QUESTION #1

Problem 1: String Matching

1. Read the groups and nums array correctly from the file.
2. Iterate through each dish in the groups list.
3. Create an array and for each dish, store the count of ingredients found in the nums array.
4. The number of ingredients in the nums array has to be equal to or greater than the ingredients in the groups array.
5. Iterate through the nums array using a sliding window of the size equal to the number of ingredients in the current dish.
6. Check if all the ingredients in the current dish are present in the current sliding window of the nums array. If yes, then remove that ingredient from the nums array and continue with the next dish.
7. If not, continue sliding the window through the nums array until you find a valid subarray or reach the end of the nums array.
8. If you reach the end of the nums array without finding a valid subarray for the current dish, return False.
9. If you find valid subarrays for all dishes in the groups' list, return True.

Pseudocode for my algorithm:

function read_data_from_file(filename, groups, test_cases):

Open file with given filename

Set is_groups_section to true

group_idx = 0

Loop through each line in the file:

If line contains "Test cases:":

Set is_groups_section to false

Continue to next iteration

If is_groups_section is true:

Process groups section line and store ingredients in groups array

Else:

Process test cases section line and store ingredients in test_cases vector

Close file

function check_and_remove_window(groups, nums, count):

Initialize index, other, and array of booleans

Loop through count array:

Extract dish from groups, sort dish

Loop through nums to find matching dish:

Extract nums_window, sort nums_window

If dish and nums_window are equal:

Remove nums_window from nums

Set corresponding element in array to true

Break inner loop

If all elements in array are true, return true

Else, return false

function can_prepare_all_dishes(flattened_groups, count, nums):

If check_and_remove_window(flattened_groups, nums, count) returns false:

Return false

Return true

function flatten_groups(groups, group_count):

Initialize flattened_groups vector

Loop through groups:

Loop through ingredients in each group:

Process and add ingredient to flattened_groups

Return flattened_groups

MAIN:

Initialize filename, groups array, and test_cases vector

Call read_data_from_file with filename, groups, and test_cases

Process groups and store the count of ingredients in each group

Call flatten_groups to get flattened_groups

Loop through test_cases:

Call can_prepare_all_dishes with flattened_groups, count, and test case

Print result

Let N be the total number of ingredients in the groups array

Let M be the total number of ingredients in the nums array

Let T be the number of test cases

Time complexity:

File reading= O(N+M)

Flatten group=O(N*M)

Can_prepare_all_dishes= worst case is O(N*M)

Main loop= O(T*N*M)

Overall time complexity= O(N*M)

Space Complexity:

To store nums and groups into array= O(N+M)

Flatten group=O(N*M)

Can_prepare_all_dishes= worst case is O(M)

Overall Space Complexity= O(N+M)

QUESTION # 2

Find Hamiltonian Circuit(G,N,V,T)

```
1)      CircuitTime <-- 0
2)      IsCircuitExist <-- false
3)      isEdgeExist <-- false
4)      Circuit [ N ]
5)      Circuit [ 0 ] <-- “h”
6)      For i=1 to N
7)              Do
8)                      Circuit [ i ] <-- “#”
9)      End for
10)             ptr <-- 1
11)             Running <-- true
12)     while Running
13)             Do
14)                     isEdgeExist <-- false
15)                     Ciruit [ ptr ] <-- getNextVertex( V, CV, N )
16)                     If Circuit [ 1 ] = “#”
17)                     Then
18)                             Running <-- false
19)                             break
20)                     End If
21)                     If ptr = 1
22)                     Then
23)                             Circuit_Time <-- 0
24)                     End If
25)                     If Circuit [ ptr ] = “#”
26)                     Then
27)                             ptr <-- ptr –1
```

```
28)                     continue
29)                 End If
30)                 If   VExist ( Circuit , ptr ) = false
31)                 Then
32)                     isEdgeExist <-- EdgeExist( Circuit[ptr-1],Circuit[ptr])
33)                 End If
34)                 If isEdgeExist = true
35)                 Then
36)                     Circuit_Time = getCircuitTime ( Circuit, ptr, N , G)
37)                     If Circuit_Time < T AND ptr + 1 < N
38)                     Then
39)                         ptr <-- ptr +1
40)                     End If
41)                 End if
42)                 If  ptr = N-1
43)                 Then
44)                  isEdgeExist <-- EdgeExist( Circuit[0],Circuit[ptr])
45)                     If isEdgeExist = true
46)                     Then
47)                         Circuit_Time+=EdgeWeight(Circuit[0],Circuit[ptr])
48)                         If Circuit_Time < T
49)                         Then
50)                             isCircuitExist = true
51)                             PRINT CIRCUIT
52)                          End If
53)                     End If
54)                 End If
55)     END WHILE
56)     If isCircuitExist = false
57)     Then
58)         PRINT “NO CIRCUIT EXIST”
59)     End If
```

string getNextVertex(V, CV, N)

```
1)  If  CV = "#"
2)  Then
3)          Return V [ 0 ]
4)  End If
5)  For  i = 0 to N
6)  Do
7)          If V[ i ] = CV
8)          Then
9)                  Return V[ i+1 ]
10)         End If
11)     End For
```

Bool VExist (Circuit, ptr)

```
1)  For i=0 to ptr
2)  Do
3)          If Circuit [  i ] = Circuit [ ptr ]
4)          Then
5)                  Retrun true
6)          End If
7)  End For
8)  Return false
```

Int getCircuitTime(Circuit, ptr, N, G)

```
1)  CTime <-- 0
2)  For i= 1 to ptr
3)  Do
4)          If Circuit [ i ] = "#"
5)          Then
6)                  continue
7)          End If
```

```
8)          CTIme += DeliveryTime ( Circuit [ i ] )
9)          CTIme  += EdgeWeight ( Circuit [ i-1 ] , Circuit [ i ] )
10)     End For
11)      Return CTime
```

SOME STANDS FOR'S

G = Graph

V = Vertex List

N = No of Vertices in Graph

T = Given Time

CV = Current Vertex

Analysis

The function **VEXIST(C,ptr)** runs in O(N) time, it just checks whether the current vertex that is Cicrcuit[ptr] exists in Current Circuit array or not!
if it is checking for the last vertex than it will check N-1 items of array, so in worst case it will run in O(N).

The function **getNextVertex(V, CV, N)** also runs in O(N) time, we just send the current vertex and it iterates through the Vertex List array and sends us the next vertex in Vertex List.

the funciton **getCircuitTime(Circuit, ptr, N, G)** also runs in O(N) time, we send the Current Circuit to the function, it calculates and sends the the total current time of circuit which is the delivery time of the vertex plus edge weights between the Vertices in the circuits.

The main Function **Find Hamiltonian Circuit** takes Graph, total vertices, Vertex list of the graph, and Givent time.
The function first allocates an array, which stores the circuit at the moment. The Vertex List V contains vertices except h, the home vertex and an additional dummy vertex "#" which marks the termination of array.

It takes the next vertex from vertex list at the position of ptr, it then checks if the vertex and previous vertex has and edge or not, if they have edge it proceeds to next position, if they don't have edge takes another Vertex from vertex list and repeats. And if we are at last vertex, we will check if this last vertex has an edge to first vertex of circuit, if it has and edge then we print the circuit. and one important thing before moving to next vertex it checks that current circuit time has exceeded the Given time or not, at any step if it exceeds the Given time, it will stop checking the that path and **Backtracks** to next possible path.

Overall Complexity of algo

Basically this algo is finding all permutations of Vertices, to check all possible paths. For each index in Circuit Array it updates and checks all possible Vertices from Vertex List. So it is running n times for n vertices i.e N^N which is **O(N!).**

QUESTION # 3

Part(a):

```
double emails(int n)
  begin:
   Double pointer  stored as new double pointer [n];
```

```
        stored[0] ß 1;
        stored[1] ß 1;
for loop runs for 2 to N
        {
        stored[i] = stored[i - 1] + stored[i - 2];
        }
End for
  return stored[n];

End emails
```

Asymptotic analysis:

Since loop runs for N-2 times its O() notation is O(n) which is linear time complexity.

“n” times email sent	num of ways
8	34

75 | 3.41645e+15

1225 | 7.40222e+255

Part(b) :

Void printdist (int* dist,int m)

Begin:

Display ”distance from point 1 to end:”

Display dist[n-1]

End function printdist

int min(bool* visited, int* dist, int n)

begin:

int minimum = infinity;

```
        int index = 0;
        int tnodes = 0;

        for  loop from 1 to n

        if (visited[i] == false && dist[i] <= minimum)

        minimum ß dist[i];
        index ß i;

End for
        return index;
end function min
void disjktra(int** matrix, int n)
begin:
  int* dist  as new int[n];
        bool* visited as new bool[n];
        int* path as new int[n];
        for loop from 0 to n-1
        dist[i] ß infinity;
        visited[i] ß false;
```

```
        end for
        dist[0] ß 0;
for loop from 0 to n-1
        int m = min( visited, dist,n);
        visited[m] = true;
        for loop from 0 to n-1
        if (!visited[k] && matrix[m][k]
        && dist[m] != infinity
        && dist[m] + matrix[m][k] < dist[k])
        dist[k] ß dist[m] + matrix[m][k];
        //path[k] = m;
        End for

        End for

 call   printdist(dist , n);
```

end function dijsktra

Asymptotic analysis:

Printdist runs only 1 statement so complexity of that function is O(1)

In min function for loop runs from o to N times so O(n)

In dijsktra function first for loop for assigning initial values is O(n)

Then a for loop run n times and insides that loop:

Min function called which runs N times

Then loop k runs n times

So total = $n^2 + n^2 = 2(n^2)+n+n$

Whole algo complexity = $O(n^2)$ quadratic complexity

Trace:

Sources:

GeeksForGeeks website for reference for identifying shortest path algorithm and dynamic behavior of fibonucci series