CSE 321 – HOMEWORK 3

■ Q1:

Version1 (DFS Version)

Use Deep for Search . Find final order of resulting DFS. Then reverse the DFS result.

The graph is represented by Dictionary ADT in Python.

```
graph = {
        "CSE102" : ["CSE241"],
        "CSE241" : ["CSE222"],
        "CSE222" : ["CSE321"],
        "CSE211" : ["CSE321"],
        "CSE321" : ["CSE422"],
        "CSE422" : []
        }
```

Firstly DFS is used for topological sort. Then the result is reversed to obtain actual result.

```
def  dfs(graph):

    visited = []
    finishOrder = []
    stack = []


    def dfs(node):
        if(node not in visited):
            visited.append(node)
            stack.append(node)
            for node in graph[node]:
                dfs(node)
            finishOrder.append(stack.pop())


    for node in graph.keys():
        if(node not in visited):

            dfs(node)

    return finishOrder
```

- Worst Case Analiysis of DFS

This algorithm is like adjacency list format. Each vertex has some edges. This algorithm uses list to represent stack and visited nodes.

Number of edges = |E|

Number of vertices = |V|

SthList.append() takes O(1).

Average case of in depends on ADT. Average case of “in” within the if statement that used for list take O(N) time. N = |V|.

We must traversal all the vertices and their edges. **So WORST CASE = O(|E| +|V|).**

```
def topologicalSortV1(graph):
    result = dfs(graph)
    result.reverse()
    return result
```

- Worst Case Analysis of TOPOLOGICAL SORT V1:

DFS TAKES O(|V| + |E|).

result.reverse() takes O(|V|).

WORST CASE ->> O(|V| + |E|).

Version2 (None DFS Version)

Explanation:

Repeatedly , identify the source node (with no incoming edge) of remaining graph . Then note it down and delete it from along with all ongoing edges from it.

```
def topologicalSortV2(graph):

    finishOrder = []
    count = {}
    keyList = graph.keys()
    def fillWithZero():

        for key in keyList:
            count[key] = 0

    def findCount():
        for key in keyList:
            for node in graph[key]:
                count[node] = count[node] +1

    def findSource():
        for key in keyList:
            if(count[key] == 0):
                finishOrder.append(key)
                count[key] = -1
                decraseCount(graph[key])

    def decraseCount(vertex):

        for edges in vertex:
            count[edges] -=1


    fillWithZero()
    findCount()
    findSource()
    return finishOrder
```

Worst Case Analysis:

- fillWithZero() = Default Count of Ongoing Edges of Each Vertecies .

  Complexity ->> O|V|. Number of vertices.

- findCount() = Find Total Count of Ongoing Edges of Each Vertecies.

  Complexity ->> O|V| *O|E| = O(|V|*|E|).

- findSource() = Find the Source Node's. If the source node is found, then delete it along with all the ongoing edges from it.

  Complexity ->>
  Append to the list take O(1).
  Traverse all the vertices and their edges. O(|V| * O|E|).

- decreaseCount() = decrease count of Ongoing Edges of desired vertex by one .
  Complextiy ->> Traverse all the edges of vertex. O(|E|).

  Total Worst Case Compexity ->> O(|V|*|E|).

■ Q2:

FINDING EXPONANTIAL WITH LOGN TIME.

Explanation:

$a^{2n} = a^n * a^n$

$a^n = a^{n/2} * a^{n/2}$

…

$a^1 = a^1$

So the total step is required log(2n) in base 2.

Ex:

$5^{16} = 5^8 * 5^8$

$5^8 = 5^4 * 5^4$

$5^4 = 5^2 * 5^2$

$5^1 = 5^1$

```
def calculateExponantial(base, exp):

    if(exp == 1):
        return base
    if(exp % 2 == 0):
        return (base**(exp/2))*calculateExponantial(base , (exp/2))
    else:
        return (base**((exp+1)/2))*calculateExponantial(base , ((exp-1)/2))
```

- WORST CASE ANALYSIS

T(N) = 2T(N/2)+1.

We use Master Theorem .

$T(N) = aT\left(\frac{N}{b}\right) + f(N).$

$f(N) = n^k (\log_2 n)^p$

So formula is Θ ($n^k * (\log_2 n)^{p+1}$) and k = 0 , p = 0. So the WORST CASE ->> $\Theta (\log_2 n)$

■ Q3:

9X9 SUDOKU SOLVER

Explanation:

Each empty cell represented by 0. As shown below picture.

First, the not empty cell is found. Then the cell fill with value in range of 1 to 9.Then the cell value is checked if it matches correct value according to its row , column , and 3x3 subgroup.

If the value is suited for this cell then the other cell is checked regarding sudoku border. This rules is applied recursively. If the value of cell is not suited then one step back. This method is called backtracking.

This rules is applied to the reach successful solution of sudoku.

```
#SUDOKU REPRESENTATION USING 2D ARRAY
sudoku  = [ [0,0,3,9,0,0,7,6,0],
            [0,4,0,0,0,6,0,0,9],
            [6,0,7,0,1,0,0,0,4],
            [2,0,0,6,7,0,0,9,0],
            [0,0,4,3,0,5,6,0,0],
            [0,1,0,0,4,9,0,0,7],
            [7,0,0,0,9,0,2,0,1],
            [3,0,0,2,0,0,0,4,0],
            [0,2,9,0,0,8,5,0,0]]
```

CODE

```python
def sudokuSolver9x9(sudoku):

    def isValid(val,r,c):
            result = True
            if(val in sudoku[r]):
                result = False
            if(result and (val in [sudoku[i][c] for i in range(0,9)])):
                result = False
            if(result and (val in [sudoku[i][j] for i in range(((r//3)*3),(((r//3)*3)+3)) for j in range(((c//3)*3),(((c//3)*3)+3))])):
                result = False

            return result

    def solve (sudoku , r=0,c=0):

        if(r == 9):
            return True
        elif(c == 9):
            return solve(sudoku,r+1,0)
        elif (sudoku[r][c] != 0):
            return solve(sudoku , r,c+1)
        else :
            for i in range (1,10):
                if(isValid(i,r,c)):
                    sudoku[r][c] = i
                    if(solve(sudoku,r,c+1)):
                        return True
                else:
                    sudoku[r][c] = 0
            return False

    solve(sudoku)
```

- WORST CASE ANALYSIS:

Each cell have 9 possible value that is range of 1 to 9. And suppose we have n empty cell.

So Worst Case Complexity ->> $O(9^n)$

■ Q4:

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

- **Bubble Sort ->**

Pseudo code :

```
function (array, n)
    i = n -1
    flag = 0
    while i >0
        For j = 1 to i
            If a[j-1] > a[j]
                swap a[j-1] and a[j]
                flag =1
        if flag == 0 return
        flag = 0
        --i
```

- array = {6, 8, 8, 3, 3, 9, 12} i = 6
- array = {6, 8, 3, 3, 8, 9, 12} i = 5
- array = {6, 3, 3, 8, 8, 9, 12} i = 4
- array = {3, 3, 6, 8, 8, 9, 12} i = 3
- array is already sorted , return array i =2

Bubble sort is **stable** algorithm. Since the elements are swapped only if previous element is less than next element . If the elements are equal then there is no swap.

- **Quick Sort ->**

Algorithm :

Procedure quicksort (first , last)

If first < last then

Partition the elements in the subarray first to last , then return new pivot index

Recursive call : quicksort(first , pivot -1)

Recursive call : quicksort(pivot+1, last)

STEPS:

- array = { 3,3,6,8,9,8,12} first = 0 , last = 6 , pivot = array[2] = 6
- array = {3,3,6,8,9,8,12} first = 0 , last = 1 , pivot = array[1] = 3 ,
  the stability of the algorithm is lost there. Array[0] and array[1] = 3 but they are swapped .
- array ={3,3,6,8,8,9,12 } first =3 , last = 6 , pivot = array[4] .
  Also the stability is lost here . array[3] = array[4] = 8 , but they are swapped .
- array = { {3,3,6,8,8,9,12 } first = 4 , last = 6 . pivot = array[5].
  So array is sorted but **The algorithm is not stable.**

- **Insertion Sort ->**

Pseudo Code

```
Procedure insertionsort(array [0: n-1])

For i = 1 to n do

        current = array[i]

        pos = i -1

        while pos >=0 and current < array [ pos ]

                array[pos+1] = array[pos]

                pos = pos -1

array [pos+1] = key
```

STEPS

- array = {6,8,9,8,3,3,12} i = 1 j =0
- array = {6,8,9,8,3,3,12} i = 2 j =1
- array = {6,8,9,8,3,3,12}
  - {6,8,9,9,3,3,12} i=3 j =2
  - {6,8,8,9,3,3,12} i=2 j =1
- array = {6,8,8,9,3,3,12}
  - {6,8,8,9,9,3,12} I = 4 pos = 3
  - {6,8,8,8,9,3,12} i = 4 pos =2
  - {6,8,8,8,9,3,12} i = 4 pos = 1
  - {6,6,8,8,9,3,12} i = 4 pos = 0
  - {3,6,8,8,9,3,12} i=4 pos = -1
- array = {3,6,8,8,9,3,12}
  - {3,6,8,8,9,9,12} i=5 pos =4
  - {3,6,8,8,8,9,12} I = 5 pos = 3
  - {3,6,8,8,8,9,12} i= 5 pos =2
  - {3,6,6,8,8,9,12} i =5 pos =1
  - {3,3,6,8,8,9,12} I = 5 pos = 0
- array = {3,3,6,8,8,9,12}
  - array = {3,3,6,8,8,9,12} i = 6 , pos =5

Algorithm is stable since we only shift the value , if the current value is less than array[pos] . There is no shifting process between equal values.

■ Q5:

a- Definition of brute force and exhaustive search

Brute force search is applied for finite domain set of given problem. It search all the possible answer for problem solution even if the answer is correct or not. Then find correct value from them .

Exhaustive is subset of brute force search. Whereas exhaustive search works with some constrains to minimize or maximize the solution set. In the other hand , brute force does not have any constrain to reach correct solution.

b- Ceaser and AES (Rijndael) vulnerability to brute force attack.

Both are vulnerable to brute force like all the other cryptographic algorithms. Finding a cipher key of Ceaser is highly easier. Since the each letter of plaintext is shifted as much as letter size of used language. So the key is found easily using brute force.

In the other hand , The AES (Rijndael Block Cipher) key could be 128, 192 or 256. So finding the key and calculation's for decryption to find plaintext could be taken billion years using brute force works on super computer.

c- NAVIE PRIMARITY TEST

Even if the naïve primality test algorithm complexity is seemed as O($\sqrt{n}$), We assume that the arithmetic operation takes O(1) time . Essentially It depends on number of bits of given number . Number of bits of given number is $\log_2 n$. so the complexity becomes O(√n) = O((√$2^{\log_2 n}$)). As a result , the complexity grows exponential.

MUHAMMED SINAN PEHLIVANOĞLU

190 104 2664