**Name: Kanwar Muhammad Daniyal**

**Abdul Subhan 21I-1223**

**Shahzeb Umer 21i-0893**

**Section: P**

**Subject: Analysis of Algorithm**

**Question no. 1**

**Iterative Bottom-Up algorithm with Memoization :**

**Main Pseudocode**

Function maximizeProfitBottomUp(customTotalSize, customPrices)

Create dp vector of size (customTotalSize + 1) with default CustomBurfi objects

For i from 1 to customTotalSize

For each price in customPrices

Calculate customPortionSize

If customPortionSize is less than or equal to i

potentialResult = dp[i - customPortionSize]

potentialProfit = customPrice + potentialResult.customProfit

If potentialProfit is greater than dp[i].customProfit

Update dp[i].customProfit with potentialProfit

Update dp[i].customPath with potentialResult.customPath + customPortionSize

Return dp[customTotalSize]

End Function  
  
**Time Complexity Analysis**

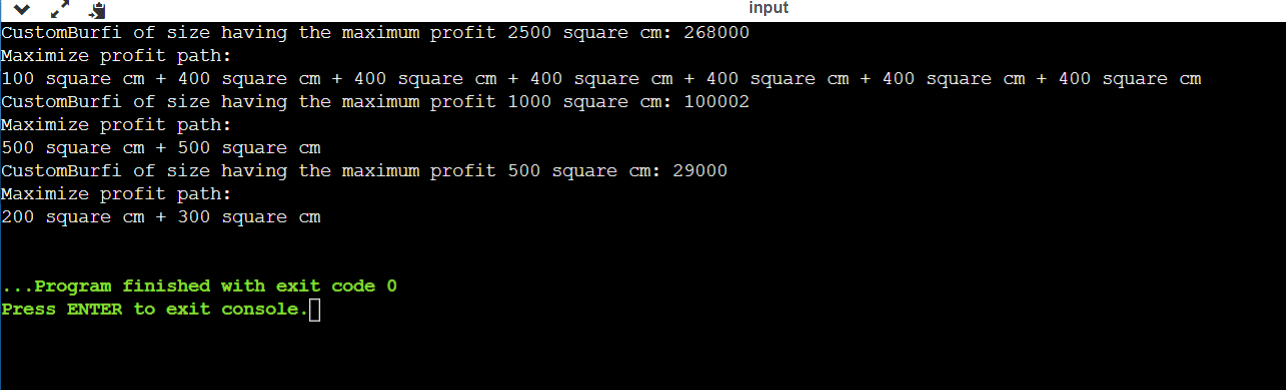
The time complexity of maximizeProfitBottomUp can be analyzed as follows:

• The function has two nested loops.

• The outer loop runs from 1 to customTotalSize, so it runs customTotalSize times.

• The inner loop iterates over all customPrices. Let's assume the number of prices is n.

• Inside the inner loop, the operations (calculating profit, updating dp array) are constant time operations.  
  
Therefore, the time complexity is **O(customTotalSize×n)**, where customTotalSize is the total size for which we are calculating the profit and n is the number of different portion sizes.



**Simple Recursive Top-Down algorithm without Memoization :**

**Main Pseudocode :**

Function maximizeProfitRecursive(customTotalSize, customPrices)

If customTotalSize is 0

Return a new CustomBurfi with 0 profit and empty path

Initialize customMaxResult as a new CustomBurfi

For each price in customPrices

Calculate customPortionSize

If customPortionSize is less than or equal to customTotalSize

Recursive call to maximizeProfitRecursive with reduced size

Calculate customCurrentProfit

If customCurrentProfit is greater than customMaxResult's profit

Update customMaxResult's profit and path

Return customMaxResult

End Function

**Time Complexity Analysis :**

The maximizeProfitRecursive function uses a recursive approach, and its time complexity can be analyzed as follows:

* Let n be the size of customPrices, and T be customTotalSize.
* The function is called recursively for each portion size, reducing the customTotalSize by the portion size in each call.
* At each level of recursion, the function iterates over n portion sizes.
* The recursion depth can go up to T/100 (as the smallest portion is 100 square cm).

The **worst-case time** complexity is approximately *O*(*n^T*/100), which is exponential and can be very inefficient for large values of **T** and **n**.

**Comparison with Other Solutions**

**1. Iterative Bottom-Up Dynamic Programming:**

• This approach would build a table iteratively, storing the maximum profit for each size up to T.

• Time complexity: O(T×n), significantly more efficient than the recursive approach for large values of T and n.

• Space complexity: Also requires O(T) space for the table, but avoids the stack space overhead of recursion.

**2. Recursive Approach with Memoization:**

• Similar to the provided recursive approach, but stores results of subproblems in a cache to avoid redundant calculations.

• Time complexity: Improved to O(T×n), similar to the iterative approach.

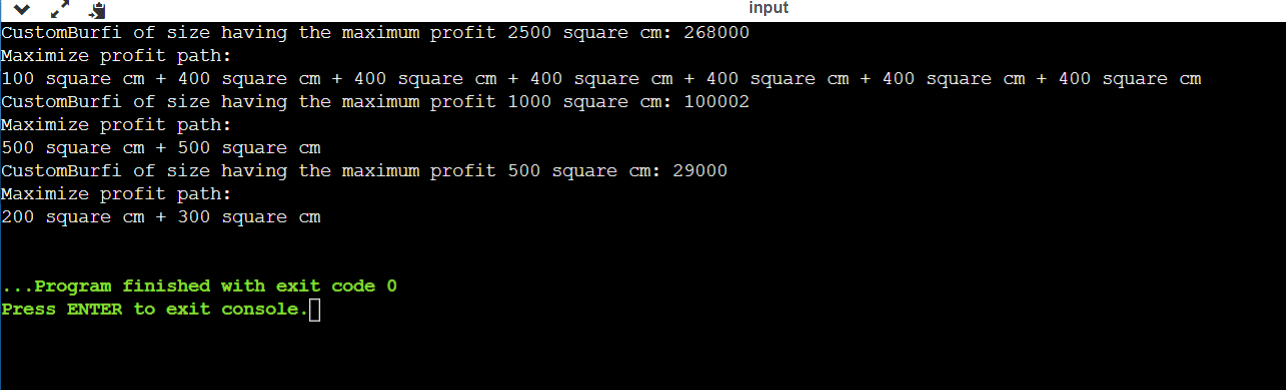
• Still suffers from potential stack overflow issues for deep recursion.

**3. Naive Recursive Approach (Current Solution):**

• Simple to implement but highly inefficient for large inputs due to exponential time complexity.

• Can quickly become impractical as T and n increase.

In summary, while the current naive recursive solution is straightforward and easy to understand, it is significantly less efficient compared to an iterative bottom-up dynamic programming approach or a recursive approach with memoization, especially for larger input sizes. The exponential growth of the recursive calls in the naive approach makes it unsuitable for practical use in cases where T and n are large.



**Recursive Top-Down algorithm with Memoization :**

**Main Pseudocode :**

Function maximizeProfitRecursive(customTotalSize, customPrices)

Check if customTotalSize is in memoization cache

If yes, return the cached result

If customTotalSize is 0

Return a new CustomBurfi object with zero profit and empty path

Initialize an empty CustomBurfi object customMaxResult

Loop through each price in customPrices

Calculate customPortionSize

If customPortionSize is less than or equal to customTotalSize

Recursively call maximizeProfitRecursive for the remaining size

Calculate the total profit for this portion size

If the calculated profit is greater than customMaxResult's profit

Update customMaxResult with this new profit and path

Store customMaxResult in the memoization cache for customTotalSize

Return customMaxResult

End Function  
  
**Time Complexity Analysis:**

Time Complexity Analysis

The function maximizeProfitRecursive uses memoization to improve the efficiency of the recursive solution. The time complexity can be analyzed as follows:

* Let n be the size of customPrices, and T be the customTotalSize.
* The function recursively explores each combination of portion sizes, but with memoization, each state (each unique value of customTotalSize) is solved only once.
* The recursion depth and the number of subproblems to solve are both proportional to T/100 (as the smallest portion size is 100 square cm).
* At each recursive call, the function iterates over all n prices.

### With memoization, the time complexity is significantly reduced to *O*(*T*×*n*). Without memoization, this would have been exponential, approximately *O*(*n^T*/100).

**Comparison with Other Solutions**

1. **Naive Recursive Solution:**

• Without memoization, the solution would have an exponential time complexity of approximately O(nT/100), becoming impractical for larger values of T and n.

2. **Iterative Bottom-Up Dynamic Programming:**

• An alternative approach that also achieves O(T×n) time complexity.

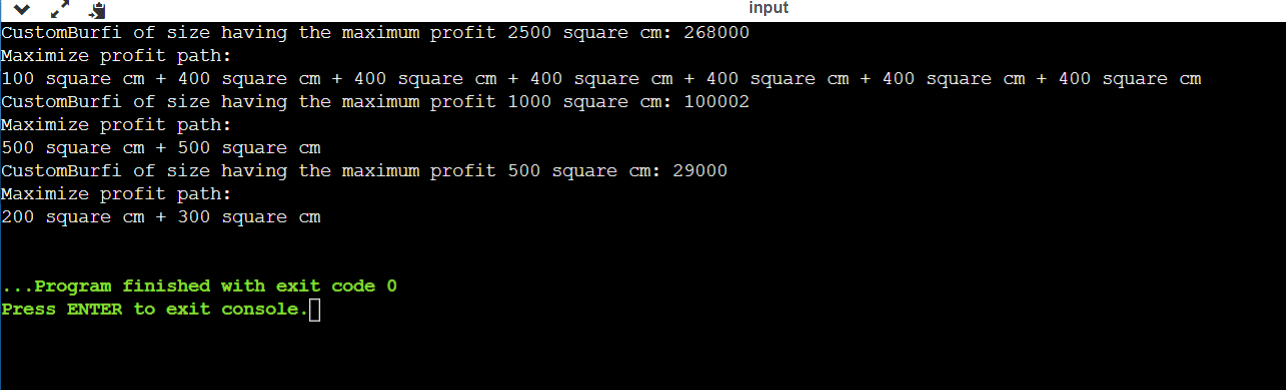
• This method avoids the overhead of recursive calls and is generally more space-efficient.

3. **Recursive Approach with Memoization (Current Solution):**

• The provided solution uses recursion with memoization, reducing the time complexity from exponential to O(T×n).

• This method is more efficient than the naive recursive solution and is comparable in time complexity to the iterative bottom-up approach. However, it might still suffer from stack overflow for very deep recursion.

In conclusion, the provided solution with memoization is significantly more efficient than a naive recursive approach and is comparable to an iterative bottom-up dynamic programming solution in terms of time complexity. However, the iterative approach might still have advantages in terms of space efficiency and avoiding stack overflow issues.



**Optimize storage in the Iterative algorithm if possible:**

**Report on Custom Burfi Cutting Problem :**

**Main Pseudocode :**

Function maximizeProfitBottomUp(customTotalSize, customPrices)

Initialize a vector maxProfit with size customTotalSize+1, all values set to 0

Initialize a vector lastPortion with size customTotalSize+1, all values set to -1

For i from 1 to customTotalSize inclusive

For each price in customPrices

Calculate customPortionSize as (index of price + 1) \* 100

If customPortionSize is less than or equal to i

Calculate potentialProfit as current price + maxProfit at (i - customPortionSize)

If potentialProfit is greater than current maxProfit at i

Update maxProfit at i to potentialProfit

Update lastPortion at i to customPortionSize

Initialize an empty path vector

Set a variable size to customTotalSize

While size is greater than 0

Get the portion size from lastPortion at index size

If portion size is not -1

Add portion size to path

Subtract portion size from size

Create a CustomBurfi object with calculated maxProfit and path

Return the CustomBurfi object

End Function  
  
**Time Complexity Analysis**

The time complexity of maximizeProfitBottomUp is O(Tn), where T is the customTotalSize and n is the number of different portion sizes (the length of customPrices).

* The outer loop runs from 1 to customTotalSize, iterating T times.
* The inner loop iterates over the customPrices array, which has n elements.
* The operations inside the inner loop are constant time.

Hence, the overall time complexity is the product of these two loops, i.e., O(Tn).

**Comparison with Simpler Solutions :**

**Naive Recursive Solution:**

* Without memoization, a recursive approach would have an exponential time complexity, approximately **O(2^n)**, which becomes impractical for large n.
* It would explore all possible combinations, leading to a large number of redundant calculations.

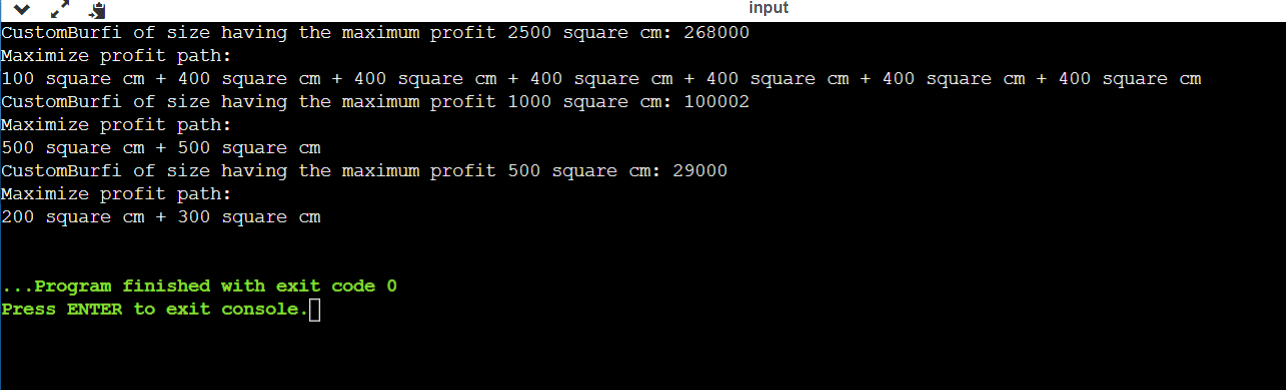
**Recursive with Memoization:**

* This approach would improve the naive recursive solution by storing results of sub-problems.
* Time complexity would be reduced to O(Tn), similar to the bottom-up approach.
* However, it might suffer from stack overflow due to deep recursion.

**Iterative Bottom-Up Dynamic Programming (Current Solution):**

* Avoids the overhead of recursive calls.
* Efficiently stores and reuses sub-problem solutions, avoiding redundant calculations.
* Provides a clear improvement in both time and space complexity over a naive recursive approach.
* Comparable in efficiency to the recursive approach with memoization but without the risk of stack overflow.

In conclusion, the current bottom-up dynamic programming solution provides an efficient and scalable approach to solving the problem. It avoids the inefficiencies of a naive recursive solution and offers a practical alternative to a recursive approach with memoization, especially for large input sizes.



**Question no. 2**

**Analysis:**

**Reading File and Matrix Initialization:**

The code opens a file ("1.txt") and reads its contents to initialize a square matrix and a pattern matrix.

It reads the first line to determine the size of the square matrix.

It initializes matrices charArray and pattArray based on the provided data.

Matrix Comparison for Pattern:

It compares the provided pattern matrix against different diagonal positions in the larger matrix to count occurrences.  
**Time Complexity:**

Reading File: Reading the file involves iterating through its contents once. This process has a time complexity of O(N), where N is the total number of characters in the file.

Matrix Initialization: Initializing matrices charArray and pattArray takes O(N) time, where N is the size of the matrix.

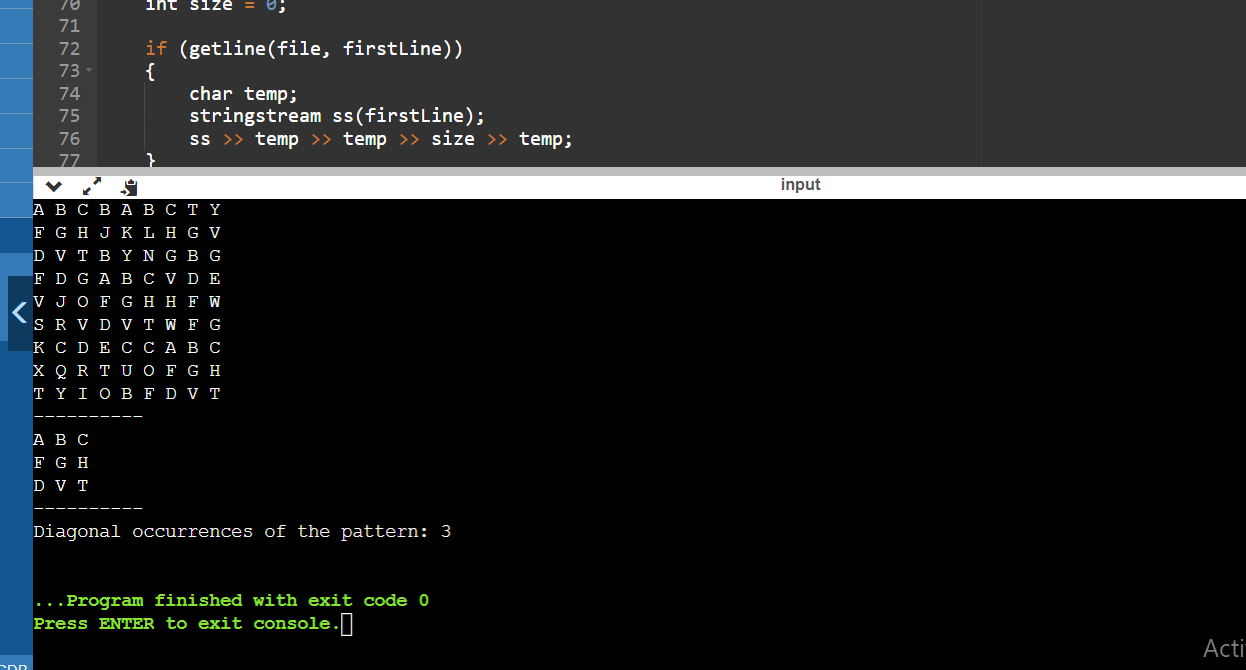
Pattern Search: The function findDiagonalOccurrences iterates through the larger matrix and checks diagonal positions. This search involves nested loops and compares elements of the pattern matrix, resulting in a time complexity of O(N^2 \* M^2), where N is the size of the matrix and M is the size of the pattern matrix.

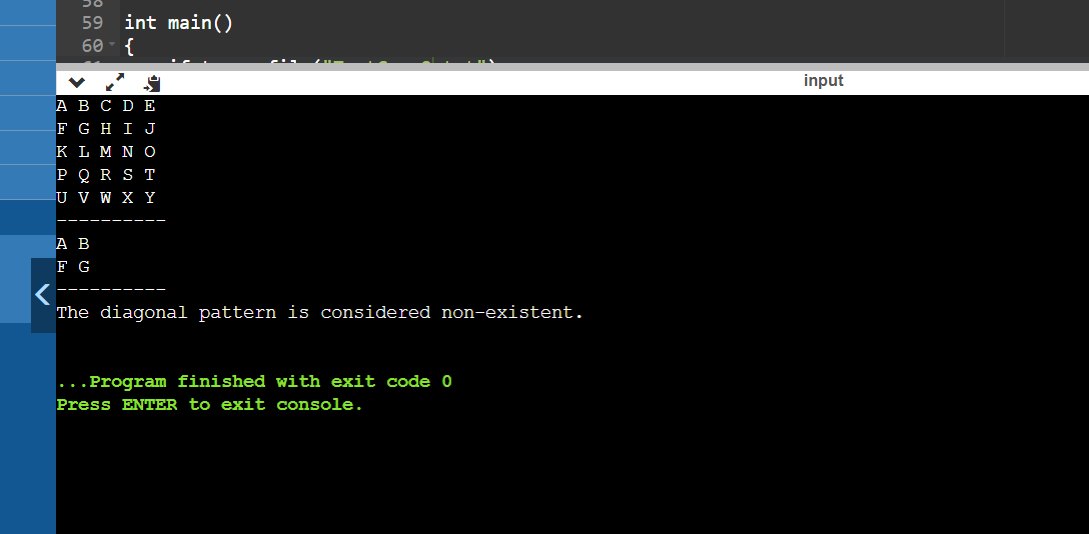
**Space Complexity:**

Matrix Storage: The code uses dynamic memory allocation to create and store matrices charArray and pattArray. The space complexity for these matrices is O(N^2), where N is the size of the matrix.

Vector Storage: It uses vectors text and pattern to represent the matrices extracted from the file. These vectors store the characters of the matrices and have a space complexity of O(N^2), where N is the size of the matrix.

Other Variables: Additional space is used for variables like string and int, but they have a negligible impact compared to the matrix storage.





**Question no. 3**

**Part A**

**Main Pseudocode :**Function isWhitespaceOrEmpty(line)

For each character ch in line

If ch is not a whitespace

Return false

Return true

Function calculatePathTime(path)

Split path into nodes

Initialize totalTime to 0

For each segment in nodes

Find segment time in path\_times

If segment time is found

Add segment time to totalTime

Else

Output error and return -1

Return totalTime

Main

Open input file "testcase1.txt"

Initialize variables for path times and paths to calculate

Read lines from file

If line is empty

Switch to reading paths to calculate

Else If reading path times

Extract path and time, store in path\_times map

Else

Store path in pathsToCalculate

For each path in pathsToCalculate

Calculate time for path

If time is valid

Output travel time for path

Add time to total

Increment validPathsCount

Calculate and output average time

Close file  
  
**Time Complexity Analysis**

* Reading path times and paths to calculate: O(n), where n is the number of lines in the file.
* calculatePathTime: O(m), where m is the number of nodes in a path. In the worst case, m could be comparable to the length of the path string.
* Main loop for paths calculation: O(p \* m), where p is the number of paths to calculate.

Overall, the time complexity is O(n + p \* m). In most practical scenarios, since m (number of nodes in a path) is generally small and fixed, the complexity can be approximated as O(n + p), dominated by the file reading and paths processing.

**Comparison with Simpler Solutions**

**Naive Approach (Without Preprocessing Path Times):**

* Would involve recalculating path times for each query, leading to O(p \* n) complexity.
* Inefficient for large numbers of queries (p) or when path times (n) are extensive.

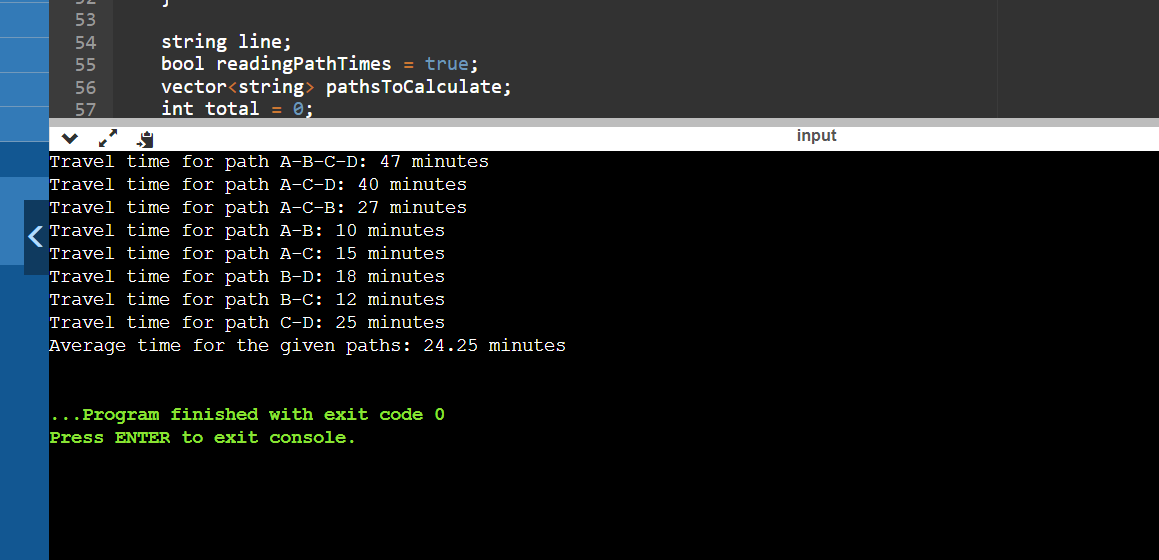
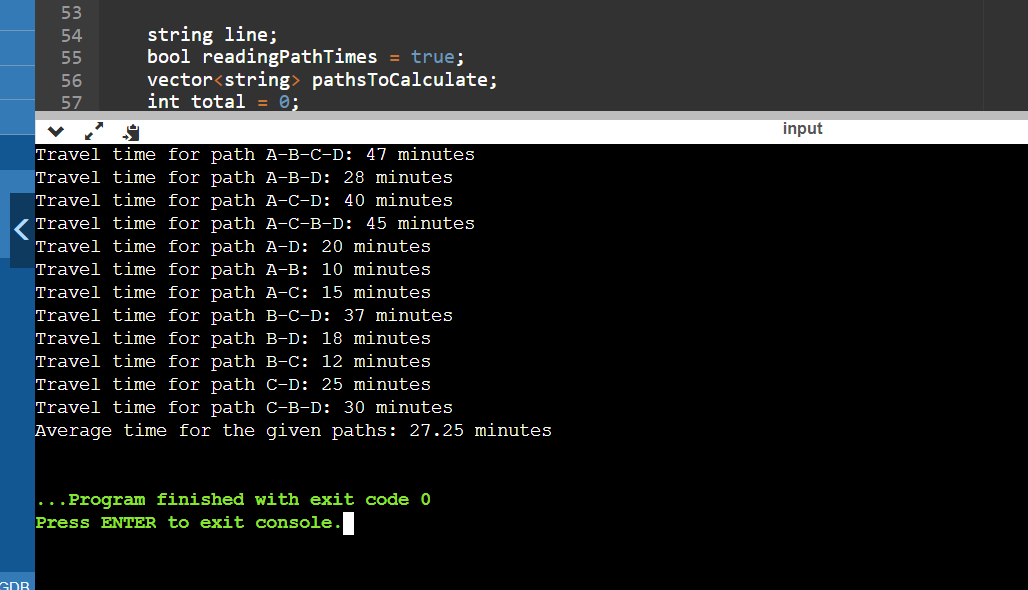
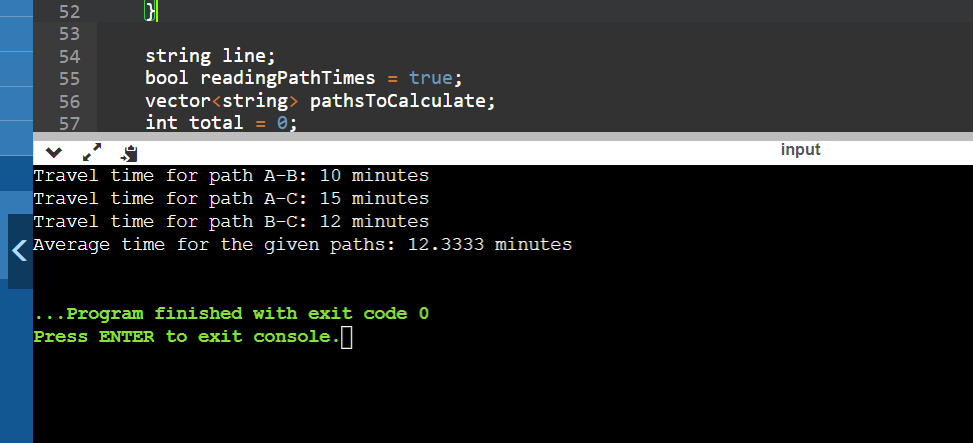
**Using a Preprocessed Map (Current Solution):**

* Preprocessing path times into a map reduces the time complexity to O(1) for each path time lookup.
* Significantly more efficient for repeated queries.

**Space Trade-off:**

* The current solution uses additional space for the map to store path times.
* This trade-off is beneficial for time efficiency, especially with multiple path queries.

In conclusion, the current solution is efficient for scenarios with multiple path queries, offering a significant time-saving over a naive approach. The use of a map for storing path times allows for quick lookups, optimizing the program for repeated use.



**Question # 03**

**Part B**

# Analysis:

* The algorithm uses a depth-first search (DFS) approach to explore the graph and find the shortest cycle.
* The DFS is performed for each vertex to ensure that cycles starting from different vertices are considered.
* The algorithm keeps track of the distance from the starting vertex to each visited vertex and uses this information to identify cycles.
* The construction of the cycle path involves traversing the parent pointers, allowing the algorithm to identify and store the vertices in the cycle.

The time complexity is influenced by both the number of vertices (V) and the number of edges (E). In the worst case, the algorithm needs to explore all vertices and edges to identify the shortest cycle. The use of DFS ensures that all possible paths are considered, leading to a comprehensive exploration of the graph.

## Time Complexity:

The time complexity of the provided code is O(V \* (V + E)), where V is the number of vertices and E is the number of edges in the graph.

Let's break down the time complexity:

1. **Outer Loop (for each vertex): O(V)**
   * The outer loop runs for each vertex in the graph.
2. **DFS (for each edge): O(V + E)**
   * The DFS function is called for each vertex.
   * In the worst case, the DFS can visit each vertex and each edge once.
3. **Constructing the cycle path: O(V)**
   * Constructing the cycle path involves traversing the parent pointers, which takes O(V) time.

Combining these components, we get a time complexity of O(V \* (V + E)).

## Space Complexity:

The space complexity is mainly determined by the vectors used for storing adjacency lists, distances, and parents, as well as the recursion stack during DFS.

* The adjacency list requires *O*(*V*+*E*) space.
* The vectors for distances and parents require *O*(*V*) space.
* The recursion stack for DFS can go up to the depth of the recursion tree, which is *O*(*V*) in the worst case.

Therefore, the overall space complexity is *O*(*V*+*E*). The space complexity is influenced by the number of vertices and edges of the graph.

# Test Cases Implementation Pictures:



