SEMESTER PROJECT

DESIGN AND ANALYSIS OF ALGORITHMS

SUBMITTED TO:

MA'AM NOOR-UL-AIN

SUBMITTED BY:

WAIZ HASSAN ZUBAIRI I21-0715

MOMINA KHALID I21-0517

FIZZA MUMTAZ 21I-0437

(SECTION : A)

DATE:

15 DEC, 2023

Q:NO:01

Task1

PseudoCode:

```
Procedure rdTstCse(filename, n, prices):
   file = open(filename)
   if not file.is_open():
      print("Error opening file: ", filename)
      exit(EXIT_FAILURE)

   file >> n
   line = read_line(file)  # Consume the newline after reading 'n'

   while (line = read_line(file)):
      if sscanf(line, "(%d,%d),{%d}", w1, h1, pr):
         prices.append({Blk(w1, h1), pr})
      else if sscanf(line, "(%d,%d),(%d,%d),{%d}", w1, h1, w2, h2, pr):
         prices.append({Blk(w1, h1), pr})
         prices.append({Blk(w2, h2), pr})
      else:
         print("Error reading line: ", line)
         exit(EXIT_FAILURE)

Procedure simRcvWthPth(prices, n):
   if n <= 0:
      return {0, {}}
```

```
    mxPft = INT_MIN
    selectedBlocks = {}

    for i = 0 to size(prices) - 1:
        size = prices[i].width * prices[i].height
        if n >= size:
            sbprb = simRcvWthPth(prices, n - size)
            currentProfit = prices[i].price + sbprb[0]

            if currentProfit > mxPft:
                mxPft = currentProfit
                selectedBlocks = sbprb[1]
                selectedBlocks.append({prices[i].width, prices[i].height, prices[i].price})

    return {mxPft, selectedBlocks}

Procedure main():
    filename = "TestCase2.txt"
    n = 0
    prices = []

    rdTstCse(filename, n, prices)
    ans = simRcvWthPth(prices, n)

    print("Maximized Profit: ", ans[0])
    print("Selected Blocks: ")
    for j = 0 to size(ans[1]) - 1:
```

```
print("(", ans[1][j].width, ",", ans[1][j].height, ") ")
```

Time Complexity Analysis And Efficiency Comparison

The time complexity of the provided solution is exponential, O(2^n), where n is the input size. This is because the algorithm explores all possible combinations of blocks, leading to an exponential growth in the number of recursive calls.The provided solution, although correct, may not be efficient for large inputs due to its exponential time complexity. For larger instances of the problem, it's recommended to explore dynamic programming approaches, such as memoization or bottom-up tabulation, to optimize the solution and reduce the time complexity to polynomial or linear levels.

Task2:

Pseudocode:

```
Procedure genericFileReading(filename, num, boxes):
  file = open(filename)
  if not file.is_open():
    print("Error opening file: ", filename)
    exit(EXIT_FAILURE)


  file >> num
  line = read_line(file)


  while (line = read_line(file)):
    if sscanf(line, "(%d,%d),{%d}", len1, brdth1, price):
      boxes.append({Box(len1, brdth1), price})
    else if sscanf(line, "(%d,%d),(%d,%d),{%d}", len1, brdth1, len2, brdth2, price):
```

```
        boxes.append({Box(len1, brdth1), price})
        boxes.append({Box(len2, brdth2), price})
    else:
        print("Error reading line")
        exit(EXIT_FAILURE)

Procedure memorecursion(boxes, num, memo):
  if memo[num] != -1:
    return memo[num]

  if num <= 0:
    return 0

  maxProfit = INT_MIN
  it = boxes.begin()

  while it != boxes.end():
    size = it->first.length * it->first.breadth

    if num >= size:
      maxProfit = max(maxProfit, it->second + memorecursion(boxes, num - size, memo))

    it = it + 1

  memo[num] = maxProfit
  return maxProfit
```

```
Procedure main():
    filename = "TestCase1.txt"
    num = 0
    boxes = []

    genericFileReading(filename, num, boxes)
    memo2 = create_vector(num + 1, -1)
    ans = memorecursion(boxes, num, memo2)

    print("Maximum Profit: ", ans)
```

Time Complexity Analysis And Efficiency Comparison:

The time complexity of the provided solution is O(n * m), where n is the target size and m is the number of boxes. The function memorecursion explores all possible combinations of boxes for each target size up to n, and for each combination, it iterates through the boxes (m). Therefore, the overall time complexity is O(n * m). The dynamic programming approach with memoization in the current implementation significantly improves efficiency compared to the original simple recursive solution. By storing and reusing intermediate results, the time complexity is reduced from exponential to O(n * m), making it more practical for larger inputs. This optimization minimizes redundant computations and enhances the algorithm's performance for a greater number of boxes and target sizes.

Task3:

Pseduocode:

```
function readFile(fileName, n, boxes):
    open file with fileName
    exit if file is not open
    read n from file
    initialize line variable
```

```
    initialize l1, b1, l2, b2, p variables
    read and discard the first line
    while there are lines in the file:
        if line matches the pattern "(%d,%d),{%d}":
            add Box(l1, b1) with price p to boxes
        else if line matches the pattern "(%d,%d),(%d,%d),{%d}":
            add Box(l1, b1) and Box(l2, b2) with price p to boxes
        else:
            print error and exit
    close the file

function bottomUpDynamicProgramming(boxes, n, path):
    initialize memo array with size n + 1, filled with -1
    initialize lastBoxIndex array with size n + 1, filled with -1
    set memo[0] to 0
    for i from 1 to n:
        set maxProfit to INT_MIN
        set lastBox to -1
        for each box in boxes:
            calculate size as box.length * box.breadth
            if i >= size:
                calculate currentProfit as box.price + memo[i - size]
                if currentProfit > maxProfit:
                    set maxProfit to currentProfit
                    set lastBox to index of the current box
        set memo[i] to maxProfit
        set lastBoxIndex[i] to lastBox
```

```
    for currentIndex from n to 0 while lastBoxIndex[currentIndex] != -1:
        add lastBoxIndex[currentIndex] to path
        decrease currentIndex by size of the box at lastBoxIndex[currentIndex]


function main():
    initialize n variable
    set fileName to "TestCase3.txt"
    initialize boxes vector
    call readFile(fileName, n, boxes)
    initialize path vector
    set result to result of calling bottomUpDynamicProgramming(boxes, n, path)
    print "Maximized Profit:", result
    print "Path used to maximize profit:"
    for i from size of path - 1 to 0:
        print "(" + boxes[path[i]].first.len + "," + boxes[path[i]].first.brd + ") "
```

Time Complexity Analysis:

The time complexity of the bottomUpDynamicProgramming function is O(n * m), where n is the target size and m is the number of boxes. The function iterates over each target size from 1 to n and, for each size, iterates over all the boxes (m) to find the maximum profit. The overall time complexity is dominated by these nested iterations.

Efficiency Comparison:

The bottom-up dynamic programming approach with memoization in this implementation significantly improves efficiency compared to the original simple recursive solution. By avoiding redundant computations and storing intermediate results, the time complexity is reduced to O(n * m), making it more practical for larger inputs. This optimization enhances the algorithm's performance, especially for scenarios with a greater number of boxes and larger target sizes.

Task4:

Pseudocode:

```
function readFile(fileName, n, boxes):
    open file with fileName
    exit if file is not open
    read n from file
    initialize line variable
    initialize l1, b1, l2, b2, p variables
    read and discard the first line
    while there are lines in the file:
        if line matches the pattern "(%d,%d),{%d}":
            add Box(l1, b1) with price p to boxes
        else if line matches the pattern "(%d,%d),(%d,%d),{%d}":
            add Box(l1, b1) and Box(l2, b2) with price p to boxes
        else:
            print error and exit
    close the file

function bottomUpDynamicProgramming(boxes, n, path):
    initialize memo array with size n + 1, filled with -1
    initialize lastBoxIndex array with size n + 1, filled with -1
    set memo[0] to 0
    for i from 1 to n:
        set maxProfit to INT_MIN
        set lastBox to -1
        for each box in boxes:
            calculate size as box.length * box.breadth
```

```
            if i >= size:
                calculate currentProfit as box.price + memo[i - size]
                if currentProfit > maxProfit:
                    set maxProfit to currentProfit
                    set lastBox to index of the current box
        set memo[i] to maxProfit
        set lastBoxIndex[i] to lastBox
    for currentIndex from n to 0 while lastBoxIndex[currentIndex] != -1:
        add lastBoxIndex[currentIndex] to path
        decrease currentIndex by size of the box at lastBoxIndex[currentIndex]


function main():
    initialize n variable
    set fileName to "TestCase2.txt"
    initialize boxes vector
    call readFile(fileName, n, boxes)
    initialize path vector
    set result to result of calling bottomUpDynamicProgramming(boxes, n, path)
    print "Maximized Profit:", result
    print "Path used to maximize profit:"
    for i from size of path - 1 to 0:
        print "(" + boxes[path[i]].first.len + "," + boxes[path[i]].first.brd + ") "
```

Time Complexity Analysis:

The time complexity of the bottomUpDynamicProgramming function is O(n * m), where n is the target size and m is the number of boxes. The function iterates over each target size from 1 to n and, for each size, iterates over all the boxes (m) to find the maximum profit. The overall time complexity is dominated by these nested iterations.

Efficiency and Storage Optimization:

This bottom-up dynamic programming approach with memoization optimizes storage by using two arrays (memo and lastBoxIndex) instead of a two-dimensional array. The memo array stores the maximum profit for each target size, and the lastBoxIndex array keeps track of the index of the last selected box for each target size. This avoids the need for a two-dimensional array to store intermediate results, leading to better memory efficiency, especially for larger inputs.

Q:NO:02

Pseudo code:

```
Function diagonalOccurence(fileName):


    Open the file with fileName

    If file is not open, print an error and return 1


    Read the first line and ignore it


    Initialize variables:

       N = 0

       Data = “”  // to store whole data from file

       columnCounter = 0  // for calculating the size of array’s row

       allow = false

       allow2 = true



    Read characters from the file until the end:

       Accumulate characters in data
```

```
    If allow2 is true and current character is not '\n':
        Increment columnCounter



Create a 2D array array2 of size [columnCounter][columnCounter]

Create a 1D array array3 of size columnCounter

// … populate array2 and array3 …

Create a 2D array array4 of size [stringCounter][stringCounter]

// … populate array4 …

Initialize variables for indexes and counters:
    X1 = 0, y1 = 0  // Indexes of Main Array
    X2 = 0, y2 = 0  // Indexes of Pattern Array
    Helper1 = 0  // for helping traversing pattern
    Helper2 = 0  // for helping traversing array
    misMatch = 0  // for count of mismatching ones
    countOfMatch = 0  // count element matched
    noOfElements = columnCounter * columnCounter  // No of Elements in Main Array
    noOfElements2 = index  // No of Elements in Pattern Array

while noOfElements is not 0:
    if x1 is greater than or equal to columnCounter, set x1 to columnCounter – 1
    if y1 is greater than or equal to columnCounter, set y1 to columnCounter – 1
```

```
        if array2[x1][y1] is not equal to array4[x2][y2]:
          if y1 is equal to columnCounter, break
          increment y1
          increment misMatch
        else:
          increment countOfMatch, y1, and y2



        Decrement noOfElements


      Return countOfMatch / index


Function main():
  Occurances = diagonalOccurence("TestCase3.txt")
  If occurances is greater than 1:
    Print "Total Diagonal Occurrences are:", occurances
  Else:
    Print "Total Diagonal Occurrences are: 0"
```

<u>Time Complexity Analysis:</u>

File Reading and Array Population:

The process begins by reading the file character by character, involving a single iteration over each character. The arrays, namely array2, array3, and array4, are populated concurrently by iterating over the characters in the file. These operations are linear in complexity and contribute O(N) time, where N represents the total count of characters in the file.

Occurrence Detection in Arrays:

The core process involves a main loop that traverses through the elements of the arrays, performing comparisons. In the worst-case scenario, this loop iterates over all elements present in the arrays, contributing to a time complexity of O(N), where N signifies the total count of elements across the arrays.

Overall Time Complexity:

The dominant factor is the linear iteration over the characters in the file and the elements in the arrays. If we follow any other solution, rather than linear iteration, the complexity increases. Thus this algorithm serves as the best time and space effient amongst all .

Thus, the overall time complexity is O(N), where N is the total number of characters in the file or the total number of elements in the arrays.

Q:NO:03(A)

PSEUDO-CODE:

```
// Node class to represent individual locations
Class Node:
    Properties: location

    // Constructor to set the location of the node
    Method Node(char data):
        Set location to data


// Edge class representing connections between nodes
Class Edge:
    Properties: destination, distance
```

```
    // Constructor to set the destination node and distance of an edge
    Method Edge(Node* dst, int dist):
        Set destination to dst
        Set distance to dist

// Graph class to manage the graph structure
Class Graph:
    Property: adjacencyList (List of Lists of Edges)

    // Constructor to initialize the graph with an empty adjacency list
    Method Graph():
        Initialize adjacencyList

    // Method to add a new node to the graph
    Method addNode(Node* node):
        newList = new List of Edges
        Add a self-loop Edge (node, distance 0) to newList
        Add newList to adjacencyList

    // Method to add an edge between nodes in the graph
    Method addEdge(int src, int dst, int distance):
        sourceList = find list in adjacencyList at index src
        destList = find list in adjacencyList at index dst
        Add Edge (destination from destList, distance) to sourceList

    // Method to check if an edge exists between nodes
    Method checkEdge(int src, int dst):
```

```
    currentList = find list in adjacencyList at index src
    dstList = find list in adjacencyList at index dst
    For each edge in currentList:
        If edge's destination equals destination from dstList:
            Return true
    Return false

// Method to print the adjacency list representation of the graph
Method print():
    For each edgeList in adjacencyList:
        Print node and its edges

// Method to print the distances among nodes in the graph
Method printDistances():
    Create distances matrix with zeros
    For each edgeList in adjacencyList:
        For each edge in edgeList:
            destinationIndex = edge's destination location - 'A'
            distances[sourceIndex][destinationIndex] = edge's distance

    Print distances matrix

// Method to get the distance between two nodes in the graph
Method getDistanceBetweenNodes(char srcNode, char dstNode):
    sourceIndex = srcNode - 'A'
    destinationIndex = dstNode - 'A'
    For each edgeList in adjacencyList:
```

```
        If edgeList is not empty and edgeList's first destination equals srcNode:
            For each edge in edgeList:
                If edge's destination equals dstNode:
                    Return edge's distance
    Return 0

// Main function to create the graph and calculate average time
Main function:
    Create a new instance of Graph, named g
    Create a new Node instance with data 'A', named nodeA
    Create a new Node instance with data 'B', named nodeB
    Create a new Node instance with data 'C', named nodeC
    Create a new Node instance with data 'D', named nodeD

    Add nodeA to the graph g using the addNode method
    Add nodeB to the graph g using the addNode method
    Add nodeC to the graph g using the addNode method
    Add nodeD to the graph g using the addNode method

    // File handling
    Open file 'testcase3.txt'
    If file is open:
        Read lines from file
        For each line in file:
            If line is not empty:
                Extract source, destination, and distance from line
                Add edge between source and destination with the given distance in the graph g
```

```
    Close file

Print adjacency list representation of g

Initialize totalTimeTaken to 0
Initialize countPathLines to -1
Set foundEmptyLine to false

Open file 'testcase3.txt'
If file is open:
    Read lines from file
    For each pathLine in file:
        If pathLine is empty and foundEmptyLine is false:
            Set foundEmptyLine to true
        Else if foundEmptyLine is true:
            Increment countPathLines
            For each character in pathLine:
                If character is not '-' and next character is '-':
                    Add distance between nodes to totalTimeTaken using getDistanceBetweenNodes
function
            If pathLine is empty:
                Break the loop

    Close file
Calculate AverageTime as totalTimeTaken divided by countPathLines
Print calculated AverageTime
```

End of Main function

Time Complexity and Comparison:

- The provided code implements a graph data structure using an adjacency list representation. This data structure offers an efficient way to manage connections between nodes. The time complexity of this implementation is **O(V + E)**, where V denotes the number of vertices (nodes) and E represents the number of edges. In very rare cases, the complexity will be worse of O(V*2).

- **The primary factor contributing to this time complexity is the traversal of edges via the adjacency list**. This traversal is vital for operations like adding edges between nodes, checking edge existence, calculating distances, and printing node connections.

- The code's choice of an adjacency list is particularly beneficial for this problem. It is **memory-efficient**, especially for graphs with relatively few connections between nodes, as it only stores information about connected nodes. In contrast, an adjacency matrix would occupy O(V^2) space, which might be impractical for larger graphs.

- Although a queue-based approach could be suitable for certain graph algorithms like Breadth-First Search (BFS) or Depth-First Search (DFS), it may not be as efficient when directly accessing specific node distances, as required in this scenario.

- Hence, the code's usage of an adjacency list provides an efficient and scalable means to represent and navigate the graph. This method is especially effective for calculating the average time taken to travel between locations within a vast city or any similar scenario.

Q:NO:3(B)

Pseudo code:

Function shortestPath(fileName):

Open the file with fileName

If file is not open, print an error and return 1

Read the first line and store it in size

Create a 2D array array1 of size [stoi(size)][2]

Read the entire file content into fileContent using stringstream

Initialize variables:

X1 = 0, y1 = 0 // for controlling array1

I = 0

Counter = 0 // for the number of elements

Iterate through fileContent:

If current character is not ‘ ‘ and not ‘\n’:

Increment counter

Store the character in array1[x1][y1]

Increment y1

Else if current character is ‘\n’:

Increment x1

Set y1 to 0

If current character is ‘\n’ and next character is ‘\n’:

Increment i

Break

Increment i

Create an array array2 of size x1

```
Initialize variables:
   X3 = 0  // to iterate array2
   Index = x1
   Val1 = '\0'  // picking values from array
   X2 = 1  // index to iterate array
   J = 0  // loop iteration needed in global
   Flag = true

While x2 is less than index:
   Val1 = array1[x2][1]

   Iterate through array1 up to x2:
      If val1 is equal to array1[j][0]:
         Store x2 – j in array2[x3]
         Increment x3
         Store j in array2[x3]
         Increment x3
         Set flag to false
         Break

   Increment x2

If not flag:
   Initialize variables:
      minElement = array2[0]
      index2 = 0
```

```
        iterate through array2 in steps of 2:
            if array2[a] is less than minElement:
                set minElement to array2[a]
                set index2 to a

        print array2[index2] + 1

        iterate from array2[index2 + 1] to array2[index2] + array2[index2 + 1]:
            print array1[x][0] + " -> "

        print array1[array2[index2 + 1]][0]

    else:
        print -1

function main():
    call shortestPath("testcase1.txt")
```

Time Complexity Analysis:

Reading Files:

Reading the file involves iterating over each character once.

Populating the 2D array array1 involves iterating over the characters of the file once.

These operations contribute O(N) time complexity, where N is the total number of characters in the file.

Building Connections (array2):

The loop that builds connections in array2 has a nested loop iterating over a decreasing range.

The worst-case scenario is when the loop iterates over all elements in array2.

This contributes O(N^2) time complexity, where N is the size of array2.

Finding Minimum Element in array2:

After building connections, finding the minimum element in array2 involves iterating over the array once.

This contributes O(N) time complexity, where N is the size of array2.

Printing Result:

The print statements involve iterating over the elements in array2 and array1.

These contribute **O(N)** time complexity.

Overall Time Complexity:

The dominant factor is the loop building connections in array2 (O(N^2)).

Thus, the overall time complexity is O(N^2).