Hands-on Activity 12.1	
Algorithmic Strategies	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11 - 01 - 25
Section: CPE21S4	**Date Submitted:** 11 - 01 - 25
Name(s): Santiago, David Owen A.	**Instructor:** Engr. Jimlord Quejado

6. Output

ILO A: Demonstrate an understanding of algorithmic strategies (Pre-Lab Review)

Table 12-1 Algorithmic Strategies and Examples

For the pre-lab review, you must fill out the table below. Using algorithms implemented in the previous activities, identify which algorithm satisfies the given algorithmic strategy. For the analysis, you must explicitly provide what part of the algorithm you have identified matches the corresponding strategy.

Strategy	Algorithm	Analysis
Recursion	Binary Search Algorithm	Binary Search can be implemented using recursion where the algorithm calls itself until the middle value matches the target value. The recursion happens every time we have to divide the array to test if arr[mid] matches the target element, otherwise, the element is not in the array. Recursion shortens and simplifies the code compared to the iterative version.
Brute Force	Bubble Sort Algorithm	The bubble sort algorithm uses a brute force strategy because it switches each element with the rest iteratively until it sorts the array. This makes it very inefficient and memory intensive compared to other strategies like binary searching.
Backtracking	Depth-First Searching Techniques	Depth-first searching techniques require backtracking since we have to go to the deepest node to the left first before marking it as “visited” and going back to the parent node to visit the other child nodes. Basically, the search can only continue by going back to previous nodes and visiting other nodes.
Greedy		
Divide and Conquer	Merge Sort and Quicksort	Merge Sort uses a divide and conquer strategy by repeatedly splitting the vector and creating subvectors to store each half. This then sorts the smaller subvectors and is then merged at the end to return the already sorted vector. Similarly, quicksort uses the same

		logic by dividing the array using partitioning then sorting each half. This speeds up the process and simplifies the sorting since we're breaking down the larger array into smaller ones.

Table 12-2 Memoization Implementation

Code:

```
#include <iostream>
#include <algorithm>

int n;
int memo[1000]; // we will initialize the elements to -1 ( -1 means,
not solved it yet )
int getMinSteps( int n )
{
   if ( n == 1 ) return 0; // base case
   if( memo[n] != -1 ) return memo[n]; // we have solved it already :)
   int r = 1 + getMinSteps( n - 1 ); // '-1' step . 'r' will contain
the optimal answer finally
   if( n%2 == 0 ) r = std::min( r , 1 + getMinSteps( n / 2 ) ) ; //
'/2' step
   if( n%3 == 0 ) r = std::min( r , 1 + getMinSteps( n / 3 ) ) ; //
'/3' step
   memo[n] = r ; // save the result. If you forget this step, then it's
the same as plain recursion.
   return r;
}
int main() {
   std::cout << "Input n: ";
   std::cin >> n;
   std::fill(memo, memo + n + 1, -1);
   std::cout << "\nMinimum steps: " << getMinSteps( n);

   return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 01
Input n:7

Minimum steps: 3
Process finished with exit code 0
```

The memoization implementation starts by breaking it down and solving the problem. In the implementation, we begin by defining the base cases where we already solve the problem. Otherwise, we continue recursively calling to solve the problems. Afterwards, the resulting values are stored in the memo array to determine the minimum number of steps taken. This begins by solving the problem first before storing the result.

Table 12-3 Bottom-Up Dynamic Programming Implementation

```
#include <iostream>
#include <algorithm>

int n;
int memo[1000]; // we will initialize the elements to -1 ( -1 means,
not solved it yet )
int getMinSteps(int n) {
   if (n == 1) return 0; // base case
   if (memo[n] != -1) return memo[n]; // we have solved it already :)
   int r = 1 + getMinSteps(n - 1); // '-1' step . 'r' will contain the
optimal answer finally
   if (n % 2 == 0) r = std::min(r, 1 + getMinSteps(n / 2)); // '/2'
step
   if (n % 3 == 0) r = std::min(r, 1 + getMinSteps(n / 3)); // '/3'
step
   memo[n] = r; // save the result. If you forget this step, then it's
same as plain recursion.
   return r;
}

int getMinStepsBottomUp(int n) {
   int dp[n + 1], i;
   dp[1] = 0; // trivial case
   for (i = 2; i <= n; i++) {
       dp[i] = 1 + dp[i - 1];
       if (i % 2 == 0) dp[i] = std::min(dp[i], 1 + dp[i / 2]);
       if (i % 3 == 0) dp[i] = std::min(dp[i], 1 + dp[i / 3]);
   }
   return dp[n];
```

```
}
int main() {
    std::cout << "Input n: ";
    std::cin >> n;
    std::fill(memo, memo + n + 1, -1);
    std::cout << "\nMinimum steps: " << getMinSteps( n);
    std::cout << "\nMinimum steps bottom up: " <<
getMinStepsBottomUp(n);


    return 0;
}
```

Output

Unlike memoization, we begin by iterating through the many using the for loop and we begin at the start until we reach the intended value to solve the problem. Instead of solving n immediately, we start at n = 1 to solve the sub problems until we solve the main problem. It's similar to memoization, but memoization breaks down the problem through recursion. Meanwhile, bottom-up starts at the smaller problems, before solving the bigger main problem.

7. Supplementary Activity

Pseudocode

```
countPaths(i, j, cost, mat, dp):
   int M // number of rows in mat
   int N // number of columns in mat

   IF i ≥ M OR j ≥ N OR cost < mat[i][j]:
      RETURN 0  // Out of bounds or cost too low

   IF i = M - 1 AND j = N - 1:
      RETURN 1 IF cost = mat[i][j] ELSE 0  // Destination check

   IF dp[i][j][cost] ≠ -1:
      RETURN dp[i][j][cost]  // Return cached result

   remaining = cost - mat[i][j]

   // Recursive calls: move right and down
   right = countPaths(i, j + 1, remaining, mat, dp)
   down  = countPaths(i + 1, j, remaining, mat, dp)

   dp[i][j][cost] = right + down  // Store result
   RETURN dp[i][j][cost]


MAIN:
   int mat // 2D matrix of integers
   int targetCost // desired total cost
   int M // number of rows in mat
   int N // number of columns in mat

   // Initialize 3D memoization table with -1
   int vector dp // 3D array of size [M][N][targetCost + 1], filled with -1

   int totalPaths =  countPaths(0, 0, targetCost, mat, dp)
   PRINT "Number of paths with cost", targetCost, ":", totalPaths
```

Manual Counting of Paths:

Path 1: R R R D D D
(0,0) → (0,1) → (0,2) → (0,3) → (1,3) → (2,3) → (3,3)
Values: 4 + 7 + 1 + 6 + 9 + 2 + 3 = 32 **X**

Path 2: R D R D R D
(0,0) → (0,1) → (1,1) → (1,2) → (2,2) → (2,3) → (3,3)

Values: 4 + 7 + 7 + 3 + 1 + 2 + 3 = 27 **X**

Path 3: D R D R D R
(0,0) → (1,0) → (1,1) → (2,1) → (2,2) → (3,2) → (3,3)
Values: 4 + 6 + 7 + 8 + 1 + 7 + 3 = 36 **X**

Path 4: R D D R R D
(0,0) → (0,1) → (1,1) → (2,1) → (2,2) → (2,3) → (3,3)
Values: 4 + 7 + 7 + 8 + 1 + 2 + 3 = 32 **X**

Path 5: R D D R R D
(0,0) → (1,0) → (2,0) → (3,0) → (3,1) → (3,2) → (3,3)
Values: 4 + 6 + 3 + 7 + 1 + 7 + 3 = 31 **X**

Thus, there are no paths that have an exact 25 cost to reach (3, 3) starting from (0, 0).

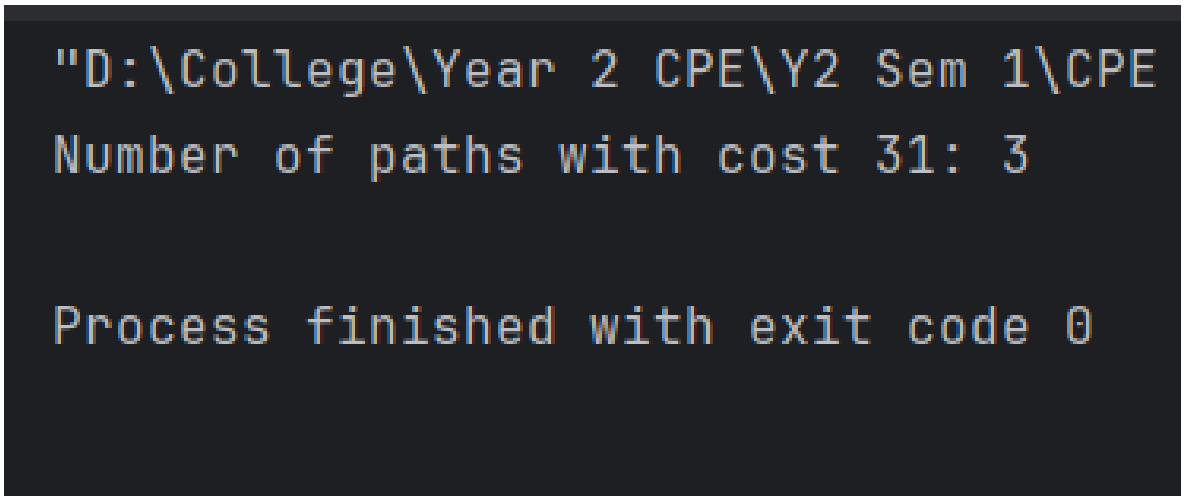
Code:

```cpp
#include <iostream>
#include <vector>

// Recursive function with memoization
int countPaths(int i, int j, int cost, const
std::vector<std::vector<int>>& mat,
std::vector<std::vector<std::vector<int>>>& dp) {
    int M = mat.size();
    int N = mat[0].size();

    // Out of bounds or cost too low
    if (i >= M || j >= N || cost < mat[i][j])
        return 0;

    // Destination reached
    if (i == M - 1 && j == N - 1)
        return (cost == mat[i][j]) ? 1 : 0;

    // Already computed
    if (dp[i][j][cost] != -1)
        return dp[i][j][cost];

    int remaining = cost - mat[i][j];

    // Move right and down
    int right = countPaths(i, j + 1, remaining, mat, dp);
    int down  = countPaths(i + 1, j, remaining, mat, dp);

    // Store result
    dp[i][j][cost] = right + down;
    return dp[i][j][cost];
}
```

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

    int targetCost = 25;
    int M = mat.size();
    int N = mat[0].size();

    // 3D DP table: dp[i][j][cost]
    std::vector<std::vector<std::vector<int>>> dp(M,
std::vector<std::vector<int>>(N, std::vector<int>(targetCost + 1,
-1)));

    int totalPaths = countPaths(0, 0, targetCost, mat, dp);
    std::cout << "Number of paths with cost " << targetCost << ": " <<
totalPaths << std::endl;

    return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE
Number of paths with cost 25: 0


Process finished with exit code 0
```



Analysis

The program implements the DP method by storing the possible paths in a 3D vector table that stores the positions M and N and the cost. Then, the function countPaths take the parameters: M and N, the required cost, the vector matrix to be explored, and the 3D vector to store each path taken. It then goes through each possible path from the starting position at (0, 0) and stores it in the 3D vector. It moves through the vector through recursive calls where the remaining cost is subtracted from the value of the cost of each position visited in the vector. This is done to keep track of the cost. Then, once we reach the destination (3, 3), we return the stored value in the DP table that corresponds to the number of paths that reach the target destination with the given cost.

8. Conclusion

In conclusion, this activity explores the different algorithmic strategies and describes them in terms of effectiveness and reusability. Here, I was able to recall the different activities such as—searching, sorting, traversing, and more. I learned how these activities utilized the different algorithmic strategies like the brute forcing of the bubble sort and the divide and conquer strategy of merge sort. Additionally, I was able to understand the broader and more advanced algorithmic strategies like dynamic programming that break down the problems to find the best solution.

In the procedures, I was able to explore the logic of dynamic programming; both the memoization and bottom-up implementations. I'll admit that I still don't fully understand how it works but I get the gist of it. It basically breaks down the problem, either through recursion or the ground up, and solves the smaller problems until we solve the original problem. We then store the steps we've taken, usually in an array to record the steps taken to come up with the most optimal solution or the least amount of steps.

In the supplementary, I was able to apply the concept of dynamic programming and I was also able to recall two and three-dimensional arrays. While the problem itself involves a 2D array or matrix, the application of dynamic programming involves the use of a 3D vector to dynamically store the results and steps taken. This way, I was able to take note of the i, j, and cost variables as I recursively counted the paths taken with respect to cost. Then, to get the number of paths that match all three variables (we reach (3, 3) at the appropriate cost), we simply get the current location in the 3d vector and return that value. The main backbone of the operation is memoization since we start recursively calling the function to traverse through the matrix.

All in all, I believe this activity is crucial in learning about the nature of algorithms, since we are now dealing with optimized techniques that yield the best results. I was able to compare and contrast all the different algorithmic strategies; ranking them from most effective to least effective, and even learning about new techniques like dynamic programming. Not only that, I was also able to apply these strategies in solving different problems and getting the most optimal solutions.

9. Assessment Rubric