CSE321

Homework #3

Abdullah Çelik - 171044002

1)

a) Design:

- Initialize a list ‘my_list’ to store the topological ordering, and a set ‘my_set’ to keep track of the visited nodes.
- Start from any unvisited node in the DAG and perform a DFS on it.
- During the DFS, whenever a node is visited for the first time, add it to the list ‘my_list’.
- After the DFS has finished, ‘my_list’ will contain the nodes in a topological ordering, with the nodes that were visited first appearing earlier in the list.

This algorithm guarantees that the topological order will be correct because it follows the directed edges of the graph and adds nodes to the topological order only after all of their outgoing edges have been traversed. Additionally, the use of DFS allows the algorithm to efficiently traverse the graph and visit all nodes.

Worst-Case Analysis:

The worst-case time complexity of the code that finds a topological ordering of a DAG using a depth-first search (DFS) is θ(V + E), where V is the number of nodes in the DAG and E is the number of edges. This is because the DFS algorithm visits each node and each edge exactly once. In the topological_sort_with_DFS function, the outer for loop iterates over all nodes in the DAG, which takes θ(V) time. The inner dfs function iterates over all outgoing edges from each node, which takes θ(E) time in total. Therefore, the total time complexity of the topological_sort function is θ(V + E) in worst-case sceneario.

b) Design:

- Initialize a list ‘my_list’ to store the topological ordering, and a queue ‘my_queue’ to keep track of the nodes with indegree 0.
- Add all nodes with indegree 0 to the queue ‘my_queue’.
- While the queue is not empty, repeat the following steps:
    a) Remove a node n from the front of the queue.
    b) Add n to the list ‘my_list’.
    c) Decrement the indegree of all nodes that are connected to n by outgoing edges.
    d) If a node m has indegree 0 after its indegree is decremented, add m to the queue.
- If the list ‘my_list’ contains all nodes in the DAG, return ‘my_list’ as the topological ordering. Otherwise, return an error message indicating that the DAG has a cycle.

Worst-Case Analysis:

The worst-case time complexity of the code is same the question a. It works θ (V + E) in worst-case where V is the number of nodes in the DAG and E is the number of edges.

2)

```
def power(base, exponent):
    if(exponent == 0):                          --> θ(1)
        return 1                                --> θ(1)

    result = power(base, int(exponent/2))       --> θ(logn)
    result *= result;                           --> θ(1)

    if(exponent % 2 == 1):                      --> θ(1)
        result *= base                          --> θ(1)

    return result;                              --> θ(1)
```

Explanation: If a problem works O(logn) in worst-case scenario, it means: we divide the problem in two at each iteration and we get the result. We can see that we can also use this bisection operation in power operation. In cases where the exponent is even, if I can calculate half of the exponent, namely power(a, n/2), the square of this result will give the answer (power(a, n/2) * power(a, n/2)). If the exponent is odd, one base multiplication left in this product. So the result will be power(a, n/2) * power(a, n/2) * a. After writing the base case, I have obtained an algorithm that works logn in the worst-case.

3) Design:

- Initialize a 9x9 grid to represent the sudoku game.
- Define a function to check if a given number is a valid choice for a given cell in the sudoku grid. This function should check if the number is not already present in the same row, column or 3x3 sub-grid as the given cell.
- Define a recursive function that takes the sudoku grid and a cell position (row and column) as input.
- If the cell position is out of bounds, meaning all cells have been filled, return true.
- If the cell at the given position already contains a number, move to the next cell and call the recursive function again with the updated cell position.
- Otherwise, try all possible numbers (1 to 9) for the given cell and call the recursive function again with the updated grid and the next cell position. If any of the recursive calls return true, return true.
- If none of the recursive calls return true, return false.
- Start the recursive function with the initial sudoku grid and cell position (0, 0).

Worst-Case Time Complexity:

- The algorithm uses a recursive approach to solve the sudoku game. It tries all possible numbers for a given cell and calls the recursive function again with the updated grid and the next cell position.
- In the worst-case, the algorithm has to try all 9 possible numbers for each cell in the grid, there is 81 cells and algorithm calculate 9^81 possibility in worst-case scenario.
- The time complexity of this algorithm in the worst-case is θ(9^81).
- If we generalize nxn sudoku game, time complexity of this algorithm in the worst case is $\theta(n^{n*n})$.

4)

Stability: Stable sorting algorithm preserves the order of duplicate keys.

Insertion Sort:

Insertion sort is stable sorting algorithm. It preserves the order of duplicate keys.

Input array = {6, 8, 9, 8, 3, 3, 12}

- Algorithm basically starts with the second element to last element and move each element to corresponding place on the left side.
- Move 8 to corresponding place in {6}. 8 is in the right position.
- Move 9 to corresponding place in {6, 8}. 9 is in the right position.
- Move 8 to corresponding place in {6, 8, 9}. Element 8 (in index 3) will be placed between 8 and 9.
- Move 3 to corresponding place in {6, 8, 8, 9}. Element 3 (in index 4) will be placed in front of 6.
- Move 3 to corresponding place in {3, 6, 8, 8, 9}. Element 3 (in index 5) will be placed between 3 and 6.
- Move 12 to corresponding place in {3, 3, 6, 8, 8, 9}. 12 is in the right position.
- Finally array is {3, 3, 6, 8, 8, 9, 12}

Quick Sort:

Quick sort is unstable sorting algorithm. Swap operation takes place according to the pivot position. This does not preserve duplicate keys.

Input array = {6, 8, 9, 8, 3, 3, 12}

- Algorithm basically select middle element as pivot. Moves pivot to end. Partition the subarray. Move left bound to the right until it reaches a value greater than or equal to the pivot. When the right bound crosses the left bound, all elements to the left of the left bound are less than the pivot and all elements to the right are greater than or equal to the pivot. Move pivot to its final location. Then recursively call quicksort for left subarray and right subarray until subarray has one element.
- Select middle element(8) as pivot. Move pivot to end. Partition the subarray. Move left bound to the right until it reaches a value greater than or equal to the pivot. When the right bound crosses the left bound, all elements to the left of the left

bound are less than the pivot and all elements to the right are greater than or equal to the pivot. Move pivot to its final location. Array is {6, 3, 3, 8, 9, 8, 12}

- Recursive call on the left sublist.{6, 3, 3}
- Select middle element(3) as pivot. Move pivot to end. Partition the subarray. Move pivot to its final location. Array is {3, 3, 6, 8, 9, 8, 12}.
- Recursive call on the right sublist {3, 6}
- Select middle element(3) as pivot. Move pivot to end. Partition the subarray. Move pivot to its final location. Array is {3, 3, 6, 8, 9, 8, 12}
- Recursive call on the right sublist {9, 8, 12}
- Select middle element(8) as pivot. Move pivot to end. Partition the subarray. Move pivot to its final location. Array is {3, 3, 6, 8, 8, 12, 9}
- Recursive call on the right sublist {12, 9}
- Select middle element(12) as pivot. Move pivot to end. Partition the subarray. Move pivot to its final location. Array is {3, 3, 6, 8, 8, 9, 12}
- Finally array is {3, 3, 6, 8, 8, 9, 12}.

Bubble Sort:

Bubble sort is stable sorting algorithm. It preserver the order of duplicate keys.

Input array = {6, 8, 9, 8, 3, 3, 12}

- The algorithm basically starts from first element to last element, it makes comparisons with the element to its right in each iteration and swaps accordingly. In this operation, it finds the largest element and puts it in the last part of the array. It does these operations by arranging the last iteration until the array is ordered.
- In first iteration, it compares these duos and swaps if necessary.
  {6, 8 -> no swapping}, {8, 9 -> no swapping}, {9, 8 -> swapping}, {9, 3 -> swapping}, {9, 3 -> swapping}, {9, 12 -> no swapping}. Array is {6, 8, 8, 3, 3, 9, 12}
- In second iteration,
  {6, 8 -> no swapping}, {8, 8 -> no swapping}, {8, 3 -> swapping}, {8, 3 -> swapping}, {8, 9 -> no swapping}. Array is {6, 8, 3, 3, 8, 9, 12}
- In third iteration,
  {6, 8 -> no swapping}, {8, 3 -> swapping}, {8, 3 -> swapping}, {8, 8 -> no swapping}. Array is {6, 3, 3, 8, 8, 9, 12}
- In forth iteration,
  {6, 3 -> swapping}, {6, 3 -> swapping}, {6, 8 -> no swapping}. Array is {3, 3, 6, 8, 8, 9, 12}
- In fifth iteration,
  {3, 3 -> no swapping}, {3, 6 -> no swapping}. Array is {3, 3, 6, 8, 8, 9, 12}
- In last iteration,
  {3, 3 -> no swapping}. Array is {3, 3, 6, 8, 8, 9, 12}
- Finally array is {3, 3, 6, 8, 8, 9, 12}

5)

(a) Brute force and exhaustive search are two different methods of solving a problem. Brute force involves trying every possible solution until the correct one is found, while exhaustive search involves systematically enumerating all possible solutions and checking if each one is correct. While both methods involve trying all possible solutions, brute force is more general and can be applied to any problem, while exhaustive search is more specific and only applicable to certain types of problems. However, exhaustive search is guaranteed to find the correct solution, whereas brute force may not. Additionally, brute force can be efficient for small numbers of possible solutions, but it becomes impractical for large numbers of solutions, whereas exhaustive search can be very time-consuming for large numbers of solutions.

(b) Caesar's Cipher is a simple encryption method that shifts each letter of a message by a certain number of places in the alphabet. For example, with a shift value of 2, A would be replaced with C, B would be replaced with D, and so on. Caesar's Cipher is vulnerable to brute force attacks, where an attacker tries all possible shift values to decode the message. Because there are only 26 possible shift values, a brute force attack on Caesar's Cipher can be easily carried out by trying all possible values until the message is decoded.

AES (Advanced Encryption Standard) is a more advanced encryption algorithm that uses complex mathematical operations to encode a message, making it much harder to decode without the correct key. Unlike Caesar's Cipher, AES is not vulnerable to brute force attacks, as it would take an impractical amount of time and computational power to try all possible keys. Instead, attackers must use other methods, such as cryptanalysis, to try to break AES encryption.

(c) This can take up to n-1 divisions, which is sub-linear in the value of n but exponential in the length of n. For example, a number n slightly less than 10,000,000,000 would require up to approximately 9,999,999,999 divisions, even though the length of n is only 11 digits. Moreover one can easily write down an input (say, a 300-digit number) for which this algorithm is impractical. Since computational complexity measures difficulty with respect to the length of the (encoded) input, this naive algorithm is actually exponential. It is, however, pseudo-polynomial time.