Chocolate Pickup

Created by Chandra Prakash

Problem Statement

You are given a `m x n` matrix called `grid`, where each cell contains some chocolates (represented by a non-negative integer). Two friends, **Alice** and **Bob**, start from the top row of the grid:

- **Alice** starts from the top-left cell `(0, 0)`.
- **Bob** starts from the top-right cell `(0, n - 1)`.

Both friends move simultaneously from the top to the bottom of the grid. In each step, they can move to the **next row**, and from the current column, they can either:

- Stay in the same column `j`
- Move to the left column `j - 1` (if within bounds)
- Move to the right column `j + 1` (if within bounds)

Each friend collects **all chocolates in the cells they pass through**. If both friends land on the **same cell** during any move, chocolates in that cell are collected **only once** (no double counting).

Your task is to find the **maximum number of chocolates** that can be collected by Alice and Bob combined, by optimally choosing their paths.

Input

- An integer `T` : number of test cases.

  For each test case:

  - Two integers `m` and `n` : number of rows and columns in the matrix `grid`.
  - `m` rows follow, each with `n` integers : representing the number of chocolates in each cell.

Constraints:

- `1 <= T <= 10`
- `2 <= m, n <= 50`
- `0 <= grid[i][j] <= 100`

Output

For each test case, print a single line with the **maximum number of chocolates** that can be collected.

Sample Input 1

```
2
3 4
2 3 1 2
3 4 2 2
5 6 3 5
2 2
1 1
1 2
```

Sample Output 1

```
21
5
```

Explanation:

Test Case 1:

Alice starts at (0, 0) and follows the path: (0, 0) → (1, 1) → (2, 1) → Chocolates: 2 + 4 + 6 = 12

Bob starts at (0, 3) and follows the path: (0, 3) → (1, 3) → (2, 3) → Chocolates: 2 + 2 + 5 = 9

Total = 12 + 9 = **21**

Test Case 2:

Alice: (0, 0) → (1, 0) → Chocolates: 1 + 1 = 2

Bob: (0, 1) → (1, 1) → Chocolates: 1 + 2 = 3

Total = 2 + 3 = **5**

Editorial

Note that we have a fixed starting point and a variable ending point.

Greedy fails.

So we need to try all paths and take the maximum path.

There will be two scenarios.

1 Doing it separately for Alice and Bob, make sure remove the common cell.

2 Do it for both Alice and Bob.

The second approach is possible and better than the first approach.

So, apply recursion for both of them at once.

Recursion

Approach

Direction: Note that we have a fixed points at the start of the grid. Hence, it is good to start the recursion from the fixed points.

Steps: Indices and base case, Explore all paths, get the ans(max or min).

Indices: We need to track Alice and Bob's position, i.e., `(i1, j1)` and `(i2, j2)`.

Note: Question says both are moving to the next row simultaneously. It means, the index `i` is going to be the same for both of them.

Final Indices: `i`, `j1` and `j2`

Base cases:

Note:

First write the out of the boundary conditions.

Then the destination base base.

Boundary condition:

Note: Return some large negative number but not INT_MIN.

Because, Using `-1e9` instead of `INT_MIN` avoids potential **integer overflow** when adding it to positive integers during recursive or DP calculations. It leads to **undefined behavior** at runtime, like wrapping around to positive values, producing **incorrect results** silently

```
if(j1 < 0 || j1 > n-1 || j2 < 0 || j2 > n-1) return -1e9;
```

Destination:

According to the question, they will reach the last row simultaneously.

Note: At the last row, Alice and Bob may either be in the same cell or in different cells.

If Same: Return only one cell value.

Else: Return the sum of both the cells.

```
if(i == n-1){
    if(j1 == j2) return grid[i][j1];
    else return grid[i][j1] + grid[i][j2];
}
```

Explore all the paths:

Each person has three possible moves at each step:

- Down: `(i+1, j)`
- Down-left: `(i+1, j-1)`
- Down-right: `(i+1, j+1)`

We need to move both of them simultaneously. Hence, for every move Alice makes, Bob has three corresponding choices, resulting in a total of 3 × 3 = 9 possible move combinations.

To cover all 9 possible move combinations for Alice and Bob at each step, it is useful to define a direction vector like {-1, 0, 1}. We then use two nested loops over this vector, one for Alice's direction and one for Bob's, so that all combinations of their moves are handled in a clean, concise, and efficient way.

Since the next index(for j1) is `j1-1`, `j1` or `j1+1` we can write it like `j1 + d1` and for bob's move it will be `j2 + d2`.

Change of state: `f(i+1, j1+d1, j2+d2)`

Note:

If Alice and Bob are at the same cell: Add it once and call next.

Else: Add both the cells and call next.

We need to store the maximum of all these 9 combinations' if and else.

Hence,

```
vector<int> dj = {-1, 0, 1};
int max1 = 0;
for(int d1 : dj) {
   for(int d2 : dir) {
        if(j1 == j2) max1 = max(max1, grid[i][j1] + f(i+1, j1+d1, j2+d2));
        else max1 = max(max1, grid[i][j1] + grid[i][j2] + f(i+1, j1+d1, j2+d2));
```

```
    }
}
```

Finally return the maximum value.

Code

```
int f(int i, int j1, int j2, vector<vector<int>>& grid, int m, int n) {
    if(j1 < 0 || j1 >= n || j2 < 0 || j2 >= n) return -1e9;

    if(i == m - 1) {
        if(j1 == j2) return grid[i][j1];
        else return grid[i][j1] + grid[i][j2];
    }

    int maxi = -1e9;
    vector<int> dj = {-1, 0, 1};

    for(int d1 : dj) {
        for(int d2 : dj) {
            int next = f(i + 1, j1 + d1, j2 + d2, grid, m, n);
            if(j1 == j2)
                maxi = max(maxi, grid[i][j1] + next);
            else
                maxi = max(maxi, grid[i][j1] + grid[i][j2] + next);
        }
    }

    return maxi;
}
```

Complexity Analysis

Time Complexity: `O(3^m * 3^m)` - Since we explore 9 combinations at every level, and there are `m` levels. Very inefficient.

Space Complexity: `O(m)` - Stack space for recursion depth.

Memoization

Approach

Use a 3D DP array `dp[i][j1][j2]` – it stores the **maximum chocolates collected from row** `i` **to the last row**, when Alice is at column `j1` and Bob is at column `j2`.

- Store the result.
- Return value if already calculated.

Code

```
int f(int i, int j1, int j2, vector<vector<int>> &grid, int m, int n, vector<vector<vector<int>>> &dp) {
    if (j1 < 0 || j1 >= n || j2 < 0 || j2 >= n) return -1e9;
```

```
    if (i == m - 1) {
        if (j1 == j2) return grid[i][j1];
        else return grid[i][j1] + grid[i][j2];
    }

    if (dp[i][j1][j2] != -1) return dp[i][j1][j2];

    int maxi = -1e9;
    for (int dj1 = -1; dj1 <= 1; dj1++) {
        for (int dj2 = -1; dj2 <= 1; dj2++) {
            int val = (j1 == j2) ? grid[i][j1] : grid[i][j1] + grid[i][j2];
            val += f(i + 1, j1 + dj1, j2 + dj2, grid, m, n, dp);
            maxi = max(maxi, val);
        }
    }

    return dp[i][j1][j2] = maxi;
}
```

Complexity Analysis

Time Complexity: `O(m * n * n * 9)` → `O(m * n^2)`

(There are `m * n * n` states, and each state does `9` computations.)

Space Complexity: `O(m * n * n)` for memo table + `O(m)` recursion stack.

Tabulation

Approach

- Same size dp table.
- Write the base case first.
- Initialize base row `m - 1`.
- Then iterate from `i = m - 2` to `0`.
- For each cell, try all 9 combinations from `j1`, `j2`.

Code

```
int maximumChocolates(int m, int n, vector<vector<int>>& grid) {
    vector<vector<vector<int>>> dp(m, vector<vector<int>>(n, vector<int>(n, 0)));

    for (int j1 = 0; j1 < n; j1++) {
        for (int j2 = 0; j2 < n; j2++) {
            if (j1 == j2) dp[m-1][j1][j2] = grid[m-1][j1];
            else dp[m-1][j1][j2] = grid[m-1][j1] + grid[m-1][j2];
        }
    }

    for (int i = m - 2; i >= 0; i--) {
        for (int j1 = 0; j1 < n; j1++) {
```

```
            for (int j2 = 0; j2 < n; j2++) {
                int maxi = -1e9;
                for (int dj1 = -1; dj1 <= 1; dj1++) {
                    for (int dj2 = -1; dj2 <= 1; dj2++) {
                        int nj1 = j1 + dj1, nj2 = j2 + dj2;
                        if (nj1 >= 0 && nj1 < n && nj2 >= 0 && nj2 < n) {
                            int val = (j1 == j2) ? grid[i][j1] : grid[i][j1] + grid[i][j2];
                            val += dp[i + 1][nj1][nj2];
                            maxi = max(maxi, val);
                        }
                    }
                }
                dp[i][j1][j2] = maxi;
            }
        }
    }

    return dp[0][0][n-1];
}
```

Complexity Analysis

Time Complexity: `O(m * n^2 * 9)` → `O(m * n^2)`

Space Complexity: `O(m * n^2)` - 3D DP array.

Space Optimization

Approach

Since each `dp[i][j1][j2]` depends only on `dp[i+1][*][*]`, we can use two 2D arrays: `curr` and `next`.

Code

```
int maximumChocolates(int m, int n, vector<vector<int>>& grid) {
    vector<vector<int>> next(n, vector<int>(n, 0)), curr(n, vector<int>(n, 0));

    for (int j1 = 0; j1 < n; j1++) {
        for (int j2 = 0; j2 < n; j2++) {
            if (j1 == j2) next[j1][j2] = grid[m-1][j1];
            else next[j1][j2] = grid[m-1][j1] + grid[m-1][j2];
        }
    }

    for (int i = m - 2; i >= 0; i--) {
        for (int j1 = 0; j1 < n; j1++) {
            for (int j2 = 0; j2 < n; j2++) {
                int maxi = -1e9;
                for (int dj1 = -1; dj1 <= 1; dj1++) {
                    for (int dj2 = -1; dj2 <= 1; dj2++) {
                        int nj1 = j1 + dj1, nj2 = j2 + dj2;
                        if (nj1 >= 0 && nj1 < n && nj2 >= 0 && nj2 < n) {
```

```
                int val = (j1 == j2) ? grid[i][j1] : grid[i][j1] + grid[i][j2];
                val += next[nj1][nj2];
                maxi = max(maxi, val);
              }
            }
          }
          curr[j1][j2] = maxi;
        }
      }
      next = curr;
    }

    return next[0][n-1];
  }
```

Complexity Analysis

Time Complexity: `O(m * n^2 * 9)` → `O(m * n^2)`

Space Complexity: `O(n^2)` - Only two 2D arrays needed.

THE END