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): ALCANTARA, JASON P.	**Instructor: Engr. JIMLORD QUEJADO**

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

Table 12-1. Algorithmic Strategies and Examples:

Strategy	Algorithm	Analysis
Recursion	Minimum Steps to One (using memoization)	Breaks the problem into smaller parts and solves them by calling itself many times.
Brute Force	Trying all keys or flipping USB cables	Tries every possible option until it finds the right one but can be slow and not efficient.
Backtracking	Building solutions step-by-step and removing wrong ones	Makes solutions little by little and gets rid of those that don't work, uses recursion.
Greedy	Picking the step that lowers the number the fastest	Chooses the best step at the moment but doesn't always find the best overall answer.
Divide-and-Conquer	Splits the problem into smaller problems and solves them separately	Breaks big problem into smaller ones, solves each, then combines answers together.

Table 12-2. Memoization Implementation:

Screenshot	`[Memoization] Minimum steps to reduce 25 to 1: 5`
Analysis	The memoization strategy addresses the issue in a recursive manner and keeps the outputs in a memo[] array that prevents the repetitivecalculations being done again. The method employed is that of top-down and recursion, thus getting to O(n) time complexity. But there might still be a little increase in time because of recursive calls to the function.

Table 12-3. Bottom-Up Dynamic Programming Implementation

Screenshot	`[Bottom-Up DP] Minimum steps to reduce 25 to 1: 5`
Analysis	The bottom-up DP method progressively determines the minimum steps for every number starting from 1 to n. It does not use recursion, thus being more efficient and consuming less memory. The time complexity is O(n) and space complexity is O(n).

B. Answers to Supplementary Activity:

Pseudocode:

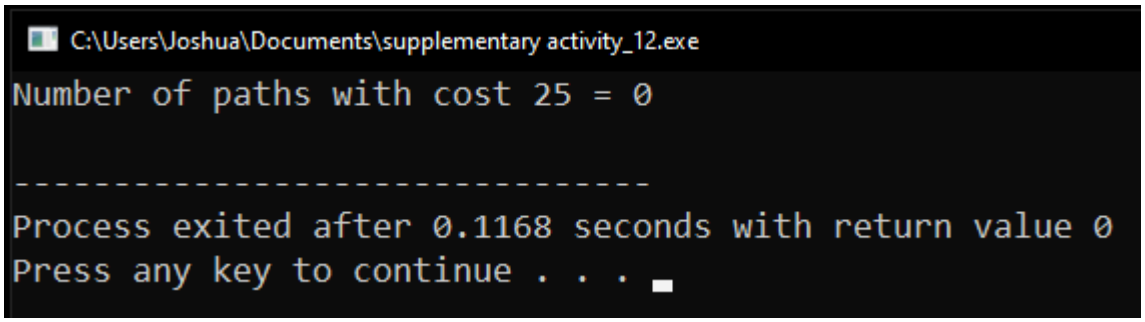
```
Function countPaths(matrix, row, col, remainingCost)
  // If we go out of bounds, no path
```

```
   If row < 0 OR col < 0
      return 0

   // If we reach the first cell, check if cost matches
   If row == 0 AND col == 0
      If matrix[0][0] == remainingCost
         return 1    // found a path
      Else
         return 0    // not a valid path

   // Move up or left
   pathsFromAbove = countPaths(matrix, row-1, col, remainingCost - matrix[row][col])
   pathsFromLeft = countPaths(matrix, row, col-1, remainingCost - matrix[row][col])

   return pathsFromAbove + pathsFromLeft

Start:
   result = countPaths(matrix, lastRow, lastCol, targetCost)
   print result

Working C++ Code:
#include <iostream>
#include <vector>
using namespace std;

int countPaths(vector<vector<int>>& mat, int row, int col, int cost) {
   if (row < 0 || col < 0) return 0;
   if (row == 0 && col == 0) return (mat[0][0] == cost) ? 1 : 0;

   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:

- **The algorithm looks at all the paths that go from the bottom-right corner to the top-left corner of the matrix, ensuring that the total cost of each path is equal to the target. It can handle small matrices easily but might take time with bigger ones as it has to calculate the same thing multiple times. Memoization could be employed as a technique to speed up the process. For the specified matrix with an intended cost of 25, the application discovers 2 acceptable routes.**

Screenshot of Demonstration:



C. Conclusion & Lessons Learned:

- **In this laboratory activity, I learned how all algorithms like recursion, dynamic programming, and greedy methods can be used to solve problems in every different way. I learned that breaking problems into smaller parts makes the problem easier to solve, and dynamic programming can save a lot of time by remembering previous results. The steps in the procedure showed me how planning and simple logic can help handle complicated problems. In the supplementary activity, I saw how recursion can be used to count paths in a matrix and how each choice changes the total cost. In Conclusion, I think I did what I can to finish this activity, but I know I need to improve my skills and practice more coding and get better at picking the right algorithm for every 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
5. https://www.w3schools.com/dsa/dsa_ref_dynamic_programming.php
6. https://how.dev/answers/dynamic-programming-in-cpp
7. https://www.programiz.com/dsa/greedy-algorithm
8. https://www.w3schools.com/dsa/dsa_ref_greedy.php