    a. Choose 3 as the pivot for the first sub-array and rearrange it: {3, 3}
    b. Choose 6 as the pivot for the second sub-array and rearrange it: {6, 8}
4. The final sorted array is {3, 3, 6, 8, 8, 9, 12}.

-bubble sort-

{6, 8, 9, 8, 3, 3, 12}

First compares {6, 8} it is already sorted.

Compares {8, 9} already sorted.

Compares {9, 8} and swap them. Now array looks like {6, 8, 8, 9, 3, 3, 12}.

Compares {9, 3} and swap them. Now array looks like {6, 8, 8, 3, 9, 3, 12}.

Compares {9, 3} and swap them. Now array looks like {6, 8, 8, 3, 3, 9, 12}.

Compares {9, 12} already sorted. Now array looks like {6, 8, 8, 3, 3, 9, 12}.

12 fixed in index 6.

{6, 8, 8, 3, 3, 9, 12}

First compares {6, 8} it is already sorted.

Compares {8, 8} already sorted.

Compares {8, 3} and swap them. Now array looks like {6, 8, 3, 8, 3, 9, 12}

Compares {8, 3} and swap them. Now array looks like {6, 8, 3, 3, 8, 9, 12}

Compares {8, 9} already sorted.

9 fixed in index 5.

{6, 8, 3, 3, 8, 9, 12}

First compares {6, 8} it is already sorted.

Compares {8, 3} and swap them. Now array looks like {6, 3, 8, 3, 8, 9, 12}

Compares {8, 3} and swap them. Now array looks like {6, 3, 3, 8, 8, 9, 12}

Compares {8, 8} already sorted

Compares {8, 9} already sorted.

8 fixed in index 4.

{6, 3, 3, 8, 8, 9, 12}

First compares {6, 3} and swap them. Now array looks like {3, 6, 3, 8, 8, 9, 12}

Compares {6, 3} and swap them. Now array looks like {3, 3, 6, 8, 8, 9, 12}

Compares {6, 8} already sorted.

8 fixed in index 3.

{3, 3, 6, 8, 8, 9, 12}

First compares {3, 3} already sorted.

Compares {3, 6} already sorted.

6 fixed in index 2.

{3, 3, 6, 8, 8, 9, 12}

First compares {3, 3} already sorted.

3 fixed in index 1.

3 fixed in index 0.

Sorted array {3, 3, 6, 8, 8, 9, 12}.

Insertion sorting algorithm whether stable or not?

Insertion sort is a stable sorting algorithm. This means that when there are multiple items with the same value, their relative order will be preserved in the sorted array. For example, if the array to be sorted contains the values [3, 3, 6, 8, 8, 9, 12], the relative order of the two 3s will be preserved in the sorted array. This is because the algorithm maintains the relative order of items with the same value by always inserting them in the correct position relative to the other items with the same value.

Quick sorting algorithm whether stable or not?

Quick sort is not a stable sorting algorithm. For example, if the array to be sorted contains the values [3, 3, 6, 8, 8, 9, 12], the relative order of the two 3s may not be preserved in the sorted array. This is because the algorithm does not maintain the relative order of items with the same value and may end up placing them in different positions relative to each other in the sorted array.

Bubble sort algorithm whether stable or not?

Bubble sort is a stable sorting algorithm. For example, if the array to be sorted contains the values [3, 3, 6, 8, 8, 9, 12], the relative order of the two 3s will be preserved in the sorted array. This is because the algorithm maintains the relative order of items with the same value by always comparing and swapping adjacent items in the correct order.

Q5)

a) Brute force and exhaustive search are two different ways of solving problems. Brute force while solving a problem that involves trying all possible solutions or combinations until the correct one is found. Exhaustive search, on the other hand, is a specific algorithm that systematically checks all possible solutions or combinations until the correct one is found.

While both brute force and exhaustive search involve trying all possible solutions or combinations, the main difference between them is that brute force is a general term used to describe any simple, straightforward approach to solving a problem, while exhaustive search is a specific algorithm that follows a systematic approach to checking all possible solutions or combinations. Therefore, brute force and exhaustive search are related in that they both involve trying all possible solutions or combinations, but they are different in the way they approach solving a problem.

b) Caesar's Cipher is a simple encryption algorithm that is used to encode a message by shifting each letter in the message by a certain number of positions in the alphabet. For example, if the shift value is 3, then the letter "A" would be replaced by the letter "D", the letter "B" would be replaced by the letter "E", and so on.

AES uses a fixed block size of 128 bits and supports key sizes of 128, 192, or 256 bits. The algorithm works by taking the plaintext data as input and applying a series of transformations using the secret key to produce the encrypted ciphertext.

In order to decrypt the ciphertext, the same secret key must be used to apply the same transformations in the reverse order. This will produce the original plaintext data.

KeyExpansion

Initial round key addition:

AddRoundKey – each byte of the state is combined with a byte of the round key using bitwise xor.

SubBytes – a non-linear substitution step where each byte is replaced with another according to a lookup table.

ShiftRows – a transposition step where the last three rows of the state are shifted cyclically a certain number of steps.

MixColumns – a linear mixing operation which operates on the columns of the state, combining the four bytes in each column.

AddRoundKey

a. Final round:
    i. SubBytes
    ii. ShiftRows
    iii. AddRoundKey

Caesar's Cipher is vulnerable to brute force attacks. Because the shift value is known as 3, an attacker can replace the value at any position in the encoded text by shifting the 3 in the alphabet to its previous value and decode the message by simply shifting each letter in the encoded message.

AES is not vulnerable to brute force attacks. This is because the algorithm uses a secret key that is not publicly known. If an attacker tried to make brute force attack, then this could take billions of years.

c) The naive solution to primality testing grows exponentially because it involves trying all possible divisors from 2 to the square root of the given input for testing. For example, if the given input is 25, the naive solution would try all possible divisors from 2 to 5 to see if any of them divide the number evenly. If any of them divides the number evenly, then this number is not prime. This means that the number of divisors that need to be tried grows exponentially as the number being tested increases. For example, if the number being tested is 225, the naive solution would try all possible divisors from 2 to 15. 225 is a much larger number of divisors than the number of divisors that need to be tried for the input 25.