Analysis of Algorithms Project

Submitted By:

Zainab Ali Khan – I210492

Abdullah Chaudhary - I210844

Aqib Ansari – I210420

Submitted To:

Owais Idrees

Noor Ul Ain

Table of Contents

Question 1

Problem

Ali, a professional Halwai, wants to maximize profit by dividing a burfi block of size 'n' into saleable portions according to certain regulations. The burfi block has dimensions of 1000 x 1000 sq. cm and must be divided into saleable portions of 100 sq. cm or multiples of 100 sq. cm, with each dimension being a multiple of 5. Different dimensions that sum up to the same size yield the same associated price. The goal is to determine the optimal division for maximizing profit without further subdividing the burfi.

Tasks:

Simple Recursive Top-Down Algorithm:

Develop a recursive algorithm without memoization to explore different splits for maximizing profit.

Recursive Top-Down Algorithm with Memoization:

Enhance the recursive algorithm by incorporating memoization to avoid redundant calculations and improve performance.

Iterative Bottom-Up Algorithm with Memoization:

Design an iterative bottom-up algorithm using memoization to find the optimal solution.

Optimize Storage in Iterative Algorithm:

Optimize storage in the iterative algorithm if possible, ensuring efficient memory usage.

C++ Files and Time Complexity Analysis:

Provide separate C++ files for each of the four algorithms. Analyze the time complexity of the Iterative Bottom-Up algorithm.

Generic and Adaptable Solution:

Ensure that the algorithms can handle any 'n' value and price 'P' (a vector of prices against each burfi size in multiples of 100 sq. cm).

Test the solution across various test cases to demonstrate its generic and adaptable nature.

Pseudocode

```
1. File Handling:
2. function readDataFromFile(filename, dimensions,
3. prices, n)
4. file = openFile(filename)
5.
6. if file is not open
7. print "Error opening file: ", filename
8. return
9. end if
10.
```

```
pattern = createRegexPattern()

line = readLine(file)
n = parseInt(line)

while line = readLine(file)
match = matchPattern(line, pattern)

if match is successful
width1 = parseInt(match[1])
height1 = parseInt(match[2])
width2 = parseInt(match[3])
height2 = parseInt(match[4])
value = parseInt(match[5])

addDimension(dimensions, width1, height1,
width2, height2)
addPrice(prices, value)
else
print "Invalid line format: ", line
end if
end while

closeFile(file)
end function
```

First Approach

```
function calculateMaximumProfit(dimensions, prices, n,
optimalDimensions)
if n is 0 or n % 100 is not 0
return 0

maxProfit = 0
selectedDimension = -1
i = 0

while i < size of dimensions
size = dimensions[i].x1 * dimensions[i].y1
if size is less than or equal to n
currentDimensions = empty vector of Dimension
currentProfit = prices[i] +
calculateMaximumProfit(dimensions, prices, n - size,
currentDimensions)

if currentProfit is greater than maxProfit
```

```
19.
20. maxProfit = currentProfit
21. selectedDimension = i
22. optimalDimensions = move currentDimensions
23.
24. i = i + 1
25.
26. if selectedDimension is not -1
27. append dimensions[selectedDimension] to
28. optimalDimensions
29.
30. return maxProfit
```

Second Approach

```
1.  function findOptimalDivisions(dimensions, prices, n,
2.  optimalDimensions, memo)
3.  if n is 0 or n % 100 is not 0
4.  return 0
5.
6.  if memo[n / 100][n % 100] is not -1
7.  return memo[n / 100][n % 100]
8.
9.  maxProfit = 0
10. selectedDimension = -1
11. i = 0
12.
13. while i < size of dimensions
14. size = dimensions[i].x1 * dimensions[i].y1
15. if size is less than or equal to n
16. currentDimensions = empty vector of Dimension
17. currentProfit = prices[i] +
18. findOptimalDivisions(dimensions, prices, n - size,
19. currentDimensions, memo)
20.
21. if currentProfit is greater than maxProfit
22. maxProfit = currentProfit
23. selectedDimension = i
24. optimalDimensions = move currentDimensions
25.
26. i = i + 1
27.
28. if selectedDimension is not -1
29. append dimensions[selectedDimension] to
```

30. optimalDimensions
31.
32. memo[n / 100][n % 100] = maxProfit
33. return maxProfit

Third Approach

1. function findOptimalSelections(sizes, values, numSizes)
2. maxValues = initializeArray(numSizes + 1, 0)
3. optimalSelections = initializeArrayOfArrays(numSizes +
4. 1)
5.
6. i = 1
7.
8. while i <= numSizes
9. maxValue = 0
10. optimalIndex = -1
11. j = 0
12.
13. while j < size(sizes)
14. area = sizes[j].width1 * sizes[j].height1
15.
16. if area <= i
17. currentMaxValue = values[j] + maxValues[i -
18. area]
19.
20. if currentMaxValue > maxValue
21. maxValue = currentMaxValue
22. optimalIndex = j
23. end if
24.
25. end if
26.
27. j = j + 1
28. end while
29.
30. maxValues[i] = maxValue
31.
32. if optimalIndex != -1
33. copyArray(optimalSelections[i -
34. sizes[optimalIndex].width1 *
35. sizes[optimalIndex].height1], optimalSelections[i])
36. appendElement(optimalIndex,
37. optimalSelections[i])
38. end if
39.

```
40. i = i + 1
41. end while
42.
43. print "Maximum Value: ", maxValues[numSizes]
44.
45. print "Optimal Selections: "
46. call printOptimalSizes(sizes,
47. optimalSelections[numSizes])
48. print " = ", maxValues[numSizes], "k"
49. end function print "Optimal Divisions:", dimensions in
50. divisions[n], "=", dp[n], "k"
```

Fourth Approach

```
function findOptimalDivisions(dimensions, prices, n)

dp = initializeArray(n + 1, 0)

optimalIndices = initializeArray(n + 1, -1)


i = 1

while i <= n

maxProfit = 0


optimalIndex = -1

j = 0


while j < size(dimensions)

size = dimensions[j].width1 *

dimensions[j].height1


if size <= i

currentProfit = prices[j] + dp[i - size]


if currentProfit > maxProfit

maxProfit = currentProfit

optimalIndex = j
```

```
end if
end if

j = j + 1
end while

dp[i] = maxProfit
optimalIndices[i] = optimalIndex
i = i + 1
end while

print "Maximum Profit: ", dp[n]
print "Optimal Divisions: "

currentSize = n
while currentSize > 0
index = optimalIndices[currentSize]
dim = dimensions[index]
print dim.width1, "x", dim.height1
currentSize = currentSize - dim.width1 * dim.height1
if currentSize > 0
print " + "
end if
end while

print " = ", dp[n]
end function
```

Comparison with other Solutions

- The first approach is likely to have the highest time complexity.

- The second approach improves upon the first by using memoization.
- The third and fourth approaches are more efficient, utilizing dynamic programming with tabulation. The third approach stores optimal divisions separately in a 2D array, while the fourth approach uses a single 1D array for both profit and optimal indices.
- The overall time complexity would be computed by adding the file reading time complexity with complexity of each approach.

The third and fourth approaches are preferable for their efficiency and ease of implementation. Choose between them based on your preference for storing and printing optimal divisions.

Question 2

Problem

You are required to provide a C++ implementation of an algorithm that adheres to the above-mentioned conditions. Additionally, analyze the time complexity of your algorithm. The efficiency of the algorithm will be compared with the best-known solution. Ensure your logic uses no more than a constant amount of extra space. It is advisable to generate all possible combinations before attempting implementation.

Pseudocode

1. Open the input file (TestCase3.txt)

2. If the file fails to open, print an error message and exit with a non-zero status.

3. Initialize variables:

- matrixT as a vector of vectors of strings for the text matrix

- matrixP as a vector of vectors of strings for the pattern matrix

- pattern as a string for storing the pattern

- expectedResult as a string for storing the expected result

- currentRow as an integer to keep track of the current row being processed

- isFirstLine as a boolean to skip the first line of the matrix

- isMatrixT as a boolean to determine if the lines being read are for matrix T

- isPattern as a boolean to determine if the lines being read are for the pattern 4. Read lines from the file:

a. If the line is empty:

- If isMatrixT is true, set isMatrixT to false and continue.

- Set isPattern to the opposite of its current value.

- Continue to the next iteration.

b. If isMatrixT is true:

- If isFirstLine is true, set isFirstLine to false and continue to the next iteration.

- Split the line using ',' as a delimiter and populate the current row of matrixT.

- Increment currentRow.

c. Else if isPattern is true:

- Store the line in the pattern string.

d. Else:

- Store the line in the expectedResult string.

5. Close the input file.

6. Print the text matrix (matrixT).

7. Calculate the size of the pattern matrix (patternSize) using the square root of the size of the pattern string.

8. Initialize matrixP as a vector of vectors of strings with size patternSize.

9. Split the pattern string and populate matrixP.

10. Print the pattern matrix (matrixP).

11. Print the expected result (expectedResult).

12. Check if the pattern exists in the text:

- Use the isPatternExists function with matrixT and matrixP.

13. Count the number of diagonal occurrences of the pattern in the text:

- Use the diagonalPatternExistence function with matrixT and matrixP.

14. Print the result:

- If the pattern exists, print "Pattern exists in the text."

- Print the number of diagonal occurrences.

15. If the pattern does not exist, print "Pattern does not exist in the text."

16. Exit the program with a status of 0.

Analysis of Time Complexity

Let's break down the time complexity of the given program:

Reading Input:
Reading the input file line by line: Let the number of lines in the file be n.
Time Complexity: O(n)

Parsing Input into Matrices:
Building matrixT and matrixP from the input involves iterating through the lines and columns.
The size of matrixT is determined by the number of lines in the file and the number of elements in each line.
matrixP is determined based on the pattern's size, which is calculated as the square root of the total characters read from the file.
Time Complexity: O(n) for matrixT, and O(sqrt(n)) for matrixP (where n is the total number of characters in the input).
Pattern Matching:
isPatternExists function checks for the existence of the pattern in the text using nested loops.
diagonalPatternExistence function also involves nested loops to search for diagonal occurrences of the pattern.
These functions iterate through the text matrix and perform comparisons for pattern matching.
Time Complexity: O(textRows * textCols * patternRows * patternCols) for isPatternExists and diagonalPatternExistence.
Summary:
Reading input: O(n)
Parsing input into matrices: O(n) + O(sqrt(n))
Pattern Matching: O(textRows * textCols * patternRows * patternCols)
Therefore, the overall time complexity of the given program can be summarized as follows:

O(n) + O(n) + O(sqrt(n)) + O(textRows * textCols * patternRows * patternCols)

Final Time Complexity:

Big O = O (N ^ 2 * M ^ 2)

The exact time complexity may vary based on the actual sizes of the matrices and the input file. The patterns within the text can also affect the runtime performance, especially for the pattern matching section, where comparisons are made between the matrices.

Efficiency over Other Simpler Solutions:

The provided code is relatively straightforward and should perform well for small to medium-sized matrices. However, it's important to note the factors that contribute to its efficiency:

- Algorithm Simplicity:

The algorithm used in the code is relatively simple and easy to understand. It involves iterating through the text matrix and checking for the existence of the pattern as well as counting diagonal occurrences.

- Memory Usage:

The code uses two matrices (matrixT and matrixP) to represent the text and pattern. While this increases memory usage, it provides a clear structure for handling the matrices.

- Readability and Maintainability:

The code is structured in a way that separates different tasks into functions, enhancing readability. This can be beneficial for maintenance and future modifications.

- Specificity to the Problem:

The code is designed to address the specific requirements of checking both the general existence of the pattern and counting diagonal occurrences. This specificity might be advantageous in scenarios where both checks are necessary.

- Ease of Implementation:

The code uses straightforward logic without employing complex algorithms, making it relatively easy to implement and debug.

- Scalability for Small to Medium-Sized Matrices:

The provided code should perform well for matrices of moderate size. It's well-suited for scenarios where the matrices are not exceptionally large.

- Optimization for the Task:

The code's focus on both checking general pattern existence and counting diagonal occurrences suggests it is optimized for the specific requirements of the problem.

While the code has these advantages, it's important to note that for very large datasets or matrices, the time complexity (O(N * M * P * Q)) might become a limiting factor. In such cases, more advanced algorithms like string matching algorithms or other optimization techniques might be considered.

Question 3

Part A

Problem : Calculation of total time between paths and average time

Calculate the average travel time between various key locations within a vibrant cityscape.

Nodes & Distances:

- Park (A) to Library (B): 10 minutes
- Park (A) to Cafe (C): 15 minutes
- Park (A) to Museum (D): 20 minutes
- Library (B) to Cafe (C): 12 minutes
- Library (B) to Museum (D): 18 minutes

- Cafe (C) to Museum (D): 25 minutes

Scenario: In this bustling cityscape, various locations hold distinct charms and activities for its residents.

- The Park (A) serves as a green space, inviting families and fitness enthusiasts alike.
- Library (B) is a sanctuary for book lovers and students seeking a quiet study environment.
- The Cafe (C) acts as a cozy spot where people gather to chat over coffee or work on laptops.
- The Museum (D) highlights intriguing exhibits, attracting art enthusiasts and history buffs.

Calculation: The assignment involves computing the average travel time between these significant locations through various potential routes.

Pseudocode

```
1.  // Define constant for the maximum number of nodes
2.  const MAX_NODES = 26
3.
4.  // Function to read distance data from a file
5.  int readDistanceData(filePath: string, distances: int[MAX_NODES][MAX_NODES]) -> int:
6.      rows = 0
7.      open file at filePath
8.      if file is open:
9.          loop while there are lines in the file:
10.             read a line from the file
11.             node1, node2, distance = parseLine(line)
12.             distances[node1 - 'A'][node2 - 'A'] = distance
13.             distances[node2 - 'A'][node1 - 'A'] = distance
14.             print node1, node2, distance
15.             increment rows
16.         close the file
17.     else:
18.         print "Unable to open file: filePath"
19.
20.     return rows
21.
22. // Function to parse a line and extract node1, node2, and distance
23. (node1: char, node2: char, distance: int) parseLine(line: string) -> (char, char, int):
24.     initialize node1, node2, distance
25.     count = 0
26.     loop through characters in line:
27.         if count is 0:
```

```
            set node1 to the current character
        else if count is 2:
            set node2 to the current character
        else if count > 3:
            append the current character to distance
        increment count

    return (node1, node2, stoi(distance))

// Function to read paths data from a file
int readPaths(filePath: string, distances: int[MAX_NODES][MAX_NODES], rowCount: int, paths:
    float) -> int:
    totalSumOfTimes = 0
    open file at filePath
    if file is open:
        loop while there are lines in the file:
            read a line from the file
            if rowCount is less than or equal to current row:
                totalSumOfTimes += calculateTotalTime(distances, line)
                increment paths
            increment row

        close the file
    else:
        print "Unable to open file: filePath"

    print "Total number of paths: paths"
    return totalSumOfTimes

// Function to calculate the total time for a given path
int calculateTotalTime(distances: int[MAX_NODES][MAX_NODES], line: string) -> int:
    totalSumOfTimes = 0
    node1, node2 = parsePath(line)
    totalSumOfTimes += distances[node1 - 'A'][node2 - 'A']

    return totalSumOfTimes

// Function to parse a path and extract node1 and node2
(node1: char, node2: char) parsePath(line: string) -> (char, char):
    initialize node1, node2
    count = 0
    loop through characters in line:
        if count is 0:
            set node1 to the current character
```

```
        else if count is 2:
            set node2 to the current character
        else if count % 2 is 0:
            set node1 to node2
            set node2 to the current character
        increment count

    return (node1, node2)

// Main function
int main() -> int:
    filePath = "testcase2.txt"
    paths = 0
    distances[MAX_NODES][MAX_NODES] = { 0 }
    row = readDistanceData(filePath, distances)
    totalSum = readPaths(filePath, distances, row, paths)
    print "Total sum of path is: totalSum"
    avgTimeBetweenLocations = totalSum / paths
    print "Average Time to move between locations is: avgTimeBetweenLocations"
    return 0
```

Important Libraries

- <iostream>:

Importance: This library is fundamental for handling input and output operations in C++. It provides functionality for standard input (cin) and standard output (cout). In this code, it is used for printing messages to the console, such as error messages or intermediate results.

- <fstream>:

Importance: This library provides facilities for file input and output operations in C++. It is used to open, read, and close files in code. In this specific code, it is essential for reading data from an external file (testcase2.txt).

- <string>:

Importance: The **<string>** library provides functionality for working with strings in C++. It is used in the code to manipulate and store strings, particularly for extracting characters from lines read from the file.

Analysis of Time Complexity

L is the number of total paths provided in the file.

- Calls readDistanceData() function, which has a time complexity of O(L).
- Calls readPaths() function, which also has a time complexity of O(L).

The total time complexity is dominated by the file reading operations.

Overall Time Complexity: O(L), where L is the total number of characters in the file.

Summary:

The time complexity of the provided code is linear with respect to the total number of characters in the input file. It's important to note that the complexity is influenced by the length of the lines in the file. If the length of the lines is constant or relatively small, the code's time complexity effectively becomes linear with respect to the number of lines in the file.

It's worth mentioning that this analysis assumes that the stoi function and other standard C++ library functions used in the code operate in constant time.

Effectiveness over other Algorithms

The provided C++ code is designed to read distances between nodes from a file, calculate the total travel time for specific paths, and then determine the average travel time between key locations in a cityscape. It follows a straightforward approach using nested loops to parse the input file and calculate path costs.

Key Features of the Code:

- **Modularity**: The code is modularized into functions (readDistanceData, readPaths) to enhance readability and maintainability.
- **Clear Logic**: The logic for parsing the input file and calculating path costs is relatively clear and easy to follow, making the code understandable.
- **Backtracking Approach**: The algorithm uses a backtracking approach to generate all possible paths, contributing to its flexibility and adaptability for different scenarios.

Efficiency Comparison:

- **Flexibility**: The code can handle different scenarios and variable path configurations by dynamically calculating distances between nodes.
- **Readability**: The code is well-structured and readable, which is advantageous for maintenance and future modifications.
- **Dynamic Path Generation**: The use of backtracking allows the code to adapt to changes in the number of nodes or paths.

Part B

Problem: Shortest Cycle in a Communication Network

Sarah, an aspiring computer scientist, is working on a project involving network analysis. She is tasked with identifying the shortest cycle in a communication network to optimize data transfer. A cycle, in this context, is a path that starts and ends at the same node, utilizing each connection exactly once.

Sarah needs to write a function that takes the number of vertices and the connections between them as input. Each vertex is assigned a unique label ranging from 0 to n - 1. The connections between vertices are represented by a 2D integer array where each array, edges[i] = [ui, vi], denotes a link between vertex

Ui and vertex vi. It is important to note that there is at most one connection between any pair of vertices, and no vertex has a direct connection to itself.

The task is to return the length of the shortest cycle in the network. If there is no cycle in the network, the function should return -1.

Pseudocode

1. Set filePath = "testcase3.txt" // Replace with the actual path to your file
2. Initialize an empty string line
3. Open the file at filePath
4. If the file is open, read the first line and extract the number of vertices (numVertex)
5. Close the file
6. Parse numVertex into an integer and assign it to n (Number of vertices)
7. Initialize an array of linked lists arrayLinkedList of size n
8. For each vertex i from 0 to n-1: a. Add vertex i to the linked list at arrayLinkedList[i]
9. Reopen the file at filePath
10. If the file is open, read each line starting from the second line:
11. Parse the first and second characters of the line as vertex1 and vertex2
12. Parse vertex1 and vertex2 into integers ver1 and ver2
13. For each linked list at arrayLinkedList[j]:
14. Find ver1 in the linked list
15. If ver1 is found, add ver2 to the end of the list
16. Close the file
17. Initialize sizeCycle = INT_MAX, cycleNum = 0, flag = 0
18. For each vertex i from 0 to n-1:
19. Initialize count = 0
20. For each element it in the linked list at arrayLinkedList[i]:
21. If it is equal to the first element in the linked list:
22. - Set flag = 1
23. - If sizeCycle > count + 1, update sizeCycle and cycleNum
24. - Break from the loop
25. Increment count
26. If flag is true:
27. Print "Size of the shortest cycle is: " + sizeCycle
28. Print "The shortest cycle of this network is: "
29. For each element it in the linked list at arrayLinkedList[cycleNum]:
30. Print it
31. If it is not the last element, print " -> "
32. Print a newline
33. If flag is false, print "-1"
34. Return 0

Important Libraries

- **<iostream>:**

This library is used for input and output operations. In the provided code, it is used for printing messages to the console using cout.

- **<fstream>:**

This library is used for file input and output operations. In the code, it is utilized to read from an external file (testcase3.txt). The ifstream (input file stream) class is employed to open and read from the file.

- **<string>**:

This library is used for string operations. In the code, it is used to manipulate strings, such as concatenating characters to form a line read from the file and converting strings to integers using stoi (string to integer) for parsing the number of vertices, vertex labels, etc.

- **<list>:**

This library provides the list container, which is a doubly linked list implementation in C++. In the code, it is used to represent an array of linked lists (arrayLinkedList). Each linked list corresponds to a vertex, and it stores the vertices connected to that particular vertex.

Analysis of Time Complexity

Reading the Number of Vertices (Lines 3-11):

This part involves reading the file, extracting the number of vertices, and initializing an array of linked lists.

Time complexity: O(n), where n is the number of vertices.

Reading Connections and Updating Linked Lists (Lines 13-29):

For each line (connection), the code iterates through the linked lists to find and update the connections.

For each connection, it performs operations on the linked lists, and the worst-case scenario is iterating through all n linked lists.

Time complexity: O(n^2 * m), where m is the number of connections.

Finding the Shortest Cycle (Lines 31-58):

The code iterates through each vertex and each linked list to find the shortest cycle.

The worst-case scenario is that each vertex is part of a cycle, and each linked list is traversed completely.

Time complexity: O(n^2).

Overall, the dominant factor in the time complexity is the iteration over connections and vertices. The time complexity of the entire code is approximately O(n^2 * m).

It's important to note that this analysis assumes a worst-case scenario, and the actual time complexity may be lower in practice depending on the network structure. Additionally, the constants hidden in the big-O notation are not considered in this analysis.

Time Complexity: O (n ^ 2 * m).

Effectiveness over other Algorithms

The provided code uses an adjacency list representation with linked lists to represent the graph, and it employs a straightforward approach to finding the shortest cycle in the network. Let's discuss its efficiency and compare it to other potential solutions:

Efficiency:

Space Complexity:

The code uses a list of linked lists to represent the graph, resulting in a relatively low space complexity. The space used is proportional to the number of vertices (O(n)).

Time Complexity:

The time complexity of the provided code is O(n^2 * m), where n is the number of vertices and m is the number of connections. This time complexity arises from the nested loops iterating over vertices and linked lists.

The algorithm is efficient for small to medium-sized networks but may become less optimal as the number of vertices and connections increases.

Comparison with Other Solutions:

Breadth-First Search (BFS) or Depth-First Search (DFS):

The provided solution is similar to a DFS-based approach. While BFS or DFS can be used to find cycles, the exact implementation details can vary.

The efficiency of BFS or DFS depends on the structure of the network. For dense graphs, the provided solution's time complexity may be competitive, but for sparse graphs, more optimized algorithms might be faster.

Floyd-Warshall Algorithm:

The Floyd-Warshall algorithm is a dynamic programming approach that can find the shortest cycles in a weighted graph (all-pairs shortest paths).

It has a higher time complexity (O(n^3)) but is more suitable for dense graphs with weighted edges.

The provided code is simpler and may perform better for sparse graphs.

Johnson's Algorithm:

Johnson's algorithm combines Dijkstra's algorithm and the Bellman-Ford algorithm to find all-pairs shortest paths.

It is more complex and has a higher time complexity but is efficient for dense graphs with both positive and negative weights.

The code provided is simpler and may be more suitable for sparse graphs without negative weights.

Conclusion: The efficiency of the code provided depends on the characteristics of the communication network. For small to medium-sized sparse networks, the code may offer a reasonable trade-off

between simplicity and performance. However, for large or dense networks, or networks with specific characteristics, more specialized algorithms might be considered for better efficiency. It's important to consider the network structure and constraints when choosing an algorithm for network analysis.