**بسم الله الرحمن الرحيم**

Dynamic Programming Project

Maximum Board Value

Contents

[Introduction: 3](#_Toc157370834)

[Abstract: 3](#_Toc157370835)

[Part1: Divide & Conquer: 4](#_Toc157370836)

[Part2: Dynamic Programming: 11](#_Toc157370837)

[Conclusion: 15](#_Toc157370838)

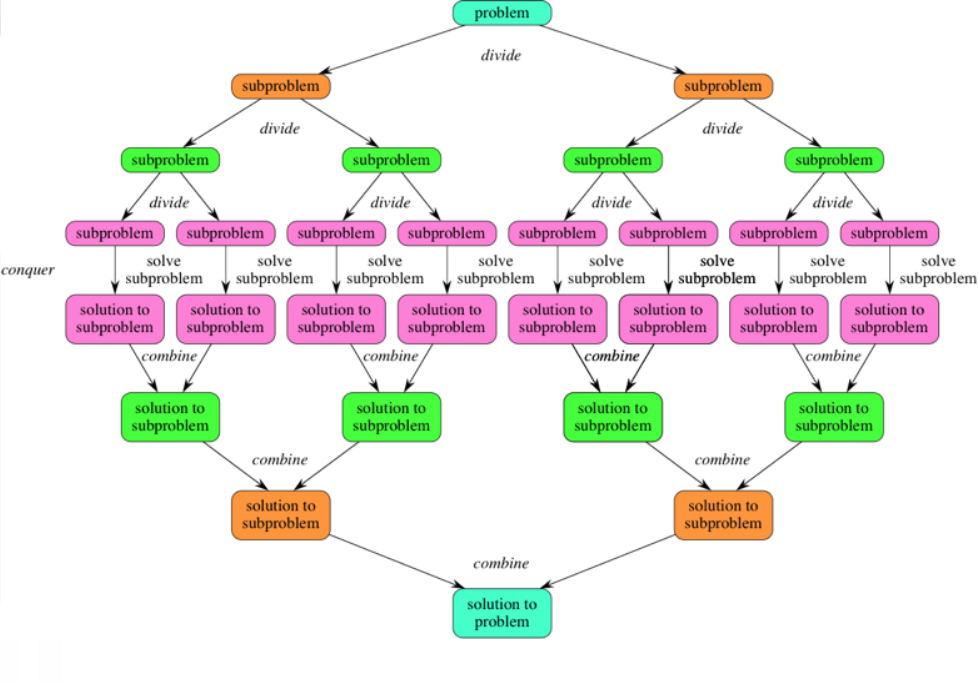
# Introduction:

This project's objective is to follow a set of movement guidelines and maximize the total value on a NxN board. The player's goal is to collect cell values by only moving down or to the right as they go across the board. The challenging element is figuring out the optimal path that yields the highest value. Divide and conquer and dynamic programming are two alternative approaches to solve this problem that are examined in this study.

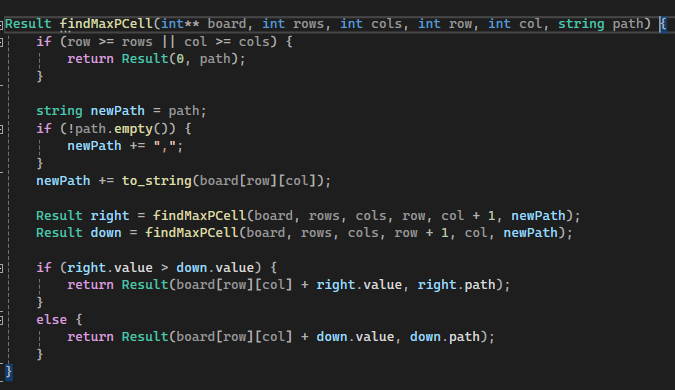
# Abstract:

In this study, two problem-solving strategies—divide and conquer and dynamic programming—are thoroughly examined and their application to a board game scenario is discussed. The goal is to find the largest value that can be obtained by a player by moving solely down or to the right on a NxN board, starting from any cell. The implementation of both strategies is described in depth in the study, together with code samples, explanations, and illustrations. A comparative study of the time and space difficulties is included with the results for sample boards. Each approach's advantages and disadvantages are examined, providing insight into how well-suited each is for certain problem scenarios. Key conclusions and insights into the advantages of using these strategies for effective problem-solving are provided in the report's conclusion.

# Part1: Divide & Conquer:



**1- Define the value returned by the function f which we want to optimize.**



The object of type Result that the function findMaxPCell returns has two fields in it:

worth The greatest sum of the values along the path, which begins in the current cell and moves either down or right until going off the edges of the board, is stored in this field.

path: The values of the cells that make up the path that results in the highest sum are stored in this field.

**2- Define the parameters which f depends on.**

**board:** The game board is represented as a 2D array of integers that is referenced by this parameter. All of the board's cell values are contained in it.

**row:** This option indicates the cell's current row index as it moves across the board.

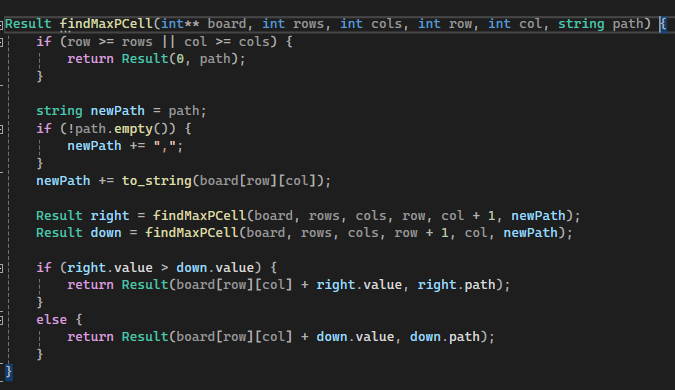
**col:** This parameter indicates the cell's current column index as it moves across the board.

**path:** The path from the initial cell to the current cell is represented by a string in this parameter. The values of every cell the path visits are concatenated.

**The recursive function findMaxPathFromCell needs these parameters in order to move around the board, calculate the maximum value that may be accumulated along various paths, and create the appropriate path string.**

The coordinates of the current cell (row, col) on the board determine the function f.

**4- Write the recursive (divide and conquer) code to solve the question.**



The object of type Result that the function findMaxPCell returns has two fields in it:

worth The greatest sum of the values along the path, which begins in the current cell and moves either down or right, is stored in this field.

path: The values of the cells that make up the path that results in the highest sum are stored in this field.

Base Case: If the current cell position (row, col) is outside of the board's boundaries, return the accumulated path and a Result with a value of 0.

Recurring Calls:

Make a recursive call for the right neighbor cell (row, col + 1).

Make a recursive call for the downward neighbor cell (row + 1, col).

Choose the Maximum Value Path:

Analyze the outcomes of the recursive calls to the right and down.

Choose the path that produces the most value.

Build Outcome:

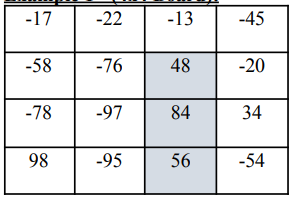
To the value of the current cell, add the value of the chosen path.

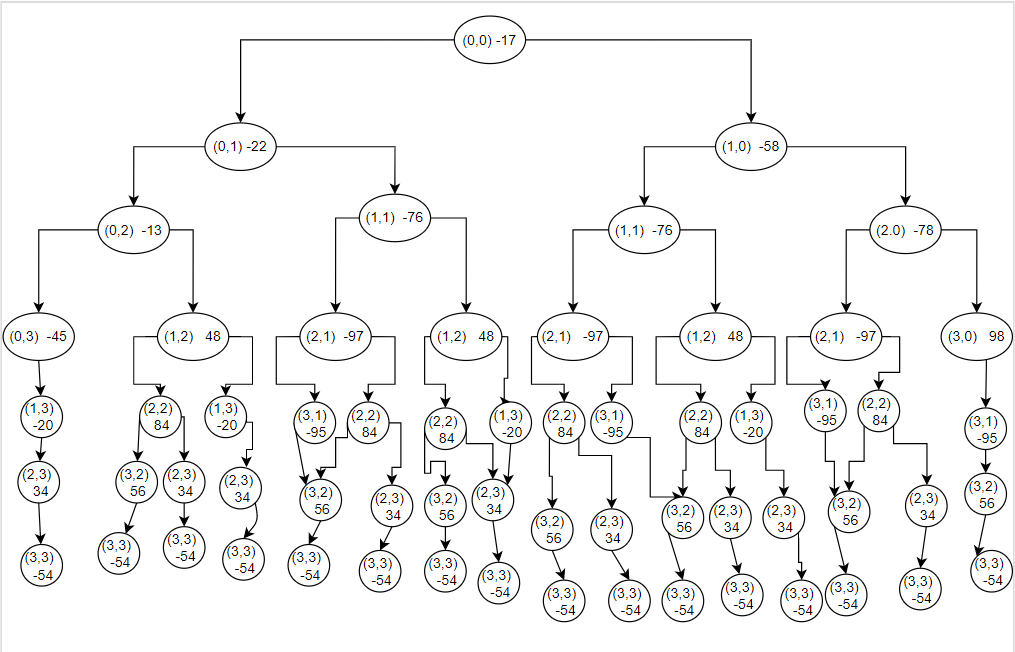
To update the path, add the value of the current cell.

Return Result:

Return a Result object containing the maximum value and the revised path.

**3- Draw the recursion tree for f using the values from the example above.**

**Example1:** 



**In this example :** we divide a big problem to a simple parts problem

**The maximum value is:188**

**Sequence of cell values: 48, 84, 56**

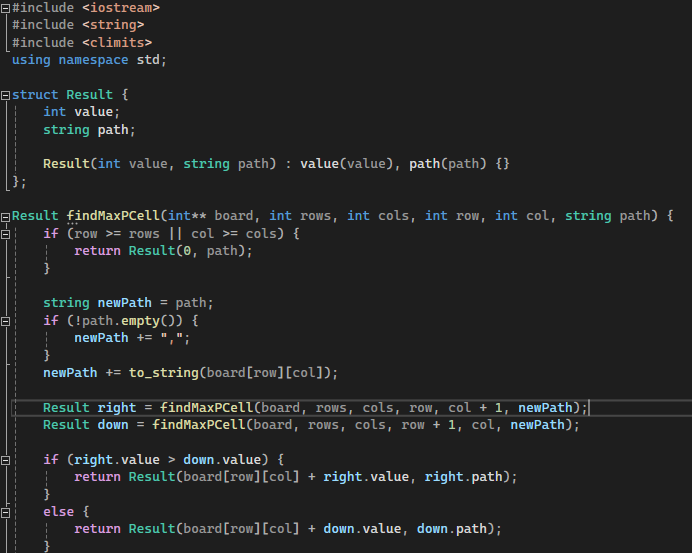
**Recursion Tree**

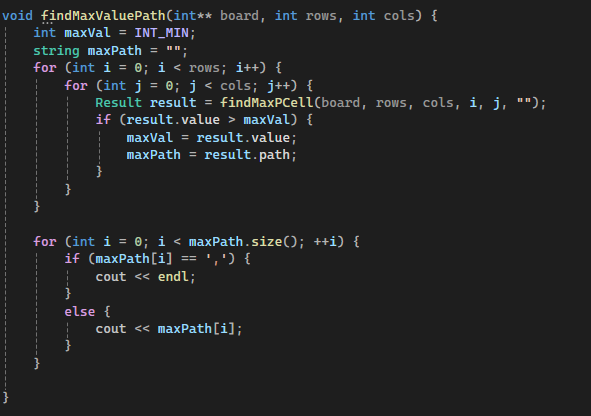
**For a given cell (row, col), the recursion tree has two branches:**

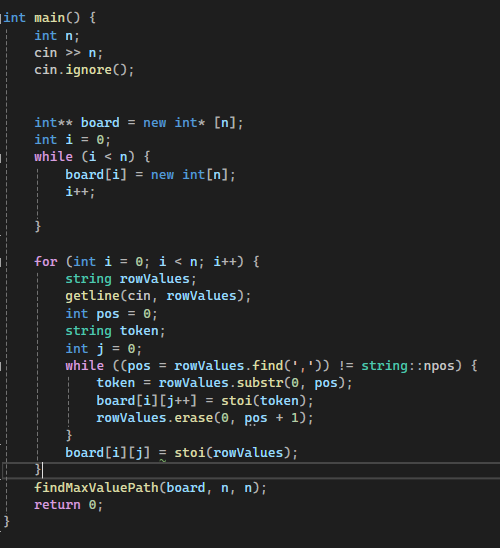
**One branch going down to the cell (row+1, col)**

**Another branch going right to the cell (row, col+1)**

**4- Write the recursive (divide and conquer) code to solve the question.**



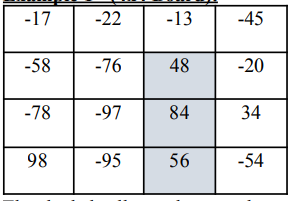




# Part2: Dynamic Programming:

**5- Draw the table and determine the dependencies between the table cells.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **j** | | | |
| **i** | **(0,0)** | **(0,1)** | **(0,2)** | **(0,3)** |
| **(1,0)** | **(1,1)** | **(1,2)** | **(1,3)** |
| **(2,0)** | **(2,1)** | **(2,2)** | **(2,3)** |
| **(3,0)** | **(3,1)** | **(3,2)** | **(3,3)** |



**For ex1:**

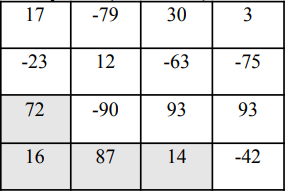
Example 1 - (4x4 Board):

The maximum value is:188

Sequence of cell values: 48, 84, 56

|  |  |  |  |
| --- | --- | --- | --- |
| **-17** | **-22** | **-13** | **-45** |
| **-58** | **-76** | **48** | **28** |
| **-78** | **-97** | **132** | **166** |
| **98** | **3** | **188** | **134** |

**For ex2:**



Example - (4x4 Board):

The maximum value is:189

Sequence of cell values: 72, 16, 87,14

|  |  |  |  |
| --- | --- | --- | --- |
| **17** | **-62** | **30** | **33** |
| **-6** | **12** | **-33** | **-42** |
| **72** | **-18** | **89** | **186** |
| **88** | **175** | **189** | **147** |

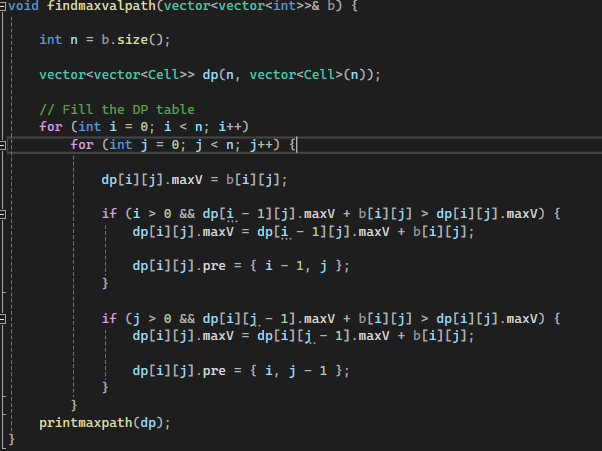
**Dependencies:** Each cell, dp[i][j], is dependent upon the cells above (dp[i-1][j]) and below (dp[i][j-1]).

The dependency is based on the maximum value path, which is the maximum of the paths that come from the top or left. It determines the maximum value of a cell.

Each Cell struct has a pre vector that constructs a backward path from any given cell to the start, indicating the best path to that cell.

**6- Determine the direction of movement within the table.**

Within the table, movement is limited to the right and bottom only. This indicates that you cannot travel up, down, or diagonally from any given cell; instead, you can only advance to the cell below or to the cell to the right. By taking into account only these two directions of movement, the dynamic programming method is guided by this restriction to effectively compute the greatest value feasible up to each cell.



A-Filling Out the DP Table:

The table is filled row by row, from top to bottom and from left to right within each row.

Each cell, dp[i][j], is calculated using the maximum value that can be reached by moving down (to dp[i+1][j]) or right (to dp[i][j+1]).

This is represented in the code by two if conditions inside of nested loops.

B-Dependencies Between Table Cells:

Each and every cell, dp[i][j], depends on the cells below (dp[i+1][j]) and directly to the right (dp[i][j+1]).

However, the dependency is also affected by the values of the board's cells. The actual value of dp[i][j] is the board value at [i][j] plus the greater of the two dependent cells (dp[i+1][j] and dp[i][j+1]).

C-Keeping Up the path:

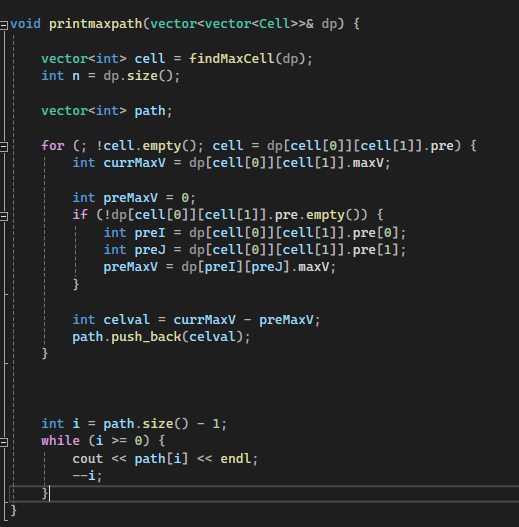
The path of the maximum value is traced backward from the cell containing the maximum value. This is done by using the pre attribute of each cell, which records the coordinates of the previous cell.

The printmaxpath function iteratively returns to the beginning of the path from the cell with the maximum value (located by findMaxCell) by following the pre pointers.

This gives the sequence of actions (or the route) that yields the maximum value.

**In summary, moving forward is the main direction of movement when filling the table (either right or down). In contrast, the movement is backward (following the pre pointers) when tracing the path that yields the maximum value. Table cells show interdependencies, meaning that a given cell's value depends on the maximum values of the cells directly to its right and below it, and can be adjusted by the cell's board value.**

**8- Write the code that will print the sequence of moves that get you the solution.**



# Conclusion:

We tackled the subject of determining the highest value that can be amassed by traveling inside a NxN board, where each cell has an integer value, in this dynamic programming project. Beginning in an arbitrary cell, the player takes one step at a time, moving either down or to the right, collecting the value of each visited cell. When a player collides with any of the board's edges, the game is over.

**Divide & Conquer:**We defined the function f to represent the maximum cumulative value on the board, which is the value returned by our optimization issue.

The current position on the board is represented by the parameters i and j.

Using the problem statement's examples, we created the recursion tree for f.

Next, in order to determine the maximum value and the order of moves, we put into practice a recursive (divide and conquer) approach.

**Dynamic Programming:**

The relationships between the table cells were ascertained when we created a table that represented the NxN board. Each cell's value is determined by the values of the cells next to it.

We established that the table may only move in one of two directions: either down or right.

To put the maximum amount of collected values in the table, we wrote Dynamic Programming code.

We also built code to print the steps that need to be taken in order to solve the problem.

We were able to ascertain the maximum value and the proper move order required to reach it on the board with speed thanks to these processes. This approach ensures that we avoid pointless computations and find the optimal solution in polynomial time complexity. Combining the two approaches results in an efficient and reliable solution to the problem.