WOBURN CHALLENGE

2018-19 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: An Honest Day's Work

With P litres of paint and each badge requiring B litres of it, James will produce $\lfloor P / B \rfloor$ badges. There will be $P - \lfloor P / B \rfloor \times B$ litres of paint left over, which is equivalent to P modulo B (the remainder when P is divided by B, written as the "`%`" operator in most programming languages). Therefore, the number of Pokédollars made will be $\lfloor P / B \rfloor \times D + (P \text{ modulo } B)$.

Official Solution (C++)

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

int main() {
  int P, B, D;
  cin >> P >> B >> D;
  cout << (P / B)*D + (P % B) << endl;
  return 0;
}
```

Problem J2: Net Weight

We'll iterate over the N Pikachus while maintaining two values m_1 and m_2 – the largest and second-largest weights of catchable Pikachus seen so far, respectively. Initially, $m_1 = m_2 = 0$.

When considering the i-th Pikachu, we should ignore it completely if $W_i > 100$. Otherwise, if $W_i > m_1$, then we've found a new heaviest catchable Pikachu, demoting the previous heaviest one to now be the second-heaviest – therefore, we should first set m_2 to equal m_1, and then set m_1 to equal W_i. Otherwise, if $W_i > m_2$, then we've found a new second-heaviest catchable Pikachu, so we should simply set m_2 to be equal to W_i.

After considering all N Pikachus in this fashion, $m_1 + m_2$ comes out to the sum of the two heaviest catchable Pikachus' weights, which is the answer we're looking for. Note that if there were fewer than two catchable Pikachus, m_1 and/or m_2 will still be equal to 0, which is desirable as it represents Jessie and/or James remaining empty-handed.

Official Solution (C++)

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

int main() {
  int N;
  cin >> N;
  int max1 = 0, max2 = 0;
  for (int i = 0, W; i < N; i++) {
    cin >> W;
    if (W <= 100) {
      if (W > max1) {
        max2 = max1;
        max1 = W;
      } else if (W > max2) {
        max2 = W;
      }
    }
  }
  cout << max1 + max2 << endl;
  return 0;
}
```

Problem J3/I1: R

We'll go row by row from top to bottom, determining and outputting each one as we go.

The first row consists of $R - 1$ "#"s followed by one ".".

The next $R - 2$ rows each consist of one "#", followed by $R - 2$ "."s, followed by one "#".

The next row is the same as the first row.

Finally, $R - 1$ rows remain. The i-th of these contains a "#" in column 1 and another in column $i + 1$. In other words, it consists of one "#", followed by $i - 1$ "."s, followed by one "#", followed by $R - i - 1$ "."s.

Official Solution (C++)

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

int main() {
  int S;
  cin >> S;
  for (int r = 0; r < S; r++) {
    for (int c = 0; c < S; c++) {
      if (r == 0 || r == S - 1) {
        cout << (c < S - 1 ? "#" : ".");
      } else {
        cout << (c == 0 || c == S - 1 ? "#" : ".");
      }
    }
    cout << endl;
  }
  for (int r = 1; r < S; r++) {
    for (int c = 0; c < S; c++) {
      cout << (c == 0 || c == r ? "#" : ".");
    }
    cout << endl;
  }
  return 0;
}
```

Problem J4/I2: Leveling Up

Let Z be the maximum possible position (Z = 100,000), and let WM_p and WG_p be the M and G values of the wild Pokémon at position p (if any), with $WM_p = WG_p = 0$ if there's no such Pokémon. Then, $WM_1, \ldots, WM_Z$ and $WG_1, \ldots, WG_Z$ may be populated in $O(Z + N)$ time by iterating over the N Pokémon.

We'll greedily simulate Jessie and Arbok's training process. Let p_1 be the smallest position which Jessie can currently freely reach, p_2 be the largest one, and a be Arbok's current level. Initially, $p_1 = p_2 = S$ and $a = L$.

At any point in time, if $p_1 - 1 \geq 1$ and $a \geq WM_{p_1 - 1}$, then Jessie is able to expand her reachable range to encompass position p_1 - 1, and has no reason not to do so. Therefore, we can increase a by $WG_{p_1 - 1}$, and then decrement p_1. Similarly, if $p_2 + 1 \leq Z$ and $a \geq WM_{p_2 + 1}$, then Jessie might as well expand her reachable range to encompass position $p_2 + 1$. Therefore, we can increase a by $WG_{p_2 + 1}$, and then increment p_2.

We should repeat the above process of expanding Jessie's range of reachable positions left and right whenever possible until we arrive in a situation in which it cannot be expanded in either direction. At that point, nothing more can be done, and we can proceed to output the current value of a.

The time complexity of the above solution is $O(Z + N)$. A similar algorithm running in $O(N \log N)$ time is also possible, if we sort and consider only positions of interest (Jessie's starting position as well all wild Pokémons' positions) instead of all Z possible positions.

Official Solution (C++)

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

const int MAXP = 100000;

int M[MAXP + 1], G[MAXP + 1];

int main() {
  int N, S, L;
  cin >> N >> S >> L;
  for (int i = 0; i < N; i++) {
    int P;
    cin >> P;
    cin >> M[P] >> G[P];
  }
  // Simulate expanding reachable interval left/right.
  int pLeft = S, pRight = S;
  for (;;) {
    if (pLeft > 1 && L >= M[pLeft - 1]) {
      // Can move left, so do so.
      L += G[--pLeft];
    } else if (pRight < MAXP && L >= M[pRight + 1]) {
      // Can move right, so do so.
      L += G[++pRight];
    } else {
      // Can move neither left nor right, so stop.
      break;
    }
  }
  cout << L << endl;
  return 0;
}
```

Problem I3/S1: The Perfect Team

The highest-level Pokémon of each type 1..K should certainly be chosen. In addition to those K Pokémon, the $M - K$ highest-level Pokémon of the remaining N – K ones should also be chosen to round out the team. We can refer to these additional $M - K$ Pokémon as "extras".

Let's sort all N Pokémon in non-increasing order of level, such that the highest-level ones come first, and then iterate over them in that order while greedily determining which ones to choose. Along the way, we'll maintain several pieces of information: the level sum of Pokémon chosen so far, an array indicating whether or not we've already chosen a Pokémon of type t (for each possible t), and the number of extras chosen so far.

When considering the i-th Pokémon, if we have not yet chosen a Pokémon of type T_i, then we should go ahead and choose this one, as it must be the highest-level Pokémon of its type. Otherwise, if we have not yet chosen $M - K$ extras, then we should go ahead and choose this one as an extra, as it must be the highest-level valid extra.

Upon processing all N Pokémon in this manner, we're guaranteed to have successfully chosen the highest-level one of each type as well as the $M - K$ highest-level valid extras, meaning that our aggregated sum of chosen Pokémon levels must be optimal. The time complexity of this solution is $O(N \log N)$, necessary for sorting the Pokémon.

Official Solution (C++)

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

const int MAXN = 300001;

pair<int, int> P[MAXN];
bool seen[MAXN];

int main() {
  int N, M, K;
  cin >> N >> M >> K;
  for (int i = 0; i < N; i++) {
    cin >> P[i].second >> P[i].first;
  }
  // Iterate over Pokemon in non-increasing order of level.
  int extra = M - K;
  long long ans = 0;
  sort(P, P + N);
  for (int i = N - 1; i >= 0; i--) {
    int T = P[i].second, L = P[i].first;
    if (!seen[T]) {
      // First time seeing a Pokemon of this type, definitely include it.
      seen[T] = true;
      ans += L;
    } else if (extra > 0) {
      // Already seen a Pokemon of this type, but there's room for this one too.
      extra--;
      ans += L;
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem I4/S2: Gym Tour

There are two possibly optimal strategies:

1. Never Fly, just visit all of the gyms by walking
2. Walk directly to town F, and then visit all of the gyms while Flying freely

We'll evaluate the minimum amount of time required to execute each strategy, and then use the faster one.

In order to evaluate strategy 1, we can think of the network of towns and roads as a rooted tree with N nodes (one for each town), rooted at node 1. Now, let's first consider a variation in which Team Rocket is required to return to node 1 at the end. In this variation, for each node i (aside from the root) such that there's at least one gym anywhere in i's subtree, Team Rocket will need to walk down from i's parent to i once, and then walk back up from i to i's parent once. In other words, the number of days required is equal to two times the number of such nodes i. The only difference between this variation and the actual strategy is that Team Rocket will save returning from their last visited gym to node 1, which takes a number of days equal to the depth of that gym in the tree. Therefore, to save

as much time as possible, they should visit the deepest gym last. This means that, to evaluate strategy 1, we only need to determine the number of nodes i whose subtrees contain at least one gym as well the maximum depth of any gym, both of which can be found by recursively traversing the tree in $O(N)$ time.

In order to evaluate strategy 2, we can make a similar observation: for each node i (aside from the root) such that its subtree contains either at least one gym or node F, Team Rocket will need to walk down from i's parent to i once, and that's it. Therefore, this strategy can be solved very similarly to strategy 1, by counting the number of such nodes recursively. It's possible to even reuse the same recursive function for both strategies.

Official Solution (C++)

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

const int MAXN = 300001;

int N, K, F;
bool req[MAXN];
vector<int> adj[MAXN];
int numEdges, maxDepth;

// Returns true if any required nodes are in i's subtree, and populates numEdges / maxDepth.
bool DFS(int i, int dep = 0, int par = 0) {
  bool hasReq = req[i];
  if (req[i]) {
    maxDepth = max(maxDepth, dep);
  }
  for (int j = 0; j < adj[i].size(); j++) {
    if (adj[i][j] != par && DFS(adj[i][j], dep + 1, i)) {
      numEdges++;
      hasReq = true;
    }
  }
  return hasReq;
}

int main() {
  cin >> N >> K >> F;
  for (int i = 0, G; i < K; i++) {
    cin >> G;
    req[G] = true;
  }
  for (int i = 0, a, b; i < N - 1; i++) {
    cin >> a >> b;
    adj[a].push_back(b);
    adj[b].push_back(a);
  }
  // Without Fly.
  numEdges = maxDepth = 0;
  DFS(1);
  int ans1 = numEdges*2 - maxDepth;
  // With Fly.
  req[F] = true;
  numEdges = maxDepth = 0;
  DFS(1);
  int ans2 = numEdges;
  cout << min(ans1, ans2) << endl;
  return 0;
}
```

Problem S3: Counterpicking

For each Pokémon trainer i, let $R_i = X_i / Y_i$. We can observe that this ratio is sufficient to determine which of Jessie's Pokémon is optimal to use against that trainer. We can furthermore observe that each of Jessie's Pokémon is either never optimal, or is optimal for a single interval of ratios.

Our objective will be to pre-process Jessie's set of N Pokémon into a sequence of Pokémon $P_{1..K}$ and corresponding splitting points $S_{1..(K-1)}$, such that P_1 is optimal for ratios in the interval $(-\infty, P_1)$, P_2 is optimal for ratios in the interval $[S_1, S_2)$, and so on, with P_K being optimal for ratios in the interval $[S_{K-1}, \infty)$. If we can perform this pre-processing, then we'll be able to obtain the answer for each trainer i in $O(\log N)$ time by binary searching on $S_{1..(K-1)}$ for the interval containing R_i, and using the corresponding optimal Pokémon.

Given two of Jessie's Pokémon i and j, let's consider when one is more optimal than the other. If $A_i = A_j$, then whichever Pokémon has a larger B value is simply always better. Otherwise, if we assume that $A_i < A_j$ and let $f(i, j) = (B_j - B_i) / (A_i - A_j)$, then the two Pokémon are equally optimal for the ratio $f(i, j)$, Pokémon i is more optimal for all ratios smaller than $f(i, j)$, and Pokémon j is more optimal for all ratios greater than $f(i, j)$.

Given that fact, let's sort Jessie's Pokémon in non-decreasing order of A values and then iterate over them in that order, while ignoring any which are known to never be optimal (due to having an equal A value to another Pokémon and a smaller B value). As we go, we'll maintain our sequence P of optimal Pokémon, which is initially empty. Each time we process a new Pokémon, we'll add it onto the end of P, but first we may need to remove zero or more Pokémon from the end of P if their intervals of optimal ratios have been rendered empty. For example, if the last two Pokémon in P are P_{K-1} and P_K, we're currently processing Pokémon i, and $f(P_{K-1}, P_K) > f(P_K, i)$, then there is no ratio for which Pokémon P_K is optimal, and it should be removed from P.

Sorting Jessie's Pokémon takes $O(N \log N)$ time, and then processing them to generate the sequence $P_{1..K}$ takes another $O(N)$ time. The sequence of splitting points $S_{1..(K-1)}$ may then be generated based on it in $O(N)$ time, with $S_i = f(P_i, P_{i+1})$ for each i. Finally, we can use this to compute the optimal answers for all of the Pokémon trainers as described above, bringing the total time complexity up to $O((N + M) \log N)$.

Official Solution (C++)

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

const int MAXN = 300000;
typedef pair<int, int> point;
#define x first
#define y second

point P[MAXN];
long double Q[MAXN];

long double InterQ(const point &A, const point &B) {
  return (long double)(B.y - A.y) / (A.x - B.x);
}

int main() {
  int N;
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> P[i].x >> P[i].y;
  }
```

```
  // Sort pairs, and remove ones strictly worse than others.
  sort(P, P + N);
  int n = 0;
  for (int i = 0; i < N; i++) {
    while (n >= 1 && P[n - 1].x == P[i].x) {
      n--;
    }
    while (n >= 2 && InterQ(P[n - 2], P[n - 1]) > InterQ(P[n - 1], P[i])) {
      n--;
    }
    P[n++] = P[i];
  }
  N = n;
  // Compute break-even quotients.
  for (int i = 0; i < N - 1; i++) {
    Q[i] = InterQ(P[i], P[i + 1]);
  }
  // Process queries.
  int M;
  cin >> M;
  while (M--) {
    long long X, Y;
    cin >> X >> Y;
    int i = lower_bound(Q, Q + N - 1, (long double)X / Y) - Q;
    cout << X*P[i].x + Y*P[i].y << endl;
  }
  return 0;
}
```

Problem S4: Holey Travels

There must always exist an optimal circle which is tangent to at least one trainer *i*'s line (it can't be better to choose a circle tangent to no lines, as it can always be shifted until it is tangent to one). We'll consider each possible line *i* to place the circle tangent to. Given such a line, the circle can then lie on either of the two sides of the line, and we'll consider both possible sides.

Given a line *i* and one of its sides, the set of possible circle centers forms a line *c* running parallel to line *i*, *R* units away from it. We'll compare this line *c* against each trainer *j*'s line:

- If line *j* is parallel to line *c*, then a circle centered anywhere on line *c* will either always or never intersect with line *j*, depending on whether the distance between those two lines is greater than *R*. Let *a* be the sum of *P* values corresponding to all such lines *j* which will always be intersected (note that these always include line *i* itself).
- Otherwise, if line *j* is not parallel to line *c*, then there exists a single interval of circle centers along line *c* for which the circle will intersect with line *j*. This interval is centered at the intersection of lines *c* and *j*, and its boundaries may be computed with some trigonometry. In particular, we'll want to imagine that line *c* is a number line (with an arbitrary origin), and represent each interval's start and end points as scalar points along it.

The number of Pokémon trapped by centering the circle at a given point along line *c* is then equal to *a* plus the sum of *P* values corresponding to all intervals overlapping with that point. The maximum possible value of this expression can be found by performing a line sweep along line *c*.

For a given line *c*, comparing it to all of the trainers' lines takes $O(N)$ time, and then performing a line sweep on the resulting intervals takes $O(N \log N)$ time. We'll repeat this whole process twice for each trainer *i*'s line (once per side) to determine the maximum number of Pokémon which can be trapped by any possibly optimal circle, bringing the total time complexity up to $O(N^2 \log N)$.

Official Solution (C++)

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

const int MAXN = 1000;

struct point {
  long double x, y;

  point() : x(0), y(0) {}
  point(long double x, long double y) : x(x), y(y) {}
  point(const point &p) : x(p.x), y(p.y) {}

  point operator+(const point &p) const { return point(x + p.x, y + p.y); }
  point operator-(const point &p) const { return point(x - p.x, y - p.y); }
  point operator*(long double c) const { return point(x * c, y * c); }
  point operator/(long double c) const { return point(x / c, y / c); }

  point rotate90cw() const { return point(y, -x); }
  point rotate90ccw() const { return point(-y, x); }

  long double dot(const point &p) const { return x*p.x + y*p.y; }
  long double norm() const { return sqrt(x*x + y*y); }
  long double cross(const point &p) const { return x*p.y - y*p.x; }
};


point ProjectPointLine(const point &a, const point &b, const point &c) {
  return a + (b - a)*(c - a).dot(b - a)/(b - a).dot(b - a);
}

bool LinesParallel(const point &a, const point &b,
                   const point &c, const point &d) {
  return fabs((b - a).cross(c - d)) < 1e-6;
}

point ComputeLineIntersection(const point &a, const point &b,
                              const point &c, const point &d) {
  b = b - a;
  d = c - d;
  c = c - a;
  return a + b*c.cross(d) / b.cross(d);
}

long double ComputeAngleBetweenLines(const point &a, const point &b,
                                     const point &c, const point &d) {
  b = b - a;
  d = c - d;
  return acos(b.dot(d) / (b.norm()*d.norm()));
}

int N;
long double R;
point A[MAXN], B[MAXN];
int P[MAXN];

int main() {
  cin >> N >> R;
  for (int i = 0; i < N; i++) {
    cin >> A[i].x >> A[i].y >> B[i].x >> B[i].y >> P[i];
  }
```

```
  // Consider each line to place the circle tangent to.
  int ans = 0;
  for (int i = 0; i < N; i++) {
    // Consider both sides of the line.
    for (int s = 0; s < 2; s++) {
      // Get the line on which to place the circle's center.
      point dir = B[i] - A[i];
      dir = dir / dir.norm();
      point tang = s ? dir.rotate90ccw() : dir.rotate90cw();
      point lnA = A[i] + tang * R;
      point lnB = lnA + dir;
      // Compare against all lines.
      int curSum = 0;
      vector < pair<long double, int> > evs;
      for (int j = 0; j < N; j++) {
        // Parallel?
        if (LinesParallel(lnA, lnB, A[j], B[j])) {
          // Close enough?
          if ((A[j] - ProjectPointLine(lnA, lnB, A[j])).norm() < R + 1e-7) {
            curSum += P[j];
          }
        } else {
          // Compute intersection point of lines (as signed distance along line).
          point inter = ComputeLineIntersection(lnA, lnB, A[j], B[j]);
          long double interDist = (inter - lnA).norm();
          if ((inter - lnA).cross(inter - lnA + tang) < 0) {
            interDist = -interDist;
          }
          // Compute max distance of circle center from intersection point.
          long double ang = ComputeAngleBetweenLines(lnA, lnB, A[j], B[j]);
          long double dist = R / sin(ang);
          // Store range of valid circle center positions (as signed distances along line).
          evs.push_back(make_pair(interDist - dist, P[j]));
          evs.push_back(make_pair(interDist + dist, -P[j]));
        }
      }
      // Line sweep over possible circle centers.
      ans = max(ans, curSum);
      sort(evs.begin(), evs.end());
      for (int j = 0; j < evs.size(); j++) {
        curSum += evs[j].second;
        ans = max(ans, curSum);
      }
    }
  }
  cout << ans << endl;
  return 0;
}
```