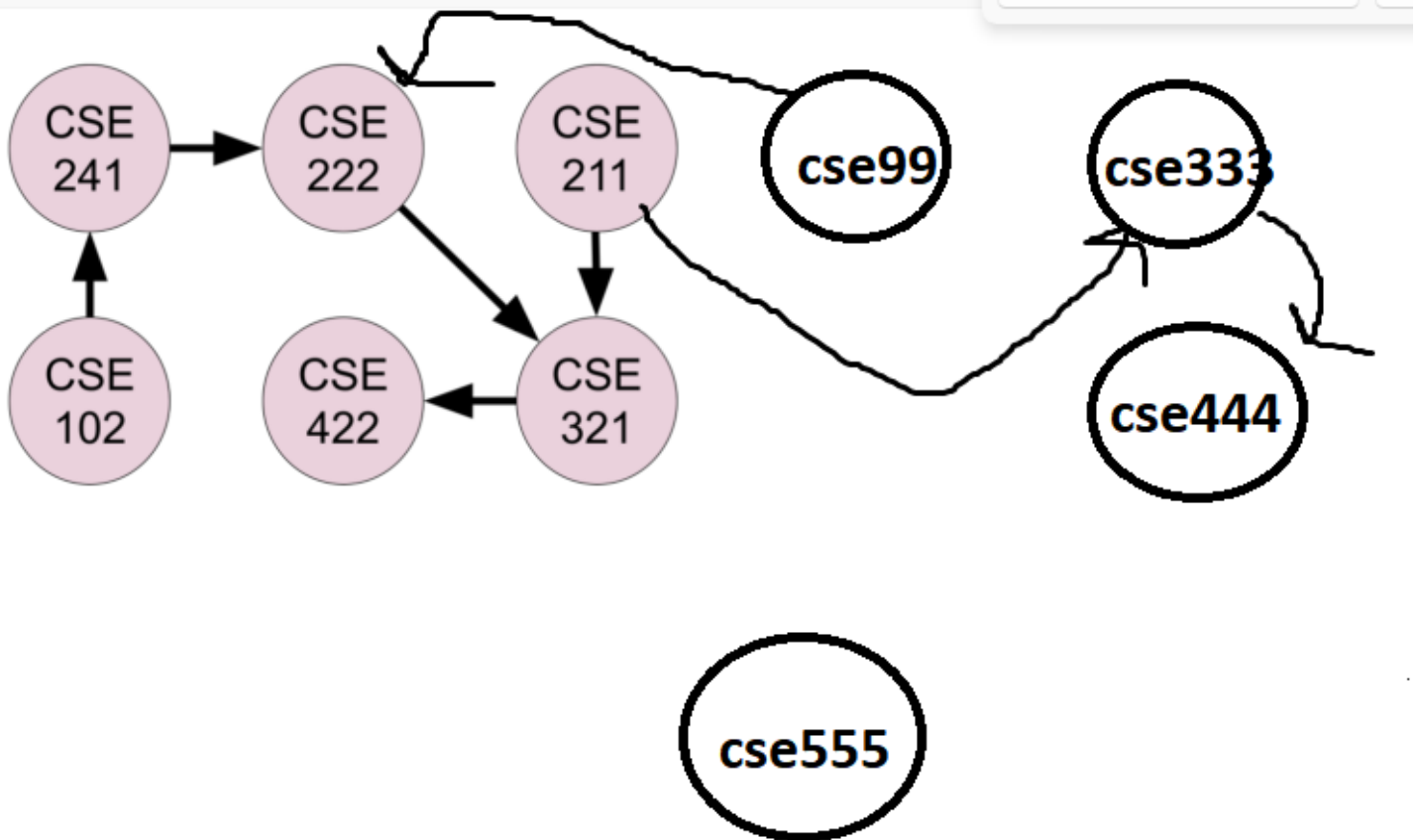


Question 1-a

Solution:

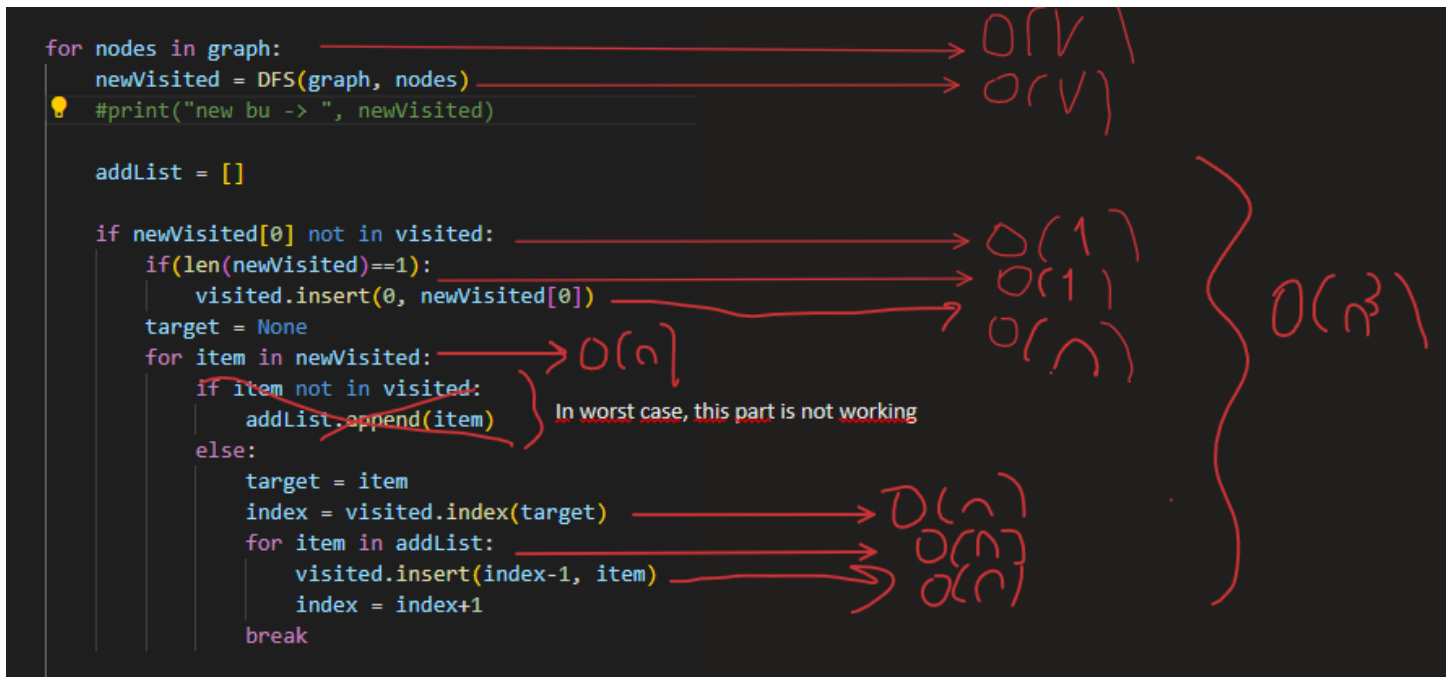
I have run DFS many times by giving all the nodes separately as start nodes. If the first element of a DFS result is not in the current result list, that means we have found a new path. I'm adding the elements from this new list to the correct index in the current result list until I encounter an already existing element in the result list. So finally, I get topological sort with all nodes. I did not touch the original DFS algorithm. I make changes out of DFS.

Note: There is an example for this question in the code. I want to show the example graph drawing. You can see its result when you run the code.



Analyse:

Running DFS is $O(\text{Vertex Number})$. We run DFS as much as vertex number $O(V)$. And while this operation, in each new graph, we check whether all the elements are in the visited list one by one. $O(V)$.



I do most operations using List. If I use Hash Map, I could decrease the complexity but there is no restriction in the question. So, general complexity worst case is: $O(V) * (O(n^3)) = O(n^4)$

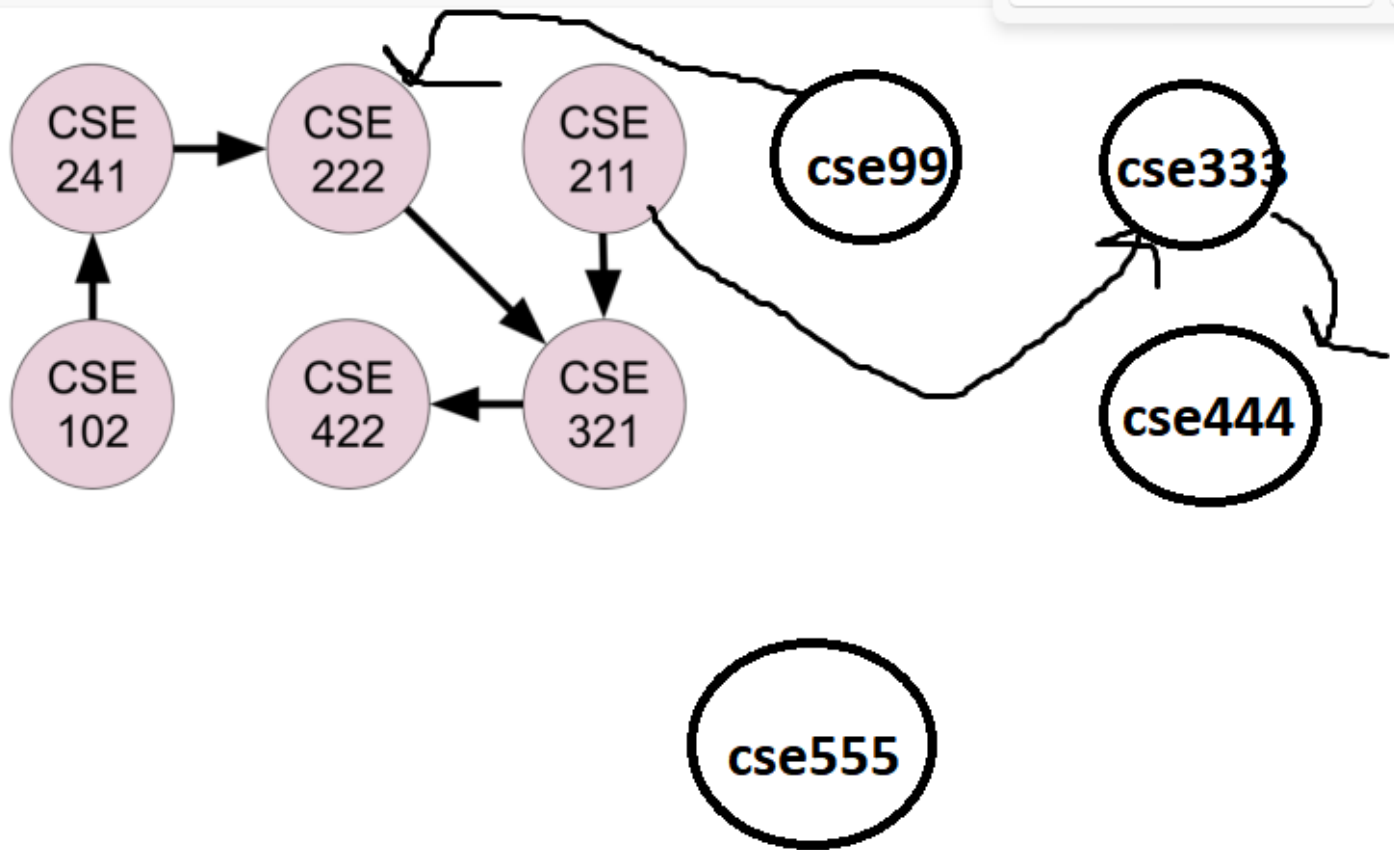
Question 1-b

Solution:

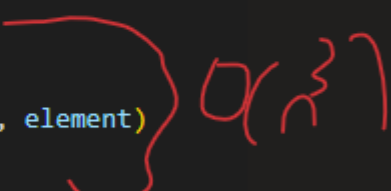
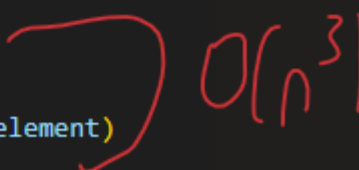
BFS is exactly the algorithm I was looking for. Again, we make BFS from all vertexes separately. If the node whose neighbours we are going to is in the current result list, we put all its neighbours in the indexes after `result[node]`. If the node we go to neighbours is not in the current result list, we find where to add the node to which we go to its neighbours, add it there and continue. BFS has been modified.

Note: There is an example for this question in the code. I want to show the example graph drawing. You can see its result when you run the code.

Canvas



Analyse:

```
def updated_breadth_first_search(graph, answer, start_node):  
  
    visited = []  
  
    queue = []  
  
    queue.append(start_node)  
    print(queue)  
  
    while queue:  
        node = queue.pop(0)  
  
        if node not in visited:  
            visited.append(node)  
            queue.extend(graph[node])  
  
            if node in answer:  
                index = answer.index(node)  
                for element in graph[node]:  
                    if element not in answer:  
                        answer.insert(index+1, element)  
                        index = index+1  
              
  
            else:  
                answer.insert(0, node)  
                index = answer.index(node)  
                for element in graph[node]:  
                    if element not in answer:  
                        answer.insert(index+1, element)  
                        index = index+1  
              
  
    return visited
```

Regular BFS algorithm complexity is $O(V+E)$. I add some lines to the algorithm to solve the problem in the question. Those lines cost (n^3) complexity. This cost come from traversing lists as nested and insert an element to list. Like previous question, I use List but if would use Hash Map, I would decrease the complexity. So, general complexity worst case is : $O(V+E) * O(n^3) = O(n^4)$

Question-2

If we are asked to construct a logarithmic complexity function, we should first think of halving the size of the problem. In this question, there is a recursive algorithm. Its base case is constant time. In body part, it halves the exponential of number and pass the recursive function. Base case depends on exponential part. So, the algorithm is simply a logarithmic algorithm. This algorithm is called Exponentiation by Squaring. I have seen it in CSE102.

Question-3

Solution:

We are going to make an exhaustive search. We start from the first empty cell, put there 0 and continue. In the next empty cell we try to put 0. If it is not a correct number, we will increase it to 1. We will try until 9. If 9 is not okay, too, we are going to step back to the previous step. We put the 0 in previous step. We will make it 1 now. And continue. The main idea is this. We are going to try 9 numbers for a cell. One of them will be okay. Brute Force is this.

Analyse:

It is a classic brute force sudoku solver. It tries to put a number from 0 to 9 to a cell. Then continues. When it notices there is no number to put in a cell, it moves back and increases the number in the previous step.

Sudoku is a 9x9 board game. I think, we cannot change the table size. If you change, it is not a sudoku game anymore. So, my analysis is based on empty cells. So, all the helper functions `check_column`, `check_row`, `find_first_empty` are constant since they include nested loops based on a *constant*. (They will run 81 times in worst case, always, it does not change according to empty cells.) If we consider the solver function, it runs 9 times for each empty cell. So, when n is empty cell number, the complexity of this function according to empty cells is 9^n (polynomial) = $O(9^n)$

Question-4

Insertion Sort

{6, 8, 9, 8, 3, 3, 12} -> Insertion Sort

Step-1 => 6, 8, 9, 8, 3, 3, 12 (No change)

Step-2 => 6, 8, 9, 8, 3, 3, 12 (No change)

Step-3 => 6, 8, 9, 8, 3, 3, 12 (Change) -> 6, 8, 8, 9, 3, 3, 12 No change, it is a proof that Insertion Sort is **stable**)

Step-4 => 6, 8, 8, 9, 3, 3, 12 (Change 4 times, I wanted to show this step fast) -> 3, 6, 8, 8, 9, 3, 12

Step-5 => 3, 6, 8, 8, 9, 3, 12 (Change) -> 3, 6, 8, 8, 3, 9, 12 (Change) -> 3, 6, 3, 8, 8, 9, 12 (Change) -> 3, 3, 6, 8, 8, 9, 12 (No change, it is a proof that Insertion Sort is **stable**)

Step-6 => 3, 3, 6, 8, 8, 9, 12 (No change, we are DONE!)

Insertion sort is a **stable** sort. During the sorting we will swap the ordering of any two items if the item on the right is less than the item to its left.

The answer is on the image. I had to show compared elements. So, I must take a screenshot.

Quick Sort

{6, 8, 9, 8, 3, 3, 12} -> Quick Sort

Step-1

I select the pivot as the beginning element. 6 is pivot.

In the partition process, the Low element is 8 and the High element is 12 at the beginning.

- We will increase the Low element until it is greater than the Pivot. So, it will stop when it encounter the element 8.
- We will decrease the High element until it is less or equal than Pivot. So, it will stop when it encounter the element 3 at index 5. We will swap them.

The array will be => {6, 3, 9, 8, 3, 8, 12}. We will continue the steps above.

- The Low element will be 9 at index 2
- The High element will be 3 at index 4. We will swap the elements at the index 2 and 4.

The array will be => {6, 3, 3, 8, 9, 8, 12}

This step is done. We will swap the pivot and the 3 at index 2.

The final array in this step is => {3, 3, 6, 8, 9, 8, 12}

Step-2

The right subtree of the step above that comes here recursively is {8, 9, 8, 12}

Pivot is 8.

In the partition process, the Low element is 9 and the High element is 12 at the beginning.

- We will increase the Low element until it is greater than the Pivot. So, it will stop when it encounter the element 9.
- We will decrease the High element until it is less or equal to the Pivot. So, it will stop when it encounter the element 8. Swap them.

The array will be $\Rightarrow \{8, 8, 9, 12\}$

There will be no swap since ($low \leq high$) in the next step. Low element will stop when it encounter the element 8 at the index 1. We will swap it with the pivot. And, as you see although the numbers 8 and 8 is equal, we are swapping them. So, the 8 that is behind before sorting the array, will be ahead of the other 8 from now on. So, Quick Sort is NOT a **stable** algorithm.

Step-3

The left subtree of the step-1 that comes here recursively is $\{3, 3\}$

Pivot is 3.

In the partition process, the Low element is 3 and the High element is 3 at the beginning. They are the same element. So, there will be no swap.

However, according to algorithm, we will swap the Low and Pivot before the terminate the function. So, 3's will be swap. It means that, the 3 that is behind before sorting the array, will be ahead of the other 3 from now on. So, Quick Sort is NOT a **stable** algorithm.

Now the whole array is: $\{3, 3, 8, 8, 9, 12\}$. It is sorted. There is no need to show other steps. It is proved that Quick Sort is NOT **stable**.

Bubble Sort

{6, 8, 9, 8, 3, 3, 12} -> Bubble Sort

Step-1 => 6, 8, 9, 8, 3, 3, 12 (No change)

6, 8, 9, 8, 3, 3, 12 (No change)

6, 8, 9, 8, 3, 3, 12 (Change)

6, 8, 8, 9, 3, 3, 12 (Change)

6, 8, 8, 3, 9, 3, 12 (Change)

6, 8, 8, 3, 3, 9, 12 (No change)

Step-2 => 6, 8, 8, 3, 3, 9, 12 (No change)

6, 8, 8, 3, 3, 9, 12 (No change, it is a proof that Bubble Sort is **stable**.)

6, 8, 8, 4, 4, 9, 12 (Change)

6, 8, 4, 8, 4, 9, 12 (Change)

6, 8, 4, 4, 8, 9, 12 (No change)

Step-3 => 6, 8, 4, 4, 8, 9, 12 (No change)

6, 8, 4, 4, 8, 9, 12 (Change)

6, 4, 8, 4, 8, 9, 12 (Change)

6, 4, 4, 8, 8, 9, 12 (No change, it is a proof that Bubble Sort is **stable**.)

Step-4 => 6, 4, 4, 8, 8, 9, 12 (Change)

4, 6, 4, 8, 8, 9, 12 (Change)

4, 4, 6, 8, 8, 9, 12 (No change)

Step-5 => 4, 4, 6, 8, 8, 9, 12 (No change, it is a proof that Bubble Sort is **stable**.)

4, 4, 6, 8, 8, 9, 12 (No change)

Step-6 => 4, 4, 6, 8, 8, 9, 12 (No change, it is a proof that Bubble Sort is **stable**.)

So, Bubble Sort is A **stable** algorithm. We swap elements only when the element on the left is greater than the element on the right.

Question-5-a

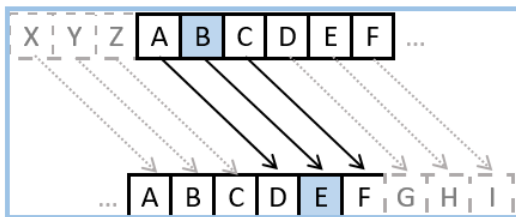
Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved. It has a "Just Do It!" strategy. Exhaustive Search is a special case of Brute Force. Exhaustive search is simply a Brute Force Algorithm.

Question-5-b

Caesar Cipher

Search Result: Caesar Cipher is one of the oldest and simplest encryption techniques. It has a very simple logic. We give the algorithm a key, it changes the letter in the string with "key" number of positions down the alphabet. For example, the string is "emre", and key is 1; encrypted text is: fnsf

Answer: Definitely, it is vulnerable against Brute Force. How many characters can we have, at most? If we use only alphabet letters it is only 26. The only thing we are going to do is, moving [1-26] move step back from the encrypted text to backward. Then, we only check the outputs. One of them will be meaningful. It is easy since there are only 26 outputs. If we use extra some specific characters in ASCII, decrypting will be easy using Brute Force, again.



SHIFT +3

This Caesar cipher has a shift of 3 characters.

The letter 'A' becomes a 'D'. The letter 'B' becomes 'E'.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Plaintext

Ciphertext

AES

Search Result: AES is a specification for the encryption of electronic data. It takes 128 bits as input and 128 bits as output as encrypted. AES relies on substitution and permutation network principle which means it is performed using a series of linked operations which involves replacing and shuffling of the input data.

If the input is 128 bits, there are 10 rounds to encrypt the input. The round steps are:

- Sub Bytes: Each byte is substituted by another byte.
- Shift Rows: Each row is shifted a specific number of times
- Mix Columns: This step is matrix multiplication. Each column is multiplied with a specific matrix.
- Add Round Keys: Result of the previous steps is processed by a XOR operation with a round key. It is the most important step in terms of the question in the Assignment.

Answer: To attack this encryption method using Brute Force, we must try every possible key until we found a meaningful thing. A key is 128 bits, and it means that there can be 2^{128} possible

keys. If your purpose is trying all possible keys, it is a fact that your life is NOT going to be enough to see the result.

<https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/>

<https://bilgisayarkavramlari.com/2008/02/21/sezar-sifrelemesi-caesars-cipher-shift-cipher-kaydirma-sifrelemelesi/>

<https://www.geeksforgeeks.org/advanced-encryption-standard-aes/>

<https://bilgisayarkavramlari.com/2009/06/03/aes-ve-rijndael-sifreleme/>

Question-5-c

- Firstly, I must explain the naïve solution of primality test. Its naïve solution is a kind of brute force approach. We get the input, and we try all numbers from 2 to this square root of input itself. If a number divides the input without remaining, it means that the number is NOT prime.
- In this algorithm we can say that the loop runs as much as the square root of number itself. But, if we will talk in asymptotic language, we must draw a bound in terms of input size. If you think the asymptotic notation of this algorithm in terms of magnitude of number, you are going to say It is $O(\log n)$. But, in terms of input size of the input n that consists of m bit, it is $O(2^m)$ since $m = \log(n)$.

So, the complexity of this brute force approach to test primality, is exponential in terms of input size.