GTU

**DEPARTMENT OF COMPUTER ENGINEERING**

**CSE 321 – Autumn 2022**

**HOMEWORK 3  
REPORT**

SÜLEYMAN GÖLBOL 1801042656

# DIRECTED ACYCLIC GRAPH

For input of directed acyclic graph, I thought if graph node doesn’t have in-degree, then it doesn’t have parent. If it doesn’t have out-degree, it doesn’t have children.   
a) For topological depth first search I thought the using the idea of backtracking. Creating a stack called stack, and a set called visited. In every node I visit, I add that node to visited set. Then I check that node’s children one by one and for each of them, I apply depth first search recursively. After that I insert node at the head of stack. This way stack list holds the topological sort of DAG.

b) For topological non depth first search, I used the idea of there should be exists vertex with in-degree 0. Because if there was an edge goes back to a node already visited, that shows the existence of a cycle. So, I find in-degrees first. And I the ones with indegree 0’s to a queue. After that, until queue becomes empty, I pop from queue as node value and add it to my topologically ordered list. This way makes ordering of it topologically.

**Worst Time Complexity:**

We say V is number of nodes and E is number of edges.

a) Every edge is visited 1 time when DFS finished, it has O(E) complexity. Because of it is a directed graph; unlike undirected graphs, edges will not appear 2 times. In the worst case all vertices need to be visited. So O(E) + O(V) = O(E+V)

b) Indegree initilization = O(V), iterating on each edge O(E). So totally O(E+V)

1. **EXPONENTIATION COMPLEXITY**

For the complexity of exponentiation, I thought about if exponent number (n) is odd or even. Because, when n is odd, finding the result can be done as an  = a \* (a2)n/2 and when n is even, it can be done as an = (a2)n/2So, in a while loop we can use this way for achieve that complexity.

**result = 1**

**while(n >= 1):**

**if n % 2 == 1:**

**result \*= a**

**a = a\*a**

**n //= 2**

**return result**

**Worst Time Complexity:**

In the worst case, we need to iterate on whole while loop, since counter n decreases by dividing 2, it is logarithmic. So it’s O(logn)

**3) SUDOKU SOLVER**

To solve sudoku with exhaustively, I used 2 main functions.   
One of them is is\_legal() which takes row, column, and value as parameters. This function checks for 3 things. If there exist a number in cell (except 0), then returns false. If value is already in that row or column, it returns false. Also it checks for the containing 3x3 box if same value is already there, it returns false. If none of them happens it returns true.

My solver function function checks until it reaches to 8th row and last column to understand if we’re done. At the end of every row, it resets column and updates row value. It checks for every cell that is not filled, then controls if putting value is proper or not. Then it decides on 2 things;

# if self.is\_legal(row, column, num):

# self.sudoku[row][column] = num

# # If solved, control the next

# if self.solver(row, column + 1):

# return True

# # if not solved return back to 0

# self.sudoku[row][column] = 0

**Worst Time Complexity:**

# In the worst case, for example if cell is empty(0) there are 9 different options because there are 9 characters that can be fit from 1 to 9. And there are m empty cells. It’s 9m. For 9x9 fixed grid it would be O(1) but I assume that grid size can be changed, so for nxn grid, It’s O(9n\*n).

# *Note:* My solution contains both user input option and auto-unit test. Press 2 for user input.

# 

**4) ARRAY SORTING**

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

**Quick Sort**

[6 8 9 8 3 3 12] Select 6 as pivot so sort [3, 3] and [8, 9, 8, 12]

[3, 3] 3 is pivot, no smaller / bigger value. Sorting [] and [] (No need sort)

[8, 9, 8, 12] 8 is pivot. Sorting [] and [9, 12]

[9, 12] 9 is pivot. Sorting [] and [12]

[12] 12 is pivot. Sorting [] and [] (No need sort)

So, step by step:  
6 8 9 8 3 3 12  
3 3 6 8 9 8 12  
3 3 6 8 9 8 12  
3 3 6 8 8 9 12

**Insertion Sort**

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

First, it inserts second element. Then compares beginning with every element in array while it’s greater.

Inserted element: 8  
Comparing if 6 > 8 and if true move 6 to one right. FALSE  
6 8 9 8 3 3 12

Inserted element: 9  
Comparing if 8 > 9 and if true move 8 to one right. FALSE  
6 8 9 8 3 3 12

Inserted element: 8  
Comparing if 9 > 8 and if true move 9 to one right. TRUE  
Moving element: 9 to one right  
6 8 8 9 3 3 12

Inserted element: 3  
Comparing if 9 > 3 and if true move 9 to one right. TRUE  
Moving element: 9 to one right  
Comparing if 8 > 3 and if true move 8 to one right. TRUE  
Moving element: 8 to one right  
Comparing if 8 > 3 and if true move 8 to one right. TRUE  
Moving element: 8 to one right  
Comparing if 6 > 3 and if true move 6 to one right. TRUE  
Moving element: 6 to one right  
3 6 8 8 9 3 12

Inserted element: 3  
Comparing if 9 > 3 and if true move 9 to one right. TRUE  
Moving element: 9 to one right  
Comparing if 8 > 3 and if true move 8 to one right. TRUE  
Moving element: 8 to one right  
Comparing if 8 > 3 and if true move 8 to one right. TRUE  
Moving element: 8 to one right  
Comparing if 6 > 3 and if true move 6 to one right. TRUE  
Moving element: 6 to one right  
3 3 6 8 8 9 12

Inserted element: 12  
Comparing if 9 > 12 and if true move 9 to one right. FALSE  
3 3 6 8 8 9 12  
3 3 6 8 8 9 12

**Bubble Sort**

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

Save index of every element one by one.

Then again for every element in array until size-index-1, compare with next. If true swap.

Comparing if 6 > 8 and if true swap them. FALSE

6 8 9 8 3 3 12

Comparing if 8 > 9 and if true swap them. FALSE

6 8 9 8 3 3 12

Comparing if 9 > 8 and if true swap them. TRUE

6 8 8 9 3 3 12

Comparing if 9 > 3 and if true swap them. TRUE

6 8 8 3 9 3 12

Comparing if 9 > 3 and if true swap them. TRUE

6 8 8 3 3 9 12

Comparing if 9 > 12 and if true swap them. FALSE

6 8 8 3 3 9 12

Comparing if 6 > 8 and if true swap them. FALSE

6 8 8 3 3 9 12

Comparing if 8 > 8 and if true swap them. FALSE

6 8 8 3 3 9 12

Comparing if 8 > 3 and if true swap them. TRUE

6 8 3 8 3 9 12

Comparing if 8 > 3 and if true swap them. TRUE

6 8 3 3 8 9 12

Comparing if 8 > 9 and if true swap them. FALSE

6 8 3 3 8 9 12

Comparing if 6 > 8 and if true swap them. FALSE

6 8 3 3 8 9 12

Comparing if 8 > 3 and if true swap them. TRUE

6 3 8 3 8 9 12

Comparing if 8 > 3 and if true swap them. TRUE

6 3 3 8 8 9 12

Comparing if 8 > 8 and if true swap them. FALSE

6 3 3 8 8 9 12

Comparing if 6 > 3 and if true swap them. TRUE

3 6 3 8 8 9 12

Comparing if 6 > 3 and if true swap them. TRUE

3 3 6 8 8 9 12

Comparing if 6 > 8 and if true swap them. FALSE

3 3 6 8 8 9 12

Comparing if 3 > 3 and if true swap them. FALSE

3 3 6 8 8 9 12

Comparing if 3 > 6 and if true swap them. FALSE

3 3 6 8 8 9 12

Comparing if 3 > 3 and if true swap them. FALSE

3 3 6 8 8 9 12

**5) Q5**

1. Brute force is a technique that doesn’t use complicated methods to solve the problems. It just iterates until it finds the wanted match. Brute force can be used for many algorithms.

Exhaustive search is a subcase of the brute force search. It’s mostly used for huge combinational problems etc.

1. Caesar’s cipher is vulnerable to brute force. The reason for that caesar’s cipher has minimum security, we can try every possible decryption. It’s only encrypted by shifting characters on text using alphabet. For 26 different characters, printing values with 1-26 shift amounts is enough to crack.   
   Normally we encrypt data by changing data using a key. But AES has key expansion which are round keys created in many ways. These are XOR’ing until all info are encrypted. With brute force, trying all possible keys takes more than 1000 billion years so its infeasible.
2. To check primality, we need to check until n-1 if x divides or not.

For a number N, we need size of the input. We can check how many bits it contains using log\_2(N) + 1’s floor value. For ex., 8 will take log\_2(N) + 1 = 3+1 = 4 bits to store.

From 2 to n-1, there are n-2 numbers. It will take n-2 steps to check until n-1 to find primality. **Input = n-2**, so **input size = log\_2(n-2) + 1**.

Let’s input size be equal to S, then **log\_2(n-2) + 1 = S**.

Then n = 2s-1 + 2 which is exponential.