WOBURN CHALLENGE

2015-16 Online Round 4

Solutions

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

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

Problem J1: Telling Time

Given a positive integer G and a list of N other positive integers, this question asks to find the number of integers in the list which are multiples of G. The solution is to just loop through the list and check if each integer is a multiple of G. If so, increment a running counter variable.

Checking if a number F is a multiple of G can be done with the modulo (or remainder) operator in most programming languages (the % operator in C++/Java/Python, for instance). That is, F is a multiple of G if and only if F % G is equal to 0. Since we loop through an array of size N, this runs in $O(N)$.

Official Solution (C++)

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

int N, G, F, ans = 0;

int main() {
  cin >> N >> G;
  for (int i = 0; i < N; i++) {
    cin >> F;
    if (F % G == 0) ans++;
  }
  cout << ans << endl;
  return 0;
}
```

Problem J2: Mission Briefing

Given a string, we want to count how many substrings from the set '001'...'009' occur. By reading the question carefully, one should notice that multiple instances of the same string should be counted only once. Namely, '001.abc.001a' should only count 1 towards the final answer. Furthermore, from the sample input and output, one should understand that the string can be surrounded by anything, including other digits. Namely, an occurrence of '.20001.' should still be counted as 001. We should therefore not assume that strings are surrounded by punctuation. Finally, we should not consider '000' to be a valid agent.

Assuming the string has 0-based indices, one way to solve the problem is keep track of occurrences with a Boolean array (accessible from indices 1 to 9) initialized to false. We loop through the input string, starting at index 2 and ending at index N–1, where N is the length of the string. We check the last 2 indices to make sure both of them are the digit '0', and then check to make sure the current index is a digit from '1' to '9' (most programming languages support directly comparing ASCII characters). If so, set the corresponding Boolean variable to true. To get the answer, count the number of true values in the Boolean array. This $O(N)$ solution is implemented on the right.

An alternate solution is to just use string searching functions in the standard library of your programming language (e.g. string::find() in C++), which should take $O(N)$ per call on the length of the string. We have to make 9 calls, one for each digit from 1 through 9.

Official Solution (C++)

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

string s;
int ans = 0;
bool found[10];

int main() {
  cin >> s;
  for (int i = 2; i < s.length(); i++) {
    if (s[i-2] == '0' && s[i-1] == '0'
        && s[i] >= '1' && s[i] <= '9') {
      found[s[i] - '0'] = true;
    }
  }
  for (int i = 1; i <= 9; i++) {
    if (found[i]) {
      ans++;
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem J3/S1: Shootout

We are given N henchmen and M doors located at distinct positions on the number line. Henchman j is in Bond's line of sight if and only if every door i with D_i value less than H_j is opened. That is, for the i-th line of output, we should count the number of henchmen such that no D value from i+1 to M is greater than H. To illustrate the intended computations, a naïve solution is given here which runs in $O(NM^2)$.

This solution is only meant to be given only 3/17 of the points (through the first subtask, where $N, M \leq 100$, so NM^2 is no larger than $100^3 = 1{,}000{,}000$). The second subtask was designed for various $O(MN \log N)$ solutions which we will not discuss here.

Naïve Solution - $O(NM^2)$

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

int N, M, H[200005], D[200005];

int main() {
  cin >> N >> M;
  for (int i = 0; i < N; i++) cin >> H[i];
  for (int i = 0; i < M; i++) cin >> D[i];
  for (int i = 0; i < M; i++) {
    int ans = 0;
    for (int j = 0; j < N; j++) {
      bool blocked = false;
      for (int k = i + 1; k < M; k++)
        if (D[i] < H[j]) blocked = true;
      if (!blocked) ans++;
    }
    cout << ans << endl;
  }
  return 0;
}
```

For up to the third subtask, we can go for an $O(NM)$ solution which eliminates the innermost for-loop of k. How? Observe that the inner loop is asking the question "does there exist a value in $D[(i+1)...M]$ which is less than the current $H[j]$?" This is actually equivalent to asking if the *minimum* value in $D[(i+1)...M]$ is less than $H[i]$. Let the array *minD*[] be such that *minD*[i] stores the minimum D value from i to M. To compute this array, just loop backwards from M to 1 and store the minimum of the minimum so far with the current D value. This simple solution is given below and is given 11/17 of the points.

Precomputing Minimums – $O(NM)$

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

int N, M, H[200005], D[200005], minD[200005];

int main() {
  cin >> N >> M;
  for (int i = 0; i < N; i++) cin >> H[i];
  for (int i = 0; i < M; i++) cin >> D[i];
  minD[M] = 1000000001;
  for (int i = M - 1; i >= 0; i--)
    minD[i] = min(minD[i + 1], D[i]);
  for (int i = 0; i < M; i++) {
    int ans = 0;
    for (int j = 0; j < N; j++)
      if (minD[i + 1] > H[j]) ans++;
    cout << ans << endl;
  }
  return 0;
}
```

It is also possible to get 11/17 points with a more complex solution than the code below, which involves sorting, searching, and erasing from an array.

For a full solution, lets start by sorting the N henchmen in increasing order of position, which can be done using any $O(N \log N)$ time sorting algorithm. Let's similarly sort the M doors in $O(M \log M)$ time – however in this case, we'll need to keep each door's original index associated with its position after the sort (as opposed to only sorting the M door positions independently, where info about the original position is lost). To accomplish this, we can represent each door as a pair of integers (position, original index), and compare these pairs by only their position value in the sorting algorithm. To use this with the built-in sorting function of certain languages, we may need to define a custom class and/or comparator, or use something like C++'s std::pair like the official solution below.

Now, as we simulate opening all of the doors in order, we'd like to maintain one running index for each of the two sorted lists. Firstly, we need to keep track of the index h of the current first henchmen not in Bond's line of fire (or $N + 1$ if there's no such henchman). Secondly, we need the index d of the current first closed door (or $M + 1$ if there's no such door). We can observe that exactly the first $h - 1$ sorted henchmen will be in Bond's line of fire, such that the h-th henchmen is the first one further from Bond than the d-th door in the sorted list.

When door i is opened, we can update d by repeatedly incrementing it as long as it's no larger than M, and the original index of the d-th door in the sorted list is smaller than or equal to i (note that d never decreases). When a given door is opened, d might stay the same or be incremented many times, but over the course of the entire simulation, it will only be incremented M times, so these updates will take $O(M)$ time in total.

After the index d has been updated for door i, we can use it to update h accordingly (which will give us the i-th answer). Similarly to the above approach, we can repeatedly increment h as long as it's no larger than N, and the index of the h-th henchman in the sorted list is before the index of the d-th door in the sorted list. Likewise, i will be incremented a total of N times throughout the entire simulation, which can be handled in $O(N)$ time.

The sorting of the two arrays dominate over the simulation, so the overall running time for the code given on the right is $O(N \log N + M \log M)$.

Alternatively, we don't have to keep track of the h value. In this case, the second while-loop may be replaced with a binary search on the sorted array of henchmen, querying for the first henchman position which is not less than the position of the current door. This position is the answer, since all henchmen before will also be in the line of sight.

Alternatively, we can put all henchmen and doors into the same array of pairs and sort them by position (but still remembering the index of each door). When we open a door, we erase the door i from the sorted array and loop d upwards until we skip over all of the erased door while counting the number of henchmen encountered in the same loop. The sorting also dominates in this variation, but as we are sorting a list of size $N + M$, the solution will run in $O((N+M) \log(N + M))$ time.

Official Solution (C++)

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

int N, M, position, h = 0, d = 0;
vector< int > H;
vector< pair<int, int> > D;

int main() {
  cin >> N >> M;
  for (int i = 0; i < N; i++) {
    cin >> position;
    H.push_back(position);
  }
  for (int i = 0; i < M; i++) {
    cin >> position;
    D.push_back(make_pair(position, i));
  }
  D.push_back(make_pair(1000000001, M));
  sort(H.begin(), H.end());
  sort(D.begin(), D.end());
  for (int i = 0; i < M; i++) {
    while (d < M && D[d].second <= i) d++;
    while (h < N && H[h] < D[d].first) h++;
    cout << h << endl;
  }
  return 0;
}
```

Problem J4: Target Practice

We have N regions of a target (1 circle surrounded by $N - 1$ rings) and M coordinates. We want to assign N given point values to the N regions such that the total score is minimized (and then again so it's maximized). The score is defined as the sum of the products of each ring's score with the number of shots that land in each ring. Formally, the problem asks to find the minimum and maximum possible *cross products* between the hit-counts of each ring and the point value we assign to each ring. By definition, the cross-product of two size N arrays A and B is equal to the sum $A[1]\times B[1] + A[2]\times B[2] + \ldots + A[N] \times B[N]$.

By the simple equation of a circle centered at the origin, we know that a shot i lands in ring j iff $\text{sqrt}(X_i^2 + Y_i^2)$ is strictly less than or equal to the outer radius R_j, and strictly greater than R_{j-1} (if $j > 1$). To avoid any possible precision issues with floating point numbers (in cases where a shot lands very close to the border of a ring), this process can be handled entirely using (64-bit) integers, by comparing squared distances instead (for example, comparing $X^2 + Y^2$ against R^2). The hit counts can be precomputed naïvely by looping through each of the M rings, and then counting how many of the N darts land in the ring using the formula above. This will have a running time of $O(NM)$. Then, we can try naïvely permuting all $N!$ orderings of P and assign them to each ring, and then for each permutation computing the cross product of the current permutation and the hit-counts of each ring. Doing this should pass the first subtask in $O(NM + N!) = O(N!)$ time (since $N \leq 10$, so $N! \leq 10! = 3{,}628{,}800$), and is thus given 6/40 points.

Suppose we already have the hit-counts for each ring. Now intuitively, in order to maximize Bond's score, we should assign the largest P values to the rings which are hit by the largest number of shots, and the smallest P values to those which are hit the least (and vice versa to minimize his score). This is as easy as sorting P and the counts in increasing order, calculating the cross-product (for the maximum), then reversing P and calculating the cross-products again (for the minimum). Since each sort takes $N \log N$, we've reduced the time complexity to $O(N \log N + NM) = O(NM)$. This is sufficient to pass all but the last set of cases, obtaining 17/35 points. However, to get full marks, we must handle the bottleneck of computing the hit-counts efficiently.

If we just consider the rings in terms of their distances, then we actually just get a sorted array of integers. To find where a point lands, we want to find the first R value which is not less than the distance of the shot from the origin. Since the distances are sorted, we can use binary search in $O(\log N)$ time to pinpoint the exact position where it lands. Across M shots, the time complexity will be $O(M \log N)$. So, the overall solution will run in $O(M \log N + N \log N)$.

Official Solution (C++)

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

int N, M, P[100005], cnt[1000005] = {0}, ans = 0;
long long R[100005], X[1000005], Y[1000005];

int main() {
  ios_base::sync_with_stdio(false);
  cin >> N >> M;
  for (int i = 0; i < N; i++) {
    cin >> R[i];
    R[i] = R[i] * R[i];
  }
  for (int i = 0; i < N; i++) cin >> P[i];
  for (int i = 0; i < M; i++) {
    cin >> X[i] >> Y[i];
    long long sqdist = X[i] * X[i] + Y[i] * Y[i];
    cnt[lower_bound(R, R + N, sqdist) - R]++;
  }
  sort(P, P + N);
  sort(cnt, cnt + N);
  for (int i = 0; i < N; i++) {
    ans += cnt[i] * P[N - i - 1];
  }
  cout << ans << endl;
  ans = 0;
  for (int i = 0; i < N; i++) {
    ans += cnt[i] * P[i];
  }
  cout << ans << endl;
  return 0;
}
```

Problem S2: World Tour

The cities and flights represent a directed graph with N nodes, each with 1 outgoing edge. For each "component" in the graph (a maximal set of nodes which would be connected to one another if the graph was undirected), the graph structure guarantees that there must be exactly one cycle, with the remaining nodes in the component connecting to that cycle (possible indirectly). An obvious solution would be to start at each node and follow the path until we have reached a node that is already visited, and then stop. Doing so for each node yields an $O(N^2)$ solution which is given 7/23 of the points. To improve on this, we can process one component at a time, finding its cycle and then handling all of its non-cyclic nodes.

We can iterate over the nodes from 1 to N. If the answer A_i for node i hasn't been computed yet, then we'll process node i's entire component right away. The first step is to locate any node which is part of the component's cycle. This can be done by repeatedly following edges forward from i, marking nodes as having been visited, until a node j is reached which has already been visited – this node must be part of the cycle. Next, we need to get the cycle's size s (the number of nodes which are part of it). We can do this by repeatedly following edges forward from j until we return j, and counting the number of nodes visited along the way.

Now, for each node a in the cycle, $A_a = s$ (if Jaws starts in the cycle, he'll just visit all s nodes in the cycle). For each non-cyclic node b which is an additional distance of d away from any node in the cycle, $A_b = d + s$ (Jaws will visit d non-cycle nodes on his way to the cycle, and then visit all s nodes in the cycle). As such, we can compute the answers for all nodes in the component in linear time using breadth-first search, by pushing all the nodes in the cycle onto a queue, and then expanding outwards from the cycle.

If we're processing node x and there's an edge from a non-cycle node y to x, then we can set $A_y = A_x + 1$ and push y onto the queue. Note that this will require precomputing the list of incoming edges for each node. At the end, we can output the computed values $A_{1..N}$. The total time complexity of this algorithm is $O(N)$.

Official Solution (C++)

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

int N, C[500005], ans[500005] = {0};
bool vis[500005] = {0};
vector<int> in[500005];
queue< pair<int, int> > q;

int main() {
  ios_base::sync_with_stdio(false);
  cin >> N;
  for (int i = 1; i <= N; i++) {
    cin >> C[i];
    in[C[i]].push_back(i);
  }
  for (int i = 1; i <= N; i++) {
    if (ans[i] > 0) continue;
    int curr;
    for (curr = i; !vis[curr]; curr = C[curr])
      vis[curr] = true;
    int start = curr, cyclelen = 0, dist;
    do {
      cyclelen++;
      q.push(make_pair(curr, 0));
      ans[curr] = -1;
      curr = C[curr];
    } while (curr != start);
    while (!q.empty()) {
      curr = q.front().first;
      dist = q.front().second;
      q.pop();
      if (dist == 0) dist = cyclelen;
      ans[curr] = dist;
      for (int j = 0; j < in[curr].size(); j++)
        if (!ans[in[curr][j]])
          q.push(make_pair(in[curr][j], dist + 1));
    }
  }
  for (int i = 1; i <= N; i++)
    cout << ans[i] << endl;
  return 0;
}
```

Problem S3: Coded Paper

We are given a 2 by N matrix of numbers and want to maximize their sum by replacing any number of rectangular regions on the matrix with other rectangles, each of which has just the number R written on it. For the first subtask where $N \leq 2$, we can manually generate every possible configuration of rectangle placements. For the second subtask where $N \leq 6$, we can do some kind of brute force to find the answer (possibly using backtracking to repeatedly place rectangles of all possible sizes in all possible configurations).

For an answer that gets full marks, let's first consider the simple case in which R is non-negative. In this case, there's no benefit to using larger cardboard rectangles – they might as well all be 1 by 1 squares to maximize the number of numbers we can replace. In particular, we should just greedily cover every cell with value $C_{i,j}$ less than R.

When R is instead negative, dynamic programming is required. Let $DP[i][s]$ be the maximum possible sum of values in the first i columns, where s is an integer between 0 and 4 describing the state of any ongoing cardboard rectangles. In particular, the following possibilities exist at any column i:

0. No ongoing rectangles
1. Ongoing height-1 rectangle in row 1
2. Ongoing height-1 rectangle in row 2
3. Ongoing height-1 rectangles in both rows
4. Ongoing height-2 rectangle spanning both rows

$DP[i][s]$ is computed by considering each possible previous rectangle state s' (between 0 and 4), and maximizing the value of: DP[i–1][s'] + {sum of visible cells in column i given the rectangles described by s} + R×{number of new rectangles being started going from state s' to s}.

This requires some case analysis of the 5 possible rectangle states and the transitions between pairs of them. These base cases and transition cases are self-explanatory in the implementation given on the left.

The final answer is stored at case 0 of the last colunm, where there are no ongoing rectangles. The time complexity of this algorithm is $O(N)$.

Official Solution (C++)

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

int N, R, d, ans = 0, C[100005][2],
DP[100005][5];

void setmax(int & a, int b) { a = max(a, b); }

int main() {
  cin >> N >> R;
  for (int j = 0; j < 2; j++)
    for (int i = 0; i < N; i++)
      cin >> C[i][j];
  if (R >= 0) {
    for (int i = 0; i < N; i++)
      for (int j = 0; j < 2; j++)
        ans += max(C[i][j], R);
    cout << ans << endl;
  } else {
    for (int i = 0; i <= N; i++)
      for (int j = 0; j < 5; j++)
        DP[i][j] = -1000000001;
    DP[0][0] = 0;
    DP[0][1] = DP[0][2] = DP[0][4] = R;
    DP[0][3] = 2*R;
    for (int i = 0; i < N; i++) {
      d = DP[i][0] + C[i][0] + C[i][1]; //case 0
      setmax(DP[i + 1][0], d);
      setmax(DP[i + 1][1], d + R);
      setmax(DP[i + 1][2], d + R);
      setmax(DP[i + 1][3], d + 2*R);
      setmax(DP[i + 1][4], d + R);
      d = DP[i][1] + C[i][1];           //case 1
      setmax(DP[i + 1][0], d);
      setmax(DP[i + 1][1], d);
      setmax(DP[i + 1][2], d + R);
      setmax(DP[i + 1][3], d + R);
      setmax(DP[i + 1][4], d + R);
      d = DP[i][2] + C[i][0];           //case 2
      setmax(DP[i + 1][0], d);
      setmax(DP[i + 1][1], d + R);
      setmax(DP[i + 1][2], d);
      setmax(DP[i + 1][3], d + R);
      setmax(DP[i + 1][4], d + R);
      d = DP[i][3];                     //case 3
      setmax(DP[i + 1][0], d);
      setmax(DP[i + 1][1], d);
      setmax(DP[i + 1][2], d);
      setmax(DP[i + 1][3], d);
      setmax(DP[i + 1][4], d + R);
      d = DP[i][4];                     //case 4
      setmax(DP[i + 1][0], d);
      setmax(DP[i + 1][1], d + R);
      setmax(DP[i + 1][2], d + R);
      setmax(DP[i + 1][3], d + 2*R);
      setmax(DP[i + 1][4], d);
    }
    cout << DP[N][0] << endl;
  }
  return 0;
}
```

Problem S4: Stakeout

For starters, we need to determine which buildings each agent can see. After sorting the positions of the buildings in $O(N \log N)$ time, we can determine the interval of buildings that each agent i can cover in $O(\log N)$ time using binary search, by searching for the first building whose position is no smaller than $A_i - R_i$, and the last building whose position is no greater than $A_i + R_i$.

As for answering the questions, we can observe that $2^i > 2^1 + 2^2 + ... + 2^{i-1}$, meaning that for any i, hiring any subset of agents with indices smaller than i is worth it if it means we can avoid hiring agent i. This suggests the approach of iterating over the agents downwards from M to 1, and for each one, only hiring them if necessary – in other words, if skipping that agent and hiring all of the agents with indices smaller than i wouldn't yield sufficient coverage of the buildings.

To implement this approach, we can start by assuming that we'll hire all M agents, and then for each agent i, we can try removing them and testing if each building is still covered by enough agents – if not, agent i must be necessary, so we must re-insert them into the set of hired agents and add 2^i onto the total necessary cost (assuming that we've precomputed the values of 2^i modulo $10^9 + 7$ for $i = 1..M$). If even hiring all M agents to start with isn't sufficient, then the answer can immediately be determined to be –1.

What remains is being able to execute the above operations efficiently. In particular, we must be able to insert and remove agents from the set of hired agents, and test if all buildings are in sight of at least C_i hired agents. If we imagine an array S such that S_i is the number of hired agents that building S is in sight of, then these operations equate to adding 1 to an interval of S values, subtracting 1 from an interval of S values, and determining the minimum S value in the array.

Now, each of these operations can be handled by a segment tree with lazy propagation in $O(\log N)$ time. Each node in the tree should store the minimum value in its interval, as well as a lazy value of how much should be added to (or subtracted from) its entire interval. Conveniently, adding a constant value c to every index in an interval results in that interval's minimum value also increasing by c.

For each question, we hire all of the agents and then iterate over all of them in an attempt to remove them, performing one or two segment tree operations each time. Therefore, the total time complexity is $O((N + QM) \log N)$.

Official Solution (C++)

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

const int LIM = 300005, MOD = 1000000007;

int N, M, Q, A, R, C, total_cost = 0;
int P[LIM], PL[LIM], PH[LIM], pow2[LIM], cov[LIM];
int tree[4 * LIM], lazy[4 * LIM];

void build(int i, int l, int h) {
  if (l + 1 == h) {
    tree[i] = cov[l];
  } else {
    int m = (l + h)/2;
    build(2*i + 1, l, m);
    build(2*i + 2, m, h);
    tree[i] = min(tree[2*i + 1], tree[2*i + 2]);
    lazy[i] = 0;
  }
}
```

```
void propagate(int i) {
  tree[2*i + 1] += lazy[i];
  lazy[2*i + 1] += lazy[i];
  tree[2*i + 2] += lazy[i];
  lazy[2*i + 2] += lazy[i];
  lazy[i] = 0;
}

int query(int i, int tl, int th, int ql, int qh) {
  if (ql <= tl && qh >= th) return tree[i];
  propagate(i);
  int tm = (tl + th)/2, ret = 1000000001;
  if (ql < tm && qh > tl) ret = min(ret, query(2*i + 1, tl, tm, ql, qh));
  if (ql < th && qh > tm) ret = min(ret, query(2*i + 2, tm, th, ql, qh));
  return ret;
}

void update(int i, int tl, int th, int ql, int qh) {
  if (ql <= tl && qh >= th) {
    tree[i]--; lazy[i]--;
    return;
  }
  if (tl + 1 == th) return;
  propagate(i);
  int tm = (tl + th)/2;
  if (ql < tm && qh > tl) update(2*i + 1, tl, tm, ql, qh);
  if (ql < th && qh > tm) update(2*i + 2, tm, th, ql, qh);
  tree[i] = min(tree[2*i + 1], tree[2*i + 2]);
}

int main() {
  ios_base::sync_with_stdio(false);
  cin >> N >> M >> Q;
  for (int i = 0; i < N; i++) cin >> P[i];
  sort(P, P + N);
  pow2[0] = 1;
  for (int i = 0; i < M; i++) {
    cin >> A >> R;
    PL[i] = lower_bound(P, P + N, A - R) - P;
    PH[i] = upper_bound(P, P + N, A + R) - P;
    cov[PL[i]]++; cov[PH[i]]--;
    pow2[i + 1] = (pow2[i] * 2) % MOD;
    total_cost = (total_cost + pow2[i + 1]) % MOD;
  }
  int init_min = cov[0];
  for (int i = 1; i < N; i++) {
    cov[i] += cov[i - 1];
    init_min = min(init_min, cov[i]);
  }
  while (Q--) {
    cin >> C;
    if (C > init_min) {
      cout << -1 << endl;
      continue;
    }
    int ans = total_cost;
    build(0, 0, N);
    for (int i = M - 1; i >= 0; i--) {
      if (PL[i] == PH[i] || query(0, 0, N, PL[i], PH[i]) > C) {
        ans -= pow2[i + 1];
        if (ans < 0) ans += MOD;
        update(0, 0, N, PL[i], PH[i]);
      }
    }
    cout << ans << endl;
  }
  return 0;
}
```