Design & Analysis of Algorithms

Final Project

* Name: Muhammad Abdullah, Muneel Haider, Abdullah Zahoor
* Roll No: 21I-0643 , 21I-0640 , 21I-2481
* Section: D

Contents

[**Problem 1**: 3](#_Toc153560626)

[**Task 1**: 3](#_Toc153560627)

[Pseudocode: 3](#_Toc153560628)

[Time Complexity: 4](#_Toc153560629)

[**Task 2**: 4](#_Toc153560630)

[Pseudocode: 4](#_Toc153560631)

[Time Complexity: 5](#_Toc153560632)

[Memoization: 5](#_Toc153560633)

[**Task 3**: 5](#_Toc153560634)

[Pseudocode: 5](#_Toc153560635)

[Time Complexity: 6](#_Toc153560636)

[**Task 4**: 6](#_Toc153560637)

[Pseudocode: 6](#_Toc153560638)

[Time Complexity: 7](#_Toc153560639)

[**Problem 2**: 7](#_Toc153560640)

[Pseudocode: **7**](#_Toc153560641)

[Time Complexity: 9](#_Toc153560642)

[**Problem 3**: 9](#_Toc153560643)

[**Part a)** 9](#_Toc153560644)

[Pseudocode: 9](#_Toc153560645)

[Time Complexity: 11](#_Toc153560646)

[**Part b)** 11](#_Toc153560647)

[Pseudocode: **11**](#_Toc153560648)

[Time Complexity: 13](#_Toc153560649)

# **Problem 1**:

## **Task 1**:

### **Pseudocode:**

***Start***

**Function** max(a, b):

If a > b, return a

Else, return b

**Function** parseLine(line, priceEntries, numEntries):

Parse line to extract dimensions and prices

Store them in priceEntries

Increment numEntries

**Function** calculateMaxProfit(length, width, priceEntries, numEntries):

If length or width is zero, return 0

Initialize maxProfit to 0

For each price entry (k):

Set pieceLength, pieceWidth, and piecePrice from priceEntries[k]

If piece fits perfectly, calculate profit

Recursively calculate profit for remaining length

Recursively calculate profit for remaining width

Update maxProfit to the maximum of these values

Return maxProfit

**Function** main:

Open file and read price entries

Calculate and print maximum profit using calculateMaxProfit

Time Complexity: O (2^n x numEntries)

***End***

## **Task 2**:

### **Pseudocode:**

***Start***

**Function** max(a, b):

Return the maximum of a and b

**Function** parseLine(line, priceEntries, numEntries):

Parse the line to get dimensions and prices of barfi

Store in priceEntries and update numEntries

**Function** calculateMaxProfit(length, width, priceEntries, numEntries):

If length or width is zero

return 0

If memo[length][width] is already computed,

return its value

Initialize maxProfit to 0

check each price entry:

If the piece can be cut perfectly

calculate profit

Recursively calculate profit for remaining length and width

Update maxProfit with the maximum profit obtained

Store maxProfit in memo[length][width]

Return maxProfit

**Function** main:

Initialize memo array to -1 ( for null values)

Read dimensions and prices from file

Calculate and print maximum profit using calculateMaxProfit

***End***

**Time Complexity:** O (numEntries x length x width)

**Memoization**: The memoization in this code and the Task 3 code converts the time complexity of the overall code of Task 1 from exponential to polynomial which is more effective. Therefore, memoization is better as it avoids repeated calculations.

**Task 3**:

**Pseudocode**:

***Start***

**Function** max(a, b):

Return the maximum of a and b

**Function** parseLine(line, priceEntries, numEntries):

Parse a line from the file to extract dimensions and prices

Store them in priceEntries and update numEntries

**Function** maxProfit(length, width, priceEntries, numEntries):

Initialize arrayProfit for storing maximum profits

For each possible piece size (i, j):

For each entry in priceEntries:

Update arrayProfit[i][j] based on piece cuts and prices

Consider additional horizontal and vertical splits

Return the maximum profit for the full piece

**Function** main:

Read dimensions and prices from a file

Compute and print the maximum profit using maxProfit

***End***

**Time Complexity**:O(n^3)

## **Task 4**:

### **Pseudocode**:

***Start***

**Function** max(a, b):

Return the maximum of a and b

**Function** parseLine(line, priceEntries, numEntries):

Parse line to extract dimensions and prices

Store dimensions and prices in priceEntries

Update numEntries

**Function** maxProfit(length, width, priceEntries, numEntries):

Initialize arrayProfit for storing maximum profits

Iterate over each possible piece size (i, j):

Iterate over each price entry (k):

Update arrayProfit[i][j] based on piece cuts and prices

Check for possible piece splits

Return arrayProfit for the full piece size

**Function** main:

Read dimensions and prices from file

Calculate maximum profit using maxProfit

Print maximum profit

***End***

**Time Complexity:** O(n^2) if number of entries value is small. Otherwise, O(n^3)

# **Problem 2**:

## **Pseudocode**:

***Start***

**function** isPatternPresent(text[], pattern, minOccurrences):

n = text.size()

occurrences = 0

for i in 0 to n - pattern.size():

for j in 0 to n - pattern.size():

isMatch = true

for k in 0 to pattern.size():

if text[i + k].substr(j + k, pattern.size()) != pattern:

isMatch = false

break

if isMatch:

occurrences += 1

return occurrences >= minOccurrences

**function** readTestCase(filename, text[], pattern, minOccurrences):

file = open file with filename

if file is not open:

print "Error opening file:", filename

return false

dimensions = read dimensions from file

resize text[] to dimensions

for i in 0 to dimensions - 1:

read text[i] from file

if reading fails:

print "Error reading text from the file."

return false

read pattern and minOccurrences from file

if reading fails:

print "Error reading pattern and/or minimum occurrences from the file."

return false

print "Dimensions:", dimensions, "x", dimensions

print "Text:"

print each line in text

print "Pattern:", pattern

print "Minimum Occurrences:", minOccurrences

return true

**function** main():

filename = "TestCase1.txt"

text[], pattern, minOccurrences = ""

if readTestCase(filename, text, pattern, minOccurrences):

print "Pattern is", (isPatternPresent(text, pattern, minOccurrences) ? "" : "not "), "present diagonally at least", minOccurrences, "times."

***End***

**Time Complexity**: O (n^3)

## **Problem 3**:

## **Part a)**

### **Pseudocode**:

***Start***

**function** nodeToIndex(node):

return node - 'A'

**function** calculateAverageTime(filename):

file = filename

if file is not open:

print "Error opening file: filename"

return -1

distances[][] = 2D array of size 26x26 initialized with zeros (for ‘A’ to ‘Z’ nodes)

sumOfTimes, numberOfPaths = 0, 0

while reading lines from file:

line = read line from file

if line is empty or starts with a digit:

break

get nodes and distances from line

update distances array

print distances

while reading lines from file:

line = read line of file

if line is empty or starts with a digit:

continue

get nodes from line

calculate pathCost using distances array

update sumOfTimes and numberOfPaths

print pathCost

close file

return (sumOfTimes / numberOfPaths) if numberOfPaths > 0 else 0

**function** main():

filename = testcasefile

averageTime = calculateAverageTime(filename)

print "Average Time to Move Between Locations:", averageTime, "minutes" if averageTime >= 0 else "An error occurred"

***End***

**Time Complexity**: O (n + m) where n is number of lines and m is number of paths

## **Part b)**

### **Pseudocode:**

***Start***

**public class** Queue:

initialize variables

function isEmpty():

return front > rear

function enqueue(item):

if rear < maximumVertices \* 2 - 1:

items[++rear] = item

function dequeue():

return -1 if isEmpty() else items[front++]

**function** shortestCycleBFS(start, n, graph[][]):

q = Queue(), visited[], level[], parent[]

fill\_n(visited, level, parent, maximumVertices, false, 0, -1)

q.enqueue(start), visited[start] = true

while not q.isEmpty():

vertex = q.dequeue()

for i in 0 to n - 1:

if graph[vertex][i] and (not visited[i] or parent[vertex] ≠ i):

visited[i], level[i], parent[i] = true, level[vertex] + 1, vertex

q.enqueue(i)

else:

return level[vertex] + level[i] + 1

return -1

**function** shortestCycleLength(n, graph[][]):

minCycle = IntMAX

for i in 0 to n - 1:

cycle = shortestCycleBFS(i, n, graph)

if cycle ≠ -1 and cycle < minCycle:

minCycle = cycle

return -1 if minCycle == IntMAX else minCycle

**function** main():

file = open file with filename

if not file.is\_open():

print "Error opening file:", filename

return -1

n, u, v = 0, 0, 0

file >> n, graph[][], fill\_n(graph, maximumVertices, false)

while file >> u >> v:

graph[u][v] = graph[v][u] = true

file.close(), print "Shortest Cycle Length:", shortestCycleLength(n, graph)

***End***

**Time Complexity**: O(V^3) where V= number of vertices