Activity No. 12.1	
Algorithmic Strategies	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: Oct 27, 2025**
Section: CPE21S4	**Date Submitted: Oct. 28, 2025**
Name(s): Crishen Luper S. Pulgado	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s)

ILO A:

Strategy	Algorithm	Analysis
Recursion	**Quick sort algorithm:** quicksort(A, low, high) begin Declare array A[N] to be sorted low = 1st element; high = last element; pivot if (low < high) begin pivot = partition (A,low,high); quicksort(A,low,pivot-1) quicksort(A,pivot+1,high) End end	The quick sort algorithm uses a recursion strategy since each call works on a smaller portion of the array which is the left side or the right side after picking the pivot. The base case for the algorithm is the if (low<high) which the algorithm will stop when the array or subarray has 0 or 1. The work toward base case is when the array gets divided into smaller parts each time. Then the recursive calls is when it calls itself to sort the smaller parts of the array.
Brute Force	**Linear Search:** N -> Boundary of the list Item -> Searching number Data -> Linear array Step 1: I := 0 Step 2: Repeat while I <= n If (item = data[i]) Print “Searching is successful” Exit Else I:= I + 1 Print “Searching is Unsuccessful” Exit	The linear search algorithm uses a brute force since it checks every element in the array until the key element has found. The repeat while I <=n keeps checking the element as long as I is within the array’s boundary.
Backtracking	**Depth First algorithm:** dfs(in v:Vertex) { // Traverses a graph beginning at vertex v // by using depth-first strategy: Iterative Version s.createStack(); // push v into the stack and mark it s.push(v); Mark v as visited; while (!s.isEmpty()) { if (no unvisited vertices are adjacent to the vertex on the top of stack) s.pop(); // backtrack	The depth first algorithm uses a backtracking since it must go back when it reaches a dead end or a node with no unvisited neighbors and tries another path to search the node. The DFS uses a stack as when it reaches a dead end and needs to go back, it uses a pop so that it will back tracks.

	else { Select an unvisited vertex u adjacent to the vertex on the top of the stack; s.push(u); Mark u as visited; } } }	
Greedy	**Selection sort**: emplate <typename T> int Routine_Smallest(T A[], int K, const int arrSize){ int position, j; //Step 1: [initialize] set smallestElem = A[K] T smallestElem = A[K]; //Step 2: [initialize] set POS = K position = K; //Step 3: for J = K+1 to N -1,repeat for(int J=K+1; J < arrSize; J++){ if(A[J] < smallestElem){ smallestElem = A[J]; 54 position = J; } } //Step 4: return POS return position; }	The selection sort uses a greedy algorithm strategy since it greedily picks the smallest remaining element which is the best local option. It only chooses once and never reconsiders the previous elements.
Divide and conquer	**Merge sort:** Declare an array Arr of length N If N=1, Arr is already sorted If N>1, Left = 0, right = N-1 Find middle = (left + right)/2 Call merge_sort(Arr,left,middle) =>sort first half recursively Call merge_sort(Arr,middle+1,right) => sort second half recursively Call merge(Arr, left, middle, right) to merge sorted arrays in above steps. Exit	The merge sort uses the divide and conquers strategy since it divides the array into half which splits it into smaller subproblems. Then, after splitting the array, it sorts the array which conquers it. Lastly, it will recombine the array or merge it to get back to the original size.

Table 12-1

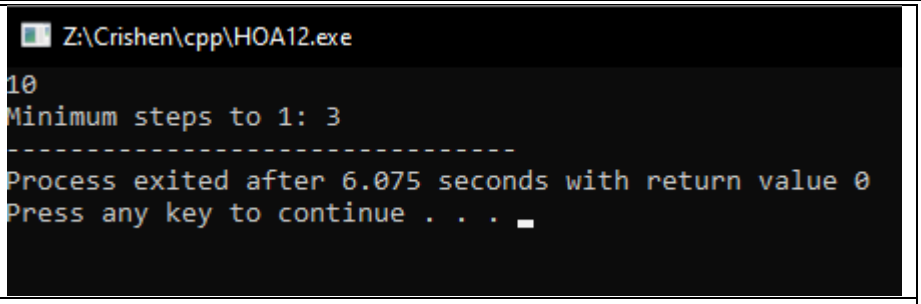
Screen Shot	
Analysis	What happens is that the memo array has an 11 elements, which 0 to 10, and it is all set to -1. So, the first call is getMinSteps(10) but it is equal to 1 so we call the value of 9 and it will begin the recursion until the value is 1. When the value of n is 2, it can get the value 1 since it has already been solved. So, this will continue until it is in the value of 10. If it is in 10, it will check the divisibility. 10 is divisible by 2 and will compute for 1 + getMinsteps(n/2) which is 5. getMinstep(5) is 3 and adds 1 which is 4. So, min(3, 4) is 3; 10 is not divisible by 3 so which is false. Hence, prints the result which is 3 steps since the memo is 3.

Table 12-2

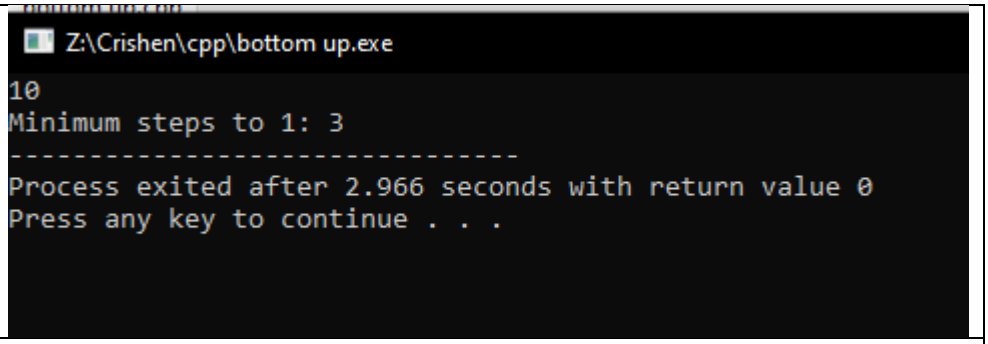
Screen Shot	
Analysis	The bottom-up will solve smaller problems first and use those solved problems to solve the bigger problems. In the code, the trivial case is is dp[1] = 0 which has 0 steps. Now, to get the dp[2], it will go back to the dp[1] since dp[i-1]. As it increases the value that were looking for, it will go back to get the previous solved dp[i] to solve that bigger problem. Hence, it is called bottom-up since we started at 1 and go upward until we it reaches the given value.

Table 12-3

B. Answers to Supplementary Activity

Pseudocode	Function countPaths(matrix, i, j, cost) If cost < 0 then return 0 // path invalid if cost goes below 0 If i == 0 and j == 0 then If cost == matrix[0][0] then return 1 // found one valid path else return 0 If i < 0 or j < 0 then return 0 // out of bounds (invalid cell) return countPaths(matrix, i-1, j, cost - matrix[i][j]) + countPaths(matrix, i, j-1, cost - matrix[i][j]) End Function

Solving Problem by hand

target cost = 25

4	7	1	6
6	7	3	9
3	8	1	2
7	1	7	3

1.) (0,0) + (0,1) + (1,1) + (1,2) + (2,2) + (2,3) + (3,3)
= 27 ×

2.) (0,0) + (0,1) + (0,2) + (1,2) + (2,2) + (2,3) + (3,3)
= 21 ×

3.) (0,0) + (0,1) + (1,1) + (2,2) + (2,3) + (3,3)
= 32 ×

4.) (0,0) + (0,1) + (0,2) + (1,2) + (2,2) + (3,2) + (3,3)
= 26 ×

5.) (0,0) + (1,0) + (2,0) + (3,0) + (3,1) + (3+2) + (3,3)
= 31 ×

6.) (0,0) + (0,1) + (0,2) + (0,3) + (1,3) + (2,3) + (3,3)
= 32 ×

7.) ~~(0,0) + (0,2) + (1,2) + (2,2) + (2,3) + (3,3)~~
(0,0) + (1,0) + (1,1) + (1,2) + (2,2) + (2,3) + (3,3)
= 26 ×

8.) (0,0) + (1,0) + (2,0) + (2,1) + (3,1) + (3,2) + (3,3)
= 32

There are no path with a cost of 25

Code	```cpp #include <iostream> #include <vector> int countPaths(std::vector<std::vector<int>>& costMatrix, int i, int j, int cost) { if (cost < 0) return 0; if (i == 0 && j == 0) return (cost == costMatrix[0][0]) ? 1 : 0; if (i < 0 || j < 0) return 0; return countPaths(costMatrix, i - 1, j, cost - costMatrix[i][j]) + countPaths(costMatrix, i, j - 1, cost - costMatrix[i][j]); } int main() { std::vector<std::vector<int>> matrix = { {4, 7, 1, 6}, {6, 7, 3, 9}, {3, 8, 1, 2}, {7, 1, 7, 3} }; int targetCost = 25; int m = matrix.size(); int n = matrix[0].size(); std::cout << "Number of paths with cost " << targetCost << ": " << countPaths(matrix, m - 1, n - 1, targetCost) << std::endl; return 0; } ```
Analysis	The goal of the problem is finding the path that has the cost of 25. It will start from the (0,0) and the bottom right of the matrix (3,3). It can only move either down or right to reach the number. The recursion works by finding all the paths from (0,0) that have a total sum equal to the given cost. The countPaths function does the job in counting how many valid paths exist. However, in the code, it moves up and left since the recursive function needs a clear base case which is the (0,0) since it's the beginning of the matrix. By starting the recursion from the end and going backward makes the base case natural a simple. Starting from (0,0) would make it complex since we have to stop at (m-1, n-1). In the end, the approach is still the same but it starts at the opposite only. If cost becomes negative it will return 0 since the path already exceed the target total which is 25. When the recursion reaches (0,0), it checks whether the remaining cost equal that cell's value which is 4. If it matches, it would return 1 since it is a valid path. If not, it would return 0. The function also checks if it is out of bound when it gives a negative index, which would return 0. The recursion of moving the cell, it either moves up (i-1, j) and subtracts the current cell's value from the remaining cost or moves to the left and does subtracting again. Each recursive call explores one possible route backward. The results were added together and returns the total number of valid paths to (I,j) with that cost.
Output	``` Number of paths with cost 25: 0 ```

C. Conclusion & Lessons Learned

To conclude, this activity has taught me algorithmic strategies. The algorithmic strategies are the recursion, brute force, backtracking, greedy algorithm, divide and conquer, and dynamic programming. These algorithmic strategies help to solve complex problems more efficient and systematic approach to solve it. The procedure demonstrates how these strategies are different from each other but the they all come up with making the complex problem into subproblems in order to solve the big problem more efficiently. The supplementary uses recursion to explore every possible path and abandons (backtrack) a path when it is invalid path. I think I understand the topic since some of the strategies are already used in the previous activities but I had a hard time implementing the code.

D. Assessment Rubric
E. External References