WOBURN CHALLENGE

2016-17 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: Fencing

For each cow i from 1 to N, we'll need to compute the number of matches m_i that they won. To do so, we can iterate over all cows j from 1 to N such that $i \neq j$, and increment m_i whenever $S_{i,j} > S_{j,i}$. Overall, we'll be looking for the cow i with the largest m_i value. To find that cow, we can keep track of the largest m_i value seen so far (as well as the value of its associated index i), and replace it whenever we come across a new maximum. At the end, we can output that associated cow index i.

Official Solution (C++)

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

int N, S[100][100];
int maxW = -1, maxC = 0;

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      cin >> S[i][j];
    }
  }
  for (int i = 0; i < N; i++) {
    int w = 0;
    for (int j = 0; j < N; j++) {
      if (i != j && S[i][j] > S[j][i]) {
        w++;
      }
    }
    if (w > maxW) {
      maxW = w;
      maxC = i;
    }
  }
  cout << maxC + 1 << endl;
  return 0;
}
```

Problem J2: Enigmoo

Overall, we want to iterate over each of the N words, and determine whether or not it can correspond to the encrypted message. If so, then we can increment a running total of possible words, which we'll output at the end.

To determine whether or not a given word i is valid, we need to consider each possible value of S between 1 and 25 (remembering to exclude 0), and check if at least one such value would result in the encrypted word matching W. If multiple S values would work (which is the case when W is made up entirely of "?" characters), we need to make sure to still only count the word once!

Finally, determining whether a given value of S makes a certain word D_i valid involves checking whether all N of its characters are independently valid. For each index j from 1 to N, we need to compute the value of $D_{i,j}$ when it's cyclically shifted forward in the alphabet by S spots, and check whether that value is equal to W_j. However, if W_j is equal to "?", then index j should instead be skipped, as any original character at that index would be valid.

Official Solution (C++)

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

int N, M, ans = 0;
string W, D;

int main() {
  cin >> N >> M >> W;
  while (M--) {
    cin >> D;
    for (int s = 1; s <= 25; s++) {
      bool valid = true;
      for (int i = 0; valid && i < N; i++) {
        if ((W[i] != '?') && ((W[i] - 'a' + s) % 26 != D[i] - 'a')) {
          valid = false;
        }
      }
      if (valid) {
        ans++;
        break;
      }
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem J3/S1: Cow-Bot Construction

For starters, if $E \leq B$, then the engineers are at least as fast as the Cow-Bot at installing modules, so they might as well just install all of them themselves. This will take them $N \times E$ minutes.

Otherwise, the Cow-Bot is faster, so our goal is to have it install as many modules as possible, with the only limitation being the modules' M requirements. As such, let's start by sorting the modules in non-decreasing order of their M values, which can be done in $O(N \log N)$ time. The modules at the end (with the largest M values) are the least likely to be able to be installed by the Cow-Bot before it's too late, so the engineers should prioritize installing those. On the other hand, the modules at the start (with the smallest M values) will be the ones which the Cow-Bot should go ahead and install whenever it can.

These ideas suggest the following greedy solution, in which we'll simulate the modules' installation. Let m be the number of modules installed so far (which we'll iterate upwards from 0 to $N - 1$), i be the first module which hasn't yet been installed (initially module 1 in the sorted list), and let t be the total amount of time taken so far (initially 0). At each step, if $m \geq M_i$, then the Cow-Bot is currently able to install the i-th module, so it should certainly do so – in this case, we'll increment i by 1, and t by B. Otherwise, there's no module currently available for the Cow-Bot to install, so the engineers must go ahead and install one from the end – in this case, we'll just increment t by E. This simulation process takes $O(N)$ time.

Official Solution (C++)

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

int N, E, B, M[200000];

int main() {
  cin >> N >> E >> B;
  if (E <= B) {
    // Engineers might as well install everything.
    cout << N*E;
    return 0;
  }
  for (int i = 0; i < N; i++) {
    cin >> M[i];
  }
  sort(M, M + N);  // Sort modules by increasing module requirement.
  // Greedily have the bot install modules when possible.
  int i = 0, ans = 0;
  for (int m = 0; m < N; m++) {
    if (m >= M[i]) {
      ans += B;
      i++;
    } else {
      ans += E;
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem J4/S2: Rational Recipes

Let's break down our counting of valid recipes by the number of monkeys m which would be served by them. Note that m uniquely determines the value of R_1 – in particular, to satisfy $m \times R_1 = F_1$, we must have $R_1 = F_1 / m$. This further implies that m must be a divisor of F_1.

This is useful due to the fact that F_1 can't have many different divisors. In fact, we can find all of them by iterating over each integer i between 1 and $\lfloor\sqrt{F_1}\rfloor$, inclusive. If F_1 modulo i is equal to 0, then i is a divisor of F_1. Furthermore, F_1 / i is also a divisor of i – however, if $i = F_1 / i$, we need to be careful to not count its recipes twice!

Now, for a given value of m, we need to be able to count the number of valid recipes which would serve m monkeys. As shown, there's a single valid value for R_1. Meanwhile, each other R_i (for $2 \le i \le N$) can independently be any positive integer such that $m \times R_i \le F_i$. The number of such valid values is simply equal to $\lfloor F_i / m \rfloor$. Overall, the number of valid recipes for a given value of m is then the product of these $N - 1$ rounded-down quotients. While computing the running product, we need to be careful to mod it by 10,007 after every iteration.

The final answer is the sum of the above counts for all of F_1's divisors, again taken modulo 10,007. The time complexity of this algorithm is $O(\sqrt{F_1} \times N)$.

Official Solution (C++)

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

const MOD = 10007;
int N, F[100], ans = 0;

// Return number of different recipes for exactly m monkeys.
int nrecipes(int m) {
  int p = 1;
  for (int i = 1; i < N; i++) {
    p = p*(F[i]/m % MOD) % MOD;
  }
  return p;
}

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> F[i];
  }
  // Find all divisors of number of bananas.
  for (int i = 1; i*i <= F[0]; i++) {
    if (F[0] % i == 0) {
      // Consider i monkeys.
      ans = (ans + nrecipes(i)) % MOD;
      // Also consider F[0]/i monkeys, if that's a different value.
      if (F[0]/i != i) {
        ans = (ans + nrecipes(F[0]/i)) % MOD;
      }
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem J5/S3: Privacy

This problem can be solved with dynamic programming. Let $DP[i][j]$ be the minimum number of treats required to satisfy the first i cows (cows 1..i) while dividing them into j troughs. The base case is $DP[0][0] = 0$, and the final answer will be $DP[N][K + 1]$, as we'll be installing K dividers to divide the N cows into $K + 1$ troughs.

From a given state *(i, j)*, the only choice is regarding how many cows should be included in the next trough. The first cow in the trough will be i, and the last will be k (for some k between i and N, inclusive). We should consider each possible value of k. For a given k, there will be $c = k - i + 1$ cows in the next trough. The number of treats t required to satisfy each of those c cows can easily be computed by iterating over them (and comparing their C values to the cow count c). We end up with the transition $DP[k][j + 1] = \min(DP[k][j + 1], DP[i][j] + t)$.

A direct implementation of the approach described above would have a time complexity of $O(N^3 K)$ – we'll consider each of the $O(NK)$ states in turn (for i from 0 to $N - 1$ and from j from 0 to K), consider each of the $O(N)$ possible transitions from it, and then compute each transition's t cost in another $O(N)$ time. This is too slow to earn full marks. However, the time complexity can be improved to $O(N^2 \times (N + K))$ by essentially just swapping the nesting order of the j and k loops. Note that each transition's t cost depends only on i and k, so there's no need to recompute it $O(K)$ times for all possible values of j.

Official Solution (C++)

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

const int INF = 0x3f3f3f3f;
int N, K, C[400], DP[401][402];

int main() {
  cin >> N >> K;
  for (int i = 0; i < N; i++) {
    cin >> C[i];
  }
  memset(DP, INF, sizeof DP);
  DP[0][0] = 0;
  for (int i = 0; i < N; i++) {
    for (int j = i; j < N; j++) {
      // Count number of treats required for trough i..j.
      int c = j - i + 1, t = 0;
      for (int k = i; k <= j; k++) {
        t += max(0, c - C[k] - 1);
      }
      // Perform DP transitions.
      for (int k = 0; k <= K; k++) {
        DP[j + 1][k + 1] = min(DP[j + 1][k + 1], DP[i][k] + t);
      }
    }
  }
  cout << DP[N][K + 1] << endl;
  return 0;
}
```

Problem S4: Bug Infestation

The monkeys' program can be modeled as a directed graph with N nodes. The i-th node corresponds to the i-th line of code, and for each line i whose debugging would result in another line gaining a bug ($L_i > 0$), there's an edge leading from node i to node L_i. Note that every node in this graph has an out-degree of at most 1. A graph of this form has some useful properties regarding the structure of its connected components – in particular, each component can have one of two types. A *type-1* component is just a tree rooted at some node (with an edge leading up from each node to its parent). Meanwhile, a *type-2* component contains a single cycle (consisting of 2 or more nodes), with each of the cycle's nodes possibly being the root of a tree of other, non-cycle nodes.

The graph's components are totally independent of one another, so our overall approach will be to find each component, traverse it to determine its type, and then compute the 2 required answer values for it (the minimum number of bugs which it can be reduced to having, and the minimum amount of time required to do so). These answers can then be tallied up over all of the components.

To find all of the graph's components, we can iterate over all N nodes in any order, while keeping track of which nodes belong to components which have already been processed (initially none). Each time we come across a node which hasn't yet been marked in this way, we'll know that we've found a new component to process. The first step will be to find the "base" of the component – in particular, its root node if it's a *type-1* component, or any node belonging to its cycle if it's a *type-2* component. This can be accomplished by repeatedly following the single outgoing edge from the current node until we either reach a node with no outgoing edge, or re-visit a node for the second time. In the former case, the final node must be the root of a *type-1* component, while in the latter case, the re-visited node must lie on a *type-2* component's cycle (given that the sequence of traversed edges has led back to it).

A *type-1* component can always be reduced to having 0 bugs, and the optimal way to do so involves essentially "propagating" all of its bugs gradually up to the root of the tree, and then debugging the root node to remove the final bug entirely. In total, each node in the tree whose subtree contains at least one bug will need to be debugged once, while any other nodes won't need to be debugged at all. Counting the number of such nodes can be done using a simple instance of dynamic programming, by recursively determining whether or not each node's subtree contains any bugs based on its children's subtrees.

A *type-2* component is trickier to deal with. Each tree branching off from the component's cycle should first be handled as if it were a *type-1* component itself (reusing the same recursive procedure), resulting in it getting reduced to its root node (which is part of the cycle) containing a bug if the tree initially contained at least one bug.

We're then left with just a cycle with some number of nodes c, which can be numbered from 1 to c by iterating around the cycle once. Each of the c nodes either contains a bug, or doesn't. Assuming there are k such bugged nodes, we can assemble a list $P_{1..k}$ of their positions in the cycle. Assuming that k is positive, not all of the bugs can ever be removed, so the best we can do is "sweep" around the cycle, essentially collecting all of the bugs as we go until only a single node is left containing a bug. However, it's unclear where along the cycle this sweep should start. We should certainly start by debugging one of bugged nodes, and then we'll end up going around until reaching the last bugged node before the initial chosen one. In particular, if we start the sweep at cycle position P_i (such that $2 \leq i \leq k$), then it will take $c - (P_i - P_{i-1})$ minutes to complete the sweep. Starting at P_1 and ending at P_k is similar. All k possible sweeps of this sort should be considered, with the fastest one being chosen for the component.

In every individual portion of the above algorithm (scanning through all of the nodes to find components, finding the base of each component, recursively solving each tree, traversing each cycle, and considering all possible sweep points around each cycle), each node in the graph is processed at most once in total. Therefore, the overall time complexity is $O(N)$.

Official Solution (C++)

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

const int MAXN = 300000, INF = 0x3f3f;
int N, Q, ans1, ans2, lcaD, B[MAXN], L[MAXN];
bool vis[MAXN];
vector<int> adj[MAXN];

// return true if i's subtree contains any bugs
bool dfs(int i, int p, int d) {
  vis[i] = true;
  int c = 0;
  // Recurse to each of i's children.
  for (int j = 0; j < (int)adj[i].size(); j++) {
    if (adj[i][j] != p && dfs(adj[i][j], i, d + 1)) {
      c++;
    }
  }
  // If i's subtree contains any bugs, i must be debugged at some point.
  if (B[i] || c) {
    ans2++;
  }
  // Update depth of LCA of bugged nodes.
  if (B[i] || c > 1) {
    lcaD = d;
  }
  return B[i] || c;
}

int main() {
  cin >> N >> Q;
  for (int i = 0; i < N; i++) {
    cin >> B[i] >> L[i];
    L[i]--;
    if (L[i] >= 0) {
      adj[L[i]].push_back(i);
    }
  }
  // Find each connected component.
  ans1 = ans2 = 0;
  for (int i = 0; i < N; i++) {
    if (!vis[i]) {
      // Locate the component's cycle (or single root node).
      int j = i;
      while (L[j] >= 0 && !vis[j]) {
        vis[j] = true;
        j = L[j];
      }
      if (L[j] < 0) {  // No cycle.
        // The component is just a tree rooted at j, recurse through it
        dfs(j, -1, 0);
        continue;
      }
      // Iterate around the cycle, noting its size and the positions of bugs.
      int p = j, k = L[j], c = 0;
      vector<int> bugs;
      do {
        // Recurse through the tree rooted at k, and determine if it contains any bugs.
        if (dfs(k, p, 0)) {
          bugs.push_back(c);
        }
        c++;
        p = k;
        k = L[k];
      } while (p != j);
```

```
      // Cycle has no bugs?
      if (bugs.empty()) {
        continue;
      }
      // Cycle has exactly 1 bugged node?
      if (bugs.size() == 1) {
        // Revert unnecessary cost of moving the tree's bug to its root on the cycle.
        ans1++;
        ans2 -= lcaD + 1;
        continue;
      }
      // Cycle has multiple bugged nodes - find the largest gap between consecutive bugs.
      ans1++;
      int m = bugs[0] - bugs[bugs.size() - 1] + c;
      for (j = 0; j < (int)bugs.size() - 1; j++) {
        m = max(m, bugs[j + 1] - bugs[j]);
      }
      ans2 += c - bugs.size() - m;
    }
  }
  if (Q == 1) {
    cout << ans1 << endl;
  } else {
    cout << ans1 << " " << ans2 << endl;
  }
  return 0;
}
```

Problem S5: Bovine Grenadiers

Let's begin by examining only the single-box case, with $N = 1$. After the first cow opens the box, each cow should clearly choose to claim the most powerful remaining grenade on each of their turns. As such, if we consider sorting the grenade powers in non-increasing order, the second cow will end up getting all of the grenades which are at odd indices in the sorted list (the 1st, 3rd, etc.), while the first cow will get the ones at even indices. Therefore, we can compute the total grenade power received by each cow in $O(G_1 \log G_1)$ time. For future convenience, let's call the box's overall value the difference between the grenade power received by the second and first cows (note that this difference is always non-negative, as the second cow can't get less than the first cow).

Let's next consider how the grenade power received by the cows changes when a single grenade's power is increased or decreased by 1. If a certain value gets increased from x to $x + 1$, then we can find the value x in the sorted list (in $O(\log G_1)$ time using binary search) and increment it in place. If x appears multiple times in the list, let's specifically find the last occurrence and increment that one. Note that, after performing this operation, the whole list is guaranteed to still be sorted in non-increasing order, without needing to reorder any elements! This means that the box's overall value can easily be updated in another $O(1)$ time. A value getting decreased by 1 can be handled in a similar fashion.

We're now ready to consider what's going on in the entire game, when there are multiple boxes. Regarding the cows' overall strategy, it can be observed that once a cow opens a box, it's optimal for both cows to then only take grenades from that box until it becomes empty – it can never help a cow to instead choose to open some other box prematurely. Now, let's refer to boxes containing even numbers of grenades as *even-boxes*, and the others as *odd-boxes*. When a cow opens an *even-box*, an odd number of total turns will go by until it's been exhausted, meaning that the other cow will end up opening the next new box. On the other hand, when a cow opens an *odd-box*, it will once again be their turn after that box has been exhausted. As we've already seen, being forced to open a box is disadvantageous, so it's optimal for the cows to repeatedly choose all of the *even-boxes* to keep passing that disadvantage back and forth, until one cow gets stuck with opening all of the *odd-boxes* at the end of the game (which cow that will be depends only on the number of *even-boxes*).

Now that we know how the game will generally play out, we can consider how to compute its exact result. We should start by computing all of the boxes' initial overall values in $O(K \log K)$ time, where K is the total number of grenades in all of them. Furthermore, whenever a grenade inside box i gets swapped out, we can go ahead and independently update box i's overall value in $O(\log G_i)$ time, as shown above.

Past that, the *odd-boxes* are simple – a certain cow is known to be the one to open each of those, so the effects of an *odd-box*'s overall value on the cows' grenade power totals for the entire game is clear, including handling when it gets updated.

As for the *even-boxes*, observe that the cows will take turns opening them, each time choosing the remaining one with the smallest overall value in order to minimize their disadvantage from having to open it. This is almost exactly the same situation as what happens to the grenades inside a single box! Therefore, we can apply the same approach of computing a sorted list of initial *even-box* overall values in $O(N \log N)$ time, and then maintain it as well as the game's overall value (the difference between the total grenade powers obtained by Bessie and Angus) in $O(\log N)$ time per update. Part of the reason this works out is that an individual absolute grenade power change of at most 1 results in its box's overall value also changing by at most 1.

The overall time complexity of this algorithm is $O(N \log N + K \log K + M \log(N + K))$, where K is the total number of grenades.

Official Solution (C++)

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

const int LIM = 300000;

int N, M, G[LIM], A[LIM], B[LIM], D[LIM];
long long V[LIM], V2[LIM], ans1 = 0, ans2 = 0;
vector<int> P[LIM], P2[LIM];

int main() {
  cin >> N >> M;
  long long sum = 0;
  for (int i = 0; i < N; i++) {
    cin >> G[i];
    P[i].resize(G[i]);
    for (int j = 0; j < G[i]; j++) {
      cin >> P[i][j];
      sum += P[i][j];
    }
    P2[i] = P[i];
    sort(P2[i].begin(), P2[i].end());  // Sort values in non-decreasing order.
    int m = 1;
    for (int j = G[i] - 1; j >= 0; j--) {  // Compute box's initial value.
      V[i] += P2[i][j] * m;
      m = -m;  // Invert sign for each value.
    }

  }
  // Sort even-parity box values in non-decreasing order.
  int K = 0;
  for (int i = 0; i < N; i++) {
    if (G[i] % 2 == 0) {
      V2[K++] = V[i];
    }
  }
  sort(V2, V2 + K);
  // Compute combined initial value.
  long long diff = 0;
```

```
  int m = 1;
  for (int i = 0; i < K; i++) {
    diff += V2[i] * m;
    m = -m;  // Invert sign for each value.
  }
  for (int i = 0; i < N; i++) {
    if (G[i] % 2 == 1) {
      diff += V[i] * m;  // Add on all odd-parity box values.
    }
  }
  while (M--) {  // Handle updates.
    int a, b, d;
    cin >> a >> b >> d;
    a--, b--;
    // Update grenade's value.
    long long v = P[a][b];
    P[a][b] += d;
    sum += d;
    // Update box's value.
    if (d == 1) {
      // Find last index with the existing value.
      int i = lower_bound(P2[a].begin(), P2[a].end(), v + 1) - P2[a].begin() - 1;
      P2[a][i]++;
      if ((G[a] - i) % 2 == 0) {  // Determine effect on box's value.
        d = -d;
      }
    } else if (d == -1) {  // Find first index with the existing value.
      int i = lower_bound(P2[a].begin(), P2[a].end(), v) - P2[a].begin();
      P2[a][i]--;
      if ((G[a] - i) % 2 == 0) {  // Determine effect on box's value.
        d = -d;
      }
    }
    v = V[a];
    V[a] += d;
    // Update combined value.
    if (G[a] % 2 == 1) {  // Odd-parity box.
      d *= m;
    } else if (d == 1) {  // Find last index with the existing value.
      int i = lower_bound(V2, V2 + K, v + 1) - V2 - 1;
      V2[i]++;
      if (i % 2 == 1) {  // Determine effect on combined value.
        d = -d;
      }
    } else if (d == -1) { // Find first index with the existing value.
      int i = lower_bound(V2, V2 + K, v) - V2;
      V2[i]--;
      if (i % 2 == 1) {  // Determine effect on combined value.
        d = -d;
      }
    }
    diff += d;
    // Update answers.
    v = sum / 2;
    if (diff < 0) {
      v -= (diff - 1) / 2;
    } else {
      v -= diff / 2;
    }
    ans1 += v;
    ans2 += sum - v;
  }
  cout << ans1 << " " << ans2 << endl;
  return 0;
}
```