ACTIVITY NO. 12

ALGORITHMIC STRATEGIES	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/26/25**
Section: CPE21S4	**Date Submitted: 10/26/25**
Name: Cruz, Axl Waltz E.	**Instructor:** Engr. Jimlord Quejado

1. Objective(s)

Create C++ Code to implement different algorithmic strategies

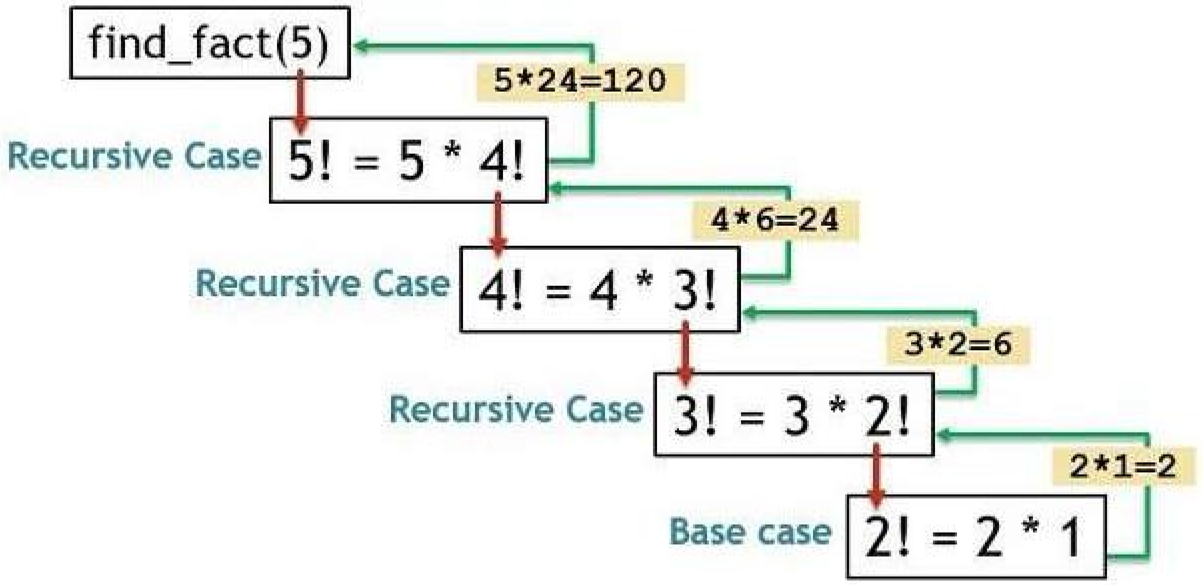
2. Intended Learning Outcomes (ILOs)

After this activity, the student should be able to:

- Demonstrate an understanding of algorithmic strategies
- Create C++ code to implement various algorithmic strategies
- Create C++ code to solve problems using learned algorithmic strategies.

3. Discussion

<u>Part A: Recursion</u>



Source: Edureka

Recursion means "defining a problem in terms of itself". This can be a very powerful tool in writing algorithms. Recursion comes directly from Mathematics, where there are many examples of expressions written in terms of themselves. For example, the Fibonacci sequence is defined as: $F(i) = F(i-1) + F(i-2)$ (back in module 1)

Parts of a Recursive Algorithm
All recursive algorithms must have the following:

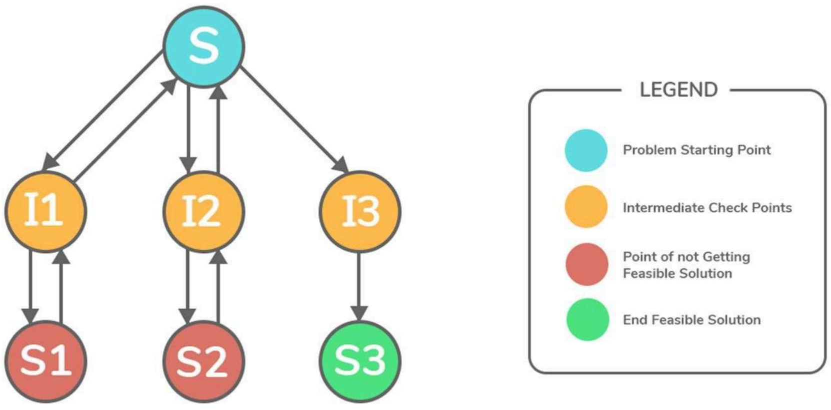
- Base Case (i.e., when to stop)
- Work toward Base Case

- Recursive Call (i.e., call ourselves)

The "work toward base case" is where we make the problem simpler (e.g., divide list into two parts, each smaller than the original). The recursive call is where we use the same algorithm to solve a simpler version of the problem. The base case is the solution to the "simplest" possible problem (For example, the base case in the problem 'find the largest number in a list' would be if the list had only one number... and if there is only one number, it is the largest).

Part B: Brute force
Simply put, a **brute force** algorithm will try all possible solutions to the problem, only stopping when it finds one that is the actual solution. A great example of a brute force algorithm in action is plugging in a USB cable. Many times, we will try one way, and if that doesn't work, flip it over and try the other. Likewise, if we have many keys but are unsure which one fits in a particular lock, we can just try each key until one works. That's the essence of the brute force approach to algorithmic design.

Part C: Backtracking


S
I1
I2
I3
S1
S2
S3
LEGEND
Problem Starting Point
Intermediate Check Points
Point of not Getting Feasible Solution
End Feasible Solution

Source: InterviewBit.com

Backtracking is an algorithmic technique where the goal is to get all solutions to a problem using the brute force approach. It consists of building a set of all the solutions incrementally. Since a problem would have constraints, the solutions that fail to satisfy them will be removed. It uses recursive calling to find a solution set by building a solution step by step, increasing levels with time. In order to find these solutions, a search tree named state-space tree is used. In a state-space tree, each branch is a variable, and each level represents a solution.

Part D: Greedy Algorithm
Another approach that is used to design algorithms for solving optimization problems. However:

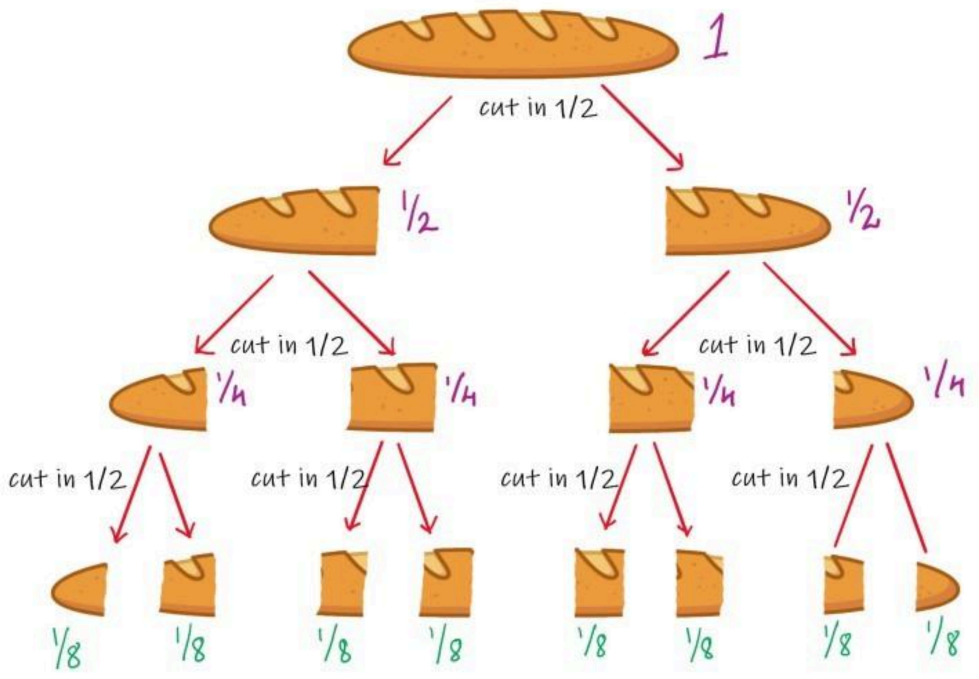
- Greedy algorithms do not always yield a genuinely optimal solution. In such cases the greedy method is frequently the basis of a heuristic approach.
- Even for problems which can be solved exactly by a greedy algorithm, establishing the correctness of the method may be a non-trivial process.

In order to give a precise description of the greedy paradigm we must first consider a more detailed definition of the environment in which typical optimisation problems occur. Thus in an optimisation problem, one will have, in the context of greedy algorithms, the following:

- A collection (set, list, etc) of candidates, e.g. nodes, edges in a graph, etc.
- A set of candidates which have already been `used'.
- A predicate (solution) to test whether a given set of candidates give a solution (not necessarily optimal).
- A predicate (feasible) to test if a set of candidates can be extended to a (not necessarily optimal) solution.
- A selection function (select) which chooses some candidate which h as not yet been used.
- An objective function which assigns a value to a solution.

In other words: An optimisation problem involves finding a subset, S, from a collection of candidates, C; the subset, S, must satisfy some specified criteria, i.e. be a solution and be such that the objective function is optimised by S.

Part E: Divide and Conquer



Source: studyalgorithms.com

Divide and conquer algorithm operates in three stages:

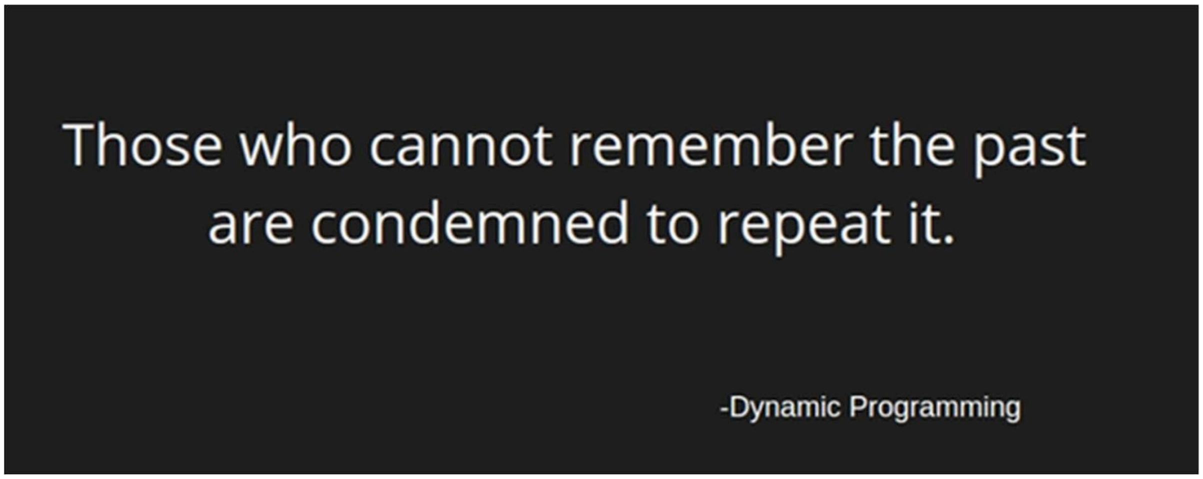
- Divide: Divide the problem recursively into smaller subproblems.
- Solve: Subproblems are solved independently.
- Combine: Combine subproblem solutions to deduce the answer to the original large problem.

Because subproblems are identical to the main problem but have smaller parameters, they can be readily solved using recursion. When a subproblem is reduced to its lowest feasible size, it is solved, and the results are recursively integrated to produce a solution to the original larger problem.

Divide and conquer is a top-down, multi-branched recursive method. Each branch represents a subproblem and calls itself with a smaller argument. Understanding and developing divide and conquer algorithms requires expertise and sound reasoning.

The divide-and-conquer approach is depicted graphically in following figure. Subproblems may not be exactly n/2 in size.

Part F: Dynamic Programming
Dynamic programming is a technique for solving problems, whose solution can be expressed recursively in terms of solutions of overlapping sub-problems.


Those who cannot remember the past are condemned to repeat it.

-Dynamic Programming

Dynamic programming (usually referred to as DP) is a very powerful technique to solve a particular class of problems. It demands very elegant formulation of the approach and simple thinking, and the coding part is very easy. The idea is very simple, If you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again.. shortly 'Remember your Past' :) . If the given problem can be broken up in to smaller sub-problems and these smaller subproblems are in turn divided in to still- smaller ones, and in this process, if you observe some over-lapping subproblems, then its a big hint for DP. Also, the optimal solutions to the subproblems contribute to the optimal solution of the given problem (referred to as the Optimal Substructure Property).

Dynamic Programming Approaches:

1. Top-Down approach – Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as **Memoization**.
2. Bottom-Up approach – Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as **Dynamic Programming**.

4. Materials and Equipment

Personal Computer with C++ IDE Recommended IDE: ● CLion (must use TIP email to download)

- DevC++ (use the embarcadero fork or configure to C++17)

5. Procedure

Pre-Lab Tip:
When you are given a problem to solve using dynamic programming, the trick is “Not to over-think, because the solution is always hidden in the problem”. Most beginners make this mistake of thinking too much, only because they thought the problem was difficult and only a complex solution would solve it. But in most cases the solution turns out to be simple.

Before you start coding, formulate the problem (not the solution) mathematically on a piece of paper. Because a problem well formulated is a problem half solved.

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

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		
Brute Force		
Backtracking		
Greedy		
Divide-and-Conquer		

This table must be included in your section 6 as table 12-1.

The Problem: MINIMUM STEPS TO ONE
On a positive integer, you can perform any one of the following 3 steps.

1.) Subtract 1 from it. (n = n - 1)
2.) If its divisible by 2, divide by 2. (If n % 2 == 0, then n = n / 2)
3.) If its divisible by 3, divide by 3. (If n % 3 == 0, then n = n / 3).

Now the question is, given a positive integer n, find the minimum number of steps that takes n to

1. Examples:

1.) For n = 1, output: 0
2.) For n = 4, output: 2 (4 /2 = 2 /2 = 1)
3.) For n = 7, output: 3 (7 -1 = 6 /3 = 2 /2 = 1)

One can think of greedily choosing the step, which makes n as low as possible and conitnue the same, till it reaches 1. If you observe carefully, the greedy strategy doesn't work here.

Eg: Given n = 10 , Greedy --> 10 /2 = 5 -1 = 4 /2 = 2 /2 = 1 (

4 steps). But the optimal way is --> 10 -1 = 9 /3 = 3 /3 = 1 (3

steps).

So, we need to try out all possible steps we can make for each possible value of n we encounter and choose the minimum of these possibilities.

It all starts with recursion.

$$F(n) = 1 + min\{ F(n-1) , F(n/2) , F(n/3) \} \; if \; (n > 1) , else \; 0 \; (i.e., F(1) = 0) .$$

Now that we have our recurrence equation, we can right way start coding the recursion. Wait. Does it have over-lapping subproblems? YES. Is the optimal solution to a given input depends on the optimal solution of its subproblems? Yes... Bingo! it is dynamic programming. So, we just store the solutions to the subproblems we solve and use them later, as in memorization, or we start from bottom and move up till the given n, as in dynamic programming. As it’s the very first problem we are looking at here, let’s see both the codes:

Memoization

```
int memo[n+1]; // 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 = min( r , 1 + getMinSteps( n / 2 ) ) ; // '/2' step
    if( n%3 == 0 ) r = min( r , 1 + getMinSteps( n / 3 ) ) ; // '/3' step
    memo[n] = r ; // save the result. If you forget this step, then its same as plain recursion.
    return r;
}
```

Run this code and show the output in table 12-2.

Bottom-Up DP

```
int getMinSteps ( 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] = min( dp[i] , 1+ dp[i/2] );
        if(i%3==0) dp[i] = min( dp[i] , 1+ dp[i/3] );
    }
    return dp[n];
}
```

6. Output

Table 12-1. Algorithmic Strategies and Examples

Strategy	**Algorithm**	**Analysis**
Recursion	Divides the task into smaller subproblems and repeatedly calls itself to find the answer.	Handles complex problems by solving simpler versions and storing previous results to avoid redundant calculations.
Brute Force	Examine every possible option until the correct one is found, regardless of efficiency.	Simple but time consuming to test all possibilities just like trying every key on a keyring until success.
Backtracking	Builds potential solutions incrementally and reverses steps when a path proves invalid.	Uses recursion to explore possibilities; when a choice doesn't work, it backtracks to try another route.
Greedy	Always selects the immediate best option at each stage.	Fast and straightforward but may not always reach the optimal overall result.
Divide-and-Conquer	Splits a large problem into smaller, manageable subproblems, solves each one, and merges their solutions.	Efficiently handles big problems by combining smaller solutions, common in sorting and searching algorithms.

Table 12-2. Memoization Implementation

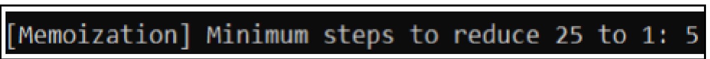
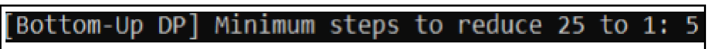
Screenshot	 `[Memoization] Minimum steps to reduce 25 to 1: 5`
Analysis	In the memoization approach, the problem is solved recursively, with results of subproblems stored in a memo[] array to avoid repeating the same calculations. This top-down technique effectively reduces redundant work, resulting in a time complexity of O(n). Despite this efficiency, the recursive nature of the method can still introduce slight delays due to repeated function calls.

Table 12-3. Bottom-Up Dynamic Programming Implementation

Screenshot	 `[Bottom-Up DP] Minimum steps to reduce 25 to 1: 5`
Analysis	The bottom-up DP method starts from the base case and builds up solutions for all values up to *n*, determining the minimum steps for each sequentially. Since it relies on iteration instead of recursion, it tends to run faster and use less memory. This approach also achieves O(n) time complexity and requires O(n) space.

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

Both the approaches are fine. But one should also take care of the lot of over head involved in the function calls in Memoization, which may give StackOverFlow error or TLE rarely.

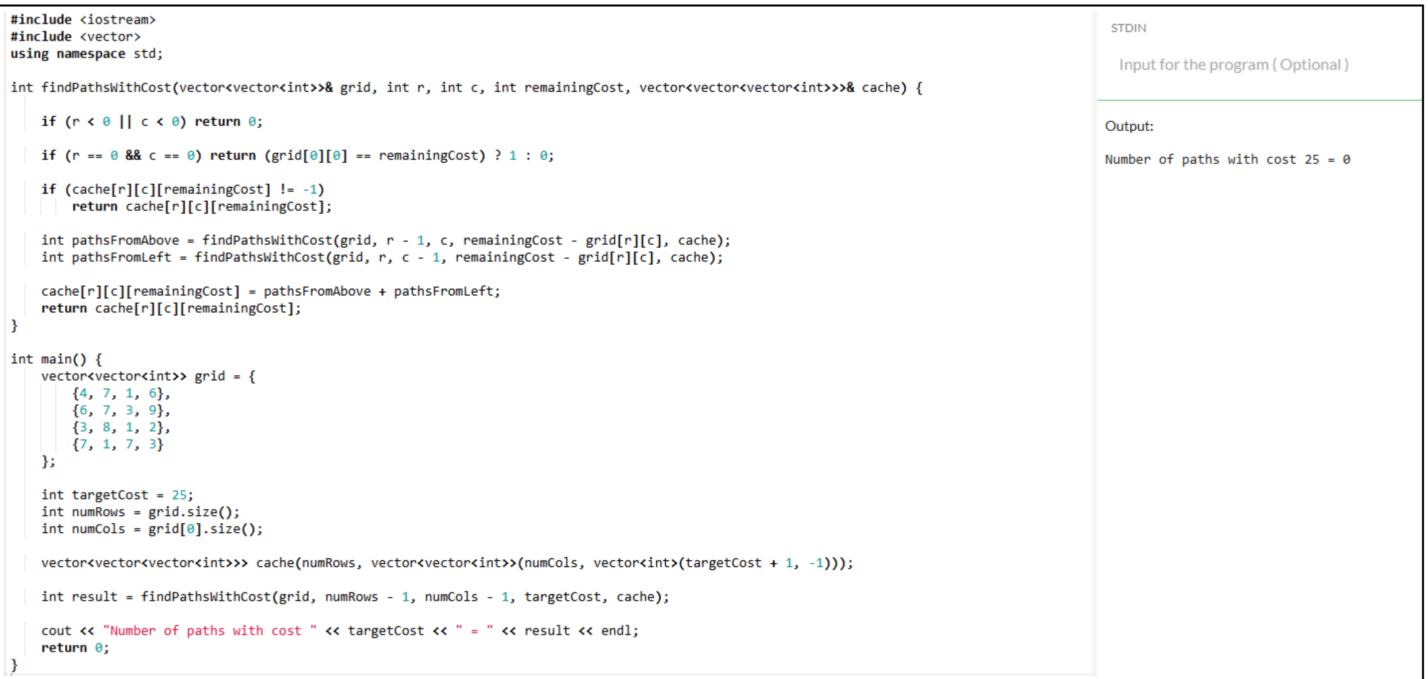
7. Supplementary Activity

ILO B: Create C++ code to implement various algorithmic strategies & ILO C: Create C++ code to solve problems using learned algorithmic strategies

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

Description:
Given an M × N integer matrix where each cell has a non-negative cost associated with it, count the number of paths to reach the last cell (M-1, N-1) of the matrix from its first cell (0, 0) such that the path has given cost. We can only move one unit right or one unit down from any cell, i.e., from cell (i, j), we can move to (i, j+1) or (i+1, j).

For example, consider the following 4 × 4 matrix where cost is 25. Two paths are having a cost of 25. A sample path has been given when the cost is 25.



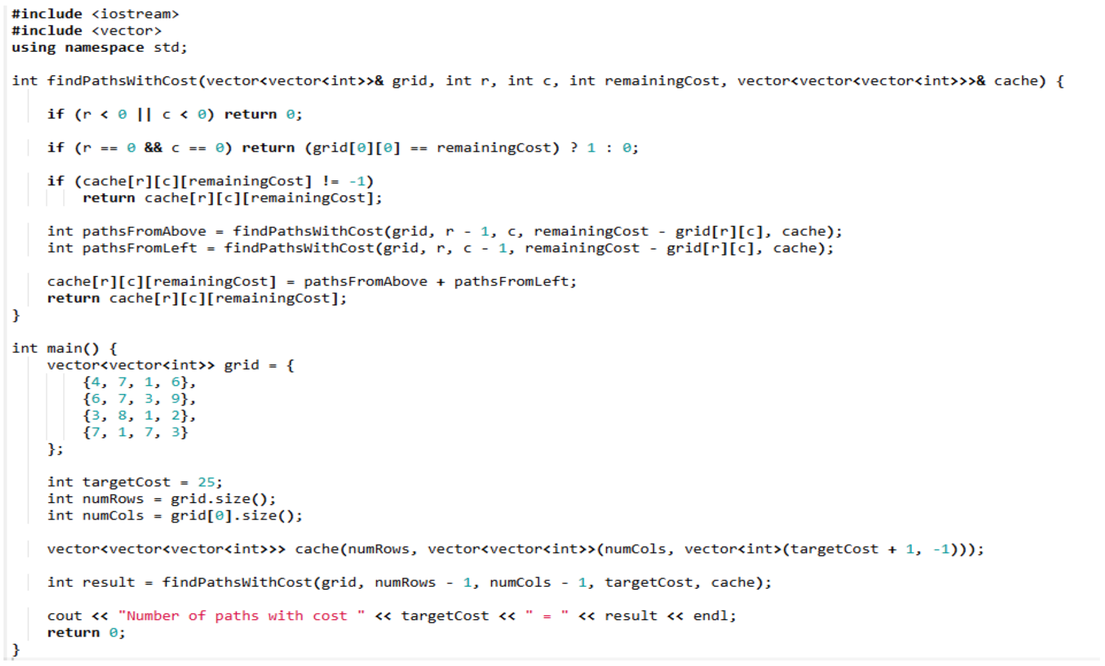
```cpp
#include <iostream>
#include <vector>
using namespace std;

int findPathsWithCost(vector<vector<int>>& grid, int r, int c, int remainingCost, vector<vector<vector<int>>>& cache) {

    if (r < 0 || c < 0) return 0;

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

    if (cache[r][c][remainingCost] != -1)
        return cache[r][c][remainingCost];

    int pathsFromAbove = findPathsWithCost(grid, r - 1, c, remainingCost - grid[r][c], cache);
    int pathsFromLeft = findPathsWithCost(grid, r, c - 1, remainingCost - grid[r][c], cache);

    cache[r][c][remainingCost] = pathsFromAbove + pathsFromLeft;
    return cache[r][c][remainingCost];
}

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

    int targetCost = 25;
    int numRows = grid.size();
    int numCols = grid[0].size();

    vector<vector<vector<int>>> cache(numRows, vector<vector<int>>(numCols, vector<int>(targetCost + 1, -1)));

    int result = findPathsWithCost(grid, numRows - 1, numCols - 1, targetCost, cache);

    cout << "Number of paths with cost " << targetCost << " = " << result << endl;
    return 0;
}
```

STDIN

Input for the program (Optional)

Output:

```
Number of paths with cost 25 = 0
```



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

int findPathsWithCost(vector<vector<int>>& grid, int r, int c, int remainingCost, vector<vector<vector<int>>>& cache) {

    if (r < 0 || c < 0) return 0;

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

    if (cache[r][c][remainingCost] != -1)
        return cache[r][c][remainingCost];

    int pathsFromAbove = findPathsWithCost(grid, r - 1, c, remainingCost - grid[r][c], cache);
    int pathsFromLeft = findPathsWithCost(grid, r, c - 1, remainingCost - grid[r][c], cache);

    cache[r][c][remainingCost] = pathsFromAbove + pathsFromLeft;
    return cache[r][c][remainingCost];
}

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

    int targetCost = 25;
    int numRows = grid.size();
    int numCols = grid[0].size();

    vector<vector<vector<int>>> cache(numRows, vector<vector<int>>(numCols, vector<int>(targetCost + 1, -1)));

    int result = findPathsWithCost(grid, numRows - 1, numCols - 1, targetCost, cache);

    cout << "Number of paths with cost " << targetCost << " = " << result << endl;
    return 0;
}
```

Output:

```
Number of paths with cost 25 = 0
```

The idea is to use recursion. The problem has optimal substructure. That means the problem can be broken down into smaller, simple "subproblems", which can further be divided into yet simpler, smaller subproblems until the solution becomes trivial.

Deliverables:

- Pseudocode (+ Solving Problem by Hand)
- Working C++ Code
- Analysis of working code
- Screenshots of demonstration

8. Conclusion

Provide the following:

- Summary of lessons learned
    - I learned how dynamic programming improves efficiency through memoization and bottom-up methods. Storing previous results helps avoid repeated work. I also applied these ideas to a pathfinding problem with cost limits.
- Analysis of the procedure
    - The activity showed how recursion and DP solve problems faster by reusing results. The bottom-up method proved more efficient than recursion. Both techniques simplified complex tasks into smaller steps.
- Analysis of the supplementary activity
    - The pathfinding task showed how DP works with multidimensional data. Using caching made finding cost-based paths easier and faster. It strengthened my understanding of practical DP use.
- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?
    - I think I did well in applying both DP approaches. I still need to improve on explaining my code clearly. Overall, the activity improved my problem-solving skills.

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