Hands-on Activity 12.1

Algorithmic Strategies

Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/28/2025
Section: CPE21S4	**Date Submitted:** 10/28/2025
Name(s): Cabrera, Gabriel A.	**Instructor:** Engr. Jimlord Quejado

6. Output

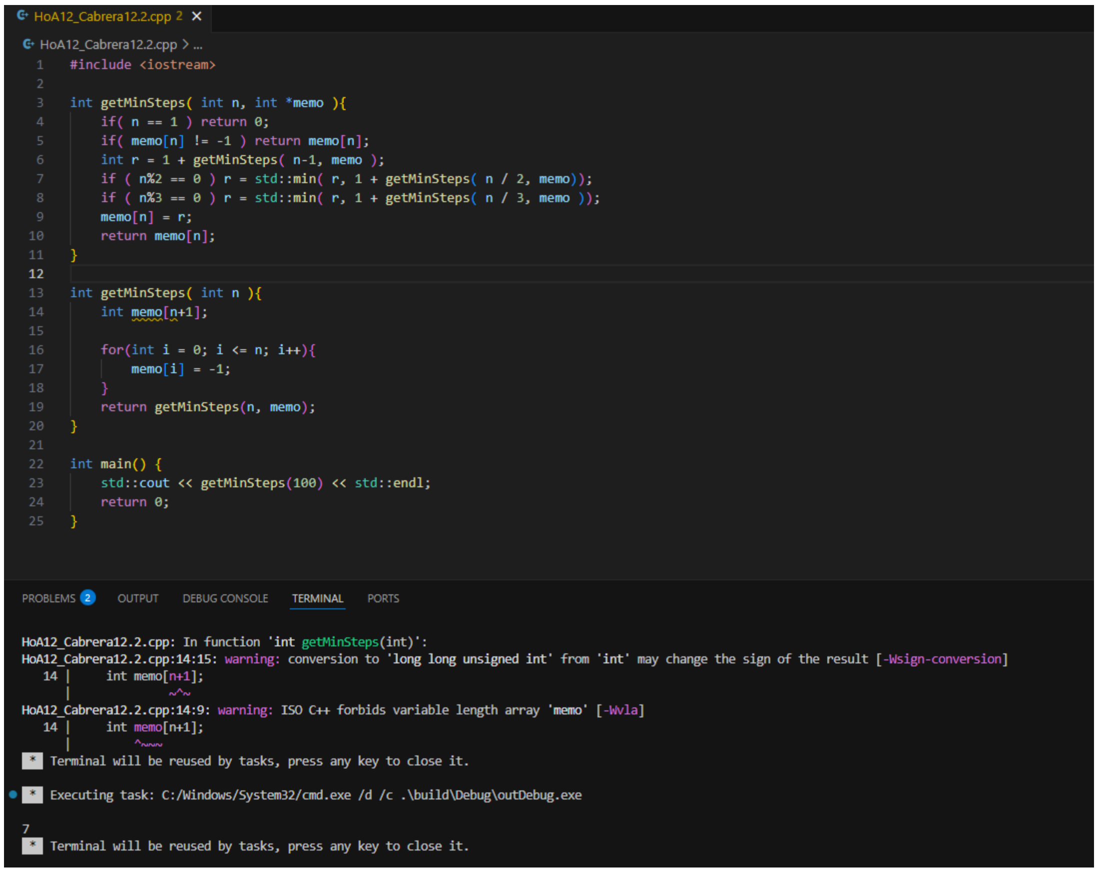
ILO A:

Strategy	Algorithm	Analysis
Recursion	Trees (In order traversal) void inorder(class<int> *p): 1 if p is not equals to NULL 2 inorder(p->child) 3 print p->data 4 inorder(p->next_child)	In Recursion strategy, the function returns the subproblem to itself until it finds the best solution for the given problem.
Brute Force	Sorting Algorithm (Bubble Sort) void bubbleSort(T *arr, const int arrSize): 1 temp 2 for I = 0 to arrSize do 3 for j = i+1 to arrSize do 4 if arr[j] less than arr[i] 5 temp equals to arr[i] 6 arr[i] equals to arr[j] 7 arr[j] equals to temp	In Brute Force algorithm, its solution to the problem is to try every possible solution until it finds the correct solution.
Backtracking	Trees (Pre order traversal) void preorderFind(class<int> *p, int key): 1 if p equals to null then return 2 if element is equal to key 3 print key 4 preorderFind(firstChild, key) 5 preorderFind(nextChild, key)	Backtracking algorithm is looking for a feasible solution for the given problem by checking each node or data if they are part of the solution or else skip/ignore the node or the data.

Greedy	Sorting Algorithm (Selection Sort) void selectionSort(T *arr, const in n) 1 POS, temp, pass = 0; 2 for I = 0 to n do 3 POS equals to routineSmallest(arr, I, n) 4 temp = arr[i] 5 swap A[K] with A[POS] 6 arr[i] = arr[POS] 7 arr[POS] = temp 8 pass++	A greedy algorithm is a problem solving method that selects the best option to solve the current problem while disregarding the next problem.
Divide-and-Conquer	Sorting Algorithm (Merge Sort) void merge(int *subArr, int first, int mid, int last): 1 int temp[last+1], indexLeft = first, indexRight = mid+1 2 int I 3 for I = first; indexLeft <= mid && indexRight <= last; do 4 if subbArr[indexLeft] < subArr[indexRight] 5 temp[i] equals to subArr[indexLeft] 6 indexLeft++ 7 else 8 temp[i] = subArr[indexRight] 9 indexRight++ 10 while indexLeft is less than or equals to mid do 11 temp[i] = subArr[indexLeft] 12 i++; indexLeft++; 13 while indexRight is less than or equals to last do 14 temp[i] = subArr[indexRight] 15 i++ indexRight++ 16 for int j = first to I do 17 print temp[j] 18 subArr[j] = temp[j]	Divide and conquer uses recursion to break down the problem into two or more subproblems of the same or similar type, until they are simple enough to be solved directly.

Table 12-1. Algorithmic Strategies and Analysis

Screenshot



HoA12_Cabrera12.2.cpp 2

HoA12_Cabrera12.2.cpp > ...

```cpp
#include <iostream>

int getMinSteps( int n, int *memo ){
    if( n == 1 ) return 0;
    if( memo[n] != -1 ) return memo[n];
    int r = 1 + getMinSteps( n-1, memo );
    if ( n%2 == 0 ) r = std::min( r, 1 + getMinSteps( n / 2, memo));
    if ( n%3 == 0 ) r = std::min( r, 1 + getMinSteps( n / 3, memo ));
    memo[n] = r;
    return memo[n];
}

int getMinSteps( int n ){
    int memo[n+1];

    for(int i = 0; i <= n; i++){
        memo[i] = -1;
    }
    return getMinSteps(n, memo);
}

int main() {
    std::cout << getMinSteps(100) << std::endl;
    return 0;
}
```

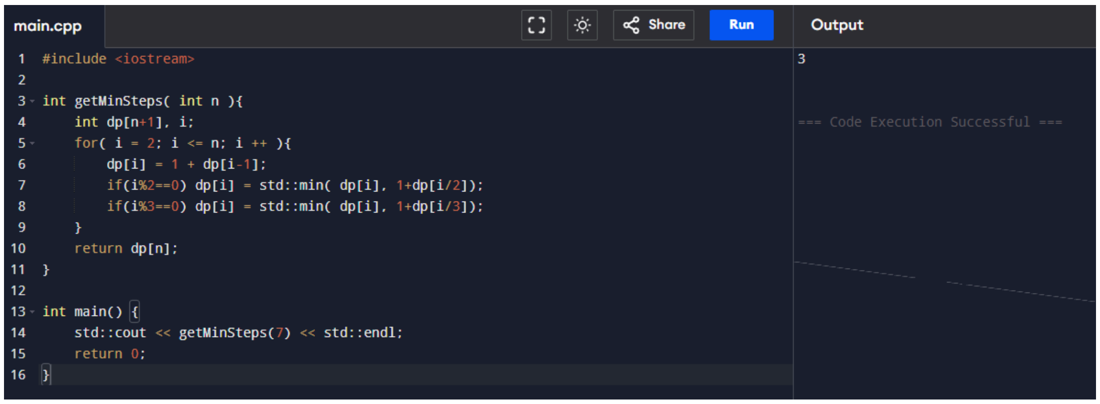
```
HoA12_Cabrera12.2.cpp: In function 'int getMinSteps(int)':
HoA12_Cabrera12.2.cpp:14:15: warning: conversion to 'long long unsigned int' from 'int' may change the sign of the result [-Wsign-conversion]
   14 |     int memo[n+1];
      |              ~^~
HoA12_Cabrera12.2.cpp:14:9: warning: ISO C++ forbids variable length array 'memo' [-Wvla]
   14 |     int memo[n+1];
      |         ^~~~
 *  Terminal will be reused by tasks, press any key to close it.

 *  Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe

7
 *  Terminal will be reused by tasks, press any key to close it.
```

Analysis

The program computes the minimum number of steps to reduce a given integer n to 1 using three operations: subtract 1, divide by 2 if divisible, or divide by 3 if divisible, applying dynamic programming with memoization to optimize performance. The getMinSteps(int n) function initializes a memo array of size n+1 with -1 to indicate uncomputed values, then calls the recursive getMinSteps(int n, int *memo) to start the computation. The recursive function uses memoization: if the result for n is already stored in memo[n], it returns the cached value; otherwise, it calculates the minimum steps by trying subtraction (n-1) and, if applicable, division by 2 or 3, adding 1 to each recursive call.

Table 12-2. Memoization Implementation

Screenshot



```cpp
#include <iostream>

int getMinSteps( int n ){
    int dp[n+1], i;
    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 << getMinSteps(7) << std::endl;
    return 0;
}
```

```
3


=== Code Execution Successful ===
```

Analysis

The program calculates the minimum number of steps to reduce an integer n to 1 using three operations: subtract 1, divide by 2 if divisible, or divide by 3 if divisible, employing a bottom-up dynamic programming approach for efficiency. The getMinSteps(int n) function creates a dp

	array of size n+1, where dp[i] stores the minimum steps needed to reach 1 from i. It initializes dp[1] implicitly through the loop starting at i=2, setting dp[i] initially to 1 + dp[i-1] (subtract 1 operation), then updates it with the minimum if i is divisible by 2 or 3 using 1 + dp[i/2] or 1 + dp[i/3], building solutions from smaller values upward.

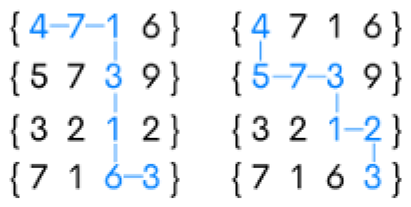
Table 12-3. Bottom-Up DP Implementation

7. Supplementary Activity

ILO B & C:

Pseudocode:

```
{4 7 8 6}
{6 7 3 9}
{3 8 1 2}
{7 1 7 3}
```



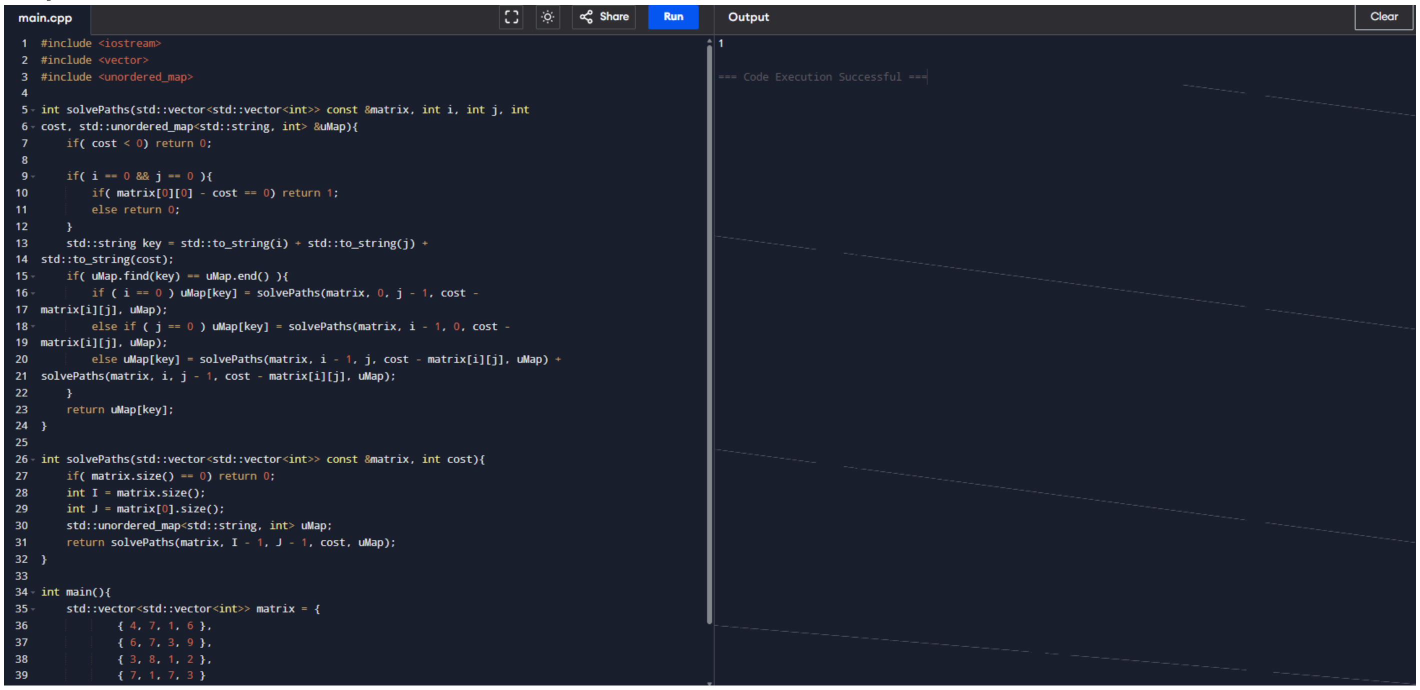
if(i equals to 0) solvePaths(int i, int j, int cost) equals to solvePaths(i, j-1, cost – matrix[i][j])
if(j equals to 0) solvePaths(int i, int j, int cost) equals to solvePaths(i - 1, j, cost – matrix[i][j])
otherwise solvePaths(int i, int j, int cost) equals to solvePaths(i, j-1, cost – matrix[i][j]) + equals to solvePaths(i - 1, j, cost – matrix[i][j])
if(matrix[i][j] equals to cost) solvePaths(0, 0, cost) equals to 1
otherwise solvePaths(0, 0, cost) equals to 0

Working Code:

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

int solvePaths(std::vector<std::vector<int>> const &matrix, int i, int j, int
cost, std::unordered_map<std::string, int> &uMap){
    if( cost < 0) return 0;

    if( i == 0 && j == 0 ){
        if( matrix[0][0] - cost == 0) return 1;
        else return 0;
    }
    std::string key = std::to_string(i) + std::to_string(j) +
std::to_string(cost);
    if( uMap.find(key) == uMap.end() ){
        if ( i == 0 ) uMap[key] = solvePaths(matrix, 0, j - 1, cost -
matrix[i][j], uMap);
        else if ( j == 0 ) uMap[key] = solvePaths(matrix, i - 1, 0, cost -
matrix[i][j], uMap);
```

```cpp
        else uMap[key] = solvePaths(matrix, i - 1, j, cost - matrix[i][j], uMap) +
solvePaths(matrix, i, j - 1, cost - matrix[i][j], uMap);
    }
    return uMap[key];
}

int solvePaths(std::vector<std::vector<int>> const &matrix, int cost){
    if( matrix.size() == 0) return 0;
    int I = matrix.size();
    int J = matrix[0].size();
    std::unordered_map<std::string, int> uMap;
    return solvePaths(matrix, I - 1, J - 1, cost, uMap);
}

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 cost = 21;
    std::cout << solvePaths(matrix, cost);
}
```

Output:



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

int solvePaths(std::vector<std::vector<int>> const &matrix, int i, int j, int
cost, std::unordered_map<std::string, int> &uMap){
    if( cost < 0) return 0;

    if( i == 0 && j == 0 ){
        if( matrix[0][0] - cost == 0) return 1;
        else return 0;
    }
    std::string key = std::to_string(i) + std::to_string(j) +
std::to_string(cost);
    if( uMap.find(key) == uMap.end() ){
        if ( i == 0 ) uMap[key] = solvePaths(matrix, 0, j - 1, cost -
matrix[i][j], uMap);
        else if ( j == 0 ) uMap[key] = solvePaths(matrix, i - 1, 0, cost -
matrix[i][j], uMap);
        else uMap[key] = solvePaths(matrix, i - 1, j, cost - matrix[i][j], uMap) +
solvePaths(matrix, i, j - 1, cost - matrix[i][j], uMap);
    }
    return uMap[key];
}

int solvePaths(std::vector<std::vector<int>> const &matrix, int cost){
    if( matrix.size() == 0) return 0;
    int I = matrix.size();
    int J = matrix[0].size();
    std::unordered_map<std::string, int> uMap;
    return solvePaths(matrix, I - 1, J - 1, cost, uMap);
}

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

```
1

=== Code Execution Successful ===
```

Analysis:

The program counts the number of paths from the bottom-right cell (M-1, N-1) to the top-left cell (0,0) in an M×N matrix, where each cell has a non-negative cost, and only paths summing exactly to a target cost are valid. Movement is restricted to left or up from any cell, meaning from position (i,j), the next step can be to (i-1,j) or (i,j-1). The recursive solvePaths function uses top-down dynamic programming with memoization via an unordered_map to store results for each state defined by current position (i,j) and remaining cost. At each cell, it subtracts the current cell's cost from the remaining cost before recursing to the left or above, avoiding negative costs with an early return of 0. The base case at (0,0) checks if the remaining cost equals the starting cell's value; if so, one valid path exists. A unique key combining i, j, and cost ensures each subproblem is computed only once, preventing exponential runtime. For the given 4×4 matrix and target cost of 21, the program outputs the total number of valid paths satisfying the exact cost constraint.

8. Conclusion

I now clearly understand algorithmic strategies such as the strategy pattern, dynamic programming, and memoization, enabling me to translate complex concepts into clean, working code. I have also learned to implement various algorithmic strategies in C++, including top-down and bottom-up approaches, which has strengthened my skills in building modular and efficient programs.

9. Assessment Rubric