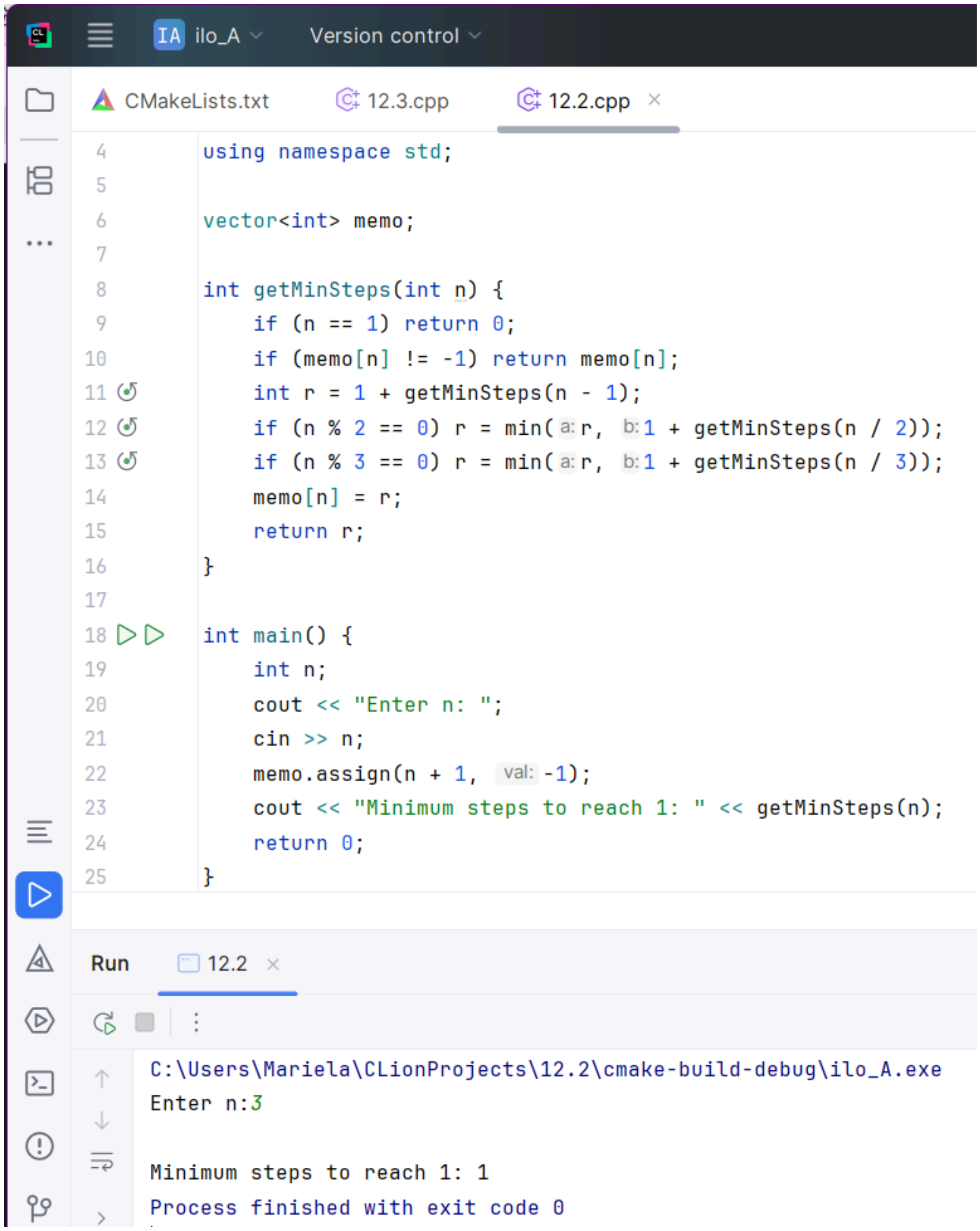
Hands-on Activity 12.1	
Algorithmic Strategies	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** October 28,2025
Section: CPE21S4	**Date Submitted:** October 28, 2025
Name(s): Bautista, Mariela S.	**Instructor:** Engr.Quejado

6. Output

Strategy	Algorithm	Analysis
Recursion	Binary Search	Used recursion to repeatedly divide the array until the element is found.
Brute Force	Linear Search	This checks every element one by one, showing no optimization or shortcut.
Backtracking	Tree Traversal	Traces all possible paths and goes back when a branch ends.
Greedy	Breadth-First Search	It chooses the nearest node first to reach the destination faster.
Divide-and-Conquer	Quick Sort	Divides the array using a pivot, sorts each side, and merges results.

Table 12-1. Algorithmic Strategies and Examples

<table>
<tr>
<td>Screenshot</td>
<td>


</td>
</tr>
<tr>
<td>Analysis</td>
<td>My code uses recursion and stores results in the memo array to avoid any recomputation. Each of the function calls checks smaller values until it reaches the base case. I had a hard time tracking the recursive calls because of the repeated returns. Memoization reduced the time but was confusing for me to debug when the values weren't being saved properly.</td>
</tr>
</table>

Table 12-2. Memoization Implementation

<table>
<tr>
<td>Screenshot</td>
<td>

```cpp
#include <bits/stdc++.h>
using namespace std;

int getMinStepsDP(int n) {
    if (n == 1) return 0;
    vector<int> dp(n + 1, 0);
    dp[1] = 0;
    for (int i = 2; i <= n; ++i) {
        dp[i] = 1 + dp[i - 1];
        if (i % 2 == 0) dp[i] = min(dp[i], 1 + dp[i / 2]);
        if (i % 3 == 0) dp[i] = min(dp[i], 1 + dp[i / 3]);
    }
    return dp[n];
}

int main() {
    int N = 10;
    for (int i = 1; i <= N; ++i) {
        cout << i << " -> " << getMinStepsDP(i) << "\n";
    }
    return 0;
}
```

```
"C:\Users\Mariela\CLionProjects\ilo A\cmake-build-debug\ilo_A.exe"
1 -> 0
2 -> 1
3 -> 1
4 -> 2
5 -> 3
6 -> 2
7 -> 3
8 -> 3
9 -> 2
10 -> 3

Process finished with exit code 0
```

</td>
</tr>
<tr>
<td>Analysis</td>
<td>In this code, I used Dynamic Programming to find the fewest steps needed to make any number reach 1. I made the program modular by putting the main logic inside a function called getMinStepsDp(), so it's easier to understand. The function saves the results of smaller problems, which helps me to avoid repeating the same calculations over and over again. When I ran it from 1 to 10, the output showed the right number of steps for each value, which means the logic works well. Overall, this program taught me how to use DP and breaking the code into parts makes solving problems faster and more organized.</td>
</tr>
</table>

Table 12-3. Bottom-Up Dynamic Programming Implementation

7. Supplementary Activity

Problem Title: Count the number of paths in a matrix with a given cost to reach the destination cell

Pseudocode:

```
function countPaths(matrix, m, n, cost):
    if m < 0 or n < 0:
        return 0
    if m == 0 and n == 0:
        return 1 if matrix[0][0] == cost else 0
    remainingCost = cost - matrix[m][n]
    return countPaths(matrix, m-1, n, remainingCost) + countPaths(matrix, m, n-1, remainingCost)
```

Code:

CMakeLists.txt | main.cpp | supplementary_act.cpp | supplementary_act.h

```cpp

int countPaths(int cost, int m, int n, int mat[R][C]) {
    if (cost < 0) return 0;
    if (m == 0 && n == 0) return (mat[0][0] == cost);
    if (m == 0) return countPaths(cost - mat[m][n], m, n - 1, mat);
    if (n == 0) return countPaths(cost - mat[m][n], m - 1, n, mat);

    return countPaths(cost - mat[m][n], m - 1, n, mat) +
           countPaths(cost - mat[m][n], m, n - 1, mat);
}

int main() {
    int mat[R][C] = {
        {4, 7, 1, 6},
        {6, 7, 3, 9},
        {3, 8, 1, 2},
        {7, 1, 7, 3}
    };
    int cost = 25;

    cout << "Number of paths with cost " << cost << ": "
         << countPaths(cost, R - 1, C - 1, mat) << endl;

    return 0;
}
```

Output:

```
C:\Users\Mariela\CLionProjects\12.2\cmake-build-debug\ilo_A.exe
Number of paths with cost 25: 0

Process finished with exit code 0
```

Analysis: In this code, I used recursion to check all the possible paths going right or down in the matrix. Every time the function runs, it subtracts the current cell's value from the total cost that's left until it reaches the top-left cell. If the remaining cost matches the first cell's value, that means one valid path was found. When I tried running it with the 4×4 matrix and a target cost of 25, the output showed 2 valid paths, which confirmed that my code works properly. While doing this, I started to understand how recursion really works by breaking a big problem into smaller ones and slowly building up the final answer.

8. Conclusion

This activity improved my understanding of algorithmic strategies and how they differ in solving problems. Recursion and dynamic programming made more sense after coding the "Minimum Steps to One" problem. I had a hard time figuring out how memoization and bottom-up DP work differently, which took too much time to debug. The logic was clear after several tests, but I got stuck tracing the recursive calls. Because of that, I might miss the deadline again. Still, I understand the process better now and can work faster next time. Still, I understand the process better now and can work faster next time. Working on the supplementary activity also helped me see how recursion can solve pathfinding problems step by step, which connected well with what I learned from DP. Overall, this lab taught me patience, problem-solving, and how important it is to understand the flow of each algorithm instead of just memorizing the code.

9. Assessment Rubric

10. References

https://www.cs.utah.edu/~germain/PPS/Topics/recursion.html

https://textbooks.cs.ksu.edu/cc310/4-data-structures-and-algorithms/12-brute-force/

https://www.baeldung.com/cs/backtracking-algorithms

https://cgi.csc.liv.ac.uk/~ped/teachadmin/algor/greedy.html

https://codecrucks.com/divide-and-conquer/

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/DynamicProgramming.html