WOBURN CHALLENGE

2017-18 On-Site Finals

Solutions

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

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

Problem J1: Cownterintelligence

Upon inputting each cow i's moo frequency M_i, we'll need to determine whether it's a power-of-2 multiple of F, and output the number i if it's not.

One way to check this is to initialize a variable x to be equal to F, and then repeatedly multiply x by 2 while checking if it's equal to M_i at each iteration. If we do find a value of x which is equal to M_i, then we know that cow i isn't an imposter, and we'll want to skip over it. On the other hand, if x never becomes equal to M_i and instead gets to be larger than M_i, then we know that cow i is an alien imposter.

Official Solution (C++)

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

int main() {
  int N, F;
  cin >> N >> F;
  // Iterate over the cows.
  for (int i = 1; i <= N; i++) {
    int M;
    cin >> M;
    // Check if it's a power-of-2 multiple of F.
    bool alien = true;
    for (int x = F; x <= M; x *= 2) {
      if (x == M) {
        alien = false;
      }
    }
    // Output i if it's an alien.
    if (alien) {
      cout << i << endl;
    }
  }
  return 0;
}
```

Problem J2: Redundant Formations

Our overall approach should be to consider each possible substring of S, check whether or not it's redundant, assemble a list of distinct substrings which are, and output the number of substrings in this list at the end.

We can do so by iterating over all pairs of indices (i, j), such that $1 \leq i < j \leq |S|$. Each such pair corresponds to the substring $S_{i..j}$, with a length of $L = j - i + 1$. We can then consider each index k between $i + 1$ and j (inclusive), and check if the substring $S_{k..(k+L-1)}$ is equal to $S_{i..j}$ (note that k should also be capped at $|S| - L + 1$ to avoid considering substrings which would go past the end of S). If such an index k is found, then a pair of overlapping equal substrings has also been found, meaning that $S_{i..j}$ must be redundant.

What remains is maintaining the list of distinct redundant substrings, and being able to add $S_{i..j}$ to it if not already present. One option is to maintain this list as an array, and loop over all of its existing entries to check if $S_{i..j}$ is equal to any of them. A more elegant option is to use a built-in data structure such as a C++ set, which will automatically ensure that its entries are all distinct.

The time complexity of the solution described above is $O(N^4)$, which is comfortably fast enough, though it's also possible to optimize it to $O(N^3)$.

Official Solution (C++)

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

int main() {
  string S;
  cin >> S;
  int N = S.size();
  // Consider all possible substrings and find redundant ones.
  set<string> ans;
  for (int i = 0; i < N; i++) {
    for (int j = i + 1; j < N; j++) {
      // Get substring at indices i..j.
      int len = j - i + 1;
      string s = S.substr(i, len);
      // Check if this substring occurs again starting between i+1 and j.
      for (int k = i + 1; k <= j && k + len <= N; k++) {
        if (S.substr(k, len) == s) {
          ans.insert(s);
          break;
        }
      }
    }
  }
  cout << ans.size() << endl;
  return 0;
}
```

Problem J3/S1: An Interspecific Army

To minimize D, it turns out that it's always optimal to pair the worst cow with the worst monkey, the second-worst cow with the second-worst monkey, and so on. To prove this, let's consider a pair of cows with combat skill levels C_1 and C_2 (such that $C_1 < C_2$), and a pair of monkeys with combat skill levels M_1 and M_2 (such that $M_1 < M_2$). We note that forming the pairs (C_1, M_2) and (C_2, M_1) can never be better than forming the pairs (C_1, M_1) and (C_2, M_2). In other words, $\max(|C_1 - M_2|, |C_2 - M_1|) \geq \max(|C_1 - M_1|, |C_2 - M_2|)$.

Having made this insight, we'll want to sort the two lists $C_1, \ldots, C_N$ and $M_1, \ldots, M_N$ in non-decreasing order, which can be done in $O(N \log N)$ time. We can then simply find and output the maximum value of $|C_i - M_i|$ over $i = 1..N$.

Official Solution (C++)

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

int main() {
  int N, C[100005], M[100005], D = 0;
  cin >> N;
  for (int i = 0; i < N; i++) cin >> C[i];
  for (int i = 0; i < N; i++) cin >> M[i];
  // Sort cow/monkey combat skill levels.
  sort(C, C + N);
  sort(M, M + N);
  // Greedily pair them up in order.
  for (int i = 0; i < N; i++) {
    D = max(D, abs(C[i] - M[i]));
  }
  cout << D << endl;
  return 0;
}
```

Problem J4/S2: Cowmmunication Network

This problem can be represented as a graph, with N nodes (one for each combat unit) and M undirected, weighted edges (one for each potential communication channel). We're then asked to find a subset of the edges with maximum total weight which result in the entire graph being connected. This sounds very similar to finding the minimum spanning tree of the graph – in this case, more of its "maximum spanning tree", which is essentially the same thing. Note that any algorithm for finding a graph's minimum spanning tree could instead find the maximum spanning tree by inverting some comparisons.

So, the only real difference is that we're allowed to use as many edges as we want, rather than exactly $N - 1$ edges. Note that this difference goes away if all edges have negative weights, as we'd want to use as few of them as possible (in other words, $N - 1$ of them).

The fact that some edges can have positive weights ends up not changing the solution too significantly. All such edges should clearly be used, even if they are not necessary for the graph's maximum spanning tree. Therefore, we can essentially find the maximum spanning tree as normal, and then additionally use all remaining edges with positive weights.

There are two standard algorithms for finding maximum spanning trees: Kruskal's and Prim's. Both can be slightly modified to solve this problem. Kruskal's works out very simply – when processing each edge (in non-increasing order of weight), it should still be used if it would connect two nodes which aren't yet connected, but it should also be used if its weight is positive. Prim's takes a bit more work, but one way to modify it is to keep track of exactly which edges get used in the maximum spanning tree, and then add on all unused positive-weight edges at the end. Both algorithms can be implemented in $O(N + M \log M)$ time.

Official Solution (C++)

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

const int MAXN = 100005;
const int MAXM = 200000;

struct Edge {
  int a, b, c;
  Edge() {}

  bool operator<(const Edge &B) const {
    return c > B.c;
  }
};

int N, M;
int Root[MAXN], Rank[MAXN];
Edge E[MAXM];

// Returns the root of i's component in the disjoint set union.
int Find(int i) {
  if (Root[i] == i) {
    return Root[i];
  }
  return Root[i] = Find(Root[i]);
}

// Returns false if i and j are already in the same component in the disjoint set union,
// otherwise merges their components and returns true.
```

```
bool Merge(int i, int j) {
  i = Find(i), j = Find(j);
  if (i == j) {
    return false;
  }
  if (Rank[i] > Rank[j]) {
    Root[j] = i;
  } else {
    if (Rank[i] == Rank[j]) {
      Rank[j]++;
    }
    Root[i] = j;
  }
  return true;
}

int main() {
  cin >> N >> M;
  for (int i = 1; i <= N; i++) {
    Root[i] = i;
  }
  for (int i = 0; i < M; i++) {
    cin >> E[i].a >> E[i].b >> E[i].c;
  }
  // Sort edges by non-increasing C.
  sort(E, E + M);
  // Greedily construct "MST" with modified Kruskal's.
  long long ans = 0;
  for (int i = 0; i < M; i++) {
    // Positive edges should always be used, others only if necessary.
    if (Merge(E[i].a, E[i].b) || E[i].c > 0) {
      ans += E[i].c;
    }
  }
  // Impossible?
  for (int i = 2; i <= N; i++) {
    if (Merge(1, i)) {
      cout << "Impossible" << endl;
      return 0;
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem J5/S3: Explosive Ordinance Disposal

In order to minimize the sum of $V_{1..N}$, we'll want to generally use the smallest V values possible. After some thought, it should become clear that very small values will indeed do the trick – in fact, limiting ourselves to just $1 \leq V_i \leq 3$ is always sufficient to produce a valid solution. However, upon further thought, we realize that those values may not quite yield an optimal solution – in particular, it may be more optimal overall to use various products of small primes. For example, an instance of $V_i = 6$ could allow many other V values to be reduced from 3 to 2. It's difficult to say exactly how large the V values in an optimal solution might get for a given value of N, but limiting ourselves to all possible values between 1 and 30 (inclusive) will be more than enough.

That insight isn't enough to directly suggest how the V values should optimally be assigned, but at that point we can use dynamic programming to efficiently find an optimal assignment. Let's represent the system of terminals and wires as a tree, rooted at an arbitrary node such as node 1. Then, let $DP[i][v]$ be the minimum sum of V values in the subtree rooted at i, such that $V_i = v$. $DP[i][v]$ can be computed recursively based on the DP values of i's children. Each child c can be handled independently, and we can simply consider all possible choices for V_c such that the

GCD of V_i and V_c satisfies the wire connecting terminals i and c, choosing the one which minimizes $DP[c][V_c]$. At the end, the answer will be $\min\{DP[1][v]\}$ over all possible values of v. The time complexity of this algorithm is $O(N)$.

Official Solution (C++)

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

const int MAXN = 201;

struct Edge {
  int i, c;
  Edge(int i, int c) : i(i), c(c) {}
};

vector<Edge> E[MAXN];
int DP[MAXN][31];  // DP[i][v] = minimum sum of labels in node i's subtree, with i having label v.

int GCD(int a, int b) {
  return b ? GCD(b, a % b) : a;
}

int Solve(int i, int v, int p) {
  // Already memoized?
  if (DP[i][v]) {
    return DP[i][v];
  }
  // Determine best label for each child node.
  int d = v;
  for (int j = 0; j < E[i].size(); j++) {
    if (E[i][j].i != p) {
      int m = 1e9;
      for (int v2 = 1; v2 <= 30; v2++) {
        int g = GCD(v, v2);
        if ((E[i][j].c == 0) == (g == 1)) {
          m = min(m, Solve(E[i][j].i, v2, i));
        }
      }
      d = min(d + m, (int)1e9);
    }
  }
  // Memoize result.
  return DP[i][v] = d;
}

int main() {
  // Input tree.
  int N;
  cin >> N;
  for (int i = 0; i < N - 1; i++) {
    int a, b, c;
    cin >> a >> b >> c;
    E[a].push_back(Edge(b, c));
    E[b].push_back(Edge(a, c));
  }
  // Root tree at node 1 and consider each possible label for it.
  int ans = 1e9;
  for (int v = 1; v <= 30; v++) {
    ans = min(ans, Solve(1, v, 0));
  }
  cout << ans << endl;
  return 0;
}
```

Problem S4: Ultimatum

Let's start by considering how to evaluate the answer to a given query i at all, before worrying about how to do so efficiently. Let h be the height of the X_i-th tallest skyscraper. Next, let c be the total number of height-h skyscrapers, and let v be the number of height-h skyscrapers which will be vaporized. Note that this means that, for any given height-h skyscraper, the probability that it'll be vaporized is v/c. The answer can then be expressed as the sum of the following 3 values:

1. The total number of citizens in the interval $[L_i, R_i]$.
2. The expected number of citizens in vaporized height-h skyscrapers outside the interval $[L_i, R_i]$. Note that this is equal to v/c multiplied by the total number of citizens in height-h skyscrapers outside the interval $[L_i, R_i]$, due to linearity of expectation.
3. The total number of citizens in skyscrapers taller than h outside the interval $[L_i, R_i]$.

Naively, at least values 2 and 3 take $O(N)$ time to compute per query, so next we'll need to look for an efficient way to share computations between queries. These two values depend on h, which depends on X_i, which suggests that processing the M queries in non-decreasing order of X may be most convenient. So, we'll want to input all of them, sort them by X, compute and store their answers, and output them in the original order at the end.

Let's imagine that X starts off at 0. Then, each time X increases by one, either h decreases to a new smaller value (at which point we can count its corresponding value of c, and set v to 1), or h stays constant and v increases by one. This means that we can easily keep track of the v/c portion of the answer. The tricky remaining part is being able to look up interval sums of C_i values efficiently, for three types of skyscrapers – all of them, only ones with exactly the current height h, and only ones taller than height h. An advanced data structure such as a binary indexed tree (or a more general segment tree) will do the trick here. For example, we can maintain three binary indexed trees, one for each of the three types of skyscrapers mentioned above, and update them appropriately whenever h decreases. In total, each C_i value will be inserted and removed at most once into each of the binary indexed trees, with a time complexity of $O(\log N)$ each, and each binary indexed tree will be queried once per query, also with a time complexity of $O(\log N)$ each. Overall, this approach can achieve a time complexity of $O((N + M) \log N)$.

Official Solution (C++)

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

const int MAXN = 200005;

typedef long long LL;
typedef long double LD;
typedef pair<int, int> PR;

int N, M, H[MAXN], C[MAXN];
PR ord[MAXN];
vector<pair<int, PR>> Q[MAXN];
LL BIT[3][MAXN];  // #0: All skyscrapers, #1: Current-height skyscrapers, #2: Taller skyscrapers.
LD ans[MAXN];

void Update(int b, int i, int v) {
  for (i++; i <= N; i += (i & -i)) {
    BIT[b][i] += v;
  }
}
```

```
LL Query(int b, int i) {
  LL v = 0;
  for (i++; i > 0; i -= (i & -i)) {
    v += BIT[b][i];
  }
  return v;
}

int main() {
  // Input, and initialize BIT #0.
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> H[i] >> C[i];
    ord[i] = make pair(H[i], i);
    Update(0, i, C[i]);
  }
  cin >> M;
  for (int i = 0; i < M; i++) {
    int X, L, R;
    cin >> X >> L >> R;
    Q[X].push_back(make_pair(i, make_pair(L - 1, R - 1)));
  }
  // Iterate over skyscrapers in order of non-increasing height.
  sort(ord, ord + N);
  reverse(ord, ord + N);
  int eqA = 0, eqB = -1;
  for (int i = 0; i <= N; i++) {
    // Skyscraper at new height?
    if (i < N && i > eqB) {
      // Move all skyscrapers at previous height from BIT #1 to BIT #2.
      for (int j = eqA; j <= eqB; j++) {
        int k = ord[j].second;
        Update(1, k, -C[k]);
        Update(2, k, C[k]);
      }
      // Find new set of skyscrapers at current height, and insert them into BIT #1.
      eqA = eqB = i;
      while (eqB + 1 < N && ord[eqB + 1].first == ord[eqA].first) {
        eqB++;
      }
      for (int j = eqA; j <= eqB; j++) {
        int k = ord[j].second;
        Update(1, k, C[k]);
      }
    }
    // Compute probability of each current-height skyscraper having been vaporized.
    int eqTot = eqB - eqA + 1;
    int eqDone = i - eqA;
    LD prob = (LD)eqDone / eqTot;
    // Process queries with the current X value.
    for (int j = 0; j < (int)Q[i].size(); j++) {
      int q = Q[i][j].first, a = Q[i][j].second.first, b = Q[i][j].second.second;
      // v1 = Sum of all skyscrapers in interval.
      LL v1 = Query(0, b) - Query(0, a - 1);
      // v2 = Expected sum of current-height skyscrapers vaporized outside interval.
      LD v2 = prob * (Query(1, N - 1) - (Query(1, b) - Query(1, a - 1)));
      // v3 = Sum of all taller skyscrapers outside interval.
      LL v3 = Query(2, N - 1) - (Query(2, b) - Query(2, a - 1));
      // Store query answer for later.
      ans[q] = v1 + v2 + v3;
    }
  }
  // Output.
  for (int i = 0; i < M; i++) {
    cout << fixed << setprecision(9) << ans[i] << endl;
  }
  return 0;
}
```

Problem S5: Crop Rectangles

With each output depending only on a pair of integers (R, C), it's safe to guess that some sort of patterns can be found for small values of (R, C) which can extend to arbitrarily large values. It may be tempting to try solving various small cases by hand to look for such patterns, and it's simple to do so for some (such as when one dimension is equal to 1 or 2), but overall it proves far too difficult to do so for others. So, our best bet will be to write some code which can (inefficiently) find answers for small grids, and then look for patterns in its results.

The most standard approach is with breadth-first search, in which the state includes information about which subset of cells have been visited, what cell the lawnmower is currently in, what direction it's facing in, and whether or not it turned just before its last move. This gives us $2^{R \times C} \times R \times C \times 4 \times 2$ states, with up to 3 transitions from each state.

Many states are unreachable, so we'll want to hash states to track their reachability rather than maintaining a Boolean array of that size. This sort of BFS ends up running in a reasonable amount of time (under a minute) for any grid such that $R \times C$ is at most 40 or so, which will suffice for our purposes.

Now, let's talk about the patterns which surface. For convenience, let $A = \min(R, C)$ and $B = \max(R, C)$, and let's talk in terms of the minimum number of "wasted" moves W (such that the answer is equal to $R \times C - 1 + W$). If we run our BFS for all possible grids with $1 \leq R, C \leq 6$, we can come up with the following observations:

- When $A = 1$, $W = 0$.
- When $A = 2$, it's impossible.
- When $A = 3$ and $B = 3$, it's impossible.
- When $A = 3$ and $B \geq 4$, *there's no clear pattern?*
- When $A = 4$, $W = 4$. If we examine the optimal lawnmower path for a 4x4 grid, it's relatively clear that it can be easily stretched out to arbitrarily large values of B, and that larger values of B can't help improve it.
- When $A \geq 5$, $W = 2$. If we examine the optimal lawnmower path for a 5x5 grid, it's relatively clear that it can be extended to larger grid sizes without wasting any additional moves (by either stretching it out along one dimension or repeatedly wrapping around the outside to increase both dimensions), and that larger grid sizes can't help improve it.

The only problematic case is when $A = 3$ and $B \geq 4$, which requires further investigation. It's once again safe to guess that a pattern will emerge for relatively small values of B which can extend to arbitrarily large values. To search for the pattern, we can try running our BFS again for as many $3 \times B$ grids as possible, which works reasonably up to around $B = 15$.

Unfortunately, this is insufficient for the pattern to become completely obvious, but it is enough to guess it correctly. The W values for $B = 4..11$ are unpredictable ([4, 4, 6, 8, 10, 10, 10, 12]), but starting at $B = 12$ they finally start to repeat regularly ([14, 14, 16, 16, 18, 18, 20, 20, ...]). To gain full confidence, it's also possible to write a more specialized brute force algorithm for $3 \times B$ grids (or even use dynamic programming) to compute all W values up to much larger values of B, though that's less viable in the limited contest time.

With that, all possible cases have been covered! For $A = 3$ and $4 \leq B \leq 11$, we can either run our BFS or hardcode the values listed above, and for all other cases we have closed-form formulae for the answers.

Official Solution (C++)

```
#include <cstring>
#include <iostream>
#include <queue>
#include <set>
using namespace std;

const int MAXA = 3;
const int MAXB = 12;
const int dy[4] = {-1, 0, 1, 0};
const int dx[4] = {0, 1, 0, -1};

int A, B;

struct State {
  int d, y, x, z, t;
  bool vis[MAXA][MAXB];

  State() {}
  State(int d, int y, int x, int z, int t) : d(d), y(y), x(x), z(z), t(t) {}

  long long hash() {
    // Compute a hash representing the state.
    long long h = 0;
    for (int i = 0; i < A; i++)  {
      for (int j = 0; j < B; j++) {
        h <<= 1;
        if (vis[i][j]) {
          h++;
        }
      }
    }
    return (((h*A + y)*B + x)*4 + z)*3 + t;
  }
};

int Solve() {
  // Initialize.
  set<long long> S;
  queue<State> Q;
  State st = State(0, 0, 0, 1, 2);
  memset(st.vis, 0, sizeof st.vis);
  Q.push(st);
  S.insert(st.hash());
  // BFS.
  while (!Q.empty()) {
    // Get state.
    State st = Q.front();
    Q.pop();
    // Update state.
    if (st.t < 2) {
      st.t++;
    }
    st.vis[st.y][st.x] = 1;
    // All cells visited?
    bool done = true;
    for (int i = 0; i < A && done; i++)  {
      for (int j = 0; j < B && done; j++) {
        if (!st.vis[i][j]) {
          done = false;
        }
      }
    }
    if (done) {
      return st.d;
    }
    // Go on.
    st.d++;
```

```
    for (int z = 0; z < 4; z++) {
      // New direction?
      int t = st.t;
      if (z != st.z) {
        // Can't reverse.
        if (z % 2 == st.z % 2) {
          continue;
        }
        // Can't turn yet?
        if (t < 2) {
          continue;
        }
        t = 0;
      }
      // Get new position.
      int y = st.y + dy[z], x = st.x + dx[z];
      if (y < 0 || y >= A || x < 0 || x >= B) {
        continue;
      }
      // Get new state.
      State st2 = st;
      st2.y = y;
      st2.x = x;
      st2.z = z;
      st2.t = t;
      if (S.count(st2.hash())) {
        continue;
      }
      Q.push(st2);
      S.insert(st2.hash());
    }
  }
  // No solution found.
  return -1;
}

int main() {
  int N, R, C;
  cin >> N;
  while (N--) {
    cin >> R >> C;
    A = min(R, C);
    B = max(R, C);
    // Handle various cases.
    int ans = A*B - 1;
    if (A == 2) {
      ans = -1;
    }
    if (A == 3) {
      if (B >= 12) {
        ans += 14 + (B - 12)/2*2;
      } else {
        ans = Solve();
      }
    }
    if (A == 4) {
      ans += 4;
    }
    if (A > 4) {
      ans += 2;
    }
    cout << ans << endl;
  }
  return 0;
}
```