** Part1: Divide & Conquer **

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

In the context of the maximum board value problem, the value provided by function *f* is the maximum value that can be obtained by the player as they move around the board in a series of steps, either down or to the right, until they move off either of the board's edges. By determining the most suitable order that provides the highest overall value, the objective is to maximize this value.

2- Define the parameters which *f* depends on

The function *f* in the context of the maximum board value problem is dependent upon the subsequent parameters:

- **Position**

  index: shows the cell's current index in the grid. It used to locate the current cell and discover paths to the right and down.

  pathStart: A flag that indicates if a path is currently at the starting point (1) or not (0). This option determines whether the value of the current cell should be included in the path or not by determining if the path just started or not.

- **Board Configuration**

  boardSize: refers to the square grid's dimensions. Based on the current index, it is used to determine the row and column indices.

  board: The grid's 2D array representation. The recursive calculations use the values of the grid's cells.

These parameters are essential for creating the recursive function MaxPathValueRecursive and applying a divide and conquer strategy to solve the maximum board value problem. The function depends on the current position, the configuration of the board, and the remaining moves to determine the maximum value that can be accumulated.

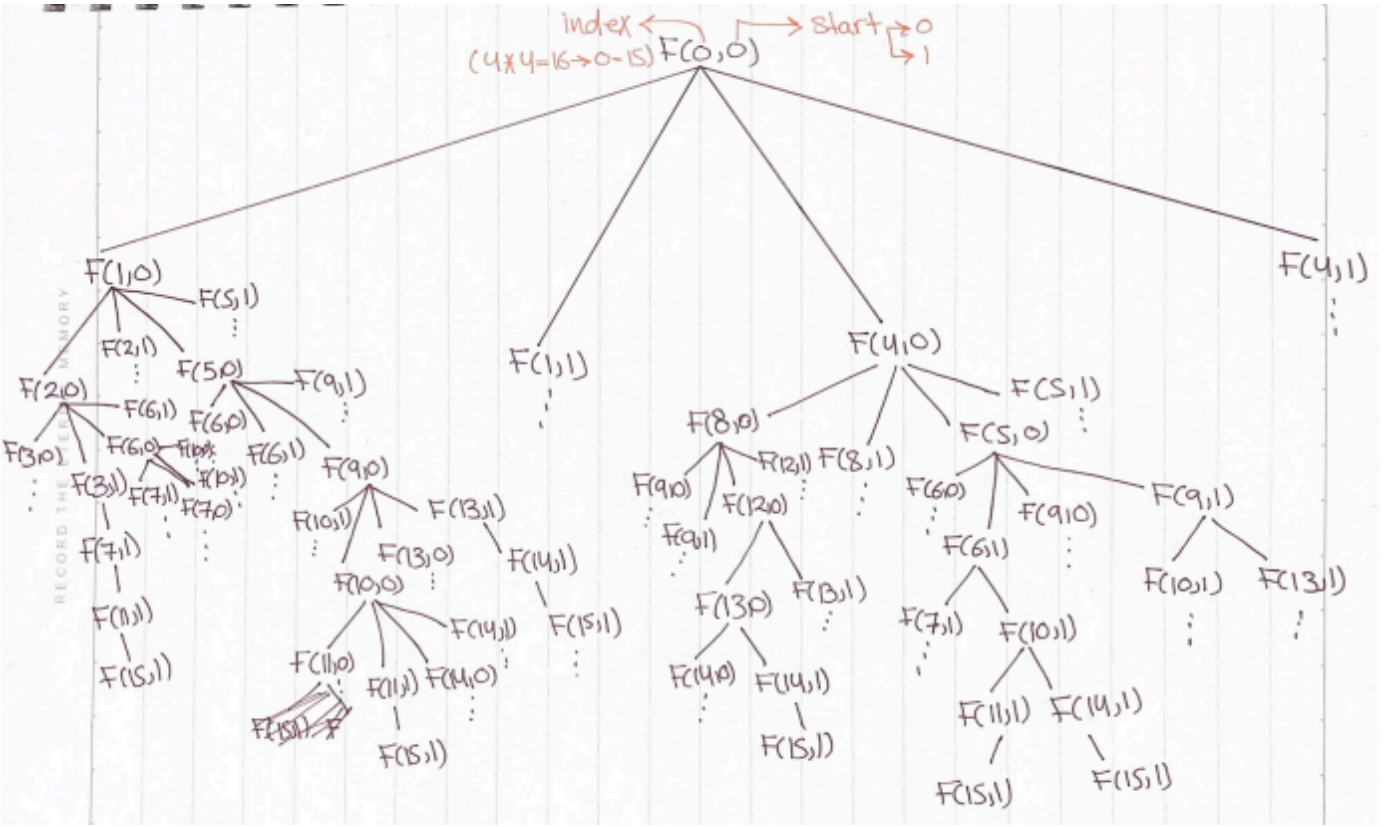
3- Draw the recursion tree for *f* using the values from the example above

For Example 1:

Example 1 - (4x4 Board):

-17	-22	-13	-45
-58	-76	48	-20
-78	-97	84	34
98	-95	56	-54

the recursion tree:



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

```
int StepDown(int index, int height) {
    int row = index / height + 1;
    int column = index % height;

    if (row >= height) {
        return -1;
    }

    return row * height + column;
}

int StepRight(int index, int width) {
    int row = index / width;
    int column = index % width + 1;

    if (column >= width) {
        return -1;
    }

    return row * width + column;
}


int MaxPathValueRecursive(int index, int pathStart, int boardSize,
int** board) {
    if (index == -1) {
        return 0;
    }

    int row = index / boardSize;
    int column = index % boardSize;

    if (pathStart == 1) {
        int indexDown = StepDown(index, boardSize);
        int indexRight = StepRight(index, boardSize);
```

```
        int down = board[column][row] +
MaxPathValueRecursive(indexDown, 1, boardSize, board);
        int right = board[column][row] +
MaxPathValueRecursive(indexRight, 1, boardSize, board);

        return std::max(right, down);
    }

    else {
        int indexDown = StepDown(index, boardSize);
        int indexRight = StepRight(index, boardSize);

        int down_withCurrent = board[column][row] +
MaxPathValueRecursive(indexDown, 1, boardSize, board);
        int down_withoutCurrent = MaxPathValueRecursive(indexDown, 0,
boardSize, board);

        int right_withCurrent = board[column][row] +
MaxPathValueRecursive(indexRight, 1, boardSize, board);
        int right_withoutCurrent = MaxPathValueRecursive(indexRight, 0,
boardSize, board);

        return std::max(down_withCurrent, std::max(right_withCurrent,
std::max(down_withoutCurrent, right_withoutCurrent)));
    }
}

int maximumPathRecursive(int** board, int size) {
    int value_startA = MaxPathValueRecursive(0, 0, size, board);
    int value_startB = MaxPathValueRecursive(0, 1, size, board);

    return std::max(value_startA, value_startB);
}
```

** Part2: Dynamic Programming **

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

For Example 1:

Example 1 - (4x4 Board):

-17	-22	-13	-45
-58	-76	48	-20
-78	-97	84	34
98	-95	56	-54

the table:

index after turning the 2D array (4*4) to 1D array (16)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	-17	-22	-13	-45	-58	-76	48	-20	-78	-97	84	34	98	-95	56	0
1	136	153	175	-31	54	112	188	14	20	43	140	34	98	-39	56	-54

start

6- Determine the direction of movement within the table

the table for example 1:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	-17	-22	-13	-45	-58	-76	48	-20	-78	-97	84	34	98	-95	56	0
1	136	153	175	-31	54	112	188	14	20	43	140	34	98	-39	56	-54

Direction

7- Write the Dynamic programming code which fills the table(s).

```
void DP(int** board, int size) {
    int boardSize = size * size; // # of cells in the grid
    int endingIndex = boardSize - 1; // index of the last cell in grid
    int* movementPath = new int[boardSize]; // store path movement of
                                            each cell
    int** values = new int*[boardSize]; // store values of each cell

    for (int i = 0; i < boardSize; ++i) {
        values[i] = new int[2];
    }

    // last cell location
    int lastVertical = endingIndex / size;
    int lastHorizontal = endingIndex % size;

    values[endingIndex][0] = MIN;
    values[endingIndex][1] = board[lastHorizontal][lastVertical];

    for (int index = endingIndex - 1; index >= 0; --index) {

        int Y_index = index / size;
        int X_index = index % size;

        values[index][0] = MIN;
        values[index][1] = board[X_index][Y_index];

        int down = StepDown(index, size);
        int right = StepRight(index, size);

        if (down != -1 &&  right != -1) {
            bool RightLarger0 = values[down][0] < values[right][0];
            bool RightLarger1 = values[down][1] < values[right][1];

            int value0 = std::max(values[down][0], values[right][0]);
            int value1 = std::max(values[down][1], values[right][1]);

            values[index][1] += std::max(0, value1);
```

```
            if (value1 > 0) {
                if (RightLarger1) {
                    movementPath[index] = 1;
                }
                else {
                    movementPath[index] = 2;
                }
            }
            values[index][0] += std::max(0, std::max(value0, value1));
        }

        else if (right == -1) {
            values[index][1] += std::max(0, values[down][1]);

            if (values[down][1] > 0) {
                movementPath[index] = 2;
            }

            values[index][0] += std::max(0, std::max(values[down][1],
values[down][0]));
        }

        else {
            if (values[right][1] > 0) {
                movementPath[index] = 1;
            }

            values[index][0] += std::max(0, std::max(values[right][1],
values[right][0]));
            values[index][1] += std::max(0, values[right][1]);
        }
    }

    // to find the maximum total value of all paths
    int maximumIndex = -1;
    int maximum = MIN;
```

```
    for (int j = 0; j < boardSize; ++j) {
        for (int k = 0; k < 2; ++k) {
            if (values[j][k] >= maximum) {
                maximum = values[j][k];
                maximumIndex = j;
            }
        }
    }

    printOptimalPath(board, movementPath, size, maximumIndex);

    for (int index = 0; index < boardSize; ++index) {
        delete[] values[index];
    }

    delete[] movementPath;
    delete[] values;
}
```

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

```
void printOptimalPath(int** board, int* movementPath, int size, int
maximumIndex) {
    int index = maximumIndex;
    int pathDirection = movementPath[maximumIndex];

    while (pathDirection != 0) {
        pathDirection = movementPath[index];

        int Y_index = index / size;
        int X_index = index % size;

        std::cout << board[X_index][Y_index] << std::endl;
```

```
        int down = StepDown(index, size);
        int right = StepRight(index, size);

        if (pathDirection == 1) {
            index = right;
        }
        else if (pathDirection == 2) {
            index = down;
        }
    }
}
```

Tala Abu soud
12027754
Dr.Samer Arandi