Dynamic Programming Project

Maximum Board Value

Part1: Divide & Conquer

1. The value that the function f returned

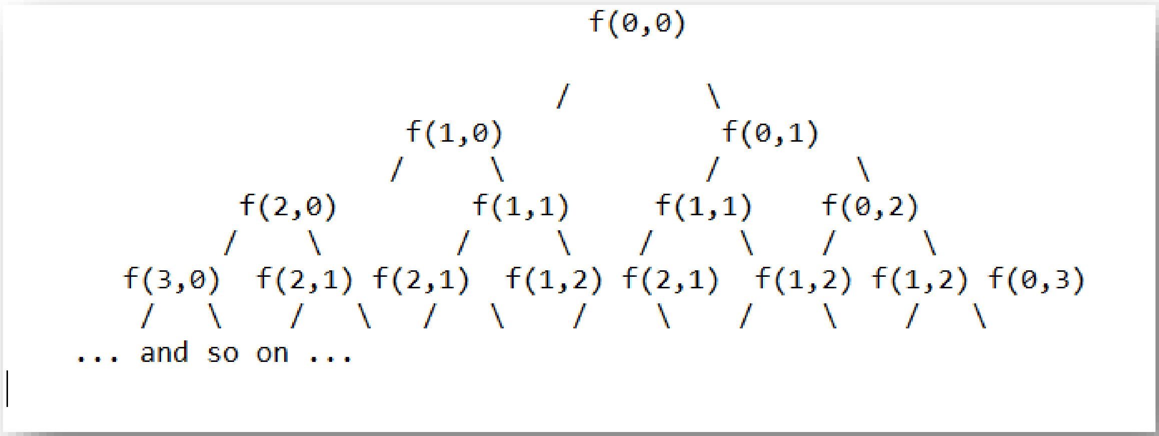
The value returned by the function f is the maximum sum of a path from the top-left corner (0, 0) to the bottom-right corner (gridsize - 1, gridsize - 1) of the grid while moving only right and down, and maximizing the sum of values encountered along the path.

2. The Parameters That Determine f

The function f depends on two parameters: x and y, representing the current position on the grid. f(x, y) returns the maximum sum path from the cell (x, y) to the bottom-right corner (gridsize - 1, gridsize - 1).

3. Recursion Tree for f

The recursion tree represents all possible paths and their respective sums.



```
                                   f(0,0)

                                /         \
                          f(1,0)              f(0,1)
                         /      \            /       \
                  f(2,0)          f(1,1)      f(1,1)     f(0,2)
                 /     \         /      \    /      \    /      \
             f(3,0)  f(2,1) f(2,1)  f(1,2) f(2,1)  f(1,2) f(1,2) f(0,3)
              /   \    /   \   /    \    /     \    /     \    /    \
   ... and so on ...
```

4. Recursive (Divide and Conquer) Code

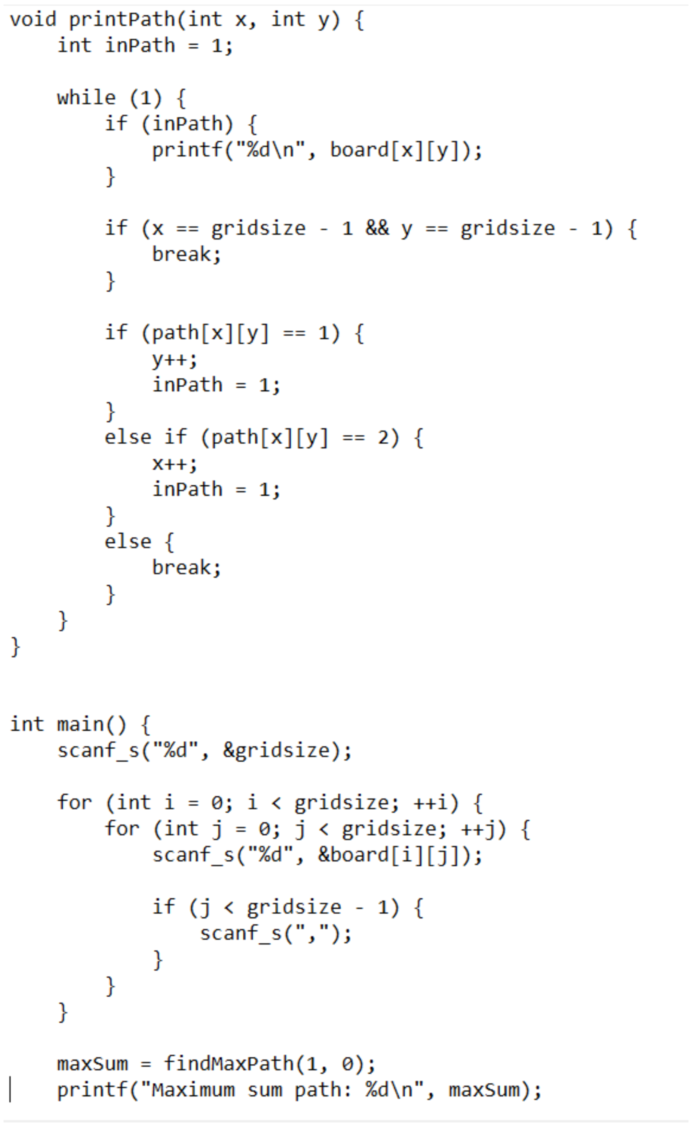
The findMaxPath function is the recursive function that calculates the maximum sum from a given cell to the bottom-right corner while considering two directions: right and down. It returns the maximum sum, and path array is used to keep track of the path direction.

The printPath function is used to print the path found by following the directions stored in the path array.

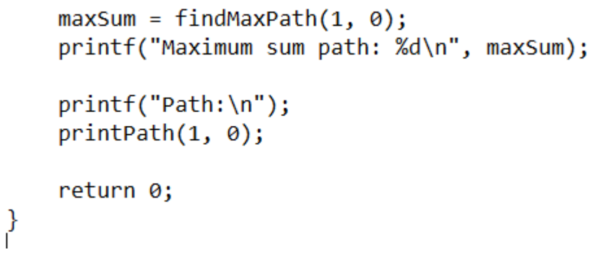
The code starts from the top-left corner (1, 0) and recursively explores all possible paths to the bottom-right corner (gridsize - 1, gridsize - 1) while maximizing the sum. It uses a divide and conquer approach to make decisions at each cell (x, y) on whether to move right or down, considering the direction with the maximum sub-path sum.

The code

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_N 1000

int board[MAX_N][MAX_N];
int gridsize;
int maxSum;
int path[MAX_N][MAX_N];

int findMaxPath(int x, int y) {
    if (x == gridsize - 1 && y == gridsize - 1) {
        return board[x][y];
    }

    int rightSum = -1e9;
    int downSum = -1e9;

    if (x < gridsize - 1) {
        downSum = findMaxPath(x + 1, y);
    }

    if (y < gridsize - 1) {
        rightSum = findMaxPath(x, y + 1);
    }

    int maxDirection = (downSum > rightSum) ? 2 : 1;
    int maxSubPathSum = (maxDirection == 2) ? downSum : rightSum;

    if (maxSubPathSum > 0) {
        path[x][y] = maxDirection;
        return maxSubPathSum + board[x][y];
    }
    else {
        path[x][y] = 0;
        return board[x][y];
    }
}
```

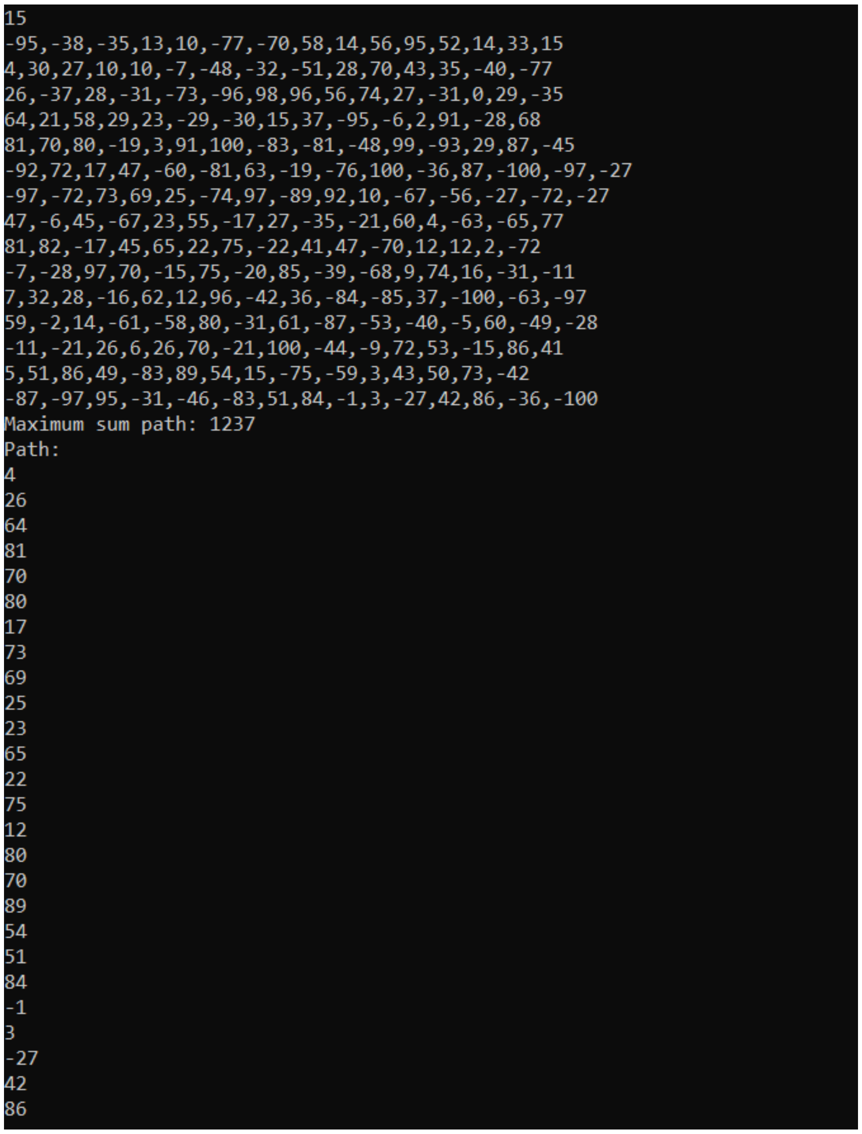


```c
void printPath(int x, int y) {
    int inPath = 1;

    while (1) {
        if (inPath) {
            printf("%d\n", board[x][y]);
        }

        if (x == gridsize - 1 && y == gridsize - 1) {
            break;
        }

        if (path[x][y] == 1) {
            y++;
            inPath = 1;
        }
        else if (path[x][y] == 2) {
            x++;
            inPath = 1;
        }
        else {
            break;
        }
    }
}


int main() {
    scanf_s("%d", &gridsize);

    for (int i = 0; i < gridsize; ++i) {
        for (int j = 0; j < gridsize; ++j) {
            scanf_s("%d", &board[i][j]);

            if (j < gridsize - 1) {
                scanf_s(",");
            }
        }
    }

    maxSum = findMaxPath(1, 0);
    printf("Maximum sum path: %d\n", maxSum);
```

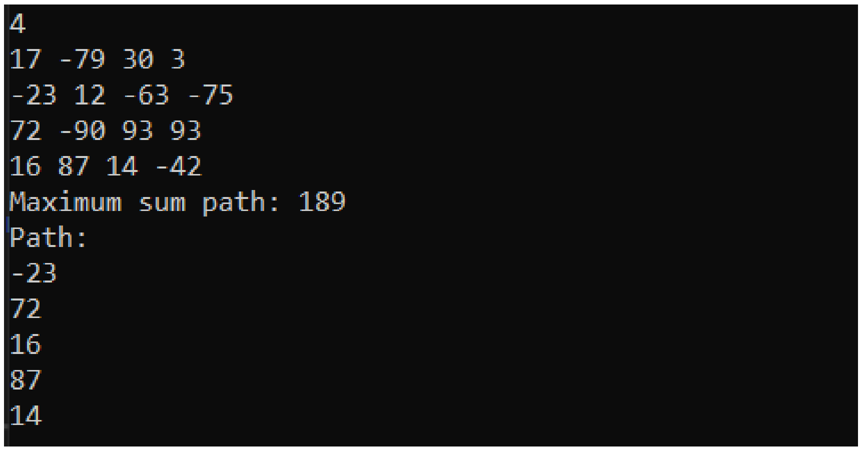


```c
    maxSum = findMaxPath(1, 0);
    printf("Maximum sum path: %d\n", maxSum);

    printf("Path:\n");
    printPath(1, 0);

    return 0;
}
```

The output:



```
15
-95,-38,-35,13,10,-77,-70,58,14,56,95,52,14,33,15
4,30,27,10,10,-7,-48,-32,-51,28,70,43,35,-40,-77
26,-37,28,-31,-73,-96,98,96,56,74,27,-31,0,29,-35
64,21,58,29,23,-29,-30,15,37,-95,-6,2,91,-28,68
81,70,80,-19,3,91,100,-83,-81,-48,99,-93,29,87,-45
-92,72,17,47,-60,-81,63,-19,-76,100,-36,87,-100,-97,-27
-97,-72,73,69,25,-74,97,-89,92,10,-67,-56,-27,-72,-27
47,-6,45,-67,23,55,-17,27,-35,-21,60,4,-63,-65,77
81,82,-17,45,65,22,75,-22,41,47,-70,12,12,2,-72
-7,-28,97,70,-15,75,-20,85,-39,-68,9,74,16,-31,-11
7,32,28,-16,62,12,96,-42,36,-84,-85,37,-100,-63,-97
59,-2,14,-61,-58,80,-31,61,-87,-53,-40,-5,60,-49,-28
-11,-21,26,6,26,70,-21,100,-44,-9,72,53,-15,86,41
5,51,86,49,-83,89,54,15,-75,-59,3,43,50,73,-42
-87,-97,95,-31,-46,-83,51,84,-1,3,-27,42,86,-36,-100
Maximum sum path: 1237
Path:
4
26
64
81
70
80
17
73
69
25
23
65
22
75
12
80
70
89
54
51
84
-1
3
-27
42
86
```



```
4
17 -79 30 3
-23 12 -63 -75
72 -90 93 93
16 87 14 -42
Maximum sum path: 189
Path:
-23
72
16
87
14
```

Part2: Dynamic Programming

5. Drawing the Table and Determining Dependencies

For the given 4x4 board:

	0	**1**	**2**	**3**
0	17	-79	30	3
1	-23	12	-63	-75
2	72	-90	93	93
3	16	87	14	-42

We will fill the dynamicprog table and p array.

The **dynamicprog** table is being filled from the bottom-right corner to the top-left corner, taking the maximum of either moving right or down in each step:

1. At cell (3,3), **dynamicprog[3][3] = -42**.
2. At cell (3,2), **dynamicprog[3][2] = 14** (since 14 > 14 - 42).
3. At cell (3,1), **dynamicprog[3][1] = 87 + 14 = 101** (since 101 > 101 - 42).

Let's continue from cell (3,0):

4. At cell (3,0), **dynamicprog[3][0] = 16 + 101 = 117** (since 117 > 117 - 42).

Now, move up to the second row from the bottom:

5. At cell (2,3), **dynamicprog[2][3] = 93** (no cells to the right or down).
6. At cell (2,2), **dynamicprog[2][2] = 93 + 93 = 186** (since 186 > 186 - 42).
7. At cell (2,1), **dynamicprog[2][1] = -90 + 186 = 96** (since 96 > 96 - 42).
8. At cell (2,0), **dynamicprog[2,0] = 72 + 117 = 189** (since 189 > 189 - 42).

Move up to the third row from the bottom:

9. At cell (1,3), **dynamicprog[1,3] = -75** (no cells to the right or down).
10. At cell (1,2), **dynamicprog[1,2] = -63 + (-75) = -138** (since -138 > -138 - 42).
11. At cell (1,1), **dynamicprog[1,1] = 12 + (-75) = -63** (since -63 > -63 - 42).

12. At cell (1,0), **dynamicprog[1,0] = -23 + 96 = 73** (since 73 > 73 - 42).

Finally, the top row:

13. At cell (0,3), **dynamicprog[0,3] = 3** (no cells to the right or down).
14. At cell (0,2), **dynamicprog[0,2] = 30 + 3 = 33** (since 33 > 33 - 42).
15. At cell (0,1), **dynamicprog[0,1] = -79 + 33 = -46** (since -46 > -46 - 42).
16. At cell (0,0), **dynamicprog[0,0] = 17 + 73 = 90** (since 90 > 90 - 42).

So the filled dynamicprog table looks like this:

	0	**1**	**2**	**3**
0	90	-46	33	3
1	73	-63	-138	-75
2	189	96	186	93
3	117	101	14	-42

6. Direction of Movement:

The movement direction is determined by the **p** array, where:

- **1** indicates moving right,
- **2** indicates moving down.

Applying fillDP to the Board:

Applying fillDP to the board, you compare the current cell's sum to that of the cell to the right or below, beginning in the bottom-right corner, and select the direction that yields the maximum sum. The movement's direction is recorded in the p array, and the dynamicprog table is filled in.

Applying findMaximumPath to the Board:

The dynamicprog table's cell with the maximum value is the first thing the findMaximumPath method looks for. It's not necessary that this be in the top-left cell. It reconstructs the path from the found cell (start_i, start_j) by following the instructions in p, printing the values from the board in order.

p (directions):

	0	**1**	**2**	**3**
0	2	2	1	0
1	2	1	1	0
2	2	1	1	0
3	1	1	0	0

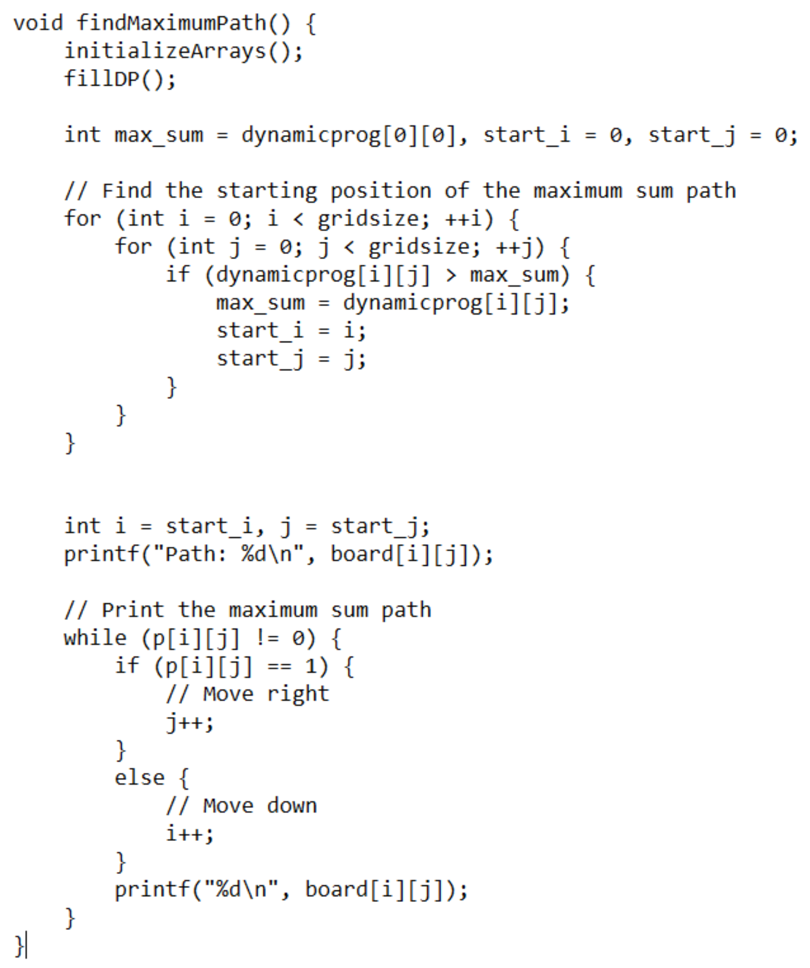
For the specific board you've given, where the maximum value is 189 and the sequence of cell values is 72, 16, 87, 14, the findMaximumPath function would identify these cells and print the path based on the p array.

7. Dynamic Programming Code to Fill the Table

```
void fillDP() {
    // Fill dynamic programming table (down and right moves)
    for (int i = gridsize - 1; i >= 0; --i) {
        for (int j = gridsize - 1; j >= 0; --j) {
            dynamicprog[i][j] = board[i][j];

            // Check if moving down gives a better result
            if (i < gridsize - 1) {
                int down = dynamicprog[i + 1][j];
                if (down + board[i][j] > dynamicprog[i][j]) {
                    dynamicprog[i][j] = down + board[i][j];
                    p[i][j] = 2; // moving down
                }
            }


            // Check if moving right gives a better result
            if (j < gridsize - 1) {
                int right = dynamicprog[i][j + 1];
                if (right + board[i][j] > dynamicprog[i][j]) {
                    dynamicprog[i][j] = right + board[i][j];
                    p[i][j] = 1; // moving right
                }
            }
        }
    }
}
```

1. **Iterate through the grid in reverse:**
    - The function loops across the grid, beginning in the top-left corner and ending in the bottom-right corner. Because each cell's value depends on the uncalculated cells to its right and below it, a reverse iteration is required. Thus, we make sure that these dependent values are already computed when needed by beginning at the bottom-right.
2. **Set the initial value of each cell:**
    - The function first sets dynamicprog[i][j] to board[i][j] for each cell (i, j). This is the initial value for the maximum sum path that encompasses this cell.
3. **Check the 'down' movement:**
    - The function determines whether it is possible to shift a cell down, or whether it is not on the last row, for each cell. The function finds the sum of the values in the current cell (board[i][j]) and the cell immediately below it (dynamicprog[i + 1][j]) in the dynamicprog table, if that is available. This total shows the largest sum path that descends first, beginning in the current cell.The function sets p[i][j] to 2 and updates dynamicprog[i][j] with this sum if it is bigger than the current value of dynamicprog[i][j]. This indicates that descending from this cell leads to a path with a greater sum.
4. **Check the 'right' movement:**
    - In a similar vein, the function determines whether moving right is feasible (that is, whether it is not in the final column). The value in the cell immediately to the right in the dynamicprog table (dynamicprog[i][j + 1]) is added to the value of the present cell (board[i][j]), if at all possible. The greatest sum path, beginning in the current cell and going right, is represented by this sum.
    - The function updates dynamicprog[i][j] with this sum and sets p[i][j] to 1, indicating that traveling right from this cell leads to a path with a bigger sum, if this sum is greater than the current value of dynamicprog[i][j].

8. Code for Printing the Sequence of Moves



```
void findMaximumPath() {
    initializeArrays();
    fillDP();

    int max_sum = dynamicprog[0][0], start_i = 0, start_j = 0;

    // Find the starting position of the maximum sum path
    for (int i = 0; i < gridsize; ++i) {
        for (int j = 0; j < gridsize; ++j) {
            if (dynamicprog[i][j] > max_sum) {
                max_sum = dynamicprog[i][j];
                start_i = i;
                start_j = j;
            }
        }
    }


    int i = start_i, j = start_j;
    printf("Path: %d\n", board[i][j]);

    // Print the maximum sum path
    while (p[i][j] != 0) {
        if (p[i][j] == 1) {
            // Move right
            j++;
        }
        else {
            // Move down
            i++;
        }
        printf("%d\n", board[i][j]);
    }
}
```

1. **Initialize Arrays:**

   - Calls initializeArrays() to set the initial values for dynamicprog and p arrays.

2. **Fill Dynamic Programming Table:**

   - Calls fillDP() to fill the p array with the optimal moves' directions (down or right) and to fill the dynamicprog table with the maximum sum pathways from each cell to the bottom-right corner.

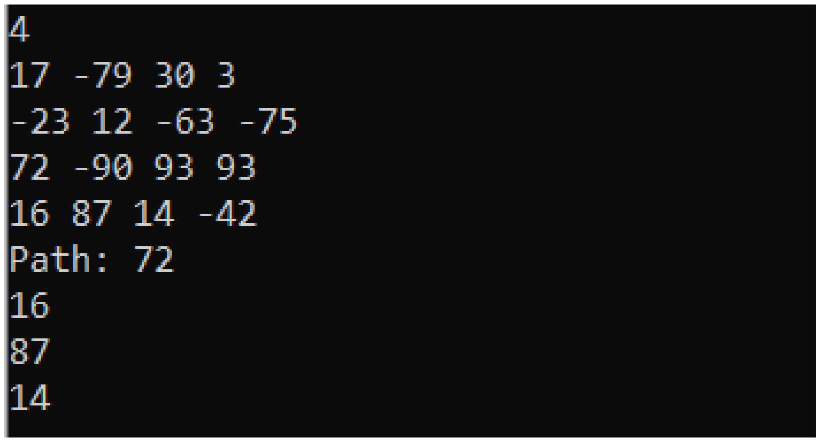
3. **Find Starting Position of the Maximum Sum Path:**

   - After that, the function initializes the variables max_sum, start_i, and start_j to record the maximum sum as well as the path's beginning point.

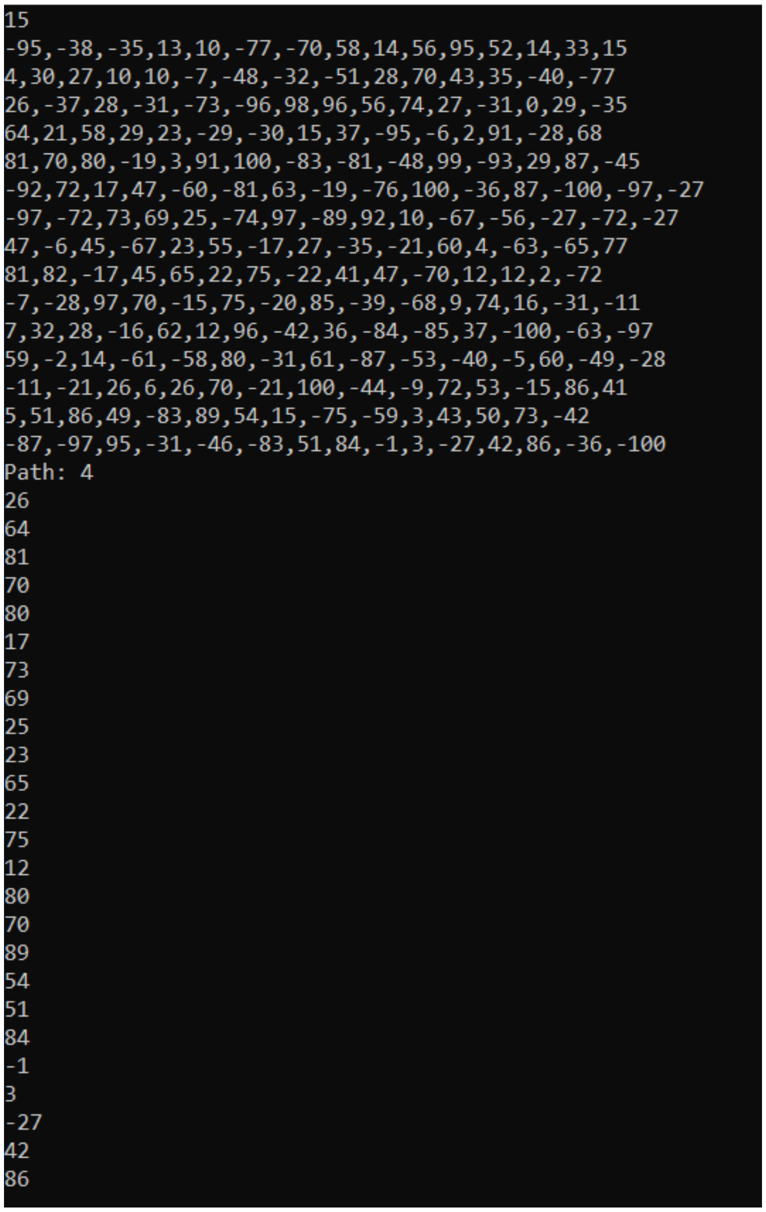
- To identify the cell with the highest sum, iteratively searches through the dynamicprog table. Any cell from which the path to the bottom-right corner (or up to a point where you can't travel any farther right or down) produces the maximum sum can be this one; it doesn't have to be the top-left cell.Whenever it locates a cell whose total exceeds the current max_sum, it updates start_i, start_j, and max_sum.

4. **Print the Maximum Sum Path:**
   - I and j are set to start_i and start_j, respectively, by the function. This is the maximum sum path's beginning point.
   - The value of the board at this first location (board[i][j]) is printed.
   - It enters a loop that continues until it reaches a cell where no further moves are possible (indicated by p[i][j] == 0). Within this loop:
     - It increases j to move right if p[i][j] indicates a move to the right (p[i][j] == 1).
     - In all other cases, it moves down by increasing i.
     - The values contained in the path are shown by printing the value of the board at the new place (board[i][j]) at each step.

The output:



```
4
17 -79 30 3
-23 12 -63 -75
72 -90 93 93
16 87 14 -42
Path: 72
16
87
14
```



```
15
-95,-38,-35,13,10,-77,-70,58,14,56,95,52,14,33,15
4,30,27,10,10,-7,-48,-32,-51,28,70,43,35,-40,-77
26,-37,28,-31,-73,-96,98,96,56,74,27,-31,0,29,-35
64,21,58,29,23,-29,-30,15,37,-95,-6,2,91,-28,68
81,70,80,-19,3,91,100,-83,-81,-48,99,-93,29,87,-45
-92,72,17,47,-60,-81,63,-19,-76,100,-36,87,-100,-97,-27
-97,-72,73,69,25,-74,97,-89,92,10,-67,-56,-27,-72,-27
47,-6,45,-67,23,55,-17,27,-35,-21,60,4,-63,-65,77
81,82,-17,45,65,22,75,-22,41,47,-70,12,12,2,-72
-7,-28,97,70,-15,75,-20,85,-39,-68,9,74,16,-31,-11
7,32,28,-16,62,12,96,-42,36,-84,-85,37,-100,-63,-97
59,-2,14,-61,-58,80,-31,61,-87,-53,-40,-5,60,-49,-28
-11,-21,26,6,26,70,-21,100,-44,-9,72,53,-15,86,41
5,51,86,49,-83,89,54,15,-75,-59,3,43,50,73,-42
-87,-97,95,-31,-46,-83,51,84,-1,3,-27,42,86,-36,-100
Path: 4
26
64
81
70
80
17
73
69
25
23
65
22
75
12
80
70
89
54
51
84
-1
3
-27
42
86
```