WOBURN CHALLENGE

2015-16 Online Round 3

Solutions

Automated grading is available for these problems at:
wcipeg.com

For problems to this contest and past contests, visit:
woburnchallenge.com

Problem J1: Battle Predictions

As a basic implementation problem, the key is to read the statement very carefully. In this case, it is simply a matter of comparing the correct values in the input with each other and not getting mixed up with what each variable means. The catch here is that to know who wins, we must compare the powers of the superheroes in the *opposite* categories, rather than the same categories. That is, attack values must always be compared with defense values and vice versa.

At that, the following straightforward program can be implemented.

Official Solution (C++)

```
#include <iostream>
using namespace std;

int AB, DB, AS, DS;

int main() {
  cin >> AB >> DB >> AS >> DS;
  if (AB > DS && DB > AS) {
    cout << "Batman" << endl;
  } else if (AS > DB && DS > AB) {
    cout << "Superman" << endl;
  } else {
    cout << "Inconclusive" << endl;
  }
  return 0;
}
```

Problem J2: Electroshock Therapy

Upon studying the diagram given in the description, we see that the wire segments can be counted in several groups. If for each building we are able to compute the length of horizontal segments on its center and top, as well as vertical segments on only its left side, then we can add it all up for each building. After that, we must account for the right side of the last building, which may be added in afterwards to get the answer.

Inspecting the diagram, we see that the number of horizontal segments for each building i (for $i = 1..N$) is equal to $H_i + 1$ (one for each floor, plus the roof). We also see that the number of vertical segments between any two adjacent buildings is equal to the maximum of their heights. So, for the left of the first building and the right of the last building, we can simply take their heights without taking the max. As such, we can loop through the buildings in $O(N)$ time and sum up these values to get the total length of wire.

Once last trick is noticing that we don't even need to store the heights in an array. We just have to always keep track of the height of the previous building and use it to take the max the next time. This yields the following code.

Official Solution (C++)

```
#include <iostream>
using namespace std;

int N, prev, curr, ans = 0;

int main() {
  prev = 0;
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> curr;
    ans += curr + max(prev, curr) + 1;
    prev = curr;
  }
  cout << ans + curr << endl;
  return 0;
}
```

Problem J3: Red Sun Simulator

In this problem, we would like to compute two values: the maximum and minimum number of minutes that Superman can be weakened for by the red sun simulator. These two values are always fixed, and knowing them will make the problem trivial to solve. The observation is that if can be weakened for both the min and max number of minutes, then he can also be weakened by any number of minutes between them. Assuming we know this range, then for every query T_i, we output "`Y`" if and only if T_i is between the min and max number of minutes, inclusive.

We can initialize the min and max to 0, and then iterate over the N minutes while updating them appropriately. Namely, in each minute i, one of three possible situations can be true:

1. Superman must be weakened (the min and max will each increase by 1)
2. Superman may or may not be weakened (only the max will increase by 1)
3. Superman cannot be weakened (nothing is changed)

Since there are up to 5 intervals to consider for each minute, we must decide how to determine whether the weakening of Superman is mandatory (case 1) or possible (case 2).

For Superman's weakening to be mandatory, either the solar frequency of minute i is equal to 1 (and so every value of every interval is a multiple), or every interval for minute i solely consists of a single integer which is a multiple of the solar frequency (convince yourself that it is impossible for two consecutive integers to both be a multiple of another integer greater than 1). In this case, if even a single interval does not have $A_{i,j} = B_{i,j}$ and $A_{i,j}$ as a multiple of F_i, then we know that there is a chance for the RSS to fail at weakening Superman during minute i. Whether Superman's weakening is mandatory can be checked using the AND operation on an initially true Boolean value as the intervals are being inputted (with no added time complexity).

The harder part is checking whether it's at least possible to weaken Superman during that minute. This is directly doable in linear time with respect to the total length of all intervals in minute i. Start with a variable "possible" set to true. Then for each interval j, we can check for multiples of F with the following for-loop in $O(B_{i,j} - A_{i,j})$ time.

```
for (int x = A; x <= B; x++)
  if (F % x == 0)
    possible = true;
```

However, since A and B may be up to 1 billion, this method will clearly not run in time. We can try mitigating this slowness with tricks like looping starting from max(F, A), or even generating the multiples of F to see if each falls between A and B like the following:

```
for (int x = F; x <= B; x += F)
  if (A <= x && x <= B)
    possible = true;
```

But ultimately, it will still be futile since for values like $F = 2$ on intervals $[1, 10^9]$, we still have to loop 500 million times. The insight required to overcome this bottleneck is ultimately this: Suppose we want to get the *smallest* multiple of F greater than A. If we take floor(A / F) and multiply it by F again, then we are guaranteed to get a value is no greater than A (since floor(A / F) $\leq A$ / F) and a multiple of F (since we just multiplied it by F). Simply adding F to the result, we will get the first multiple of F greater than A. Then, we can just check if this is less than or equal to B. This is sufficient to solve the problem (since we know how to determine whether a multiple is in any interval in constant time), but if we want something more compact, then we can play around with the idea to finally deduce that each interval contains at least one multiple of F if and only if floor((A – 1) / F) = floor(B / F).

Case 3 occurs when neither case 1 or case 2 is true, so we can simply do nothing after we determined so. Since every interval and question can be processed in constant time, the above algorithm will have a running time complexity of $O((M_1 + M_2 + ... + M_N) + Q)$. This solution is implemented as follows.

Official Solution (C++)

```
#include <iostream>
using namespace std;

int N, Q, F, M, A, B, T;
int min_win = 0, max_win = 0;

int main() {
  cin >> N >> Q;
  for (int i = 0; i < N; i++) {
    cin >> F >> M;
    bool mandatory = true, possible = false;
    for (int j = 0; j < M; j++) {
      cin >> A >> B;
      mandatory &= (F == 1 || (A == B && A % F == 0));
      possible |= ((A - 1) / F != B / F);
    }
    if (mandatory) {
      min_win++;
      max_win++;
    } else if (possible) {
      max_win++;
    }
  }
  for (int i = 0; i < Q; i++) {
    cin >> T;
    if (min_win <= T && T <= max_win) {
      cout << "Y" << endl;
    } else {
      cout << "N" << endl;
    }
  }
  return 0;
}
```

Problem J4: LexCorp Infiltration

If not for type-2 events requiring the grid to be constantly changed, this would be a relative standard flood-fill problem. As a first step, we can locate all of the control rooms, assigning them an index number from 1 to some integer K (note that $1 \leq K \leq RC/2$), determining their sizes, and remembering which room each empty cell belongs to. This can be accomplished in $O(RC)$ time using either depth first search (recursion) or breadth first search. We can iterate through the cells of the grid in any order, and when we come across an empty cell which hasn't been assigned to a room yet, we should perform a search to locate its entire room by recursively expanding to adjacent, unvisited empty cells on its top, right, bottom, and left sides (if they exist). An easy trick to remove the need for range checks is to initially pad an entire 0-based grid [0 ... R+1][0 ... C+1] with wall tiles before inputting the actual grid into 1-based indices. This effectively creates a “border” around the grid to prevent the search from going out of bounds.

For each round of filling, we can just choose the index as the current number of rooms that have been found so far, starting with 0 for the first room. We will also keep track of the area for each room’s (incrementing it whenever we reach a cell). At the end of filling each room, we record the index of the room associated with its area.

Next, we should determine the initial ranks of the rooms. To do this efficiently, we must first sort the rooms in non-increasing order of size, which can be done in $O(K \log K)$ time. To do this in most languages, we can encapsulate the room data into pairs of (area, index) and define a comparator for the sorting function to compare by the sizes rather than the room indices. Finally, their ranks can be computed in $O(K)$ time with a single sweep through them, ensuring that each room i gets the same rank as the previous room if they're of equal area, or a rank of i otherwise. When there are only type-1 events, nothing more is required – each event i can be processed by looking up the index of the room that cell (A_i, B_i) is part of, and then looking up that room's rank.

However, type-2 events make things trickier, as they can cause various rooms' ranks to change. That being said, there's no need to re-compute the ranks of all rooms – it can be observed that, when a room is eliminated, the only effect is that the ranks of all rooms with strictly smaller sizes decrease by 1. Since there are $O(K)$ such rooms each time, processing all N events has a time complexity of $O(NK)$. Since $N \leq 3000$ and $K \leq RC/2$ ($RC/2$ is as large as $1000 \times 1000/2 = 500{,}000$), that yields roughly $500{,}000 \times 3{,}000 = 1.5$ billion steps. So, even with this observation, looping over all such rooms and updating their ranks is too slow for full marks.

However, this logic can be rearranged as follows: a room's rank is equal to its initial rank, minus the number of strictly larger rooms which have been eliminated so far. Instead of keeping the ranks of every room up to date, we can instead handle each event by looping over all previous type-2 events and counting the number of larger rooms which were targeted in them (taking care to not double-count rooms which were targeted multiple times). This yields a time complexity of $O(RC + N^2)$. Plugging in the upper bounds for R, C and N, we are in the ballpark of $1000 \times 1000 + 3000^2$, or roughly 10 million steps, which is acceptable. This solution is presented on the right.

Finally, it's worth noting that this problem can be solved even more efficiently, in $O(RC + N \log K)$ time, by using an advanced data structure such as a binary indexed tree to speed up the step of counting previously-eliminated larger rooms from $O(N)$ to $O(\log K)$ running time.

Official Solution (C++)

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;

int R, C, N, T, A, B, area;
char G[1005][1005];
bool vis[1005][1005];
int idx[1005][1005], rank[1000005], prev[3005];
vector<pair<int, int> > rooms;

void dfs(int r, int c) {
  if (vis[r][c] || G[r][c] == '#') return;
  vis[r][c] = true;
  idx[r][c] = rooms.size();
  area++;
  dfs(r - 1, c); dfs(r + 1, c);
  dfs(r, c - 1); dfs(r, c + 1);
}

int main() {
  memset(G, '#', sizeof G);
  cin >> R >> C >> N;
  for (int i = 1; i <= R; i++)
    for (int j = 1; j <= C; j++) {
      cin >> G[i][j];
      vis[i][j] = false;
      idx[i][j] = -1;
    }
  for (int i = 1; i <= R; i++)
    for (int j = 1; j <= C; j++)
      if (G[i][j] == '.' && !vis[i][j]) {
        area = 0;
        dfs(i, j);
        rooms.push_back(make_pair(area, rooms.size()));
      }
  sort(rooms.rbegin(), rooms.rend());
  int curr = 0;
  for (int i = 0; i < rooms.size(); i++) {
    if (i == 0 || rooms[i].first != rooms[i - 1].first)
      curr = i;
    rank[rooms[i].second] = curr;
  }
  for (int i = 0; i < N; i++) {
    prev[i] = -1;
    cin >> T >> A >> B;
    int id = idx[A][B];
    if (id == -1 || rank[id] == -1) {
      cout << -1 << endl;
      continue;
    }
    int ans = rank[id];
    for (int j = 0; j < i; j++)
      if (prev[j] >= 0 && prev[j] < rank[id]) ans--;
    cout << ans << endl;
    if (T == 2) {
      prev[i] = rank[id];
      rank[id] = -1;
    }
  }
  return 0;
}
```

Problem S1: Energy Absorption

For this problem, we want to keep Batman's net damage positive, given that his initial damage is Q_i (positive). So, we want to determine after how many punches this will happen, or if it will ever happen. The problem is, we have to be able to compute the answer quickly for many different initial damages.

Start by observing that if Superman hasn't yet stopped punching for some index and the next punch will do positive damage, then Superman is guaranteed to following through with this punch. This is because if Batman's damage is currently positive, then positive damage will always make it more positive, and so there's no reason for Superman to stop. This point is true regardless of the initial damage of Batman. That leads to the observation that only on particular indices will Superman ever stop, regardless of the initial damage of Batman.

We must first find these "key indices" in the sequence of punches – all indices i such that the total damage dealt by the first i punches ($D_1 + D_2 + ... + D_i$) is less than any earlier such total (less than the total damage dealt by the first j punches for any $j < i$). These key indices represent the punches which could possibly cause Batman's damage to become non-negative, meaning that Superman will always stop punching right before one of them (or after all N punches). In particular, if Batman starts with Q_i damage, then we need to find the first key index k such that its total damage $D_1 + D_2 + ... + D_k$ is strictly smaller than $-Q_i$, and we'll know that Superman will perform precisely $k - 1$ punches. If there's no such key index, then Superman will perform all N punches.

Iterating over the key indices for each query to find the appropriate one results in an $O(NM)$ algorithm, which is too slow for full marks. However, the key indices have cumulative damages which are monotonic (strictly decreasing), so we can instead use binary search to improve the time complexity to $O(M \log N)$.

Official Solution (C++)

```
#include <algorithm>
#include <cstdio>
#include <utility>
#include <vector>
using namespace std;

int N, M, D, Q, prev = 0, curr = 0;
vector<pair<int, int> > v;

int main() {
  scanf("%d%d", &N, &M);
  for (int i = 0; i < N; i++) {
    scanf("%d", &D);
    curr -= D;
    if (curr > prev) {
      prev = curr;
      v.push_back(make_pair(curr, i));
    }
  }
  v.push_back(make_pair(2000000001, N));
  sort(v.begin(), v.end());
  for (int i = 0; i < M; i++) {
    scanf("%d", &Q);
    printf("%d\n", lower_bound(v.begin(), v.end(), make_pair(Q + 1, 0))->second);
  }
  return 0;
}
```

Problem S2: Detective Work

The problem of trying to make all of the strings equal can be reduced to the problem of making each *column* of the row of strings equal. So every index (from 1 to *M*) can be handled independently, and we are simply trying to transform that index across all *N* strings to the same letter. For each index, we need to find two values – the minimum number of letter replacements in the column to make that index equal across all *N* strings, and the number of ways to do so.

For a given index, we should consider each possible letter (from A to Z) to transform the entire column to. The cost to transform the column into a given letter is the number of strings which have a non-wildcard letter at the current index which is different than the chosen letter. The minimum cost for the index will then be the minimum of these costs over all 26 possible letters, while the count (error) will be the number of different letters from A to Z which have this minimal cost.

Now that we know how to get the costs and errors for each column, getting an overall answer is easy. The total cost will be the sum of these costs, since to change all of the strings, we can simply change letters in all of the columns independently. From principles of basic combinatorics, we know that the total number of ways for *M* independent events to happen is just the product of the number of ways for each event to happen. So, to get the total error (number of possible words), we just multiply the number of possible ways for each index and modulo it by 1000.

Official Solution (C++)

```
#include <iostream>
using namespace std;

int N, M, cost = 0, error = 1;
string S[1005];

int main() {
  cin >> N >> M;
  for (int i = 0; i < N; i++)
    cin >> S[i];
  for (int j = 0; j < M; j++) {
    int minfreq = 1000000000, total;
    for (char c = 'A'; c <= 'Z'; c++) {
      int freq = 0;
      for (int i = 0; i < N; i++) {
        if (S[i][j] != '?' && S[i][j] != c)
          freq++;
      }
      if (freq < minfreq) {
        minfreq = freq;
        total = 1;
      } else if (freq == minfreq) {
        total++;
      }
    }
    cost += minfreq;
    error = (error * total) % 1000;
  }
  cout << cost << " " << error << endl;
  return 0;
}
```

Problem S3: Rescue Mission

For starters, we should precompute the 2-dimensional prefix sums of all rows and columns. This precomputation can be done in $O(NM)$ time, and will be helpful later for computing subsequent rescue zone sums in constant time. Let $sum[i][j]$ denote the sum of the rectangle consisting of rows 1 to i (inclusive), and columns 1 to j (inclusive). Then, we can compute it using the recursive formula

```
sum[i][j] = P[i][j] + sum[i - 1][j] + sum[i][j - 1] - sum[i - 1][j - 1]
```

where $sum[i][j] = 0$ in the base cases of $i = 0$ or $j = 0$. The entire prefix sum array can be computed while P is being inputted, at no additional cost in time complexity.

Now, because all values of P are non-negative, each rescue zone should extend as far up and left from its central cell as possible. We can call a rescue zone "complete" if it's able to extend all the way to the top-most row (row 1) and to the left-most column (column 1), without intersecting with the other rescue zone. We can let $C[i][j]$ equal the sum of the complete rescue zone with central point (i, j). Computing $C[i][j]$ is easy now that we have the prefix sums – just subtract the sum of the rectangle with rows 1 to $i - 1$ and columns 1 to $j - 1$ from the total sum of the rectangle from the corner (1, 1) to cell (i, j). As a result, we will end up with the sum of the backwards "L"-shaped slice of the grid. That is, $C[i][j] = sum[i][j] - sum[i - 1][j - 1]$.

Now, only the following cases are possible: either both rescue zones will be "complete", or one branch of one of the rescue zones is forced to stop short in the horizontal or vertical direction by the other.

```
...S.B      .....S      ..S..B
...S.B      .....S      ..S..B
SSSS.B      SSSSSS      ..S..B
.....B      ....B.      ..SBBB
BBBBBB      BBBBB.      SSS...
```

In the first case, the central point of one rescue zone is strictly above and to the left of the central point of the other. The best possible answer for this case will be the maximum value of $C[i_1][j_1] + C[i_2][j_2]$ across any pair of coordinates (i_1, j_1) and (i_2, j_2) that satisfy $1 \le i_1 < i_2 \le N$ and $1 \le j_1 < j_2 \le M$. To actually compute this best answer, we can use dynamic programming as we iterate over all cells (i, j) from the top-left corner of the grid to the bottom-right corner.

Let $DP[i][j]$ store the maximum complete rescue zone whose central point is in the inclusive rectangle from (1, 1) to (i, j). Intuitively, we can see that $DP[i][j]$ is equal to the maximum of three values: $DP[i - 1][j]$, $DP[i][j - 1]$, and $C[i][j]$. Finally, to get the max sum of any pair of complete, non-intersecting rescue zones, we can take the max of $C[i][j] + DP[i - 1][j - 1]$ across all coordinates (i, j) in the grid. This part of the solution will take $O(NM)$ time (and can in fact be fully computed as the grid is being inputted!).

The second case is that only one of the rescue zones is complete, while the other one has to stop short in either the horizontal or vertical direction so that it doesn't intersect with the first zone. Without loss of generality, let's say that the incomplete rescue zone's central point is to the right of the complete one's (see the rightmost figure above). It could instead be below it (as in the middle figure), but we can repeat the following algorithm with the grid's axes inverted to cover that case. Inverting the grid is as simple as treating row indices as column indices and vice versa.

Now, suppose the complete zone has its central point at (i, j) and the incomplete zone has the central point (r, c) (with $r \le i$ and $c > j$) such that it includes cells $j + 1$ to c in row r, and cells 1 to r in column c. If we iterate over all possible (i, j) cells from the bottom-right to the top-left while maintaining the sum of the best incomplete zone in each row r (which can be updated in constant time as we go through each cell (i, j)), then the sum of the best accompanying incomplete zone for the complete zone centred at (i, j) can also be looked up in constant time. As such, this part of the solution can also be implemented in $O(NM)$ time.

Taking the best answer of all of the cases will yield the following $O(NM)$ solution.

Official Solution (C++)

```
#include <algorithm>
#include <iostream>
using namespace std;

int N, M, P[2005][2005], ans = 0;
int sum[2005][2005] = {0}, dp[2005][2005] = {0}, best[2][2005] = {0};

int main() {
  cin >> N >> M;
  for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
      cin >> P[i][j];
      sum[i][j] = P[i][j] + sum[i - 1][j] + sum[i][j - 1] - sum[i - 1][j - 1];
      int curr = sum[i][j] - sum[i - 1][j - 1];
      dp[i][j] = max(curr, max(dp[i - 1][j], dp[i][j - 1]));
      ans = max(ans, curr + dp[i - 1][j - 1]);
    }
  }
  for (int invert = 0; invert <= 1; invert++) {
    int h1 = (invert ? M : N), h2 = (invert ? N : M);
    int tmp[2005] = {0};
    for (int a = h1; a >= 1; a--) {
      int curr = 0;
      for (int b = 1; b <= h2; b++) {
        int p = (invert ? P[b][a] : P[a][b]);
        tmp[b] = p + max(tmp[b], curr);
        curr += p;
        best[invert][a] = max(best[invert][a], tmp[b]);
      }
    }
  }
  for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
      int curr = sum[i][j] - sum[i - 1][j - 1];
      if (i < N) ans = max(ans, curr + best[0][i + 1]);
      if (j < M) ans = max(ans, curr + best[1][j + 1]);
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem S4: Lex Luthor's Landmines

For starters, let's sort the landmines in non-decreasing order of position. This takes $O(N \log N)$ time. Our primary goal will be to compute the values $ML[i]$ and $MR[i]$ for each mine i – the minimum and maximum positions that will be engulfed in an explosion if mine i is set off. From here, one naïve thing to try would be a straightforward simulation using breadth-first search. To compute the *ML* and *MR* of each initial mine, we can add it to the queue. Processing each mine as we pop them from the queue, we can loop through all other mines to see if that mine is set off. The simulation for each mine will take $O(N^2)$. For N mines, that is $O(N^3)$ overall. To finally process each query position C, we can naively loop through all of the mines and count the number of mines that have *ML* and *MR* ranges including C. Since there are at M queries, the running time for this part is $O(NM)$. This solution is given 15/40 of the points.

We can try certain optimizations such as gradually expanding the explosion range for each initial mine. This will get each simulation from $O(N^2)$ to $O(N)$, yielding $O(N^2)$ across all N mines. However, that is still too slow for N as large as 10^6. Furthermore, answering queries also happens to be a bottleneck, with M up to 3×10^6. We will need to speed up both sections to sub-quadratic running time to pass under the time limit.

For a full solution, we can start by representing the mines as the nodes of a directed, unweighted graph. Let there be an edge from mine i to mine j if mine j's explosion would directly set off mine i (in other words, if $P[j] - L[j] \leq P[i] \leq P[j] + R[j]$). The purpose of this graph is that $ML[j]$ and $MR[j]$ can be computed based on the *ML* and *MR* values of mines which have edges leading into mine j (for example, $ML[j]$ is equal to the minimum of $L[j]$ and the *ML* values of those mines).

One issue with this graph is that it can have $O(N^2)$ edges. However, the observation can be made that keeping at most two incoming edges for each mine j will suffice – the edge leading from the mine whose position is in the interval $[P[j] - L[j], P[j] + R[j]]$ which has the minimum L value, and similarly the one which has the maximum R value. These (at most) two mines are clearly at least as important as any other mines that mine j can directly set off. Both of the mines can be found in $O(\log N)$ time by using a segment tree to perform a range minimum/maximum query on the L/R values of the mines.

We now have a graph with $O(N)$ edges which represents the relationships between the mines' *ML* and *MR* values, but it doesn't yet give us an appropriate order in which to compute these values based on one another, due to the fact that it can have cycles. If two mines are reachable from one another in this graph, they must have the same *ML* and *MR* values. As such, the next step is to locate all of the strongly connected components in the graph (in $O(N)$ time using an algorithm such as Tarjan's). Each strongly connected component can then be contracted to a single node, which stores the number of actual mines in the node as well as their minimum L value and maximum R value, and a new, directed acyclic graph can be created from these component nodes. In graph theory, this new graph is known as the *condensation* of the original, and happens to be a directed acyclic graph (DAG).

We can now perform a topological sort on the nodes of this DAG, iterating over them in order and updating each node's *ML* and *MR* values based on those of the nodes with edges leading to it. For each one, we don't need to bother looking up exactly which initial mines correspond to the component node – only how many there are that have this set of *ML* and *MR* values. This process takes $O(N)$ time.

Finally, we can perform a line sweep to get the requested answers. We'll need events for the civilians' positions, as well as for the start and end of each component node's final explosion range $[ML, MR]$ (also storing the number of mines associated with this range). Upon sorting the events in $O((N+M) \times \log(N+M))$ time, we can iterate over them while maintaining the number of mines whose final explosion range affects the current position, and noting these counts each time a civilian position is reached so that they can later be outputted in the correct order.

The official solution implementing this approach is as follows.

Official Solution (C++)

```
#include <algorithm>
#include <cstdio>
#include <queue>
#include <set>
#include <vector>
using namespace std;

const int MAXN = 1000005, SIZE = 2100000, INF = 1000000000;

int components, comp_id[MAXN], timer = 0, lowlink[MAXN] = {0};
vector<bool> vis(MAXN, false);
vector<int> adj[MAXN], stack;

void tarjan_dfs(int u) {
  lowlink[u] = timer++;
  vis[u] = true;
  stack.push_back(u);
  bool is_component_root = true;
  int v;
  for (int j = 0; j < (int)adj[u].size(); j++) {
    if (!vis[v = adj[u][j]]) tarjan_dfs(v);
    if (lowlink[u] > lowlink[v]) {
      lowlink[u] = lowlink[v];
      is_component_root = false;
    }
  }
  if (!is_component_root) return;
  do {
    vis[v = stack.back()] = true;
    stack.pop back();
    lowlink[v] = INF;
    comp_id[v] = components;
  } while (u != v);
  components++;
}

int N, M, C, x, l, r, segmax, ql, qh, qmin, qmax;
vector<int> X, L, R, dag[MAXN];
vector<pair<int, pair<int, int> > > mines, events;
int CS[MAXN], CL[MAXN], CR[MAXN], indegree[MAXN], ans[MAXN];
int min_val[SIZE], min_idx[SIZE], max_val[SIZE], max_idx[SIZE];

void query(int i, int lo, int hi) {
  if (ql <= lo && hi <= qh) {
    if (min_val[i] < L[qmin]) qmin = min_idx[i];
    if (max_val[i] > R[qmax]) qmax = max_idx[i];
    return;
  }
  int m = (lo + hi) / 2;
  if (ql <= m) query(i * 2, lo, m);
  if (qh > m) query(i * 2 + 1, m + 1, hi);
}

int main() {
  scanf("%d%d", &N, &M);
  for (int i = 0; i < N; i++) {
    scanf("%d%d%d", &x, &l, &r);
    mines.push back(make pair(x, make pair(l, r)));
  }
  for (int i = 0; i < M; i++) {
    scanf("%d", &C);
    events.push back(make pair(C, make pair(0, i)));
  }
  //sort mines
  sort(mines.begin(), mines.end());
  for (int i = 0; i < N; i++) {
    X.push_back(mines[i].first);
    L.push back(X[i] - mines[i].second.first);
    R.push_back(X[i] + mines[i].second.second);
  }
```

```
  //initialize segment tree
  for (segmax = 1; segmax < N; segmax *= 2);
  for (int i = 0; i < segmax; i++) {
    int j = segmax + i;
    min idx[j] = max idx[j] = i;
    if (i < N) {
      min_val[j] = L[i]; max_val[j] = R[i];
    } else {
      min_val[j] = INF;  max_val[j] = -INF;
    }
  }
  for (int i = segmax - 1; i > 0; i--) {
    int l = i * 2, r = i * 2 + 1;
    min_val[i] = min(min_val[l], min_val[r]);
    max_val[i] = max(max_val[l], max_val[r]);
    min idx[i] = (min val[i] == min val[l]) ? min idx[l] : min idx[r];
    max_idx[i] = (max_val[i] == max_val[l]) ? max_idx[l] : max_idx[r];
  }
  //construct graph by finding best pair of mines in range and constructing edges from them
  for (int i = 0; i < N; i++) {
    ql = lower_bound(X.begin(), X.end(), L[i]) - X.begin();
    qh = lower_bound(X.begin(), X.end(), R[i] + 1) - X.begin() - 1;
    qmin = qmax = i;
    query(1, 0, segmax - 1);
    if (qmin != i) adj[qmin].push_back(i);
    if (qmax != i) adj[qmax].push_back(i);
  }
  //find strongly connected components
  for (int i = 0; i < N; i++) if (!vis[i]) tarjan_dfs(i);
  //construct directed acyclic graph out of SCCs
  set<pair<int, int> > edges;
  for (int i = 0; i < N; i++) {
    int u = comp_id[i];
    if (++CS[u] == 1) {
      CL[u] = L[i]; CR[u] = R[i];
    } else {
      CL[u] = min(CL[u], L[i]); CR[u] = max(CR[u], R[i]);
    }
    for (int j = 0; j < (int)adj[i].size(); j++) {
      int v = comp_id[adj[i][j]];
      if (u != v && !edges.count(make_pair(u, v))) {
        edges.insert(make_pair(u, v));
        dag[u].push back(v);
        indegree[v]++;
      }
    }
  }
  //topological sort nodes of the DAG
  queue<int> q;
  for (int i = 0; i < components; i++) if (indegree[i] == 0) q.push(i);
  while (!q.empty()) {
    int u = q.front(); q.pop();
    events.push_back(make_pair(CL[u], make_pair(-1, CS[u])));
    events.push_back(make_pair(CR[u], make_pair(1, -CS[u])));
    for (int j = 0; j < (int)dag[u].size(); j++) {
      int v = dag[u][j];
      CL[v] = min(CL[v], CL[u]);
      CR[v] = max(CR[v], CR[u]);
      if (--indegree[v] == 0) q.push(v);
    }
  }
  //line sweep
  sort(events.begin(), events.end());
  int curr = 0;
  for (int i = 0; i < (int)events.size(); i++) {
    if (events[i].second.first == 0)
      ans[events[i].second.second] = curr;
    else
      curr += events[i].second.second;
  }
  for (int i = 0; i < M; i++) printf("%d\n", ans[i]);
  return 0;
}
```