Project Report

Design & Analysis of Algorithms

Muhammad Ali	21I-0887
Nuzhat ul Ain	21I-0634
Abubakar	21I-1379

Problem 1 (Strings):

Pseudocode:

```
FUNCTION rabinkarp(array, data, q, d):
    FOR find IN array:
        p = 0
        t = 0
        n = length(data)
        m = length(find)

        IF n < m:
            RETURN false

        result = 1
        FOR i = 0 to (m-1) - 1:
            result = result * d
        h= result % q
    FOR i = 0 to m-1 :
        p = p + find[i]
        t = t + data[i]

    matched = false
    i = 0

    FOR i = 0 to n - m:
        IF p == t:
            matched = true
            BREAK

        IF i < n - m:
            t = (t - data[i]) + data[i + m]

    IF NOT matched:
        RETURN false

    ne = ""
    FOR j = 0 to n - 1:
        IF j < i:
            ne = ne + data[j]
        ELSE IF j > (i + m + 1):
            ne = ne + data[j]
    data = ne

RETURN true
```

Time Complexity Analysis:

- The outer loop runs O(k) times, where k represents the number of strings in array.
- Initializing p, t, n and m take O (1) time.
- The inner power loop runs O(m) times, where m is the length of string in array.
- P and t are computed in O(m) time.
- The next inner loop runs O(n-m+1) times performing constant operations i.e. comparing p and t and updating the value of t if p and t don’t match.
- Next loop for constructing a new string runs O(n) times.

The overall time complexity is O(k(m+m+(n-m+1)+n)= O(k(2n))=O(kn).

Space Complexity Analysis:

- Variables ‘p’ and ‘t’ have a space complexity of O(m) where m is the length of the string to be found.
- The string variable ‘ne’ has a maximum space complexity of O(n). That is the worst case where the previous version of data remains unchanged.
- The remaining variables use O(1) space.
  Therefore the total space complexity is O(m+n)= O(n).

Problem 2 (Graphs):

Pseudocode:

```
FUNCTION NearestNeighbour( vertices, size, matrix, vertixCost,  totalCost):
verticesVisited = 1
dynamically declare array vertices of size ‘size’ initialized to false
visited[0]=true
declare vector indexes
indexes.push_back(0)
index=0
```

```
while (verticesVisited != size)
        min=INT_MAX
        for i from 0 to size-1
                if (visited[i] != true && matrix[index][i] != 0 && matrix[index][i] <
min)
                        min = i;
        index=min
        visited[index]=true
        indexes.push_back(index)
        verticesVisited++
indexes.push_back(0)
twoOpt(indexes, vertices, matrix, size)

FUNCTION twoOpt( indexes, vertices, matrix, size):
improve=true
while(improve)
        improve=false
        for i from 0 to size-2
                for j from i+2 to size
                        first=0
                        second=0
                        total=0
                        for i from 0 to size-1
                                first=indexes[i]
                                second=indexes[i+1]
                                total=total+matrix[first][second]
                        oldDistance=total
        while(i+1<j-1)
            int temp=indexes[i]
            indexes[i]=indexes[j]
            indexes[j]=temp
            (i+1)+1
            (j-1)-1
        first=0
        second=0
        total=0
        for i from 0 to size-1
            first=indexes[i]
            second=indexes[i+1]
            total=total+matrix[first][second]
        newDistance=total
        if( newDistance < oldDistance)
            improve=true
            break
        while(i+1<j-1)
            int temp=indexes[i]
            indexes[i]=indexes[j]
            indexes[j]=temp
            (i+1)+1
            (j-1)-1
    if(improve)
        break
```

Time Complexity Analysis:

- The function NearestNeighbour runs the main while loop for the number of vertices V where V is equals to n and the inner loop runs V times to find the next vertex whose edge has the minimum possible distance. Hence the code runs $O(V^2)$ or $O(n^2)$ times.
- Initializing all the variables take O (1) time.
- The function NearestNeighbour then calls the function twoOpt after it has executed its codes means the complexities for both the code will be added.
- The function twoOpt executes and the two outer for loop run for n-2 and n+2 times respectfully in nested format which means their complexity is $O(n^2)$ and there's an inner loop that calculates the distance each time the loops run and the while loop inside the nested loop run for n/2 times hence the total time complexity foe this code becomes $O(n^3)$.
- The main outer while loop is used to execute the code till there are improvements being made however if an improvement is made the code automatically breaks the execution of the double for loop hence the complexity of the function does not change from $O(n^3)$
- The overall time complexity is
    - $O(n^2 + n^2(n+ n/2)) = O(n^2 + n^3 + n^3/2) = O(n^2 + 2n^3) = O(n^3)$

Space Complexity Analysis:

- The array matrix has a space complexity of n^2 as it forms a 2D matrix for the number of vertices to store the value of each edge.
- There's a 1D array of size n to save the battery drained by the robot after visiting each vertex and another 1D Boolean array to check if vertices have been visited or not.
- There's 2 vectors or size n+1 to save the indexes of the path and the names of the vertices.
- In total $O(n^2 + n + n + n + n) = O(n^2 + 4n) = O(n^2)$

Websites used to learn about the algorithms used in Q2:

Nearest neighbour algorithm - Wikipedia

2-opt - Wikipedia

Traveling Salesman Problem With the 2-opt Algorithm | by Adam Davis | Medium

Problem 3 (Dynamic Programming):

PART A:

Pseudocode:

```
ComputeTotalNumberOfWays(n):
Declare a dynamic array dp of size n+1
dp[0]=1
dp[1]=1
dp[2]=2
for i from 3 to n
        dp[i]=dp[i-1]+dp[i-2]
totalWays=dp[n]
Deallocate memory for dp array
RETURN totalWays
```

Time Complexity Analysis:

- The loop runs for time complexity of O(n).
- The remaining statements take O(1) time complexity.

The overall time complexity is O(n).

Space Complexity Analysis:

- The array declared takes space complexity of O(n).
  The overall space complexity is O(n).

"n" e-mails	Number of ways
3	3
8	34
75	3416454622906707
1225	7.40×10^{255}

PART B:

Pseudocode:

```
FUNCTION compute_optimal_cost_and_path( dp[], n, c[][MAX_N] , parent[]):
dp[0]=0
for j from 1 to n-1
    dp[j]=INF
    for i from 0 to j-1
        if (dp[i] + c[i][j] < dp[j])
            dp[j] = dp[i] + c[i][j];
            parent[j] = i;

creating a stack called path of type int
curr = n-1
total_cost = dp[n-1]
while curr is not equal to 0
    push curr+1 onto path
    curr = parent [curr]
push 1 onto path
output “The optimal path is”
while path is not empty
    output path.top()
    path.pop()
    if path is not empty
        output “->”
output newline
output “The total optimal cost is” , total_cost
```

Time Complexity Analysis:

- The first outer loop runs O(n-1) times and the inner loop runs O(n-2) times which leads to time complexity of O(n^2).
- The while loop pushes path into stack which at worst time complexity will be O(n).
- Next while pops and prints the path which can be at max O(n).
- The remaining statements take constant time O(1).

The overall time complexity is O(n^2).

Space Complexity Analysis:

- The space complexity of ‘c’ is O(n^2).
- The space complexity of ‘dp’ and ‘parent’ is O(n).
- The space complexity of stack ‘path’ can be up to O(n).
- The remaining variables take constant space complexity O(1).

The overall space complexity is O(n^2).