QUESTION 1

(a)

```
def topSortDFS(self):
    for i in range(len(self.vertices)):
        if not self.visited[i]:
            self.topSortDFSHelper(i)

    print(self.stack)

def topSortDFSHelper(self, i):
    self.visited[i] = True

    for j in range(len(self.graph[self.vertices[i]])):
        if not self.visited[self.vertices.index(self.graph[self.vertices[i]][j])]:
            self.topSortDFSHelper(self.vertices.index(self.graph[self.vertices[i]][j]))

    self.stack.insert(0, self.vertices[i])
```

vertices is an array that keeps the vertices so for example:

- ["CSE 102", "CSE 241", "CSE 211", "CSE 222", "CSE 321", "CSE 422"]

visited is a boolean array that all the elements are False initially.

In topSortDFS method, we go through the each vertex and we sent the index of this vertex to helper method if it is not visited.

In helper method, we mark the given vertex as visited. Here, we go through each vertex that is adjacent to the given vertex. If that adjacent vertex is not visited before, we start the helper method to visit that vertex. At the end, we insert each vertex to the beginning of the stack after all of the vertices that can be reached from it is visited. With this way, we can get the topological order.

We go through each vertex and each edges that all the vertices has. So worst case running time for this algorithm is → $\theta(|V|+|E|)$ where V is number of vertices and E is number of edges.

(b)

```
def topSortNonDFS(self):
    indegree = [0 for i in range(len(self.vertices))]
    order = []
    orderHelper = []

    for i in self.vertices:
        for j in self.graph[i]:
            indegree[self.vertices.index(j)] += 1

    for i in range(len(indegree)):
        if indegree[i] == 0:
            orderHelper.append(self.vertices[i])

    while len(orderHelper) != 0:
        currentItem = orderHelper.pop(0)
        order.append(currentItem)

        for i in self.graph[currentItem]:
            indegree[self.vertices.index(i)] -= 1
            if indegree[self.vertices.index(i)] == 0:
                orderHelper.append(i)

    print(order)
```

In this algorithm, first we fill the indegree array that holds the indegrees of the vertices. We visit each vertex and edge so this step takes $\theta(|V|+|E|)$ time.

Afterwards, we put vertices that has indegree 0 to a helper array.

Then we iterate through these vertices that has indegree 0. We remove that vertex that has indegree 0 from helper array and add it to the order array.

Then we iterate through all neighbouring vertices of that vertex (with indegree 0). These vertices' indegrees are decreased by 1 because we processed the 1 vertex that has edge to them. If their indegrees also becomes 0, we add them to the helper array for later use.

With this way, we order the vertices starting from vertices that has indegree 0 and continuing with neighbouring vertices. At the end, we get the topological order.

Worst case for this algorithm is $\theta(|V|+|E|)$ as mentioned because we visit every vertex and every edge.

QUESTION 2

```
def power(a, n):
    if n == 0:
        return 1
    elif n%2 == 0:
        temp = power(a, n/2)
        return temp * temp
    else:
        temp = power(a, (n-1)/2)
        return temp * temp * a
```

This algorithm is decrease & conquer algorithm as we have seen at the lectures. Here we decrease by a constant factor which is 2. At each recursion step, we divide the n by 2.

$a^n = a^{n/2} \times a^{n/2}$ {if n is even}

$a^n = a^{(n-1)/2} \times a^{(n-1)/2} \times a$ {if n is odd}

At each step, we divide the n by 2 so worst-case time complexity is $\theta(\log n)$.

QUESTION 3

```
def isNumSafe(row, column, number, sudoku):
    for i in range(9):
        if sudoku[i][column] == number:
            return False

    for i in range(9):
        if sudoku[row][i] == number:
            return False

    boxStartRow = row - row%3
    boxStartCol = column - column%3
    for i in range(3):
        for j in range(3):
            if sudoku[i + boxStartRow][j + boxStartCol] == number:
                return False

    return True
```

Firstly I wrote the code to check if given number fits the given row and column by checking entries up-to-bottom, left-to-right, and 3x3 box that row & column belongs to.

```
def sudokuSolverHelper(sudoku, row, column):
    if (row == 8 and column == 9):
        return True
    elif column == 9:
        row = row + 1
        column = 0

    if sudoku[row][column] == 0:
        for i in range(1, 10):
            if isNumSafe(row, column, i, sudoku):
                sudoku[row][column] = i
                if sudokuSolverHelper(sudoku, row, column+1):
                    return True
                else:
                    sudoku[row][column] = 0
        return False
    else:
        return sudokuSolverHelper(sudoku, row, column+1)

def sudokuSolver(sudoku):
    sudokuSolverHelper(sudoku, 0, 0)
```

Here, we go through each place one by one and try each number (1, 2, 3, 4, 5, 6, 7, 8, 9) for that place.

That number may fit or may not fit. If it doesn't fit that place, it is okay, we move to the next number. If it fits, that doesn't mean it is locked forever. That entry may leave the sudoke unsolvable. Therefore we continue by calling method for the next entry and if sudoku can be solved with that entry, that place is locked, its value is found. Otherwise, we continue with the next number.

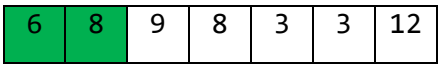
For each empty place, we have 9 options and we have 9x9 matrix for the sudoku. So worst case for this algorithm is $\theta(9^{9x9})$ which is a very big constant.

QUESTION 4

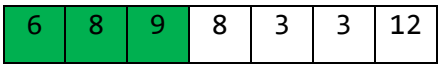
array = {6, 8, 9, 8, 3, 3, 12}

Insertion Sort

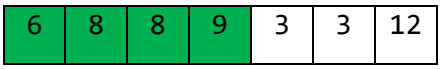
Green colored side indicates the sorted part.



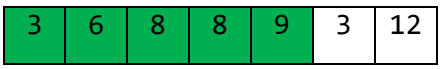
8 is in the correct place, we don't change its position.



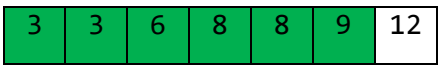
9 is in the correct place, we don't change its position.



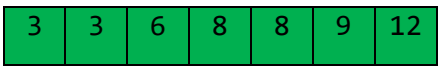
Since 8 is smaller than 9, we swap these 2 and stop because there is no more element that is bigger than 8.



We compare 3 with 9 first. Since 3 is smaller, we swap them. Then we compare it with 8 and swap, with 8 again and swap, with 6 and swap and since there is no more element, we move on.



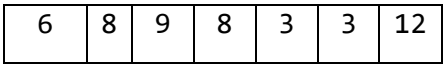
Again, we compare 3 with 9 first. Since 3 is smaller, we swap them. Then we compare it with 8 and swap, with 8 again and swap, with 6 and swap and since there is no more element that is bigger than 3, we move on.



12 is in the correct place, we don't change its position.

Insertion sort is a stable sorting algorithm. We only swap if the number is smaller than the element that it is comparing to. So the appearance of two equal numbers in the array doesn't change for input array and sorted array.

Quick Sort

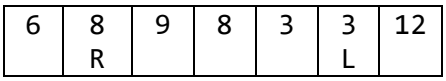


right-> <-left

current = 6

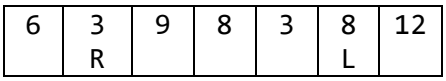
We iterate “right” to right until we find an element that is >= 6
We iterate “left” to left until we find an element that is <= 6

We repeat this until right index is bigger than left index.

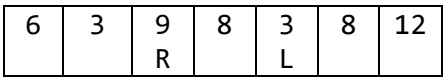


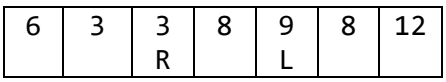
R: right L: left

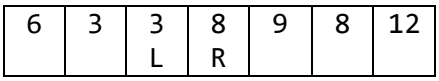
We swap left and right elements if right didn’t get through left.



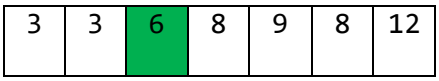
We repeat this process.



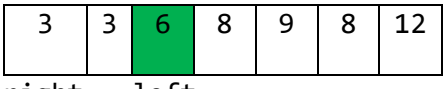




L got through R so we stop here and change current with L.

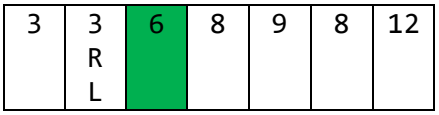


Now 6 is in its right place. We will repeat the same process again for left and right side.

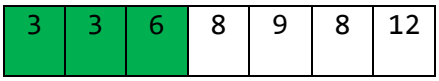


right left

current = 3

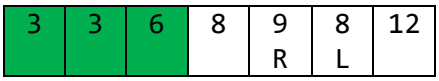


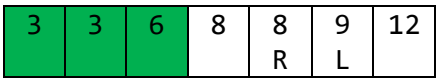
Since R and L are overlapping, we stop and change 3 with 3 (first 2 indices).

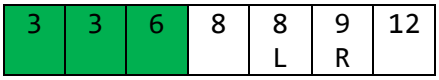


We continue with right hand side of the 6.

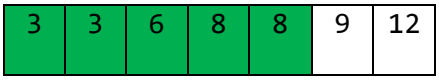
current = 8





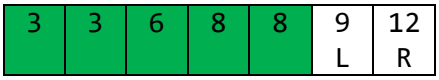


L got through R so we stop here and change current with L.

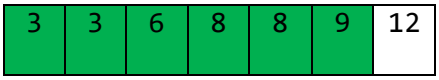


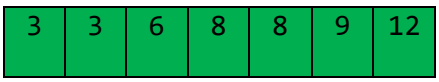
Finally we need to sort last 2 elements

current = 9



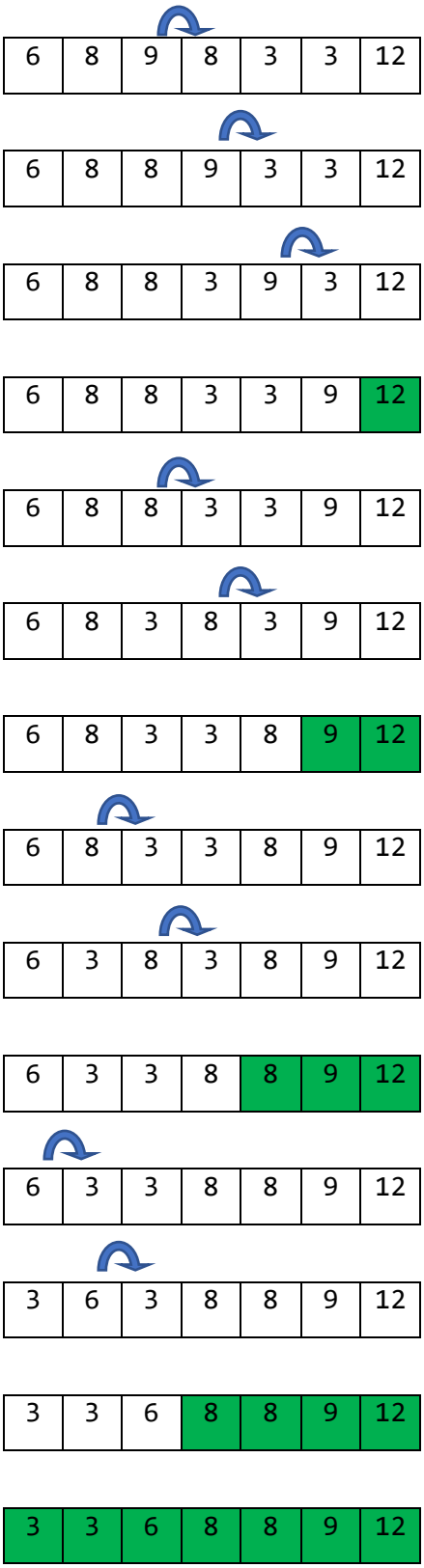
L got through R so we stop here and change current with L.





Quick sort is not a stable sorting algorithm because, for example, we swap 3 with 3 even though they are equal. The appearance of two equal numbers in the array changes for input array and sorted array.

Bubble Sort



At each iteration we look two adjacent elements and sort them. 6-8 don't swap, 8-9 don't swap, 9-8 swap because 9 needs to come later than 8. 9-3 swap, 9-3 swap, 9-12 don't swap. With this process, we put last element to its proper position.

Now last element is in its right place so we look until we reach second last element. We repeat the same process, if we find an adjacent pair that bigger element comes before the smaller element we swap immediately.

With this process, we sort the array starting from the last element.

Bubble sort is a stable sorting algorithm. We only swap if the number is bigger than the element at next index that it is comparing to. So the appearance of two equal numbers in the array doesn't change for input array and sorted array.

QUESTION 5

(a)

Firstly, brute force is solving a problem through exhaustion. We go through all possible choices until a solution is found. In cyber security, for example, brute force is used as an attack type. It is used to get login information, encrypted keys, etc. If we get the hashed version of a key and if we know the algorithm it is hashed, we can try many inputs to find the correct one that creates the hash we have.

Exhaustive search, on the other hand, is an important special case of brute force. It is simply a brute force approach to combinational problems. It suggests generating each and every element of the problem domain.

(b)

Caesar Cipher is one of the encryption techniques. With this technique, every letter in a text is changed with another letter. This another letter is determined by a fixed shift value. If the original letter is M and shift value is 3, J is replaced by M. This technique is vulnerable to brute force attack. A person can try shift numbers and find the ciphered text.

AES stands for Advanced Encryption Standard. It is used for ciphering the electronic data. There are different processes in this standard which are: AddRoundKey, SubBytes, ShiftRows, MixColumns, AddRoundKey, SubBytes, ShiftRows, AddRoundKey in order.

- AddRoundKey: Each byte's combined with a byte of the round key using xor.
- SubBytes: Each byte's replaced with other byte according to substitution table.
- ShiftRows: Shifting the rows of the state.
- MixColumns: Mixing the columns of the state.

AES is also vulnerable to brute force attack but with current and foreseeable hardware, recovering an AES-128 key takes billions of years.

(c)

As the number n grows, also amount of work (numbers that need to be checked) also grows. Here we iterate n-2 times so time complexity is O(n) in terms of n, the given number. The only thing that comes to my mind is binary system because number of bits to represent n, is simply O(logn) and therefore n is actually $O(2^b)$. So we can represent the running time as $O(2^b)$ where b is the number of bits that n consists.