WOBURN CHALLENGE

2016-17 Online Round 2

Solutions

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

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

Problem J1: The Perfect Mate

We'll need to iterate over all N warriors, inputting their information and keeping track of the best mate seen so far. We'll need to remember both this optimal mate's name (so that it can be outputted at the end), and his number of victories (so that we can determine if a superior mate is found). For convenience, we can initialize his name as "None" and victory count as –1. In this way, any suitable mate will replace him, and if no suitable mates are found, we'll proceed to output "None".

When we consider each warrior, we should simply ignore them if they've lost more than 0 battles. Otherwise, they should become the new optimal mate if they've won strictly more battles than the previous mate has. If they've only won an equal number of battles, the previous optimal mate should be retained, as they appeared earlier in the list.

Official Solution (C++)

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

int N, ansW = -1;

int main() {
  string ansS = "None";
  cin >> N;
  for (int i = 0; i < N; i++) {
    string S;
    int W, L;
    cin >> S >> W >> L;
    if (L == 0 && W > ansW) {
      ansS = S;
      ansW = W;
    }
  }
  cout << ansS << endl;
  return 0;
}
```

Problem J2: EHC

We should start by sorting all N existing holographic emitters in increasing order of distance from the mess hall. We can then iterate over them while maintaining the maximum distance d down the hall that the EHC is so far able to reach. d is initially equal to R, as there's an additional emitter at the entrance to the mass hall. When we consider emitter i, which covers the inclusive range from $E_i - R$ to $E_i + R$, the EHC can already reach this range if $d \geq E_i - R$, in which case no additional emitters must be installed right before emitter i. Otherwise, there's an intervening distance of $d' = E\_i - R - D$ which must be filled in with emitters. Each emitter spans a distance of $2R$ metres, and so the additional emitters should be evenly spaced out at intervals of $2R$ metres in order to minimize the number of them that are required.

Therefore, ceil(d' / ($2R$)) new emitters must be installed in order for the EHC to be able to reach emitter i. Either way, we can then set d to $E\_i + R$ and proceed to the next emitter. Finally, the EHC must still reach the end of the hallway from the end of the range of the furthest emitter. If we pretend that there's a dummy final emitter at a distance of $M + R$ metres from the mess hall, then this is equivalent to the EHC reaching this dummy emitter, allowing the algorithm to avoid having any other final step.

Official Solution (C++)

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

int N, M, R;

int main() {
  cin >> N >> M >> R;
  vector<int> E(N);
  for (int i = 0; i < N; i++) {
    cin >> E[i];
  }
  E.push_back(M + R);  // Dummy final emitter which must be reached.
  sort(E.begin(), E.end());  // Sort by increasing distance.
  int prev = R, ans = 0;
  for (int i = 0; i < (int)E.size(); i++) {
    int dist = E[i] - R - prev;  // Distance from prev to current.
    if (dist > 0) {
      ans += (dist - 1)/(2 * R) + 1;  // Cover intervening distance.
    }
    prev = E[i] + R;
  }
  cout << ans << endl;
  return 0;
}
```

Problem J3/S1: Most Illogical

Any Boolean expression in this problem can be thought of as consisting of one or more "clauses", where each clause consists of one or more literals `and`'ed together, with all of the clauses then `or`'ed together. For example, the expression *A* or *B* and *C* and *D* or *E* consists of 3 clauses, "*A*", "*B* and *C* and *D*", and "*E*".

If at least one of the literals in a clause is false, then the result of the whole clause is definitely false, due to the properties of the "and" operator. Otherwise, if at least one of the literals is unknown, then the result of the whole clause is unknown (it could be false if any unknown literals are false, or it could be true if all unknown literals are true). Finally, if all of the literals are true, then the result of the whole clause is true.

Similar logic can be applied to the whole expression, though inverted due to properties of the "`or`" operator. If at least one of the clauses is true, then the result of the whole expression is true. Otherwise, if at least one of the clauses is unknown, then the result of the whole expression is unknown. Finally, if all of the clauses are false, then the result of the whole expression is false.

With these facts in mind, we can process the expression from left to right, while keeping track of the result of the ongoing clause, and the result of the whole expression. When a literal is processed, the result of the ongoing clause should be updated accordingly (for example, if a "`false`" literal is encountered, then the result of the ongoing clause should be set to false). Similarly, when the end of a clause is reached (which happens after the last literal or when an "`or`" operator is encountered), the result of the whole expression should be updated according to the result of that clause, and then a new clause should be started.

Official Solution (C++)

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

int N;
string expr = "false";  // Whole expression is false until proven otherwise.
string clause = "true";  // Ongoing clause is true until proven otherwise.

int main() {
  cin >> N;
  for (int i = 0; i < N; i += 2) {
    string val, op = "";
    cin >> val;
    if (val == "false") {
      clause = "false";  // Ongoing clause is now definitely false.
    }
    if (val == "unknown" && clause == "true") {
      clause = "unknown";  // Ongoing clause is no longer true.
    }
    if (i < N - 1) {
      cin >> op;
    }
    if (op != "and") {  // End of the ongoing clause was reached.
      if (clause == "true") {
        expr = "true";  // Whole expression is now definitely true.
      }
      if (clause == "unknown" && expr == "false") {
        expr = "unknown";  // Whole expression is no longer false.
      }
      clause = "true";  // Start a new clause.
    }
  }
  cout << expr << endl;
  return 0;
}
```

Problem J4/S2: Away Mission

Let's first consider the case in which $Q = 1$. We want to assemble shirts with triples of CCC values (r, g, b) such that, for as few of them as possible, $r > m$, where $m = \max(g, b)$.

For starters, let's just consider forming pairs of the green and blue CCCs (g, b), with the goal of maximizing their resulting m values. If we consider the list of all $2N$ green/blue CCC values, the best we could ever hope to do is for the largest N of these $2N$ values to be equal to the N produced m values. As it turns out, this can always be accomplished – for example, if we pair the largest green CCC with the smallest blue CCC, the second-largest green CCC with the second-smallest blue CCC, and so on. As such, if we sort these $2N$ values and take the largest N of them, we'll get an optimal set of m values.

What remains is pairing the r values against these m values, which can be done greedily. Assuming that both lists are sorted in non-decreasing order, let's iterate upwards through the r values. For a given r value, we might as well match it with the smallest remaining m value which is larger than or equal to it, if any. This can be done by maintaining a pointer into the sorted list of m values, and advancing it forwards at each step until a sufficiently large m value is reached (or until it hits the end of the array, at which point no more non-red shirts can be created).

The other case, in which $Q = 2$, is similar – we now want to maximize the number of triples in which $r > m$. This time, we'll want to form N (g, b) pairs with the goal of minimizing their resulting m values. If we consider the largest of all of the green or blue CCC values, it will necessarily need to be one of the m values. It'll need to be matched with some value from the opposite list, so we might as well pair it with the largest of those, in an effort to minimize future m values. Therefore, it's always optimal to pair the largest green and blue CCCs together, and this can be extended to show that we should always sort the lists of green and blue CCC values independently, and then pair them up in that order to compute an optimal set of m values.

At that point, we can follow a very similar algorithm to the $Q = 1$ case in order to greedily pair up the r and m values, this time iterating downwards through the r values and matching each one against the larger remaining m value which is smaller than it (if any).

The time complexity of this algorithm in either of the two cases is dependent on sorting $O(N)$ component values. This can be done in $O(N \log N)$ time, or $O(N)$ time if we take advantage of their limited magnitudes with radix sort.

Official Solution (C++)

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

const int MAXN = 200005;
int N, Q, red = 0;
int R[MAXN], G[MAXN], B[MAXN], GB[2*MAXN];

int main() {
  cin >> N >> Q;
  for (int i = 0; i < N; i++) {
    cin >> R[i];
  }
  sort(R, R + N);
  if (Q == 1) {
    for (int i = 0; i < 2*N; i++) {
      cin >> GB[i];
    }
    sort(GB, GB + 2*N);
    int j = 2*N - 1;
    for (int i = N - 1; i >= 0; i--) {
      if (R[i] <= GB[j]) {
        j--;
      } else {
        red++;
      }
    }
  } else /* Q == 2 */ {
    for (int i = 0; i < N; i++) {
      cin >> G[i];
    }
    for (int i = 0; i < N; i++) {
      cin >> B[i];
    }
    sort(G, G + N);
    sort(B, B + N);
    for (int i = 0; i < N; i++) {
      GB[i] = max(G[i], B[i]);
    }
    int j = 0;
    for (int i = 0; i < N; i++) {
      if (R[i] > GB[j]) {
        j++;
        red++;
      }
    }
  }
  cout << red << endl;
  return 0;
}
```

Problem S3: Turbolift Testing

Solving this problem efficiently requires precomputing various useful values, and then using them to answer the questions.

Let's consider each of the N button sequences. Assuming that sequence i is executed with the turbolift starting at floor 0, let $len[i]$ be the length of the sequence, $delta[i]$ be the turbolift's final floor, $maxP[i][j]$ be the highest floor reached at any point during the first j button presses of the sequence, $minP[i][j]$ be the lowest floor reached during the first j button presses, $maxS[i]$ be the highest floor reached at any point during the whole sequence (equal to $maxP[i][len[i]]$), and $minS[i]$ be the lowest floor reached (equal to $minP[i][len[i]]$). These values can all be easily computed by simulating the sequence's button presses in $O(L[i])$ time, which is sufficient as the sum of the sequences' lengths is at most 200,000.

Next, let's repeat a similar process over the entire sequence of M button sequences. Let $len_2[i]$ be the total length of the first i executed sequences, $delta_2[i]$ be the turbolift's final floor after the first i sequences, $max_2[i]$ be the highest floor reached at any point during the first i sequences, and $min_2[i]$ be the lowest floor reached during the first i sequences. These values can similarly be easily computed in $O(M)$ time, with the help of the precomputed len, $delta$, $maxS$, and $minS$ values. Note that $len_2[0] = delta_2[0] = max_2[0] = min_2[0] = 0$.

Finally, we're set up to answer the questions efficiently. Let's say we're interested in the first b button presses. The b-th button press must occur during some sequence i ($1 \le i \le M$) – in particular, during the first sequence i such that $b \le len_2[i]$. Since the values $len_2[0..M]$ are increasing, we can use binary search to determine the value of i in $O(\log M)$ time. Now, since the first $i - 1$ button sequences have been completed before the b-th button press, the highest floor reached during the first b button presses is at least $max_2[i - 1]$. However, the first $b - len_2[i - 1]$ button presses of button sequence S_i were additionally executed after that, which is where more of our precomputed values come into play – the answer must be $\max(max_2[i - 1], delta_2[i - 1] + maxP[S_i][b - len_2[i - 1]])$. Similarly, the minimum floor reached is $\min(min_2[i - 1], delta_2[i - 1] + minP[S_i][b - len_2[i - 1]])$.

The time complexity of this algorithm is $O(\text{sum}\{L[1..N]\} + M + Q \log M)$.

Official Solution (C++)

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

const int MAXN = 200005;

int N, M, Q;
int S[MAXN],L[MAXN];
long long mu2[MAXN], md2[MAXN], ans1[MAXN],
ans2[MAXN], p2[MAXN];
pair<long long, int> P[MAXN];
vector<long long> mu[MAXN], md[MAXN];
string s;

int main() {
  cin >> N >> M >> Q;
  // Process sequences.
  for (int i = 0; i < N; i++) {
    // Input sequence.
    cin >> s;
    L[i] = s.length();
    // Precompute prefix mins/maxes for this sequence.
    mu[i].resize(L[i] + 1, 0);
    md[i].resize(L[i] + 1, 0);
    long long p = 0;
    for (int j = 0; j < L[i]; j++) {
      p += (s[j] == 'U' ? 1 : -1);
      mu[i][j + 1] = max(mu[i][j], p);
      md[i][j + 1] = min(md[i][j], p);
    }
    // Store overall min/max/delta for this sequence.
    mu2[i] = mu[i][L[i]];
    md2[i] = md[i][L[i]];
    p2[i] = p;
  }
  for (int i = 0; i < M; i++) {
    cin >> S[i];
    S[i]--;
  }
  for (int i = 0; i < Q; i++) {
    cin >> P[i].first;
    P[i].second = i;
  }
  // Process queries in increasing order.
  sort(P, P + Q);
  int j = 0;
  long long c = 0, p = 0, a1 = 0, a2 = 0;
  for (int i = 0; i < Q; i++) {
    // Process completed sequences.
    int k;
    while (P[i].first > c + L[k = S[j]]) {
      a1 = min(a1, p + md2[k]);
      a2 = max(a2, p + mu2[k]);
      p += p2[k];
      c += L[k];
      j++;
    }
    // Also consider prefix min/max for ongoing
sequence.
    int d = P[i].first - c;
    ans1[P[i].second] = min(a1, p + md[k][d]);
    ans2[P[i].second] = max(a2, p + mu[k][d]);
  }
  // Output query answers in original order.
  for (int i = 0; i < Q; i++) {
    cout << ans1[i] << ' ' << ans2[i] << '\n';
  }
  return 0;
}
```

Problem S4: Diplomacy

Let's consider plotting the temperature and gravity values on a Cartesian plane. For example, let's pretend that all temperature values are x-coordinates, and all gravity values are y-coordinates. Every race's preferred requirements that correspond to a rectangle, and the answer is the largest number of rectangles that overlap (inclusively) at any single point.

This geometric representation suggests a possible approach - line sweep. Let's consider sweeping upwards through the rectangles. As we go, we'll want to handle interesting events, namely the bottoms and tops of rectangles. *A* rectangle with lower-left corner (x_1, y_1) and upper-right corner (x_2, y_2) will correspond to a starting event at y-coordinate y_1, and an ending event at y-coordinate y_2, with both events tied to the range of x-coordinates $[x_1, x_2]$. We'll want to generate all $2N$ events and then iterate over them in increasing order of y-coordinate. Because rectangles' ranges should be inclusive, we should carefully break ties in the sorting so that we iterate over all starting events at a given y-coordinate before the ending events at the same y-coordinate.

During the line sweep, we'll want to keep track of what rectangles are currently ongoing in some fashion (in terms of their x-coordinate ranges). Importantly, we'll need to be able to efficiently determine the largest number of these 1-dimensional inclusive ranges that overlap at any single point. If we can compute this value at every step of the line sweep, then the final answer will be the largest of these values.

What remains is coming up with a data structure which will allow us to handle these updates and queries efficiently (as we'll have to do $O(N)$ of them over the course of the line sweep). Let's model exactly what we need the data structure to support. Let $A[i]$ be the number of active ranges which include point i. *A* rectangle's starting event corresponds to incrementing $A[i]$ for each i in the event's range $[x_1, x_2]$. Similarly, a rectangle's event corresponds to decrementing this range of A values. Finally, we'll need to look up the largest A value.

One thing worth noting about this A array is that it can be very large (as x-coordinates can be between 1 and 10^9). However, the exact values of the rectangles' x-coordinates don't matter, only their relative order does. As such, they can be compressed down to values between 1 and at most $2N$. This will reduce A's size down to $O(N)$, which is more manageable.

Now, what data structure can efficiently support the required set of updates and queries? A segment tree with lazy propagation can get them done in $O(\log N)$ time each, which will result in an overall time complexity of $O(N \log N)$. Each node in the tree should simply store the largest A value in its range, as well as a lazy value of how much its entire range of A values should be incremented/decremented by. This works out nicely due to the observation that, when a set of values are all incremented by a constant amount x, their maximum value will also increase by x. Therefore, range updates can be lazily applied in a standard fashion, and the maximum of all of the values can be determined from the root of the tree. It's worth noting that an interval tree can also be used in place of a segment tree to manage the state of the ongoing rectangles during the line sweep.

Official Solution (C++)

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

int sz, treeMax[1100000], treeLazy[1100000];

// Lazily propagate updates to node i.
void Prop(int i) {
  treeMax[i] += treeLazy[i];
  if (i < sz) {  // Update children's lazy values, if any.
    treeLazy[i*2] += treeLazy[i];
    treeLazy[i*2 + 1] += treeLazy[i];
  }
  treeLazy[i] = 0;  // Clear lazy value.
}

// Update range a..b by a delta of d. Currently at node i, spanning range r1..r2.
void Update(int i, int r1, int r2, int a, int b, int d) {
  Prop(i);  // Propagate this node.
  if (a <= r1 && r2 <= b) {  // Is this node contained in the update range?
    treeLazy[i] += d;
    Prop(i);
    return;
  }
  // Recurse to affected children.
  int m = (r1 + r2)/2;
  if (a <= m) {
    Update(i*2, r1, m, a, b, d);
  }
  if (b > m) {
    Update(i*2 + 1, m + 1, r2, a, b, d);
  }
  // Propagate children and update this node based on their values.
  Prop(i*2);
  Prop(i*2 + 1);
  treeMax[i] = max(treeMax[i*2], treeMax[i*2 + 1]);
}

int N, a, b, c, d;
vector<int> comp;
vector<pair<pair<int, int>, pair<int, int> > > ev;

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> a >> b >> c >> d;
    // Collect y-coordinates and create line sweep events for left/right edges of rectangle.
    comp.push back(c);
    comp.push back(d);
    ev.push back(make pair(make pair(a, 1), make pair(c, d)));
    ev.push back(make pair(make pair(b + 1, -1), make pair(c, d)));
  }
  // Get list of unique y-coordinates.
  sort(comp.begin(), comp.end());
  comp.resize(unique(comp.begin(), comp.end()) - comp.begin());
  // Initialize segment tree of sufficient size.
  for (sz = 1; sz < (int)comp.size(); sz *= 2) {}
  // Line sweep through events by increasing x-coordinate.
  sort(ev.begin(), ev.end());
  int ans = 0;
  for (int i = 0; i < (int)ev.size(); i++) {
    int d = ev[i].first.second;  // Get event delta.
    // Get event's compressed y-coordinates.
    int a = lower_bound(comp.begin(), comp.end(), ev[i].second.first) - comp.begin();
    int b = lower_bound(comp.begin(), comp.end(), ev[i].second.second) - comp.begin();
    Update(1, 0, sz - 1, a, b, d);  // Update segment tree.
    ans = max(ans, treeMax[1]);  // Consider max value in the tree.
  }
  cout << ans << endl;
  return 0;
}
```