Contest 1 Editorial

COMP4128 21T3

19th September 2021

A. Balance

Algorithm

We use brute force.

Read the input into a 2D array. We can calculate the maximum and minimum (and hence the imbalance) of each of the m rows in $O(n)$ time each, and each of the n columns in $O(m)$ time each. Then the greatest imbalance among the rows is found in $O(m)$ time, and the greatest imbalance among columns in $O(n)$ time. The answer is determined by comparing these two quantities.

This algorithm runs in $O(mn)$ time, which is sufficient for $m, n \leq 1,000$.

Implementation Notes

A small speedup is achieved by making one pass through the grids to find maxima and minima in each row and column, as in the reference solution. Looping through a 2D array by columns is a constant factor slower than looping by rows because the entries of a row are stored contiguously in memory. This is described in the tips page of the course website, under the heading 'Nested loops'.

Since each number in the grid has absolute value at most 10^9, the difference between any two numbers is at most 2×10^9. This is less than 2^{31}, so none of the values calculated in our algorithm overflow a 32-bit signed integer. Therefore `long long` variables are not required; `int` is sufficient.

Reference Solution

```
// Solution by Kevin
#include <climits>
#include <iostream>
#include <cstdio>
using namespace std;

const int N = 1005;
int a[N][N];
int row_mins[N];
int row_maxs[N];
int col_mins[N];
int col_maxs[N];

int main () {
    // scan in grid
    int m, n;
    cin >> m >> n;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cin >> a[i][j];
        }
    }

    // initialise arrays
    for (int i = 0; i < m; i++) {
        row_mins[i] = INT_MAX;
        row_maxs[i] = INT_MIN;
    }
    for (int i = 0; i < n; i++) {
        col_mins[i] = INT_MAX;
        col_maxs[i] = INT_MIN;
    }

    // work out mins and maxs of each row and column
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            row_mins[i] = min(row_mins[i], a[i][j]);
            row_maxs[i] = max(row_maxs[i], a[i][j]);
            col_mins[j] = min(col_mins[j], a[i][j]);
            col_maxs[j] = max(col_maxs[j], a[i][j]);
        }
    }

    // work out imbalances on the rows and columns
    int row_imbalance = INT_MIN;
    int col_imbalance = INT_MIN;
    for (int i = 0; i < m; i++) {
        row_imbalance = max(row_imbalance, row_maxs[i] - row_mins[i]);
    }
    for (int i = 0; i < n; i++) {
        col_imbalance = max(col_imbalance, col_maxs[i] - col_mins[i]);
    }

    // get final output
    if (row_imbalance == col_imbalance) printf("YES\n");
    else printf("NO\n");
}
```

B. Matchup

Algorithm

We use the greedy method.

Claim: Martha can pair her weakest player with the weakest player of the opposing team.

Proof: Suppose there is an arrangement in which Martha's team wins all games, but her weakest player p_1 plays an opponent p'_1 while the weakest opponent p'_2 is assigned to Martha's player p_2. Then we can swap p_1 and p_2 to obtain an arrangement in which Martha's team still wins all games, but the weakest players face each other.

It is therefore sufficient to test whether Martha's weakest player can beat the opponent's weakest player, whether Martha's second weakest player can beat the opponent's second weakest player, and so on. We can sort both arrays, and make one simultaneous pass of both sorted arrays, reporting "NO" if Martha's team loses or draws any game or "YES" otherwise.

Sorting takes $O(n \log n)$ and the rest of the algorithm takes $O(n)$, so the algorithm runs in $O(n \log n)$ time, which is sufficient for $n \leq 100,000$.

Implementation Notes

The C++ Standard Template Library provides a `sort` function in the `<algorithm>` header, which can be used with the default comparator `<` or a custom comparator. No particular sorting algorithm is required by the language standard, but the function is guaranteed to have worst case time complexity of $O(n \log n)$. `sort` is not stable; if stable sorting is required, use the `stable_sort` function (also in `<algorithm>`.

Again, all values used in this problem fit in an `int` variable without overflow.

Some students instead created two self-balancing binary search trees containing the strengths of each team's players, and iterated through both simultaneously. A correct implementation of this approach requires `multiset` from the `<set>` header. If one were to use `set` instead, duplicate entries would only be inserted once, leading to a `WRONG-ANSWER` or `RUN-ERROR`.

Reference Solution

```
// Solution by Raveen
#include <algorithm>
#include <iostream>
using namespace std;

const int N = 100100;
int a[N];
int b[N];

int main (void) {
  int n;
  cin >> n;
  for (int i = 0; i < n; i++)
    cin >> a[i];
  for (int i = 0; i < n; i++)
    cin >> b[i];

  sort(a,a+n);
  sort(b,b+n);

  for (int i = 0; i < n; i++)
    if (a[i] <= b[i]) {
      cout << "NO\n";
      return 0;
    }

  cout << "YES\n";
}
```

C. Control

Algorithm

We use brute force.

Each square is either uncontrolled, occupied by a piece, or attacked (i.e. a piece can move directly to this square). First, read the input into a 2D array to determine the occupied squares and the type of each piece. Next, we determine the squares attacked by each piece.

- For rooks, we step one square at a time in each of the four allowed directions (up, down, left, right), marking each square as attacked until we reach an occupied square or go off the edge of the board.
- For bishops, we step one square at a time in each of the four allowed directions (up left, up right, down left, down right) and mark squares as above.
- For queens, we step one square at a time in each of the eight allowed directions (up, down, left, right, up left, up right, down left, down right) and mark squares as above.
- For knights, we try all eight squares to which the knight can move, ignoring any which are already occupied or off the edge of the board.

Finally, we count all squares which are either occupied or attacked.

There are up to $O(n^2)$ pieces. For each piece, we step through $O(n)$ squares, each taking constant time. Therefore the algorithm runs in $O(n^3)$ time, which is sufficient for $n \leq 100$.

Implementation Notes

It is necessary to read the entire input before determining which squares are attacked, as the scope of a piece may be blocked by another piece.

It may be convenient to write a helper function which takes a row and column index and determines whether the referenced square is eligible to be attacked (i.e. not occupied and not off the edge of the board).

One reference solution uses a pair of constant arrays to facilitate easy lookup of the vertical and horizontal steps required.

Be *very* careful when writing two very similar blocks of code. It is tempting to write one block, copy and paste, and then edit the second block. However this is a very common source of bugs, which can end up costing you a lot of time during a contest. If you can spare the keyboard time, we generally recommend typing the second block out in full.

Reference Solutions

```
// Solution by Angus
#include <cstdio>
using namespace std;
#define MAXN 110
int n, k, occupied[MAXN][MAXN], controlled[MAXN][MAXN], x[MAXN*MAXN], y[MAXN*MAXN];
char type[MAXN*MAXN];

void mark(int i, int j) {
        if (i >= 1 && j >= 1 && i <= n && j <= n) controlled[i][j] = true;
}

void process_knight(int i, int j) {
        mark(i, j);
        mark(i-2, j-1);
        mark(i+2, j-1);
        mark(i-2, j+1);
        mark(i+2, j+1);
        mark(i-1, j-2);
        mark(i+1, j-2);
        mark(i-1, j+2);
        mark(i+1, j+2);
}
```

```
void process_rook(int i, int j) {
        mark(i, j);
        for (int x = i+1; x <= n; x++) {
                if (occupied[x][j]) break;
                mark(x, j);
        }
        for (int x = i-1; x >= 1; x--) {
                if (occupied[x][j]) break;
                mark(x, j);
        }
        for (int y = j+1; y <= n; y++) {
                if (occupied[i][y]) break;
                mark(i, y);
        }
        for (int y = j-1; y >= 1; y--) {
                if (occupied[i][y]) break;
                mark(i, y);
        }
}

void process_bishop(int i, int j) {
        mark(i, j);
        for (int x = i+1, y = j+1; x <= n && y <= n; x++, y++) {
                if (occupied[x][y]) break;
                mark(x, y);
        }
        for (int x = i-1, y = j+1; x >= 1 && y <= n; x--, y++) {
                if (occupied[x][y]) break;
                mark(x, y);
        }
        for (int x = i+1, y = j-1; x <= n && y >= 1; x++, y--) {
                if (occupied[x][y]) break;
                mark(x, y);
        }
        for (int x = i-1, y = j-1; x >= 1 && y >= 1; x--, y--) {
                if (occupied[x][y]) break;
                mark(x, y);
        }
}

void process_queen(int i, int j) {
        process_rook(i, j);
        process_bishop(i, j);
}

int main() {
        scanf("%d%d", &n, &k);
        for (int i = 0; i < k; i++) {
                scanf(" %c%d%d", &type[i], &x[i], &y[i]);
                occupied[x[i]][y[i]] = true;
        }

        for (int i = 0; i < k; i++) {
                if (type[i] == 'Q') process_queen(x[i], y[i]);
                else if (type[i] == 'R') process_rook(x[i], y[i]);
                else if (type[i] == 'B') process_bishop(x[i], y[i]);
                else if (type[i] == 'N') process_knight(x[i], y[i]);
        }

        int ans = 0;
        for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n; j++) {
                        ans += controlled[i][j];
                }
        }
        printf("%d\n", ans);
}
```

```
// Solution by Raveen
#include <iostream>
#include <vector>
using namespace std;
```

```
int n, k;
vector<int> qx, qy, rx, ry, bx, by, nx, ny;
int nq, nr, nb, nn;
const int N = 110;
int ctrl[N][N]; // 0 is free, 1 is occupied, 2 is attacked

// indices 0-3 are rook moves U D L R
// indices 4-7 are bishop moves UL UR DL DR
// indices 8-15 are knight moves
//                 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
const int dx[] = { 0, 0,-1, 1,-1, 1,-1, 1,-2,-1, 1, 2, 2, 1,-1,-2};
const int dy[] = {-1, 1, 0, 0,-1,-1, 1, 1, 1, 2, 2, 1,-1,-2,-2,-1};

bool attackable (int x, int y) {
  return x >= 1 && x <= n && y >= 1 && y <= n && ctrl[x][y] != 1;
}

int main (void) {
  cin >> n >> k;
  for (int i = 0; i < k; i++) {
    char ch;
    int x, y;
    cin >> ch >> x >> y;
    ctrl[x][y] = 1;
    if (ch == 'Q') {
      qx.push_back(x);
      qy.push_back(y);
      nq++;
    } else if (ch == 'R') {
      rx.push_back(x);
      ry.push_back(y);
      nr++;
    } else if (ch == 'B') {
      bx.push_back(x);
      by.push_back(y);
      nb++;
    } else if (ch == 'N') {
      nx.push_back(x);
      ny.push_back(y);
      nn++;
    }
  }

  for (int i = 0; i < nq; i++) {
    int x = qx[i], y = qy[i];
    for (int j = 0; j < 8; j++) {
      int _x = x, _y = y;
      while (true) {
        _x += dx[j];
        _y += dy[j];
        if (!attackable(_x,_y))
          break;
        ctrl[_x][_y] = 2;
      }
    }
  }

  for (int i = 0; i < nr; i++) {
    int x = rx[i], y = ry[i];
    for (int j = 0; j < 4; j++) {
      int _x = x, _y = y;
      while (true) {
        _x += dx[j];
        _y += dy[j];
        if (!attackable(_x,_y))
          break;
        ctrl[_x][_y] = 2;
      }
    }
  }

  for (int i = 0; i < nb; i++) {
    int x = bx[i], y = by[i];
    for (int j = 4; j < 8; j++) {
      int _x = x, _y = y;
      while (true) {
        _x += dx[j];
        _y += dy[j];
        if (!attackable(_x,_y))
          break;
        ctrl[_x][_y] = 2;
      }
    }
  }

  for (int i = 0; i < nn; i++) {
    int x = nx[i], y = ny[i];
    for (int j = 8; j < 16; j++) {
      int _x = x + dx[j];
      int _y = y + dy[j];
      if (!attackable(_x,_y))
        continue;
      ctrl[_x][_y] = 2;
    }
  }

  int ans = 0;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      if (ctrl[i][j])
        ans++;
  cout << ans << '\n';
}
```

D. Election

Algorithm

We use brute force.

Read the input, storing each candidate's preference list as one row in a 2D array.

For each candidate i, we maintain:

- v_i, the number of votes currently belonging to candidate i, and
- e_i, a boolean which is true if candidate i has been eliminated and false otherwise.

For each voter j, we maintain:

- c_j, the candidate to which voter j is currently assigned, and
- p_j, which preference of voter j is candidate c_j.

Initially, we assign each voter to their first preference candidate and initialise the above variables accordingly.

In each round, we:

- test whether any candidate has a majority, by checking each value v_i;
- if not, find the candidate with the fewest votes, by checking each value v_i and breaking ties by i;
- marking that candidate as eliminated;
- for each voter j currently assigned to that candidate, iterate forward from position p_j in their preference list until a non-eliminated candidate is reached, and update c_j as well as the vote counts v_i.

As one candidate is eliminated each round, the algorithm is guaranteed to terminate within k rounds. In each round:

- checking for a majority takes $O(k)$ time;
- identifying a candidate for elimination takes $O(k)$ time;
- identifying the eliminated candidates' voters takes $O(n)$ time;
- reassigning each of these voters appears to take $O(k)$ time (as we may have to iterate through the entire preference list), but in fact it is *amortized* $O(1)$ because each voter is reassigned $O(k)$ times across all rounds.

Therefore the algorithm runs in $O(nk)$ time, which is sufficient for $n \leq 10,000$ and $k \leq 100$.

The algorithm could be optimised further by keeping an array of each candidate's voters, rather than only the count v_i. This allows us to skip the $O(n)$ step of identifying which voters are assigned to the newly eliminated candidate. This improves the constant factor but not the asymptotics. While we only reassign at most n/k voters in the first round, $n/(k-1)$ voters in the second round, and so on in the form of a harmonic series, each reassignment is no longer amortised constant time. This can be seen in the following test case devised by Kevin.

```
6 10
10 9 8 7 6 5 4 3 2 1
10 9 8 7 6 5 4 3 2 1
10 9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1 10
9 8 7 6 5 4 3 2 1 10
9 8 7 6 5 4 3 2 1 10
```

In the first eight rounds, candidates 1 to 8 are eliminated. In the penultimate round however, candidate 9 is eliminated, and each of their $n/2$ voters has to be reassigned in $O(k)$ time. In this way, the worst case performance of the algorithm is still $O(nk)$.

Implementation Notes

Some students got `WRONG-ANSWER` due to incorrect handling of the tiebreaker condition for elimination. Note that there is no need for a tiebreaker condition when checking for a majority, since it is impossible for two different candidates to have a majority simultaneously.

Some students encountered errors in checking for a winner. Recall that in C/C++, `a/b` finds the quotient when `a` is divided by `b`, i.e. it rounds down.

Most solutions to this problem (including the reference solutions) used nested array lookups, which could be mistyped or otherwise buggy. The simplest way to identify and correct such bugs is to make additional test cases and print out lots of well-formatted debugging output (such as the contents of arrays v, e, c and p in each round), then carefully check that your program is behaving as intended. The sample cases are given only to confirm your understanding of the problem, so they are often insufficient for debugging purposes.

Reference Solution

```
// Solution by Raveen
#include <iostream>
using namespace std;

const int N = 100100;
const int K = 110;
int a[N][K];

int v[K]; // how many voters does this candidate have
bool e[K]; // has this candidate been eliminated
int c[N]; // which candidate has this voter
int p[N]; // which preference is this voter up to

int main (void) {
  int n, k;
  cin >> n >> k;

  int x;
        for (int i = 0; i < n; i++)
                for (int j = 0; j < k; j++) {
                        cin >> x;
                        a[i][j] = x-1;
                }

        for (int i = 0; i < n; i++)
                v[a[i][0]]++;

        for (int i = 0; i < n; i++)
                c[i] = a[i][0];

  while (true) {
    for (int j = 0; j < k; j++)
      if (v[j]*2 > n) {
        cout << j+1 << '\n';
        return 0;
      }

    int lo = n, lo_id = k;
    for (int j = 0; j < k; j++)
      if (!e[j] && v[j] < lo) {
        lo = v[j];
        lo_id = j;
      }

    e[lo_id] = true;
    v[lo_id] = 0;

    for (int i = 0; i < n; i++)
      if (c[i] == lo_id) {
        do {
          p[i]++;
        } while (e[a[i][p[i]]]);
        c[i] = a[i][p[i]];
```

```
            v[c[i]]++;
        }
    }
}
```

E. Illuminate

Algorithm

We use the greedy method.

If no light has $\ell_i \leq L$, we are unable to cover the first house and therefore there is no solution. Suppose instead that such a light exists.

Claim 1: The first light we use should have largest r_i among all lights with $\ell_i \leq L$.

Proof: Suppose a solution can be formed without using such a light. This solution must cover house L, using some light corresponding to an interval $[\ell_j, r_j]$ where $r_j < r_i$. Swapping this light for one covering $[\ell_i, r_i]$ constructs a solution which has the same number of lights and satisfies the above claim.

Claim 2: Suppose light i with interval $[\ell_i, r_i]$ is as per the previous claim. Then no other light j with $r_j \leq r_i$ should be used.

Proof: A solution containing light i and light j is still valid if we omit light j, and now uses strictly fewer lights.

Once we select one light satisfying Claim 1, we can discard all other lights with $\ell_i \leq L$. It remains to cover $[r_i + 1, R]$ by repeating this process.

To ensure that we consider only those lights with $\ell_i \leq L$, we can sort the lights by start point in $O(n \log n)$ time. As we scan through the list of lights, each light is handled only once and in constant time, so the main part of the algorithm runs in $O(n)$. Therefore the algorithm runs in $O(n \log n)$ overall, which is sufficient for $n = 100,000$.

Note that sorting instead by end point doesn't work; we don't know when to stop looking for the first light of the answer and move on to the next, so we may have to scan the whole list for each light comprising the answer.

Implementation Notes

The C++ STL `sort` uses the default comparator `<` unless otherwise specified. When comparing two `pairs` `a = (a.first, a.second)` and `b = (b.first, b.second)`, `a < b` is true if:

- `a.first < b.first` or
- `a.first = b.first` and `a.second < b.second`.

So if we store the lights in an array of pairs, with first entry ℓ_i and second entry r_i, the default sort will be ascending by start point with ties broken in ascending order of end point, which is fine in this problem. However, other sorted orders can also be obtained without needing to write a custom comparator, if one is willing to use tricks.

- To sort by endpoint, construct the pairs in the other order, i.e. (r, ℓ).
- To sort by descending order, we can sort the array and then use `reverse`. But what about sorting by ascending order of ℓ_i with ties broken in *descending* order of r_i? Here we can construct the pairs as $(\ell_i, -r_i)$, sort, and then negate the second entry of each pair to restore positive values. It is easy to make mistakes doing this, so use at your own risk and consider including debugging output to ensure that the behaviour is as intended.

There are two slightly different approaches to scanning the list. The first reference solution below uses a `for` loop, which treats each new light that extends the currently covered range as a temporary part of the solution and overwrites it if a better choice is encountered later. The second reference solution is more explicit in this construction, using a `while` loop in which each iteration corresponds to the selection of one new light.

Optionally, one can filter out any lights which do not overlap with the range $[L, R]$.

Values up to 10^{18} can be stored in a `long long`.

Reference Solutions

```
// Solution by Paula
#include <algorithm>
#include <iostream>
#include <utility>
using namespace std;

const int N = 100100;
pair<long long, long long> lights[N];

int main(void) {
  int _n;
  long long m, L, R;
  cin >> _n >> m >> L >> R;
  int n = 0;
  for (int i = 0; i < _n; i++) {
    long long l, r;
    cin >> l >> r;
    if (r >= L || l <= R)
      lights[n++] = make_pair(l,r);
  }

  sort(lights, lights + n);

  int ans = 1;
  bool impossible = false;
  long long currEnd = L;
  long long maxNext = 0;

  for (int i = 0; i < n; i++) {
    long long cL = lights[i].first;
    long long cR = lights[i].second;

    if (cL > currEnd) {
      if (cL > maxNext + 1) {
        impossible = true;
        break;
      } else {
        ans++;
        currEnd = maxNext + 1;
      }
    }
    maxNext = max(maxNext, cR);
    if (maxNext >= R)
      break;
  }
  if (impossible || maxNext < R)
    cout << "-1\n";
  else
    cout << ans << '\n';
}
```

```
// Solution by Raveen
#include <algorithm>
#include <iostream>
#include <utility>
using namespace std;

const int N = 100100;
pair<long long, long long> lights[N];

int main (void) {
  int n;
  long long m, L, R;
  cin >> n >> m >> L >> R;

  for (int i = 0; i < n; i++) {
    long long l, r;
    cin >> l >> r;
    lights[i].first = l;
    lights[i].second = r;
  }

  sort(lights, lights + n);

  int upto = 0;
  long long covered = L-1;
  int ans = 0;
  while (covered < R) {
    if (upto == n) {
      cout << "-1\n";
      return 0;
    }
    long long best = covered;
    while (upto < n && lights[upto].first <= covered + 1) {
      best = max(best,lights[upto++].second);
    }
    if (best == covered) {
      cout << "-1\n";
      return 0;
    } else {
      covered = best;
      ans++;
    }
  }
  cout << ans << '\n';
}
```