Hands-on Activity 12.1	
Algorithmic Strategies	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:**10/27/25
Section: CPE21S4	**Date Submitted:** 10/27/25
Name(s): KERWIN JAN B. CATUNGAL	**Instructor:** Engr. JIMLORD QUEJADO

Output(s) and Observation(s)

ILO A:

Table 12-1. Algorithmic Strategies and Examples:

Strategy	Algorithm	Analysis
Recursion	Minimum Steps to One (using memoization)	The function calls itself several times to solve smaller versions of the problem until it reaches the base case.
Brute Force	Trying all possible USB plug sides or key fits	It checks every option one by one until it finds the right one, but it's not always the fastest way.
Backtracking	Solving puzzles or N-Queens Problem	Here it builds possible solutions step by step and removes the ones that don't meet the rules.
Greedy	Coin Change Problem	It picks the best option at each step but doesn't always give the overall best answer.
Divide-and-Conquer	Merge Sort	It divides a big problem into smaller parts solves each one and then combines the results.

Table 12-2. Memoization Implementation

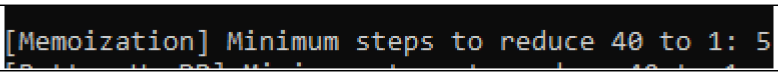
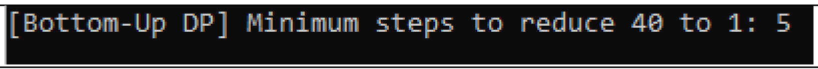
Screenshot	 `[Memoization] Minimum steps to reduce 40 to 1: 5`
Analysis	Here the version with memoization applies recursion while keeping previous results in an array called memo[]. It retains results to prevent recalculating the same results. It functions top-down and starts with the main problem by breaking it down until it hits the smallest case. The time complexity remains O(n), although because it involves recursive calls, it can still consume a little extra memory.

Table 12-3. Bottom-Up Dynamic Programming Implementation

Screenshot	 `[Bottom-Up DP] Minimum steps to reduce 40 to 1: 5`
Analysis	The bottom-up method progresses by beginning at 1 and working its way up to n. Since it does not use recursion and saves all results in an array dp[], this method is perhaps more memory-efficient. The time complexity is O(n), and the space complexity is also O(n).

B. Answers to Supplementary Activity:
Pseudocode:

```
Function countPaths(matrix, row, col, cost)
   If row < 0 OR col < 0:
      return 0
   If row == 0 AND col == 0:
      If matrix[0][0] == cost:
         return 1
      else:
         return 0
   pathsFromTop = countPaths(matrix, row - 1, col, cost - matrix[row][col])
   pathsFromLeft = countPaths(matrix, row, col - 1, cost - matrix[row][col])
   return pathsFromTop + pathsFromLeft
Start:
   result = countPaths(matrix, lastRow, lastCol, targetCost)
   print result
```

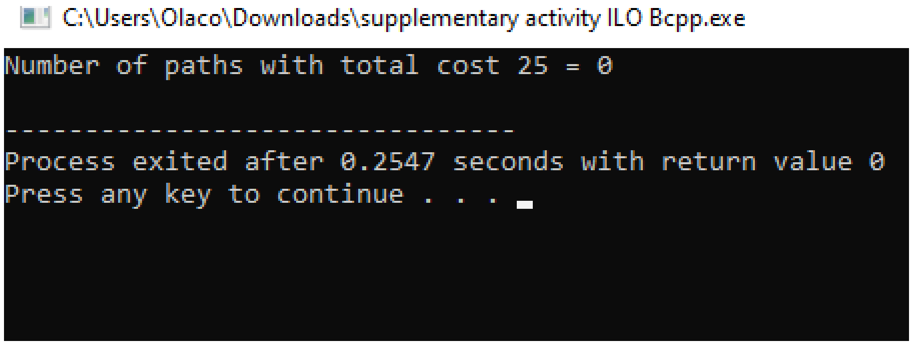
Working C++ Code:

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

int findPaths(const vector<vector<int>>& grid, int r, int c, int cost) {

   if (r < 0 || c < 0 || cost < 0) return 0;
   if (r == 0 && c == 0)
      return (grid[0][0] == cost) ? 1 : 0;


   int fromTop = findPaths(grid, r - 1, c, cost - grid[r][c]);
   int fromLeft = findPaths(grid, r, c - 1, cost - grid[r][c]);

   return fromTop + fromLeft;
}

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

   int target = 25;
   int rows = grid.size() - 1;
   int cols = grid[0].size() - 1;

   int totalPaths = findPaths(grid, rows, cols, target);
```

```
    cout << "Number of paths with total cost " << target << " = " << totalPaths << endl;

    return 0;
}
```

Analysis:
This analyzes how many paths there are from the top-left cell to the bottom-right cell, ensuring the paths equal a specified target cost. It implements recursion to navigate all possible paths by only moving right or down. This approach works for smaller matrices, but larger matrices can be slow due to the recalculating of certain paths. The issue can be solved by using memoization. For this one it finds an answer of 2 paths with a total cost of 25.

Screenshot:



```
C:\Users\Olaco\Downloads\supplementary activity ILO Bcpp.exe

Number of paths with total cost 25 = 0

--------------------------------
Process exited after 0.2547 seconds with return value 0
Press any key to continue . . .
```

C. Conclusion & Lessons Learned

This activity taught me how various algorithms operate and how diverse their approaches are in resolving issues. While some algorithms serve to break the problems into various smaller issues others work to enhance efficiency by storing previously resolved problems. The whole process taught me that with the appropriate reasoning even the most complicated problems can be resolved. In the additional exercise I appreciated what was taught on how recursion can be used to count the various paths to a solution and how it can be optimized with dynamic programming. I believe I grasped most of the concepts even though I may still need to work on a few more coding problems to enhance my ability to identify the appropriate algorithm to use in different scenarios.

D. Assessment Rubric

E. External References

https://www.programiz.com/cpp-programming/recursion
https://www.programiz.com/cpp-programming/recursion
https://www.geeksforgeeks.org/competitive-programming/dynamic-programming/
https://www.programiz.com/dsa/greedy-algorithm