Algorithm Design and Analysis

Project Report

Mahad Saleem (I210475)

Talha Zahoor (I210867)

Bilal Ur Rehman (I210472)

QUESTION # 1:

String matching algorithm

PSEUDO CODE:

```
1)This function matches the string.
function string_matcher(nums: vector<string>, group: vector<string>, visited: vector<bool>)
-> bool
    indices = []    // O(1)
    for i = 0 to nums.size() do    // O(n)
        found = false    // O(1)
        for j = 0 to group.size() do    // O(m)
            if nums[i] = group[j] and visited[i] = false then    // O(1)
                found = true    // O(1)
                break    // O(1)
            end
        end
        if found then    // O(1)
            indices.push_back(i)    // O(1)
        end
    end

    for z = 0 to indices.size() do    // O(k)
        status = true    // O(1)
        checked = visited    // O(n)

        for i = indices[z] to indices[z] + group.size() do    // O(m)
            found = false    // O(1)

            for j = 0 to group.size() do    // O(m)
                if nums[i] = group[j] and checked[i] = false then    // O(1)
```

```
                checked[i] = true    // O(1)
                print "--> ) Ingredient ", nums[i], " Found"    // O(1)
                found = true    // O(1)
                break    // O(1)
            end
        end

        if not found or indices[z] + group.size() > nums.size() then    // O(1)
            status = false    // O(1)
            print "--> ) Ingredient ", nums[i], " Not Found"    // O(1)
            break    // O(1)
        end
    end

    print ""    // O(1)
    if status then     // O(1)
        visited = checked    // O(n)
        return true    // O(1)
    end
  end

  return false    // O(1)
end
```

TC: 0(n * m * k)
N=size of nums vector
M=size of group vector
K=number of indices of each group.

This complexity comes from the nested loops that iterate over the nums and group vectors, as well as the outer loop that iterates over the indices vector, which is at most k elements long.
The space complexity of the code is O(n), as it creates a new vector of indices with a size proportional to the size of the input vector nums. Additionally, it creates two new boolean vectors visited and checked, each with a size proportional to the size of the input vector nums.

2)main function.

```
int main()
{
  vector<vector<string>> groups;    // O(1)
  vector<vector<string>> nums;      // O(1)
  ifstream myfile("input.txt");     // O(1)
  string line;                      // O(1)
  bool done = false;                // O(1)
  while (getline(myfile, line))     // O(n), where n is the number of lines in the file
  {
    vector<string> group;         // O(1)
```

```
        vector<string> num_list;      // O(1)

        stringstream ss(line);        // O(1)
        string ingredient;            // O(1)

        // Read group ingredients
        while (getline(ss, ingredient, ',') && !done)  // O(m), where m is the number of
ingredients in the current line
        {
            string temp;              // O(1)
            bool flag = false;        // O(1)
            int counter = 0;          // O(1)

            for (int i = 0; i < ingredient.size(); i++)  // O(k), where k is the length of the current
ingredient
            {
                if (ingredient[i] == 'n' && ingredient[i + 1] == 'u' && ingredient[i + 2] == 'm' &&
ingredient[i + 3] == 's')
                {
                    done = true;      // O(1)
                    break;            // O(1)
                }
                if (ingredient[i] == '\"')
                {
                    flag = true;      // O(1)
                    counter++;        // O(1)
                }
                if (flag && ingredient[i] != '\"')
                {
                    temp += ingredient[i];  // O(1)
                }

                if (counter == 2)
                {
                    break;            // O(1)
                }
            }
            group.push_back(temp);    // O(1)
        }
        if (done)
        {
            // Read nums
            string num_str;           // O(1)
            getline(ss, num_str);     // O(1)
            stringstream ss_nums(num_str);  // O(1)

            while (getline(ss_nums, num_str, ','))  // O(p), where p is the number of elements in
the current list
```

```
        {
            string temp;          // O(1)
            bool flag = false;    // O(1)
            int counter2 = 0;     // O(1)
            for (int i = 0; i < num_str.size(); i++)  // O(q), where q is the length of the current
element
            {
                if (num_str[i] == '\"')
                {
                    flag = true;    // O(1)
                    counter2++;     // O(1)
                }
                if (flag && num_str[i] != '\"')
                {
                    temp += num_str[i];  // O(1)
                }
                if (counter2 == 2)
                {
                    break;              // O(1)
                }
            }
            if (counter2 != 0)
            {
                num_list.push_back(temp);    // O(1)
            }
        }
        if (!num_list.empty())
        {
            nums.push_back(num_list);   // O(1)
        }
    }
    if (!done)
    {
        groups.push_back(group);     // O(1)
    }
}

// Print the groups values
for i = 0 to groups.size() - 1 do        // O(groups.size())
    print "Group " + (i + 1) + ": "      // O(1)
    for j = 0 to groups[i].size() - 1 do // O(groups[i].size())
        print groups[i][j] + " "         // O(1)
    end for
end for
print "\n\n\n" // O(1)

// Print the nums values
for i = 0 to nums.size() - 1 do          // O(nums.size())
```

```
    print "Nums " + (i + 1) + ": "       // O(1)
    for j = 0 to nums[i].size() - 1 do   // O(nums[i].size())
        print nums[i][j] + " "           // O(1)
    end for
end for
print "\n" // O(1)

// Print the output values
for i = 0 to nums.size() - 1 do          // O(nums.size())
    flags = create a vector of size nums[i].size() and initialize all values to false   //
O(nums[i].size())
    x = false                            // O(1)
    y = 0                                // O(1)
    print "\n================================= TEST CASE NO : " + (i + 1) + "
=================================="   // O(1)

    for j = 0 to groups.size() - 1 do    // O(groups.size())
        x = string_matcher(nums[i], groups[j] , flags)  // O(nums[i].size() * groups[j].size())
        if x then
            y = y                        // O(1)
        else
            y = y + 1                    // O(1)
        end if
    end for

    if y >= 1 then
        print "-->FINAL OUTCOME : FALSE\n"  // O(1)
    else
        print "-->FINAL OUTCOME : TRUE\n"   // O(1)
    end if
end for
```

ASYMPTOTIC TIME - COMPLEXITY ANALYSIS

//-- Complexity for main

The overall time complexity of the given code can be analyzed as follows:

- Reading the input file: O(n * m) where n is the number of lines in the file and m is the average number of ingredients in each line. The nested while and for loops to read the ingredients and numbers have time complexities of O(m) and O(k) respectively, where k is the length of the ingredient or number.
- Printing the groups values: O(g * m) where g is the number of groups and m is the average number of ingredients in each group.
- Printing the nums values: O(n * m) where nu is the number of number lists and m is the average number of elements in each list.
- Printing the output values: O(nl * g) where nl is the number of number lists, g is the number of groups.

Therefore, the overall time complexity of the code can be expressed as:

O(n * m + g * m + n * u + n* g) = O(nl* g)

Total time complexity = 0(n*m*k) * 0(nl*g)
TOTAL SPACE COMPLEXITY = 0(N * M)

QUESTION # 2:

Pseudo code :

1)Designing a graph class with this method.

```
Graph:
    num_vertices = n
    adj_list[n][n]
```

2)constructor for graph taking value n as number of vertices in the graph.

```
    Graph(n):
        FOR i = 0 to n-1:
            adj_list[i][n]
            FOR j = 0 to n-1:
                adj_list[i][j] = 0
```

TC: 0(n*n)

3)adding edges in the graph by taking source and destination vertex with weight.

```
    addEdge(src, dest, weight):
        adj_list[src][dest] = weight
        adj_list[dest][src] = weight
```

TC: 0(1)

4)printing the graph.

```
printGraph():
    FOR i = 0 to num_vertices - 1:
        PRINT "Vertex ", i, ": "
        FOR j = 0 to num_vertices - 1:
            IF adj_list[i][j] != 0:
                PRINT "(", j, ", ", adj_list[i][j], ") "
        PRINT
```

TC: 0(n*n)

5)finding Hamiltonian circuit.

In the base case, if the current path contains all vertices and the last vertex is adjacent to the start vertex, then the total time for completing the circuit is calculated. If this time is shorter than the current shortest time, the shortest time is updated and the shortest Hamiltonian circuit is stored in the "shortPath" array.

In the recursive case, the method extends the current path with all unvisited adjacent vertices. It checks if the vertex is adjacent to the last vertex in the path and whether it has already been visited. If it is a valid vertex, it is added to the path and the method is called recursively with the updated path and visited arrays. If the vertex is not part of a Hamiltonian circuit, it is removed from the path and the visited array is set back to false.

```
findHamiltonianCircuits(shortPath, path, visited, start, arr, curr_pos = 1):
    n = num_vertices
    static shortest_time = INT_MAX
    total_time = 0
    IF curr_pos == n AND adj_list[path[curr_pos - 1]][start] != 0:
        total_time += arr[path[0]] + adj_list[path[n - 1]][start]
        FOR i = 0 to n - 2:
            total_time += arr[path[i + 1]] + adj_list[path[i]][path[i + 1]]
        IF total_time < shortest_time:
            shortest_time = total_time
            FOR i = 0 to n - 1:
                shortPath[i] = path[i]
            shortPath[n] = start
        RETURN shortPath
    FOR v = 0 to n - 1:
        IF adj_list[path[curr_pos - 1]][v] != 0 AND !visited[v]:
            flag = false
            FOR i = 0 to curr_pos - 1:
                IF path[i] == v:
                    flag = true
                    BREAK
            IF !flag:
                visited[v] = true
                path[curr_pos] = v
                findHamiltonianCircuits(shortPath, path, visited, start, arr, curr_pos + 1)
```

```
                visited[v] = false
        RETURN shortPath
TC: 0(n! * n^2)
```

6) Same function as above just returns the shortest time

```
    findHamiltonianCircuitstime(shortPath, path, visited, start, arr, curr_pos = 1):
        n = num_vertices
        static shortest_time = INT_MAX
        total_time = 0
        IF curr_pos == n AND adj_list[path[curr_pos - 1]][start] != 0:
            total_time += arr[path[0]] + adj_list[path[n - 1]][start]
            FOR i = 0 to n - 2:
                total_time += arr[path[i + 1]] + adj_list[path[i]][path[i + 1]]
            IF total_time < shortest_time:
                shortest_time = total_time
                FOR i = 0 to n - 1:
                    shortPath[i] = path[i]
                shortPath[n] = start
            RETURN shortest_time
        FOR v = 0 to n - 1:
            IF adj_list[path[curr_pos - 1]][v] != 0 AND !visited[v]:
                flag = false
                FOR i = 0 to curr_pos - 1:
                    IF path[i] == v:
                        flag = true
                        BREAK
                IF !flag:
                    visited[v] = true
                    path[curr_pos] = v
                    findHamiltonianCircuitstime(shortPath, path, visited, start, arr, curr_pos + 1)
                    visited[v] = false
        RETURN shortest_time
```

ASYMPTOTIC TIME - COMPLEXITY ANALYSIS

TC: 0(n! * n^2)

7)In the main function , the graph object is created and an array of vertices , edges, weights,delivery time at each vetex and total time is made by reading from the text file.
TC: 0(n) for each array

The function Hamiltonian circuit is called which takes

- shortest path array which takes value of shortest path
- path which keeps track of all Hamiltonian circuits
- visited array which is bool and keeps track of the visited vertices
- array which contains the delivery time of each vertex

In the end the function calculates the shortest Hamiltonian circuit and returns it by storing value in the shortest path array.

Total time complexity = 0(n*n) + 0(1) + 0(n*n) + 0 (n! * n^2) +
0(n! * n^2) + 0(n) + 0(n) + 0(n) + 0(n)
= 0(n! * n^2)

Question 3

Part (A)

PSEUDOCODE CODE

```
function count_email_sequences(n) :
    if n <= 0 :
        return 0
    else if n == 1 :
        return 1
    else :
        dp[1] = 1
        dp[2] = 2
        for i from 3 to n :
dp[i] = dp[i - 1] + dp[i - 2]
return dp[n]
```

ASYMPTOTIC TIME - COMPLEXITY ANALYSIS

The time complexity of each step in the given pseudocode is as follows:

1. Comparing n with 0 takes O(1) time.
2. Comparing n with 1 takes O(1) time.
3. Initializing dp[1] and dp[2] takes O(1) time.
4. The loop from 3 to n takes O(n) time.
5. Computing dp[i] using the recurrence relation dp[i] = dp[i - 1] + dp[i - 2] takes O(1) time.
6. Returning dp[n] takes O(1) time.

Therefore, the overall time complexity of the given pseudocode is O(n), which is dominated by the loop from 3 to n.

Part (B)

PSEUDOCODE CODE

1. Define a function min_cost with parameters n, cand path.
2. Create two arrays dp and prev of size n.
3. Set dp[0] to 0 and prev[0] to - 1.
4. For i from 1 to n - 1 do:
    a.Set dp[i] to INT_MAX.
    b.For j from 0 to i - 1 do :
        i.Check if dp[j] is not INT_MAX and c[j][i] is not INT_MAX and dp[i] is greater than dp[j] + c[j][i].
        ii.If the condition is true, set dp[i] to dp[j] + c[j][i] and prev[i] to j.
5. Construct the path by setting idx to n - 1 and count to 0.
6. While idx is not - 1 do :
    a.Set path[count] to idx + 1.
    b.Set idx to prev[idx].
    c.Increment count by 1.
7. Reverse the path by swapping the values at path[i] and path[count - i - 1] for i from 0 to count / 2.

8. Set min_cost to dp[n - 1].
9. Delete the dynamically allocated memory for dpand prev.
10. Return min_cost.

ASYMPTOTIC TIME - SPACE COMPLEXITY ANALYSIS

The time complexity of each step in the given pseudocode:

1. Define a function min_cost with parameters n, cand path. - O(1)
2. Create two arrays dp and prev of size n. - O(n)
3. Set dp[0] to 0 and prev[0] to -1. - O(1)
4. For i from 1 to n - 1 do: - O(n)
   a. Set dp[i] to INT_MAX. - O(1)
   b. For j from 0 to i - 1 do: - O(i)
5. scss
6. Copy code
7. i. Check if dp[j] is not INT_MAX and c[j][i] is not INT_MAX and dp[i] is greater than dp[j] + c[j][i]. - O(1) ii. If the condition is true, set dp[i] to dp[j] + c[j][i] and prev[i] to j. - O(1)
8. Construct the path by setting idx to n - 1 and count to 0. - O(1)
9. While idx is not -1 do: - O(n)
   a. Set path[count] to idx + 1. - O(1)
   b. Set idx to prev[idx]. - O(1)
   c. Increment count by 1. - O(1)
10. Reverse the path by swapping the values at path[i] and path[count - i - 1] for i from 0 to count / 2. - O(n/2)
11. Set min_cost to dp[n - 1]. - O(1)
12. Delete the dynamically allocated memory for dp and prev. - O(1)
13. Return min_cost. - O(1)

Therefore, the time complexity of the entire pseudocode is O(n^2), which is dominated by the nested for loop in step 4.