**Design and Analysis of Algorithms**

**Project**

**Group Members:** Sania Mirza(21I-0764), Zahra Rizwan(21I-0726), Zarina Shabbir(21I-2528)

**Problem 1** (*Done By Sania*):

**Algorithm:**

1. Open input file

2. Read input file line by line

3. For each line:

3.1 If the line contains "groups":

3.1.1 Extract the list of ingredients for the group

3.1.2 Split the list of ingredients by ", " and store each ingredient in a vector

3.1.3 Add the vector to the list of groups

3.2 If the line contains "{":

3.2.1 Extract the list of ingredients

3.2.2 Split the list of ingredients by ", " and store each ingredient in a vector

3.2.3 Add the vector to the list of groups

3.3 If the line contains "case":

3.3.1 Print the line

3.4 If the line contains "nums":

3.4.1 Extract the list of ingredients

3.4.2 Split the list of ingredients by ", " and store each ingredient in a vector

3.4.3 Call the function "dish()" with the list of groups and list of ingredients as arguments

3.4.4 Print the result returned by "dish()"

4. Close input file

Function dish(groups, nums):

1. For each group in groups:

1.1 Get the number of ingredients in the group

1.2 Create a flag array of size num\_ing

1.3 Set group\_found to true

1.4 For i = 0 to nums.size() - num\_ing + 1:

1.4.1 Initialize all elements of the flag array to false

1.4.2 For j = 0 to num\_ing:

1.4.2.1 For k = 0 to num\_ing:

1.4.2.1.1 If g[j] == nums[i+k], set flag[j] to true and break

1.4.3 For j = 0 to num\_ing:

1.4.3.1 Set group\_found to true if flag[j] is true, else set group\_found to false and break

1.4.4 If group\_found is true:

1.4.4.1 Set all elements of nums[i:i+num\_ing] to ""

1.4.4.2 Break the loop

1.5 If group\_found is false, return false

2. Return true.

**Asymptotic Time Complexity Analysis:**

The time complexity of the dish function O(n^3) where n is the length of the nums vector. This is because the function has three nested loops, each iterating over the entire length of the groups vector, the number of ingredients in the group and the number of elements in the nums vector respectively.

The main function has several loops but they are all linear in time with respect to the length of the input file. Therefore the overall complexity of the program is **O(n^3)** where n is the length if the input file.

**Problem 2** (*Done By Zahra*):

(Note: the solution comprises 6 functions in total)

**Algorithm:**

1. Open the input file

2. Read the input file line by line

BEGIN

read\_file()

edge\_array, vertex\_array, Weights\_array, cost t(A), t(B), t(B), Time\_limit T

read file from

while i to endofline

if V= found

then split input by delimiter

end if

else if E= found

then split input by delimiter

end if

else if W= found

then split input by delimiter

end if

else if W= found

then split input by delimiter

end if

else if t(A), t(B), t(B), T found

then convert to integer

end if

end while

call function final circuit

2. generates the final input

final\_Circuit(graph, tA, tB, tC, T, totalvertix)

edge\_exists = new int[ROWS]

for i from 0 to ROWS - 1 do

edge\_exists[i] = -1

end for

edge\_exists[0] = 0

if Process\_Circuit(graph, edge\_exists, 1, tA, tB, tC, T) == false (call)

then print "NOT FEASIBLE FOR HAMATONIAN GRAPH!"

return false

else

printCircuit(edge\_exists, totalvertix) (call)

end if

return true

end

3. Processes the Hamiltonian circuit

Process\_Circuit(graph, edge, loc, costA, costB, costC, Time\_Limit)

if loc == ROWS then

if graph[edge[loc - 1]][edge[0]] == 1 then

return true

else

return false

end if

end if

flag = false

for i from 1 to ROWS - 1 do

if costA < Time\_Limit then

if Check(i, graph, edge, loc, costA, costB, costC, Time\_Limit) then

edge[loc] = i

if Process\_Circuit(graph, edge, loc + 1, costA, costB, costC, Time\_Limit) == true && (costA + costB + costC) < Time\_Limit

then

return true

end if

edge[loc] = -1

end if

end if

end for

return false

end

4. Check function to process the cost case and path case too

Check(vertex, graph, edge\_exists, loc, costA, costB, costC, Time\_Limit)

if sum < Time\_Limit then

if graph[edge\_exists[loc - 1]][vertix] == 0 then

return false

for i from 0 to loc - 1 do

if edge\_exists[i] == vertex then

return false

end for

end if

return true

end

5. this function is gonna be used in file reading for index checking

get\_index(arr, val)

get = -9999

for i from 0 to length of (arr – 1) do

if arr[i] == val then

get = i

end if

end for

return get

end

6.Finally this function will print the final circuit

printCircuit(edge, totalvertix)

print "FEASIBLE FOR HAMILTONIAN CIRCUIT”

for i from 0 to totalvertix - 1 do

print edge[i], " "

end for

print edge[0]

end

END.

**Asymptotic Time Complexity Analysis:**

We can calculate the overall time complexity of the code is considering all the functions and adding them up. The void read and printCircuit functions both have a time complexity of O(n^2). The bool check function and get\_index both have a time complexity of O(n). The processCircuit and finalCircuit functions both have a time complexity of O(n!). Hence it can be stated the overall complexity is O(n!).

**Problem 3** (*Done By Zarina*)

**Part a:**

**Algorithm:**

Int n

Int[n] dp

dp[0]🡨0

db[1]🡨0

for i=1 to n

dp[i]🡨 dp[i-1]+dp[i-2]

print(dp[n])

iterative method.

it’s time complexity is O(n) linear and space complexity is O(n)

table:

|  |  |
| --- | --- |
| n emails | Numbers of ways |
| 3 | 3 |
| 8 | 34 |
| 75 | 3.4 x 10^15 |
| 1225 | 7.4 x 10^255 |

**Part b:**

**Algorithm:**

Part2(arr)

int[n+1] cost🡨 infinity

int[n+1] path🡨 0

for i🡨2 to n

for j🡨1 to i

if(cost[j]+arr[j-1][i-1]<=cost[i])

cost[i]🡨 cost[j]+arr[j-1][i-1]

path[i]=j

end if

vactor <int> backtrack

backtrack.push\_back(n)

while(n greater than 1)

n=path[n];

backtrack.puch\_back(n)

end while

for i🡨 sizeofbacktrack to -1

print(backtrack[i])

// for the minimum cost

Vector<int>dp

dp[0]🡨0

int min\_cost🡨 infinity

for i🡨 1 to n

min\_cost🡨 infinity

for j🡨0 to i

min\_cost=min(min\_cost,arr[j][i]+dp[j])

dp[i]=min\_cost

**Asymptotic Time Complexity Analysis:**

const int cn = 10; - O(1), constant time complexity.

int main() - O(1), constant time complexity.

vector<int>cost(cn + 1, INT\_MAX); - O(n), where n = cn + 1, as the vector is initialized with n elements and each element takes constant time to initialize.

vector<int>path(cn + 1, 0); - O(n), where n = cn + 1, as the vector is initialized with n elements and each element takes constant time to initialize.

cost[1] = 0; - O(1), constant time complexity.

vector<vector<int>> arr = {...}; - O(1), constant time complexity, as the 2D vector initialization is performed once, during the compilation time.

for (int i = 2; i <= cn; i++) { ... } - O(n^2), as this loop iterates n-1 times, and for each iteration, an inner loop is executed n/2 times on average. The overall time complexity is n\*(n/2) = O(n^2).

if ((cost[j] + arr[j - 1][i - 1]) <= cost[i]) { ... } - O(1), constant time complexity, as it only involves arithmetic operations and comparison.

cost[i] = cost[j] + arr[j - 1][i - 1]; - O(1), constant time complexity, as it only involves arithmetic operations.

path[i] = j; - O(1), constant time complexity.

vector<int> backtrack; backtrack.push\_back(cn); - O(1), constant time complexity.

int n = cn; while (n > 1) { ... } - O(n), as in the worst case, the while loop executes n-1 times, which is the length of the backtrack vector.

n = path[n]; - O(1), constant time complexity.

backtrack.push\_back(n); - O(1), constant time complexity.

for (int i = backtrack.size() - 1; i > -1; i--) { ... } - O(n), as the loop iterates over all elements of the backtrack vector.

cout << backtrack[i]; if (i != 0)cout << "->"; - O(1), constant time complexity.

vector<int>dp(arr.size(), INT\_MAX); - O(n), where n = arr.size(), as the vector is initialized with n elements and each element takes constant time to initialize.

dp[0] = 0; - O(1), constant time complexity.

for (int i = 1; i < cn; i++) { ... } - O(n^2), as this loop iterates n-1 times, and for each iteration, an inner loop is executed i times on average. The overall time complexity is 1+2+3+...+(n-1) = O(n^2).

int min\_cost = INT\_MAX; - O(1), constant time complexity.

for (int j = 0; j < i; j++) { ... } - O(n), where n = i, as the loop iterates i times at most.

min\_cost = min(min\_cost, arr[j][i] + dp[j]); - O(1), constant time complexity, as it only.

while (n > 1) loop takes at most n iterations since n is decreasing by at least 1 in each iteration. Each iteration involves constant-time operations, so the total time complexity is O(n).

The final for loop takes O(n) time since it iterates over the backtrack vector with n elements.

REFRANCES USED FOR HELP:

<https://www.geeksforgeeks.org/hamiltonian-cycle/>

<https://www.geeksforgeeks/>

<https://www.w3schools.com>