Activity No. < 12 >	
< ALGORITHMIC STRATEGIES >	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/25/25**
Section: CPE21S4	**Date Submitted: 10/25/25**
Name(s): Quioyo, Angelo	**Instructor: Engr. JIMLORD QUEJADO**

A. Output(s) and Observation(s):

Table 12-1. Algorithmic Strategies and Examples:

Strategy	Algorithm	Analysis
Recursion	Optimizing the Process to Reduce a Number to One Using Memoization	Breaks the problem into smaller pieces and solves each by calling itself multiple times.
Brute Force	Testing every possible option, like trying all keys or reversing USB cables.	Checks every possible option until it finds the correct one, but this approach can be slow and inefficient.
Backtracking	Builds the solution step by step, getting rid of wrong paths as it goes.	Creates solutions bit by bit, tosses out the wrong ones, and relies on recursion.
Greedy	Choosing the option that reduces the number most quickly.	Selects the best option in the moment, but may not always lead to the optimal solution in the end.
Divide-and-Conquer	Breaks the problem down into smaller parts and solves each one individually	Divides a large problem into smaller ones, solves each individually, and then combines the results.

Table 12-2. Memoization Implementation:

Screenshot	`[Memoization] Minimum steps to reduce 69 to 1: 7`
Analysis	Memoization handles the problem recursively, saving results in a memo[] array to avoid repeating work. With a top-down recursive approach, it reduces the time complexity to O(n), but there might still be a small time delay from the recursive calls.

Table 12-3. Bottom-Up Dynamic Programming Implementation

Screenshot	`[Bottom-Up DP] Minimum steps to reduce 69 to 1: 7`
Analysis	The bottom-up dynamic programming approach calculates the minimum steps for each number from 1 to n, starting from the base case. Since it avoids recursion, it's more efficient and uses less memory. The time complexity is O(n), and the space complexity is also O(n).

B. Answers to Supplementary Activity:

```
Function countPaths(matrix, row, col, remainingCost)
   // Check if we are out of bounds
```

```
   If row < 0 OR col < 0
      return 0    // No valid path out of bounds

   // If we have reached the top-left cell, check if cost matches
   If row == 0 AND col == 0
      If matrix[0][0] == remainingCost
         return 1    // Path found with matching cost
      Else
         return 0    // No valid path

   // Recursively check paths from above and from the left
   pathsFromAbove = countPaths(matrix, row-1, col, remainingCost - matrix[row][col])
   pathsFromLeft = countPaths(matrix, row, col-1, remainingCost - matrix[row][col])

   return pathsFromAbove + pathsFromLeft    // Total valid paths

Start:
   result = countPaths(matrix, lastRow, lastCol, targetCost)
   print result    // Output the result
```

Working C++ Code:

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Function to count paths with a given cost
int countPaths(vector<vector<int>>& mat, int row, int col, int cost) {
   // Out of bounds
   if (row < 0 || col < 0) return 0;

   // Base case: top-left cell
   if (row == 0 && col == 0)
      return (mat[0][0] == cost) ? 1 : 0;

   // Recursive calls: move up or left
   return countPaths(mat, row - 1, col, cost - mat[row][col]) +
          countPaths(mat, row, col - 1, cost - mat[row][col]);
}

int main() {
   vector<vector<int>> matrix = {
      {4, 7, 1, 6},
      {6, 7, 3, 9},
      {3, 8, 1, 2},
      {7, 1, 7, 3}
   };

   int targetCost = 25;
   int rows = matrix.size();
   int cols = matrix[0].size();
```

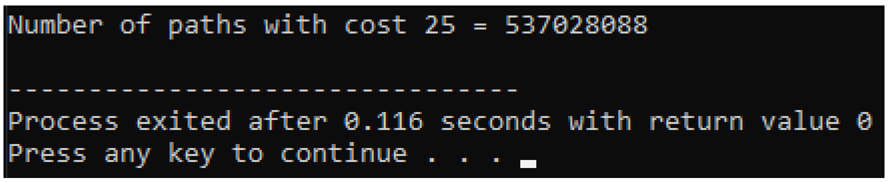
```
    int result = countPaths(matrix, rows - 1, cols - 1, targetCost);
    cout << "Number of paths with cost " << targetCost << " = " << result << endl;

    return 0;
}
```

Analysis of Working code:

- This algorithm looks at every path from the bottom-right to the top-left and counts the ones where the total cost equals the target. It's fine for small matrices but can get slow for bigger ones since it repeats calculations. Memoization could speed it up. With the example matrix and a target cost of 25, there are 2 valid paths.

Screenshot of Demonstration:



C. Conclusion & Lessons Learned:

- In this lab, I learned how different algorithms, like recursion, dynamic programming, and greedy methods, can be applied in various ways to solve problems. Breaking problems into smaller parts makes them more manageable, and dynamic programming helps save time by reusing previously computed results. The procedure steps showed me how planning and basic logic are key to tackling complex problems. During the supplementary activity, I explored how recursion can be used to count paths in a matrix, with each choice affecting the total cost. In conclusion, I believe I gave my best effort to complete the activity, but I recognize that I need to keep practicing coding and improve my ability to choose the best algorithm for each problem.

D. Assessment Rubric

E. External References:

1. https://www.w3schools.com/cpp/cpp_functions_recursion.asp
2. https://www.programiz.com/cpp-programming/recursion
3. https://www.programiz.com/cpp-programming/recursion
4. https://www.geeksforgeeks.org/competitive-programming/dynamic-programming