**Analysis of Algorithms Project Report**

**21I-0859 Hadeed Rauf**

**21I-0796 Munib Bajwa**

**21I-0631 Muzammil Hassan**

**Question No 2(PseudoCode And Complexities):**

function pattrenCounter(matrix, pattern, matrixSize, patternSize):

count = 0

for row = 0 to (matrixSize - patternSize):

for col = 0 to (matrixSize - patternSize):

flag = true

for i = 0 to patternSize:

for j = 0 to patternSize:

if matrix[row + i][col + j] != pattern[i][j]:

flag = false

break

if flag is false:

break

if flag is true:

count++

row += patternSize - 1

break

if count is 1:

count = 0

return count

function main():

expRes, pSize = 0

tSize = matrixSize

matrix = readMatrixFromFile

pattrens = readPattrenFromFile

print "Given Matrix"

displayMatrix(matrix, tSize)

print "\nGiven Pattern"

displayMatrix(pattrens, pSize)

print "\nOccurrence Expected:", expRes

occurrencesFound = pattrenCounter(matrix, pattrens, tSize, pSize)

print "\nOccurrences found:", occurrencesFound

if expRes == occurrencesFound:

print "\nTrue"

else:

print "\nFalse"

deleteMatrix(matrix, tSize)

deleteMatrix(pattrens, pSize)

return

**Time Complexity:**

Pattern Counting:

The pattrenCounter function has nested loops to iterate through the matrix and pattern. In the worst case, it has a time complexity of O((N^2)\*(M^2)), where N is the square root of matrix size and M is square root of pattern length.

**Space Complexity:**

The overall space complexity is O(N), where N is the size of the matrix.

**Question No 1 (PseudoCode and Complexities):**

Function simpleMaxProfit(n, prices, path):

if n is less than MIN\_DIMENSION or not a multiple of BLOCK\_SIZE\_MULTIPLE:

return 0

max\_profit = 0

best\_cut = 0

For i from MIN\_DIMENSION to n / 2, incrementing by MIN\_DIMENSION:

Calculate current\_profit as simpleMaxProfit(i) + simpleMaxProfit(n - i)

If current\_profit is greater than max\_profit:

Set max\_profit to current\_profit

Set best\_cut to i

End For

For j from MIN\_DIMENSION to n / 2, incrementing by MIN\_DIMENSION:

Calculate current\_profit as simpleMaxProfit(j) + simpleMaxProfit(n - j)

If current\_profit is greater than max\_profit:

Set max\_profit to current\_profit

Set best\_cut to j

End For

If max\_profit is greater than prices[n / BLOCK\_SIZE\_MULTIPLE]:

Set path[0] to best\_cut

Set path[1] to n - best\_cut

Else:

Set path[0] to n / BLOCK\_SIZE\_MULTIPLE

Return max\_profit

End Function

Function memoizedMaxProfit(n, prices, memo, path):

if n is less than MIN\_DIMENSION or not a multiple of BLOCK\_SIZE\_MULTIPLE:

return 0

If memo[n] is not -1:

Return memo[n]

max\_profit = 0

best\_cut = 0

For i from MIN\_DIMENSION to n / 2, incrementing by MIN\_DIMENSION:

Calculate current\_profit as memoizedMaxProfit(i) + memoizedMaxProfit(n - i)

If current\_profit is greater than max\_profit:

Set max\_profit to current\_profit

Set best\_cut to i

End For

For j from MIN\_DIMENSION to n / 2, incrementing by MIN\_DIMENSION:

Calculate current\_profit as memoizedMaxProfit(j) + memoizedMaxProfit(n - j)

If current\_profit is greater than max\_profit:

Set max\_profit to current\_profit

Set best\_cut to j

End For

If max\_profit is greater than prices[n / BLOCK\_SIZE\_MULTIPLE]:

Set path[0] to best\_cut

Set path[1] to n - best\_cut

Else:

Set path[0] to n / BLOCK\_SIZE\_MULTIPLE

memo[n] = max\_profit

Return max\_profit

End Function

Function bottomUpMaxProfit(n, prices, path):

Create an array memo of size (MAX\_N + 1) and initialize all elements to 0

For i from MIN\_DIMENSION to n, incrementing by MIN\_DIMENSION:

Set max\_profit and best\_cut to 0

For j from MIN\_DIMENSION to i / 2, incrementing by MIN\_DIMENSION:

Calculate current\_profit as memo[j] + memo[i - j]

If current\_profit is greater than max\_profit:

Set max\_profit to current\_profit

Set best\_cut to j

End For

For k from MIN\_DIMENSION to n / 2, incrementing by MIN\_DIMENSION:

Calculate current\_profit as memo[k] + memo[n - k]

If current\_profit is greater than max\_profit:

Set max\_profit to current\_profit

Set best\_cut to k

End For

If max\_profit is greater than prices[i / BLOCK\_SIZE\_MULTIPLE]:

Set path[0] to best\_cut

Set path[1] to i - best\_cut

Else:

Set path[0] to i / BLOCK\_SIZE\_MULTIPLE

Set memo[i] to max\_profit

End For

Return memo[n]

End Function

Function optimizedBottomUpMaxProfit(n, prices, path):

Create an array memo of size (MAX\_N + 1) and initialize all elements to 0

For i from MIN\_DIMENSION to n, incrementing by MIN\_DIMENSION:

Set max\_profit and best\_cut to 0

For j from MIN\_DIMENSION to i / 2, incrementing by MIN\_DIMENSION:

Calculate current\_profit as memo[j] + memo[i - j]

If current\_profit is greater than max\_profit:

Set max\_profit to current\_profit

Set best\_cut to j

End For

For k from MIN\_DIMENSION to n / 2, incrementing by MIN\_DIMENSION:

Calculate current\_profit as memo[k] + memo[n - k]

If current\_profit is greater than max\_profit:

Set max\_profit to current\_profit

Set best\_cut to k

End For

If max\_profit is greater than prices[i / BLOCK\_SIZE\_MULTIPLE]:

Set path[0] to best\_cut

Set path[1] to i - best\_cut

Else:

Set path[0] to i / BLOCK\_SIZE\_MULTIPLE

Set memo[i] to max\_profit

End For

Return memo[n]

End Function

**Time Complexity Analysis:**

**simpleMaxProfit: O(2^n)** - Exponential time complexity due to exploring all possible divisions recursively.

**memoizedMaxProfit:** O((n / MIN\_DIMENSION)^2) - Quadratic time complexity with memoization, avoiding redundant calculations.

**bottomUpMaxProfit:** O((n / MIN\_DIMENSION)^2) - Quadratic time complexity for the bottom-up dynamic programming approach.

**optimizedBottomUpMaxProfit**: O((n / MIN\_DIMENSION)^2) - Quadratic time complexity with further optimization.

These time complexities are based on the size of the burfi block (n) and the minimum dimension (MIN\_DIMENSION). The dynamic programming techniques, especially memoization, significantly improve the efficiency compared to the simple recursive approach.

**Question No 3 (PseudoCode and Complexities):**

**Pseudo code for Q3 PartA:**

// Constants

const MAX\_NODES = 4

// Function to calculate the average travel time for a given path

function calculateAverageTime(path, pathLength, distances)

totalTravelTime = 0

for i = 0 to pathLength - 2

source = path[i] - 'A'

destination = path[i + 1] - 'A'

totalTravelTime += distances[source][destination]

return totalTravelTime

// Main function

function main()

// Open the file

inputFile = openFile("testcase1.txt")

if not inputFile.is\_open

print "Error: Unable to open the file."

return 1

// Define nodes and distances

distances[MAX\_NODES][MAX\_NODES] = {0}

distanceMap = unordered\_map<string, int>()

// Read Paths and Distances

while readLine(inputFile, line) and not line.empty

ss = createStringStream(line)

path = ""

distance = 0

ss >> path >> distance

source = path[0]

destination = path[2]

distances[source - 'A'][destination - 'A'] = distance

distanceMap[path] = distance

// Close the file

closeFile(inputFile)

// Print distances matrix

for i = 0 to MAX\_NODES - 1

for j = 0 to MAX\_NODES - 1

print distances[i][j] + " "

print newline

// Define possible paths

paths = ["AB", "AC", "BC"]

// Calculate and print path costs

print "Path Costs:"

for path in paths

pathLength = length(path)

pathCost = 0

print path + " = "

for i = 0 to pathLength - 2

source = path[i] - 'A'

destination = path[i + 1] - 'A'

pathCost += distances[source][destination]

print distances[source][destination]

if i < pathLength - 2

print " + "

print " = " + pathCost + " minutes"

// Calculate and print total sum of times, total number of paths, and average time

totalSumOfTimes = 0

totalPaths = length(paths)

print "Results:"

for path in paths

pathLength = length(path)

pathCost = calculateAverageTime(path, pathLength, distances)

totalSumOfTimes += pathCost

print "Total sum of times = " + totalSumOfTimes + " minutes"

print "Total number of paths = " + totalPaths

print "Average time to move between locations = " + totalSumOfTimes / totalPaths + " minutes"

return 0

Time complexity Analysis:

Reading Paths and Distances:

The loop reads each line from the file and parses the path and distance. In the worst case, it iterates through all the lines in the file once. Time Complexity: O(N), where N is the number of lines in the file.

Calculating Average Time:

The loop inside the calculateAverageTime function iterates over the nodes in a path. In the worst case, it iterates through all nodes in a path. Time Complexity: O(M), where M is the number of nodes in the longest path.

Printing Paths and Costs:

There are nested loops to iterate over paths and nodes within each path. In the worst case, it iterates through all paths and nodes. Time Complexity: O(P \* M), where P is the number of paths, and M is the number of nodes in the longest path.

**Time Complexity:**

The overall time complexity is determined by the most time-consuming operation, which is printing paths and costs in the provided case. Overall Time Complexity: O(N + P \* M), where N is the number of lines in the file, P is the number of paths, and M is the number of nodes in the longest path.

**Pseudo code for Q3 PartB:**

function findShortestCycle(vertex, parent, start, depth, shortestCycle, currentCycle, shortestCycleVertices, visited, edges):

visited[vertex] = true

currentCycle[depth] = vertex

for neighbor in edges[vertex]:

if neighbor != parent:

if not visited[neighbor]:

findShortestCycle(neighbor, vertex, start, depth + 1, shortestCycle, currentCycle, shortestCycleVertices, visited, edges)

else if neighbor == start and depth > 1 and depth < shortestCycle:

shortestCycle = depth

for j in range(0, depth + 1):

shortestCycleVertices[j] = currentCycle[j]

visited[vertex] = false

function main():

open file "testcase1B.txt" as inputFile

if not inputFile:

print "Error opening file. Exiting program."

return 1

read numVertices from inputFile

if numVertices < 2 or numVertices > MAX\_VERTICES:

print "Invalid number of vertices. Exiting program."

return 1

initialize edges as a 2D vector with numVertices rows

while true:

read u from inputFile

if u == -1:

break

read v from inputFile

if v == -1:

break

if u < 0 or u >= numVertices or v < 0 or v >= numVertices:

print "Invalid vertex label. Vertex label should be between 0 and " + (numVertices - 1) + ". Exiting program."

return 1

add v to edges[u]

add u to edges[v]

close inputFile

initialize shortestCycle as MAX\_VERTICES + 1

initialize shortestCycleVertices as an array with MAX\_VERTICES elements

for i in range(0, numVertices):

initialize visited as an array with MAX\_VERTICES elements, all set to false

initialize currentCycle as an array with MAX\_VERTICES elements

call findShortestCycle(i, -1, i, 0, shortestCycle, currentCycle, shortestCycleVertices, visited, edges)

print "Assigned numbers to vertices:"

for i in range(0, numVertices):

print "Vertex " + i + ": " + i

print "Connections between vertices:"

for i in range(0, numVertices):

print "Vertex " + i + " is connected to: " + edges[i]

if shortestCycle == MAX\_VERTICES + 1:

print "No cycle found in the graph."

else:

print "Length of the shortest cycle: " + shortestCycle

print "Vertices of the shortest cycle: " + shortestCycleVertices

**Time complexity Analysis:**

findShortestCycle Function: The findShortestCycle function performs a Depth-First Search (DFS) traversal on the graph. The time complexity of the DFS traversal is O(V + E), where V is the number of vertices and E is the number of edges. The function is called for each vertex in the main loop, resulting in an overall time complexity of O(V \* (V + E)).

Main Function:

The main function iterates over each vertex and calls findShortestCycle. The loop runs V times, and for each iteration, the findShortestCycle function is called.

Thus, the overall time complexity of the main function is O(V \* (V + E)).