**CS-2009 Design and Analysis of Algorithms, Spring-2023 Project**

**Participants:**

* **Muhammad Abdullah Khan (21i-0805) [P1]**
* **Sail Ali Khan (21i-0649) [P2]**
* **Hamza Khalid (21i-0704) [P3]**

**P1 (Strings):**

**Calculate Hash Function:**

**Pseudocode:**

function calHash(num: string) -> hashes:

h <- hashes

h.hash <- 0

h.size <- length(num)

for k in [0, length(num)-1]:

h.hash <- h.hash + (num[k] - 96) \* (k+1)^10

return h

Time complexity: O(n), where n is the length of the input num.

Storage complexity: O(1).

**For file reading:**

**Pseudocode:**

procedure input()

infile <- open file "p1\_input.txt"

for each line in infile do

group <- empty vector of strings

num <- empty vector of strings

group\_h <- empty vector of hashes

num\_h <- empty vector of hashes

if line contains '{' then

start <- index of '{' + 1

end <- index of '}'

items <- substring of line from start to end - 1

ss <- stringstream initialized with items

while getline(ss, item, '\'') do

if getline(ss, item, ',') then

item <- substring of item from 0 to length of item - 1

end if

push item to group

push calHash(item) to group\_h

end while

push group to groups

push group\_h to groups\_h

else if line contains '[' then

start <- index of '[' + 1

end <- index of ']'

items <- substring of line from start to end - 1

ss <- stringstream initialized with items

while getline(ss, item, '\'') do

if getline(ss, item, ',') then

item <- substring of item from 0 to length of item - 1

end if

push item to num

push calHash(item) to num\_h

end while

push num to nums

push num\_h to nums\_h

end if

end for

close infile

end procedure

**Explaination:**

* Open the input file "p1\_input.txt"
* Create an empty vector "groups" to store groups
* Create an empty vector "nums" to store numbers
* Create an empty vector "groups\_h" to store hashes of groups
* Create an empty vector "nums\_h" to store hashes of numbers
* Read each line from the input file
* If the line contains '{', then extract group items:
  + Find the index of the first occurrence of '{' in the line
  + Find the index of the first occurrence of '}' in the line
  + Extract the substring between the above two indices (excluding the curly braces)
  + Create an empty vector "group" to store group items
  + Create an empty vector "group\_h" to store hashes of group items
  + Create a stringstream object "ss" to read items from the substring
  + Read each item from the stringstream:
    - Remove the single quotes from the item
    - If there is a comma after the item, remove it
    - Add the item to the "group" vector
    - Calculate the hash of the item using the "calHash" function and add it to the "group\_h" vector
  + Add the "group" vector to the "groups" vector
  + Add the "group\_h" vector to the "groups\_h" vector
  + If the line contains '[', then extract num items:
  + Find the index of the first occurrence of '[' in the line
  + Find the index of the first occurrence of ']' in the line
  + Extract the substring between the above two indices (excluding the square brackets)
  + Create an empty vector "num" to store num items
  + Create an empty vector "num\_h" to store hashes of num items
  + Create a stringstream object "ss" to read items from the substring
  + Read each item from the stringstream:
    - Remove the single quotes from the item
    - If there is a comma after the item, remove it
    - Add the item to the "num" vector
    - Calculate the hash of the item using the "calHash" function and add it to the "num\_h" vector
  + Add the "num" vector to the "nums" vector
  + Add the "num\_h" vector to the "nums\_h" vector
* Return from the function

**Function to find whether all dishes can be made of a certain group from a certain nums(ingredients):**

**Pseudocode:**

function canDisheshBeMade(r: int):

count <- 0

match <- false

for c in [0, length(groups\_h)-1]:

hash\_groups.size <- 0

hash\_groups.hash <- 0

for k in [0, length(groups\_h[c])-1]:

hash\_groups.hash <- hash\_groups.hash + groups\_h[c][k].hash

hash\_groups.size <- hash\_groups.size + groups\_h[c][k].size

match <- false

for k in [0, length(nums\_h[r])-length(groups\_h[c])]:

hash\_nums.size <- 0

hash\_nums.hash <- 0

for i in [0, length(groups\_h[c])-1]:

hash\_nums.hash <- hash\_nums.hash + nums\_h[r][k+i].hash

hash\_nums.size <- hash\_nums.size + nums\_h[r][k+i].size

if hash\_groups.hash == hash\_nums.hash and hash\_groups.size == hash\_nums.size:

match <- true

for i in [0, length(groups\_h[c])-1]:

nums\_h[r][k+i].hash <- -1

nums\_h[r][k+i].size <- -1

count <- count + 1

break

else:

match <- false

end for

end for

print "Test case " + (r+1) + ": "

if match and count == length(groups):

print "Pass"

else:

print "Fail"

end if

**Time complexity:** O(mnk), where m is the number of test cases, n is the number of groups in each test case, and k is the maximum length of a group or a number.

**Storage complexity:** O(mnk).

**Explanation:**

* Set count to zero and match to false.
* For each c in the range from 0 to the length of groups\_h minus 1:
* Set hash\_groups.size to zero and hash\_groups.hash to zero.
* For each k in the range from 0 to the length of groups\_h[c] minus 1:
* Add the hash and size values of groups\_h[c][k] to hash\_groups.hash and hash\_groups.size, respectively.
* Set match to false.
* For each k in the range from 0 to the difference between the length of nums\_h[r] and the length of groups\_h[c]:
  + Set hash\_nums.size to zero and hash\_nums.hash to zero.
  + For each i in the range from 0 to the length of groups\_h[c] minus 1:
    - Add the hash and size values of nums\_h[r][k+i] to hash\_nums.hash and hash\_nums.size, respectively.
  + iIf hash\_groups.hash is equal to hash\_nums.hash and hash\_groups.size is equal to hash\_nums.size, set match to true, set the hash and size values of each nums\_h[r][k+i] to -1, increment count by 1, and break out of the loop.
  + Otherwise, set match to false.
* Print "Test case " concatenated with r+1.
* If match is true and count is equal to the length of groups, print "Pass", otherwise print "Fail".

**References:**

Use of npos: <https://cplusplus.com/reference/string/string/npos/>

Use of sstream: <https://www.geeksforgeeks.org/stringstream-c-applications/>

**P2 (Graphs):**

**File Reading:**

**Pseudocode:**

**function** **ReadFromFile()** {

// Open the input file

inputFile <- OpenFile("test2.txt")

// Declare variables for storing the data

edges <- empty vector of pairs

weights <- empty vector of integers

vertexIndices <- empty map of character keys to integer values

cities <- empty vector of characters

// Read the vertices from the file

input <- ReadLine(inputFile)

startIndex <- 0

index <- 0

while (startIndex < input.size()) {

if (input[startIndex] != ',') {

cities.push\_back(input[startIndex])

vertexIndices[input[startIndex]] <- index

index <- index + 1

}

startIndex <- startIndex + 1

}

numVertices <- size of cities

// Read the edges from the file

input <- ReadLine(inputFile)

startIndex <- 0

while (startIndex < input.size()) {

if (input[startIndex] == '(') {

startIndex <- startIndex + 1

continue

}

edges.push\_back(make\_pair(input[startIndex], input[startIndex + 2]))

if (startIndex + 5 < input.size())

startIndex <- startIndex + 5

else

break

}

numEdges <- size of edges

// Read the weights from the file

input <- ReadLine(inputFile)

startIndex <- 0

while (startIndex < input.size()) {

if (input[startIndex] != ',') {

weights.push\_back(static\_cast<int>(input[startIndex])-48)

}

startIndex <- startIndex + 1

}

// Initialize the adjacency matrix with all zeros

adjMatrix <- empty vector of vectors of integers with size numVertices x numVertices

for i <- 0 to numVertices-1

for j <- 0 to numVertices-1

adjMatrix[i][j] <- 0

// Update the adjacency matrix based on the edges and weights

for i <- 0 to numEdges-1

startVertex <- edges[i].first

endVertex <- edges[i].second

weight <- weights[i]

startIndex <- vertexIndices[startVertex]

endIndex <- vertexIndices[endVertex]

adjMatrix[startIndex][endIndex] <- weight

adjMatrix[endIndex][startIndex] <- weight // For undirected graph

// Print the adjacency matrix

for i <- 0 to numVertices-1

for j <- 0 to numVertices-1

Print adjMatrix[i][j] followed by a space

Print a newline

Print a line of dashes for formatting

Close the input file

return adjMatrix

}

**function ReadTotalTime():**

// Initialize the vector with 0 at the first index

total\_time = [0]

// Open the input file

inputFile = open("test2.txt")

// Read the vertices from the file

for i in range(4):

input = inputFile.readline()

// Read the times from the file

input = inputFile.readline()

starting\_index = 0

time = 0

while starting\_index < length(input):

if input[starting\_index] == '(':

// Check for double digits

if (48 <= ord(input[starting\_index + 5]) <= 57) and (starting\_index + 5 < length(input)):

time += (ord(input[starting\_index + 4]) - 48) \* 10

time += ord(input[starting\_index + 5]) - 48

else:

time += ord(input[starting\_index + 4]) - 48

starting\_index += 3

total\_time.append(time)

time = 0

if starting\_index < length(input):

starting\_index += 1

else:

break

// Read the total time from the file

input = inputFile.readline()

time = (ord(input[2]) - 48) \* 10 + (ord(input[3]) - 48)

total\_time[0] = time

// Close the input file

inputFile.close()

// Return the vector of times

return total\_time

The 2 functions work on the basic fstream library , where we use delimiters to cut and splice each line we read. It is hardcoded in some way and can cause bugs. The First ReadFromFile Function reads the .txt files and makes the required adjacency list and returns it to the main function.

The Second Function Reads the last 2 lines of the .txt file and returns a vector where the 1st index corresponds to the maximum time limit and the other indexes hold values for time required to travel to a particular destination, i.e in order A, B , C etc times.

**Function to find all possible Hamiltonian Circuits**

**Pseudocode**

function Hamiltonian\_Circuit(Matrix: vector<vector<int>>, current\_path: vector<int>, visited: vector<bool>, current\_selection: int, time: int)

// check if cycle has been found, array must be full and last element should be connected with start

if Matrix.size() == current\_path.size() && Matrix[current\_selection][0] != 0 then

current\_path.push\_back(0) // connect with starting vertex

time += Matrix[current\_selection][0] // add ending edge time

// check if its time is less than best time and update best path array

if time < best\_time then // new better hamiltonian cycle has been found

best\_path = current\_path // copy into new best path

best\_time = time // update time

end if

print\_path(current\_path)

print("--------------------")

time -= Matrix[current\_selection][0] // remove values so that you can test for other edges in recursive

current\_path.pop\_back()

return

end if

for i from 0 to Matrix.size()-1 do

if visited[i] == false then // before adding new vertex check if it's not been visited yet

if Matrix[current\_selection][i] != 0 then // check for edge before adding

current\_path.push\_back(i)

visited[i] = true

time += Matrix[current\_selection][i] // add edge weight

Hamiltonian\_Circuit(Matrix, current\_path, visited, i, time) // recursively call func for other edges

visited[i] = false

current\_path.pop\_back()

time -= Matrix[current\_selection][i]

end if

end if

end for

end function

This function is like an explorer that travels through a graph, represented as a matrix called "Matrix". As it goes through the graph, it keeps track of the path it has taken so far in the vector "current\_path". There's also a vector called "visited" that keeps track of which vertices it has visited.

The function starts at a certain vertex called "current\_selection" and begins exploring all the possible paths it can take. It keeps track of the time it has taken so far in the variable "time".

If it finds a Hamiltonian cycle, it adds the starting vertex to the path, updates the best path and time, and prints the current path. Then it removes the ending vertex and returns.

If it hasn't found a Hamiltonian cycle yet, it explores all the edges from the current vertex and recursively calls the function for the next vertex. At each step, it adds the current vertex to the path, updates the visited vector and time taken, and removes the current vertex and updates the visited vector and time if the path does not lead to a Hamiltonian cycle. It keeps doing this until it either finds a Hamiltonian cycle or has explored every possible path.

Complexity

Time Complexity **O(n!),** where n is the number of vertices in the graph. This is because the function explores all possible paths, and for each vertex, it explores all possible edges, resulting in n! possible paths to explore.

Space complexity of this function is **O(n^2)**, where n is the number of vertices in the graph. This is because the function uses a matrix to represent the graph, which takes up O(n^2) space and also includes a lot of other vectors that adds up O(n)

**Resources Used:**

<https://iq.opengenus.org/hamiltonian-cycle/>

<https://www.youtube.com/watch?v=2UczS2hQLsI>

<https://www.youtube.com/watch?v=v3s0DjRqhKA>

<https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/>

**P3a(Storage Complexity Analysis):**

In the function find\_Sequences(), an array of size n+1 is created to store the previously calculated values. The maximum size of this array is equal to the value of input entered by the user, which determines the number of elements in the array.

In the main() function, only one variable input is declared, which has a constant storage complexity of O(1). Therefore, it does not contribute to the overall storage complexity of the program. Hence, the **overall storage complexity of the code is O(n).**

**Time Complexity => O(n)**

**Pseudocode:**

**Define a function find\_Sequences** that takes an integer **input** as an argument.

Set the constant integer n to input.

Declare a dynamic array of boost::multiprecision::int1024\_t of size n+1, and assign its address to a pointer variable arr.

Set the first and second element of arr to 1 and 2, respectively.

For i = 2 upto input:

Assign arr[i] to the sum of arr[i-1] and arr[i-2] using memoization.

Return arr[input-1], which represents the number of possible ways for the input entered.

**Define the main function:**

Declare an integer variable input.

Output the message "Please enter the value of n:" to the user.

Read in the value of input from the user.

Output a newline character.

Output the message "The number of Possible ways for n=" followed by the value of input, followed by the message " is: ", followed by the value returned by calling **find\_Sequences** with input as its argument.

|  |  |
| --- | --- |
| **“n” emails** | Number of ways |
| 3 | 3 |
| 8 | 34 |
| 75 | 3416454622906707 |

|  |  |
| --- | --- |
| 1225 | 7402219246128381535515569805363490509656655739189773170620289798498555202168614220564123097306186533878335573209493172256624481282050263922108038105180603908340999567895887244267018836917210569368888215390135498107060634049645656705056309965305780860046993 |

Resource used to consult Boost-Library:

* <https://www.geeksforgeeks.org/advanced-c-boost-library/>

**P3b (Storage Complexity Analysis):**

The storage requirements can be broken down into the following categories:

Integer variables used in the program:

int n = 8;

int count = 0;

int start = 0, end = n-1;

int min\_cost - stores the minimum cost of the shortest path

Boolean variables:

bool Is\_Visited[n] - an array of n boolean values

Arrays:

int dist[n] - an array of n integers

int prev[n] - an array of n integers

int path[n] - an array of n integers

Constants:

const int n = 8;

Function calls:

minimum\_cost function is called once

Shortest\_path function is called once

The total storage complexity of the code is the sum of the storage required by all these variables, arrays, constants, and functions.

The arrays dist, prev, Is\_Visited and path are of size n, so their storage complexity is O(n).

The integer variables count, start, end, and min\_cost take up a constant amount of memory, so their storage complexity is O (1).

The constants n and INT\_MAX also take up a constant amount of memory, so their storage complexity is O (1).

The function calls to minimum\_cost and Shortest\_path do not take up any significant amount of memory, so their storage complexity is also O (1).

Therefore, the **overall storage complexity of this program is O(n) + O (1) + O (1) = O(n) -** This is after not taking into account the cost matrix.

**Time Complexity => O(n) for shortest path**

**Time Complexity => O(n^2) for Optimal cost**

**Pseudocode:**

Define constant n as 8

Define function Shortest\_path with parameters end and prev:

a. Output "Shortest path: "

b. Define integer array path of size n and count = 0

c. For i = end down to 0 with i = prev[i], do:

i. Add i to path and increment count

d. For i = count - 1 down to 0, do:

i. If i = 0, output path[i] + 1

ii. Else output path[i] + 1 followed by "->"

e. Output new line

Define function minimum\_cost with parameters start and end:

a. Define boolean array Is\_Visited of size n and set all elements to false

b. Define integer arrays dist and prev of size n and set all elements of dist to infinity and all elements of prev to -1

c. Define two-dimensional integer array cost of size n x n and initialize it

d. Set dist[start] to 0

e. For i = 0 up to n, do:

i. Set current to -1

ii. For j = 0 up to n, do:

1. If Is\_Visited[j] is false and (current is -1 or dist[j] < dist[current]), set current to j

EndFor

iii. If current is -1, break

iv. Set Is\_Visited[current] to true

v. For j = 0 up to n, do:

a. If cost[current][j] is not equal to infinity, do:

1. Set path to dist[current] + cost[current][j]

2. If path is less than dist[j], set dist[j] to path and prev[j] to current

EndFor

f. Call Shortest\_path function with parameters end and prev

g. If dist[end] is equal to infinity, output "No path is found" and return -1

h. Return dist[end]

Define function main:

a. Set start to 0 and end to n-1

b. Set min\_cost to the result of calling minimum\_cost function with parameters start and end

c. If min\_cost is not equal to -1, output "Optimal cost is: " followed by min\_cost

d. Return 0

Resource used to consult Dijkstra-algorithm:

* <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>
* <https://www.geeksforgeeks.org/c-program-for-dijkstras-shortest-path-algorithm-greedy-algo-7/>

**Dry Run:**

**N =4;**

**Cost matrix[n][n]={**

**{0,100,20,300},**

**{INT\_MAX,0,50,10},**

**{ INT\_MAX , INT\_MAX ,0,30},**

**{ INT\_MAX , INT\_MAX , INT\_MAX,0 }**

**}**

Start =0, end =3

Dist[4]={ **INT\_MAX** , **INT\_MAX** , **INT\_MAX** , **INT\_MAX** };

Prev[4]={-1,-1,-1,-1}

Outer for loop in **minimum cost** function:

For i=0:

Current=0

Dist={0,100,20,30}

For i=1:

Current=2

Dist={0,100,20,50}

Prev={-1,0,0,2}

For i=2:

Current=3

Dist={0,100,20,50}

Prev={-1,0,0,2}

For i=3:

Current=1

Dist={0,100,20,50}

Prev={-1,0,0,2}

For optimal cost => we return dist[end] = dist[3]= **50**

**For optimal path: we send the end vertex and prev[] array to the function:**

Path[n] //declare path array with size n to store prev vertices that led to shortest path

Prev={-1,0,0,2}

For i=3(end)

Path[0] = 3

For i=2(prev[i])

Path[1]=2

For i=0(Prev[i])

Path[2]=0

Now display Path[i]+1 in reverse order: **Output is: 1->3->4**