# CyberChallenge.IT 2026
# Programming Solutions

## Contents

# 1 Scoring [40 points]

## 1.1 Problem Statement

On CTFTime.org, one of the main websites to get CTF-related info, CTF teams are ranked based on their results over the year. For each CTF a team plays, they are assigned a score based on their results. At the end of the year, the final score of the team is the sum of their best results **up to 10 CTFs**: if a team played less or equal than 10 CTFs the score is just the sum of the single scores, otherwise it is the sum of the top 10 only. You are given a list of results of several teams throughout the year, you need to compute their final score.

## 1.2 Problem Details

You will be given multiple testcases, where each testcase represents a team. For each team, you will be given a space separated list of results. It is guaranteed that each team played at least one CTF. The scores of the teams are always non-negative integers not exceeding 200 (as it is the maximum score that the real CTFTime.org gives).

### Input

The input consists of $2T + 1$ lines, where $T$ is the number of teams:

- Line 1: an integer $T$, the number of teams.
- Lines 2, ..., $T * 2 + 1$: for each team, the number $N$ of CTFs played as an integer followed by the list of scores, alternated line by line, so that line 2 contains the number of CTFs played by the first team, line 3 contains the space-separated scores of the first team, line 4 contains the number of CTFs played by the second team, line 5 contains the space-separated scores of the second team, and so on.

### Output

The output consists of $T$ lines, each containing the score of the corresponding team.

### Scoring

Your program will be tested on a number of testcases grouped in subtasks. In order to obtain the score associated to a subtask, you need to correctly solve all its testcases.

- **Subtask 1** [**20 points**]: $T = 1$, $N = 10$.
- **Subtask 2** [**20 points**]: $1 \leq T \leq 10^3$, $1 \leq N \leq 10^3$ for each team.

### Examples

| INPUT | OUTPUT |
|---|---|
| 1<br>10<br>200 0 200 0 200 0 200 0 200 0 | 1000 |

| INPUT | OUTPUT |
|---|---|
| 2<br>5<br>10 20 30 40 50<br>12<br>1 2 3 4 5 6 7 8 9 10 11 12 | 150<br>75 |

### Explanation

In the first example we have exactly 10 results, so we just need to sum them. In the second one, the first case has 5 numbers, so the sum is again enough, while the second one has 12, so we need to cut out the worst two scores before summing.

## 1.3 Solution

For each team we only need the sum of the best **up to 10** scores.
So the simplest solution is:

- read the list of scores,
- sort it in decreasing order,
- sum the first $\min(N, 10)$ values.

This is enough for all subtasks: each team has at most $10^3$ scores, so sorting is easily fast enough.
The complexity for one team is $O(N \log N)$, and the memory usage is $O(N)$.

## 1.4 Source Code

### C++

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    cin >> T;

    while (T--) {
        int N;
        cin >> N;
        vector<int> a(N);
        for (int i = 0; i < N; i++) cin >> a[i];

        sort(a.begin(), a.end(), greater<int>());

        long long ans = 0;
        for (int i = 0; i < min(N, 10); i++) ans += a[i];

        cout << ans << "\n";
    }

    return 0;
}
```

### Python

```python
T = int(input())

for _ in range(T):
    N = int(input())
    a = list(map(int, input().split()))
    a.sort(reverse=True)
    print(sum(a[:10]))
```

# 2 SneCC [80 points]

## 2.1 Problem Statement

The game of SneCC is a (bad) copy of the famous Snake game. It is played on a $R \times C$ board, where the top left corner is $(0, 0)$ and the bottom right is $(R - 1, C - 1)$. The cells of the form $(0, y)$, $(R - 1, y)$, $(x, 0)$ and $(x, C - 1)$ are walls. At the start of the game, a snake is placed on a cell $(x, y)$, facing right and has length 1 (i.e., it occupies only that cell). At every second, the snake advances by one cell in the direction it is facing.

Moreover, some cells of the grid contain food. Food, when eaten, increases the length of the snake by one unit: the snake now occupies all the cells it was occupying before plus the one where the food was, and the food disappears.

The game ends when one of the following conditions is met:

- The timer reaches $10^6$ seconds.
- The snake hits a wall.
- The snake hits itself (i.e.: after a move the snake's head is in the same place as one of another of its body parts).

You are given a series of moves that the snake performs during a game, as well as the initial positions of the food. Your job is to find, when the game ends, the final position of the head of the snake and its length.

## 2.2 Problem Details

You are given the dimensions of the grid $R$ and $C$, the starting poing $(x, y)$ of the snake, the number $F$ of food pieces and their corresponding location. It is guaranteed that all the pieces of food are at different locations, that there is no food where the snake starts and that no food is in a wall. Moreover, you are given a number $N$ and a list of $N$ moves. Moves have the form of a pair of an integer and a letter. The integer represents the second in which the move is performed, while the letter, `L` or `R`, represents the direction of the move. For example `1 R` means that at second 1 the snake turns right: if it was facing right, it will now face down.

Important notes:

- At every second, the snake moves forward in the direction it is facing (remember it faces right at start).
- Turning left or right is applied before making a move forward. See the example for clarification.
- Eating food increases the length of the snake at the end of the move.
- The entire snake body moves simultaneously.
- The first subtask contains no food.
- If the game ends by the timer reaching $10^6$ seconds, the snake still makes a move at that second (i.e., the last move happens at second $10^6$ and not $10^6 - 1$).

### Input

The input consists of $4 + F + N$ lines:

- Line 1: the dimensions $R$ and $C$ of the grid, as space separated integers.
- Line 2: the starting position of the snake, as two space separated integers.
- Line 3: the number of food pieces $F$.
- The following $F$ lines contain the coordinates of the food pieces, as space separated integers.
- Line $4 + F$: the number of moves $N$.
- The following $N$ lines contain the moves, as space separated pairs of a number and a letter.

### Output

The output consists of 2 lines: the first line contains the final position of the snake, as two space separated integers. The second line contains its length at the end of the game, as a single integer.

### Scoring

Your program will be tested on a number of testcases grouped in subtasks. In order to obtain the score associated to a subtask, you need to correctly solve all its testcases.

- **Subtask 1** **[30 points]**: $3 \leq R, C \leq 10^3$, $F = 0$, $0 \leq N \leq 200$.
- **Subtask 2** **[30 points]**: $3 \leq R, C \leq 10^3$, $0 \leq F \leq 200$, $0 \leq N \leq 200$.
- **Subtask 3** **[20 points]**: $3 \leq R, C \leq 10^9$, $0 \leq F \leq 10^3$, $0 \leq N \leq 10^3$.

### Examples

| INPUT | OUTPUT |
|---|---|
| 7 7<br>3 3<br>7<br>3 4<br>3 5<br>4 5<br>5 5<br>5 4<br>5 3<br>4 3<br>3<br>3 R<br>5 R<br>7 R | 0 3<br>8 |

### Explanation

In the example, the following happens:

- The snake start at (3,3), facing right.
- At seconds 3,5,7 it turns right, tracing a clockwise loop.
- The snake eats food at seconds 1–7 (one per step), so its length becomes 8 and its tail never moves during these seconds.
- At second 8 it moves into (3,3), a cell that was occupied by its body (the starting cell), but at the same time the tail moves away (so it does not die).
- It goes forward to the wall, where it hits the head at second 11.
- Final head position is (0,3), final length is 8.

## 2.3 Solution

The important observation is that, despite the grid possibly being huge, the game lasts for at most $10^6$ seconds. So we can directly simulate the game second by second.
We maintain:

- the current direction of the snake,
- a deque containing the body of the snake from tail to head,
- a set with the cells currently occupied by the snake,

- a set with the remaining food cells,
- the list of turns, indexed by time.

At each second:

1. if there is a scheduled turn at that second, we apply it;
2. the head moves forward by one cell;
3. if that cell is a wall, the game ends immediately;
4. if the new cell contains food, the snake grows;
5. otherwise, the tail moves away.

To correctly handle the case in which the head moves into the cell that was occupied by the tail, we must remove the tail *before* checking self-collision when no food is eaten. This matches the statement that the whole body moves simultaneously.
Since we do at most $10^6$ iterations, and each one performs only $O(1)$ operations on sets/deques, the total complexity is $O(10^6 + F + N)$.

## 2.4 Source Code

### C++

```cpp
#include <bits/stdc++.h>
using namespace std;

static long long encode(long long x, long long y) {
    return (x << 32) ^ y;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    long long R, C;
    cin >> R >> C;

    long long sx, sy;
    cin >> sx >> sy;

    int F;
    cin >> F;

    unordered_set<long long> food;
    for (int i = 0; i < F; i++) {
        long long x, y;
        cin >> x >> y;
        food.insert(encode(x, y));
    }

    int N;
    cin >> N;

    vector<char> turn(1000001, 0);
    for (int i = 0; i < N; i++) {
        int t;
        char ch;
        cin >> t >> ch;
        if (1 <= t && t <= 1000000) turn[t] = ch;
    }
```

```cpp

    // 0 = right, 1 = down, 2 = left, 3 = up
    int dir = 0;
    long long dx[4] = {0, 1, 0, -1};
    long long dy[4] = {1, 0, -1, 0};

    deque<pair<long long, long long>> body;
    body.push_back({sx, sy});

    unordered_set<long long> occupied;
    occupied.insert(encode(sx, sy));

    long long hx = sx, hy = sy;

    for (int t = 1; t <= 1000000; t++) {
        if (turn[t] == 'L') dir = (dir + 3) % 4;
        else if (turn[t] == 'R') dir = (dir + 1) % 4;

        long long nx = hx + dx[dir];
        long long ny = hy + dy[dir];

        if (nx == 0 || nx == R - 1 || ny == 0 || ny == C - 1) {
            cout << nx << " " << ny << "\n";
            cout << body.size() << "\n";
            return 0;
        }

        long long key = encode(nx, ny);
        bool grow = (food.find(key) != food.end());

        if (!grow) {
            auto [tx, ty] = body.front();
            body.pop_front();
            occupied.erase(encode(tx, ty));
        }

        if (occupied.find(key) != occupied.end()) {
            cout << nx << " " << ny << "\n";
            cout << (body.size() + 1) << "\n";
            return 0;
        }

        body.push_back({nx, ny});
        occupied.insert(key);
        hx = nx;
        hy = ny;

        if (grow) {
            food.erase(key);
        }
    }

    cout << hx << " " << hy << "\n";
    cout << body.size() << "\n";
    return 0;
}
```

### Python

```python
from collections import deque
```

```

R, C = map(int, input().split())
sx, sy = map(int, input().split())

F = int(input())
food = set()
for _ in range(F):
    x, y = map(int, input().split())
    food.add((x, y))

N = int(input())
turn = {}
for _ in range(N):
    t, ch = input().split()
    turn[int(t)] = ch

# 0 = right, 1 = down, 2 = left, 3 = up
dir = 0
dx = [0, 1, 0, -1]
dy = [1, 0, -1, 0]

body = deque()
body.append((sx, sy))
occupied = {(sx, sy)}

hx, hy = sx, sy

for t in range(1, 10**6 + 1):
    if t in turn:
        if turn[t] == 'L':
            dir = (dir + 3) % 4
        else:
            dir = (dir + 1) % 4

    nx = hx + dx[dir]
    ny = hy + dy[dir]

    if nx == 0 or nx == R - 1 or ny == 0 or ny == C - 1:
        print(nx, ny)
        print(len(body))
        exit(0)

    grow = (nx, ny) in food

    if not grow:
        tx, ty = body.popleft()
        occupied.remove((tx, ty))

    if (nx, ny) in occupied:
        print(nx, ny)
        print(len(body) + 1)
        exit(0)

    body.append((nx, ny))
    occupied.add((nx, ny))
    hx, hy = nx, ny

    if grow:
        food.remove((nx, ny))

print(hx, hy)
```

```
63 print(len(body))
```

# 3 Palindromic substrings [80 points]

## 3.1 Problem Statement

You are given a string $s$ of uppercase letters, that can be written in characters as $s_0 s_1 s_2 ...$ . You need to answer a series of queries of the following form: given two non-negative integers $a$ and $b$, is it possible to rearrange the letters in the substring $s_a s_{a+1} ... s_{b-1} s_b$ to form a palindromic string?

*Note: a string is called palindromic if the string obtained by reversing it is the same as the original one.*

## 3.2 Problem Details

It is guaranteed that $a \leq b$ for each query, and that both of them are less than the length of the string $s$. Notice that both $a$ and $b$ are included in the substring and that indexing starts from 0.

### Input

The input consists of $3 + q$ lines:

- Line 1: the length of the string $s$
- Line 2: the string $s$ itself
- Line 3: the number of queries $q$
- Lines $4, ..., 3 + q$: two space separated integers, $a$ and $b$, for each line

### Output

The output consists of $q$ lines, each containing the string `YES`, if such rearrangement exists, or `NO`, if it does not.

### Scoring

Your program will be tested on a number of testcases grouped in subtasks. In order to obtain the score associated to a subtask, you need to correctly solve all its testcases.

Denoting with $L$ the length of $s$:

- **Subtask 1** [**20 points**]: $L \leq 500$, $q \leq 200$
- **Subtask 2** [**30 points**]: $L \leq 10^5$, $q \leq 10^3$
- **Subtask 3** [**30 points**]: $L \leq 10^6$, $q \leq 10^6$

### Examples

| INPUT | OUTPUT |
|---|---|
| 10<br>AZAZBBBBCC<br>6<br>0 4<br>7 9<br>4 9<br>2 7<br>8 9<br>1 8 | YES<br>YES<br>YES<br>NO<br>YES<br>NO |

**Explanation**

- The first substring is `AZAZB`, which can be rearranged as `AZBZA` to become a palindrome string, so the answer is `YES`.
- The second substring is `BCC`, which can be rearranged as `CBC` to become a palindrome string, so the answer is `YES`.
- The third substring is `BBBBCC`, which can be rearranged as `BBCCBB` to become a palindrome string, so the answer is `YES`.
- The fourth substring is `AZBBBB`, which cannot be rearranged to become a palindrome string, so the answer is `NO`.
- The fifth substring is `CC`, which already is a palindrome, so the answer is `YES`.
- The sixth substring is `ZAZBBBBC`, which cannot be rearranged to become a palindrome string, so the answer is `NO`.

## 3.3 Solution

A string can be rearranged into a palindrome if and only if **at most one** character appears an odd number of times.
So for each query we do not need the exact frequencies: we only need, for each letter, whether its count in the substring is even or odd.
Since the alphabet contains only uppercase letters, we can store this information inside a bitmask of 26 bits:

- bit $i$ is 1 if the corresponding letter appears an odd number of times,
- bit $i$ is 0 otherwise.

Let `pref[i]` be the parity mask of the prefix $s_0 s_1 \dots s_{i-1}$. Then the parity mask of the substring $[a, b]$ is simply

`pref[b+1] XOR pref[a]`.

The answer is:

- `YES` if the resulting mask has at most one bit set,
- `NO` otherwise.

Building the prefix masks takes $O(L)$ time, and each query is answered in $O(1)$ time.

## 3.4 Source Code

### C++

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int L;
    cin >> L;

    string s;
    cin >> s;

    int q;
    cin >> q;

    vector<unsigned int> pref(L + 1, 0);
    for (int i = 0; i < L; i++) {
```

```cpp
        pref[i + 1] = pref[i] ^ (1u << (s[i] - 'A'));
    }

    while (q--) {
        int a, b;
        cin >> a >> b;
        unsigned int mask = pref[b + 1] ^ pref[a];
        if (__builtin_popcount(mask) <= 1) cout << "YES\n";
        else cout << "NO\n";
    }

    return 0;
}
```

### Python

```python
L = int(input())
s = input().strip()
q = int(input())

pref = [0] * (L + 1)
for i, ch in enumerate(s):
    pref[i + 1] = pref[i] ^ (1 << (ord(ch) - ord('A')))

for _ in range(q):
    a, b = map(int, input().split())
    mask = pref[b + 1] ^ pref[a]
    if mask.bit_count() <= 1:
        print("YES")
    else:
        print("NO")
```

# 4 Hamiltonian Grid [100 points]

## 4.1 Problem Statement

In a $R \times C$ grid ($R \leq 2$), numbered with $(0, 0)$ on the top left and $(R-1, C-1)$ on the bottom right, a non-negative integer is written in each cell, with the number on the $(0, 0)$ cell always being 0. A pawn is sitting in the $(0, 0)$ cell and, every second, it can do one of the following actions:

- Move horizontally or vertically, but not diagonally (i.e., incrementing or decrementing exactly one of its coordinates, when possible).
- Wait in that position without moving.

The pawn wants to visit each cell **exactly one time**, but it can visit a cell only if the number of seconds passed from the beginning is strictly higher than the number written in it. What is the minimum time required to visit all the cells?

### Input

The input consists of the following lines:

- Line 1: an integer, $T$, representing the number of testcases.
- The following lines describe all $T$ testcases. For each testcase:
  - The first line contains two space separated integers, $R$ and $C$, representing the number of rows and columns of the grid.
  - The following $R$ lines contain a space-separated list of $C$ integers each. It is guaranteed that the first number of the first of these lines is 0.

### Output

The output consists of $T$ lines, each of them representing the answer to one of the testcases.

### Scoring

Your program will be tested on a number of testcases grouped in subtasks. In order to obtain the score associated to a subtask, you need to correctly solve all its testcases.

- **Subtask 1** [**20 points**]: $1 \leq T \leq 100$, $R = 1$, $C \leq 10^3$
- **Subtask 2** [**40 points**]: $1 \leq T \leq 100$, $1 \leq R \leq 2$, $C \leq 10^3$
- **Subtask 3** [**40 points**]: $1 \leq T \leq 100$, $1 \leq R \leq 2$, $C \leq 10^5$

### Examples

| INPUT | OUTPUT |
|---|---|
| 3<br>1 6<br>0 2 10 10 9 7<br>1 10<br>0 99 25 49 31 12 58 74 63 77<br>1 10<br>0 0 0 0 0 0 0 0 0 0 | 14<br>108<br>9 |

| INPUT | OUTPUT |
|---|---|
| 2<br>2 4<br>0 51 16 89<br>79 32 85 28<br>2 14<br>0 545 641 33 840 459 455 879 374 17 725 106 792 217<br>978 750 965 661 991 221 855 364 164 371 987 731 887 781 | 90<br>998 |

**Explanation**

- In the first testcase of the first input file, the pawn can move to the second cell at second 3, then it will have to wait until second 11 to move to the third one. Since 10 is the maximum of the grid, after it it will move every second, reaching the end at second 14.
- In the second testcase of the first input file, the pawn can move to the second cell at second 100. Since 99 is the maximum of the grid, after it it will move every second, reaching the end at second 108.
- In the third testcase of the first input file, the pawn can already move through all cells at second 1, reaching the end at second 9.
- In the first testcase of the second input file, the highest number is 89, which is on the top right corner of the grid, which will be therefore left as the last one to visit. The pawn can start moving down at second 80; then it will move right, up and right again without waiting. Then it will wait until second 86 to go down again, then right and it will finally wait until second 90 to visit the last cell and reach the end.
- In the second testcase of the second input file, the best strategy is to proceed straight to the right until the end of the grid, then go down and then head left until the last cell is visited. The pawn will wait until second 546 to visit the first cell, then until 642 for the next cell on the right; then it will go straight right one cell and wait again until 841 for the next one; then it will go straight right two cells in a row, waiting again until 880 for the following right cell. It will then be able to visit all cells in the first row without waiting, reaching the last cell on the top right corner at second 886. Then it will go down and then left two times, having to wait until second 988 to visit the following left cell. After that, it will just be able to proceed going left without waiting, reaching the end at second 998.

## 4.2 Solution

We consider the starting cell $(0, 0)$ as already visited at time 0, exactly as in the examples.

**Subtask 1:** $R = 1$ With only one row there is only one possible order: we must visit the cells from left to right.
If we are currently at time `cur` and the next cell contains the number `x`, then the earliest time at which we can visit it is

$$\max(\texttt{cur} + 1,\ x + 1).$$

So we just scan the row from left to right and update the current time.

**Subtasks 2 and 3:** $R \leq 2$ For a $2 \times C$ grid, every Hamiltonian path starting from the top-left corner has a very rigid shape.
Choose a column $p$:

- all columns strictly before $p$ are visited with a forced zig-zag,
- then we move straight to the right on one row from column $p$ to column $C - 1$,
- then we switch row in the last column,
- finally we move straight to the left on the other row until column $p$.

So there are only $C$ candidate Hamiltonian paths. Subtask 2 can be solved easily by enumerating them, with a time complexity of $O(C^2)$. With some optimization this is enough also for subtask 3. We illustrate however a more efficient solution for this last subtask.

Fix one of the paths, and let the visited cells be

$$v_0, v_1, \ldots, v_{2C-1},$$

where $v_0 = (0, 0)$.
If a cell $v_i$ (with $i \geq 1$) contains the value $a[v_i]$, then:

- we cannot visit it before time $a[v_i] + 1$,
- after visiting it, there are exactly $(2C - 1 - i)$ moves left.

Therefore the final time of this path is

$$\max\Big(2C - 1,\ \max_{i \geq 1}\big(a[v_i] + 1 + (2C - 1 - i)\big)\Big).$$

So for each candidate path we only need the position of every cell inside the order. Using the structure of the path, the cells split into:

- a zig-zag prefix,
- a suffix on the row used to go right,
- a suffix on the row used to come back left.

This lets us precompute:

- prefix maxima for the zig-zag part,
- suffix maxima of `value - column`,
- suffix maxima of `value + column`,

and evaluate each of the $C$ candidate paths in $O(1)$ time.
Hence the whole testcase is solved in $O(C)$ time and $O(C)$ memory.

## 4.3 Source Code

### C++

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    cin >> T;

    const long long NEG = -(1LL << 60);

    while (T--) {
        int R, C;
        cin >> R >> C;

        if (R == 1) {
            vector<long long> a(C);
            for (int i = 0; i < C; i++) cin >> a[i];

            long long cur = 0;
```

```
            for (int i = 1; i < C; i++) {
                cur = max(cur + 1, a[i] + 1);
            }

            cout << cur << "\n";
            continue;
        }

        vector<long long> top(C), bot(C);
        for (int i = 0; i < C; i++) cin >> top[i];
        for (int i = 0; i < C; i++) cin >> bot[i];

        long long total_cells = 2LL * C;
        long long base = total_cells - 1;

        vector<long long> colmax(C, NEG);
        for (int c = 0; c < C; c++) {
            if (c % 2 == 0) {
                long long a = (c == 0 ? NEG : top[c] + total_cells - 2LL * c);
                long long b = bot[c] + total_cells - 2LL * c - 1;
                colmax[c] = max(a, b);
            } else {
                long long b = bot[c] + total_cells - 2LL * c;
                long long a = top[c] + total_cells - 2LL * c - 1;
                colmax[c] = max(a, b);
            }
        }

        vector<long long> pref(C + 1, NEG);
        for (int i = 1; i <= C; i++) {
            pref[i] = max(pref[i - 1], colmax[i - 1]);
        }

        vector<long long> sufTopMinus(C + 1, NEG), sufBotMinus(C + 1, NEG);
        vector<long long> sufTopPlus(C + 1, NEG), sufBotPlus(C + 1, NEG);

        for (int c = C - 1; c >= 0; c--) {
            long long topMinus = (c == 0 ? NEG : top[c] - c);
            long long botMinus = bot[c] - c;
            long long topPlus = top[c] + c;
            long long botPlus = bot[c] + c;

            sufTopMinus[c] = max(sufTopMinus[c + 1], topMinus);
            sufBotMinus[c] = max(sufBotMinus[c + 1], botMinus);
            sufTopPlus[c] = max(sufTopPlus[c + 1], topPlus);
            sufBotPlus[c] = max(sufBotPlus[c + 1], botPlus);
        }

        long long ans = (1LL << 62);

        for (int p = 0; p < C; p++) {
            long long cur = max(base, pref[p]);

            if (p % 2 == 0) {
                if (sufTopMinus[p] != NEG) {
                    cur = max(cur, total_cells - p + sufTopMinus[p]);
                }
                cur = max(cur, 1LL - p + sufBotPlus[p]);
            } else {
                cur = max(cur, total_cells - p + sufBotMinus[p]);
                cur = max(cur, 1LL - p + sufTopPlus[p]);
            }
```

```
            ans = min(ans, cur);
        }

        cout << ans << "\n";
    }

    return 0;
}
```

### Python

```python
import sys

input = sys.stdin.readline
NEG = -10**30

T = int(input())

for _ in range(T):
    R, C = map(int, input().split())

    if R == 1:
        a = list(map(int, input().split()))
        cur = 0
        for i in range(1, C):
            cur = max(cur + 1, a[i] + 1)
        print(cur)
        continue

    top = list(map(int, input().split()))
    bot = list(map(int, input().split()))

    total_cells = 2 * C
    base = total_cells - 1

    colmax = [NEG] * C
    for c in range(C):
        if c % 2 == 0:
            a = NEG if c == 0 else top[c] + total_cells - 2 * c
            b = bot[c] + total_cells - 2 * c - 1
            colmax[c] = max(a, b)
        else:
            b = bot[c] + total_cells - 2 * c
            a = top[c] + total_cells - 2 * c - 1
            colmax[c] = max(a, b)

    pref = [NEG] * (C + 1)
    for i in range(1, C + 1):
        pref[i] = max(pref[i - 1], colmax[i - 1])

    suf_top_minus = [NEG] * (C + 1)
    suf_bot_minus = [NEG] * (C + 1)
    suf_top_plus = [NEG] * (C + 1)
    suf_bot_plus = [NEG] * (C + 1)

    for c in range(C - 1, -1, -1):
        top_minus = NEG if c == 0 else top[c] - c
        bot_minus = bot[c] - c
        top_plus = top[c] + c
```

```
        bot_plus = bot[c] + c

        suf_top_minus[c] = max(suf_top_minus[c + 1], top_minus)
        suf_bot_minus[c] = max(suf_bot_minus[c + 1], bot_minus)
        suf_top_plus[c] = max(suf_top_plus[c + 1], top_plus)
        suf_bot_plus[c] = max(suf_bot_plus[c + 1], bot_plus)

    ans = 10**30

    for p in range(C):
        cur = max(base, pref[p])

        if p % 2 == 0:
            if suf_top_minus[p] != NEG:
                cur = max(cur, total_cells - p + suf_top_minus[p])
            cur = max(cur, 1 - p + suf_bot_plus[p])
        else:
            cur = max(cur, total_cells - p + suf_bot_minus[p])
            cur = max(cur, 1 - p + suf_top_plus[p])

        ans = min(ans, cur)

    print(ans)
```