WOBURN CHALLENGE

2018-19 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: A Fistful of Quarters

D dollars is equivalent to $4D$ quarters. If $Q \geq 4D$, no additional quarters are required, while if $Q < 4D$, $4D - Q$ additional quarters are required. Equivalently, the answer may be expressed as $\max(0, 4D - Q)$.

Official Solution (C++)

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

int main() {
  int D, Q;
  cin >> D >> Q;
  cout << max(4 * D - Q, 0) << endl;
  return 0;
}
```

Problem J2: Life Insurance

We'll simulate the battle while maintaining two variables: Mario's current HP (initially 50), and the number of Life Shrooms L consumed so far (initially 0). Upon inputting each value H_i, we should first increase HP by H_i (setting it equal to 50 if it would exceed 50). Then, if HP ≤ 0, we should set it equal to 10 and increment L by one. After processing all N events in this manner, we can output L.

Official Solution (C++)

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

int main() {
  int N, H, ans = 0, curH = 50;
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> H;
    // Update health, capping at 50.
    curH = min(curH + H, 50);
    // Time to consume a life shroom?
    if (curH <= 0) {
      curH = 10;
      ans++;
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem J3/I1: Inventory

Each of the C size-3 items requires its very own knapsack. This amounts to C knapsacks.

Each of the B size-2 items cannot fit into a knapsack with any other size-2 items, so this amounts to another B knapsacks. However, each of those knapsacks also has space for a size-1 item, so we should go ahead and dispose of $\min(A, B)$ size-1 items along the way.

Finally, the remaining size-1 items (of which there are $\max(0, A - B)$) should be packed into knapsacks 3 at a time, with one potentially non-full knapsack remaining at the end. This amounts to another $\lceil \max(0, A - B) / 3 \rceil$ knapsacks (or, equivalently, $\lfloor (\max(0, A - B) + 2) / 3 \rfloor$ knapsacks).

Official Solution (C++)

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

int main() {
  int A, B, C;
  cin >> A >> B >> C;
  int ans = C;  // 1 knapsack for each C.
  // 1 knapsack for each B (along with an A, if any).
  ans += B;
  A = max(0, A - B);
  // 1 knapsack for each 3 (or fewer) As.
  ans += (A + 2) / 3;
  cout << ans << endl;
  return 0;
}
```

Problem J4/I2: Your Name, Please

$N + 1$ button presses are required just to confirm a length-N name, regardless of its letters (an "A" button press upon selecting each letter, plus a final "+" button press at the end). Therefore, if $K < N + 1$, then no valid name exists. Otherwise, let $B = K - (N + 1)$ be the number of additional "<"/">" button presses required.

Let's consider how many such button presses each letter in a name contributes. Let $m(p, x)$ be the number of button presses required to move the cursor to a letter x from the previous letter p (with the previous letter before the first one assumed to be "A"). We'll either repeatedly press "<" or ">", whichever requires fewer presses. These two options require $|x - p|$ and $26 - |x - p|$ button presses (in some order), giving us $m(p, x) = \min(|x - p|, 26 - |x - p|)$. We can observe that, given any previous letter p, it's possible to choose some next letter x such that $m(p, x)$ has any wanted value between 0 and 13, inclusive.

We can then observe that a valid name exists if and only if $B \leq 13N$. Furthermore, we can construct a name for a given B value by greedily choosing letters one by one such that $m(p, x)$ is maximized each time (up to at most 13), but without exceeding the remaining allotment of required button presses.

Official Solution (C++)

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

int main() {
  int N, K;
  cin >> N >> K;
  // Factor out confirmation button presses.
  K -= N + 1;
  if (K < 0 || K > N*13) {  // Too few/many button presses required?
    cout << "Impossible" << endl;
    return 0;
  }
  // Greedily select each letter.
  for (int i = 0, let = 0; i < N; i++) {
    // Use as many button presses as possible/allowed.
    int b = min(K, 13);
    K -= b;
    let = (let + b) % 26;
    cout << (char)(let + 'A');
  }
  cout << endl;
  return 0;
}
```

Problem I3/S1: World of StarCraft

We can represent *World of StarCraft* as an undirected graph, with each node corresponding to a planet, and each usable space route corresponding to an edge. Note that only space routes connecting planets under the control of the same race should be considered at all (in other words, space routes i such that $R_{Ai} = R_{Bi}$). The i-th friend can then safely complete their objective if and only if node Y_i is reachable from node X_i.

One possible approach is to consider each friend i independently, and perform either a BFS (Breadth-First Search) or DFS (Depth-First Search) from node X_i to check whether Y_i is reachable. This would require $O(N + M)$ time per friend, resulting in an overall time complexity of $O(K(N + M))$, which is too slow to earn full marks.

To improve on this, we can use the fact that the graph is undirected to observe that, for each connected component in the graph, all nodes in the component are reachable from one another. In other words, friend i can safely complete their objective if and only if X_i and Y_i are part of the same connected component as one another. Therefore, if we can precompute which connected component each node is in, then the friends can be handled in $O(1)$ time each.

Finding all of the graph's connected components may be done in $O(N + M)$ using a process known as floodfill, in which we iterate over all N nodes in any order, and each time we come across a node whose component has yet to be determined, we perform either a BFS or DFS from it to identify all nodes in its component.

Official Solution (C++)

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

const int MAXN = 100000;

int numComps, comp[MAXN];
vector<int> adj[MAXN];
string R;

void DFS(int i) {
  comp[i] = numComps;
  for (int j = 0; j < adj[i].size(); j++) {
    int k = adj[i][j];
    if (!comp[k] && R[i] == R[k]) {
      DFS(k);
    }
  }
}

int main() {
  int N, M, K;
  cin >> N >> M >> K >> R;
  for (int i = 0, A, B; i < M; i++) {
    cin >> A >> B;
    A--, B--;
    adj[A].push_back(B);
    adj[B].push_back(A);
  }
  // Floodfill to find components of inter-reachable planets.
  for (int i = 0; i < N; i++) {
    if (!comp[i]) {
      numComps++;
      DFS(i);
    }
  }
  // Consider each friend.
  int ans = 0;
  for (int i = 0, X, Y; i < K; i++) {
    cin >> X >> Y;
    X--, Y--;
    if (comp[X] == comp[Y]) {  // In the same component?
      ans++;
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem I4/S2: Farming Simulator

At least H seconds of walking time from left to right are always required, and we're interested in minimizing the number of "extra" seconds required to also get all N trees planted along the way.

Upon sorting the holes in increasing order of their P values (which may be done in $O(N \log N)$ time), we'll consider dividing them up into one or more consecutive sequences of holes which will be handled together, such that each hole is included in exactly one sequence. For each such sequence of holes $i..j$, our strategy will be to walk right from P_i to P_j (while planting a seed in each hole), then walk left back to P_i, and finally walk right back to P_j (while watering each hole's seed, and waiting in place as necessary). An optimal strategy always exists which follows this particular pattern, for some choice of sequences.

The next question is how much extra time any given sequence $i..j$ incurs. At least $2(P_j - P_i)$ seconds will be spent walking left to P_i and back right to P_j. We can also observe that this is exactly the amount of time that will pass between the first (seed-planting) and last (seed-watering) visit to each hole in the sequence, assuming we keep moving. Extra waiting time may be required along the way if any hole has a W value larger than that. Based on this, we can arrive at the number of extra seconds incurred by the interval being $\max(2(P_j - P_i), \max(W_i, \ldots, W_j))$.

What remains is applying the above observations to determine the optimal set of sequences, using a dynamic programming approach. We'll let $DP[i]$ be the minimum number of seconds required to divide the first i holes into sequences (such that $DP[i] = 0$, and $H + DP[N]$ will be our answer). $DP[j]$ can be computed by considering each possible index i ($1 \leq i \leq j$) such that the last sequence up to there is $i..j$ (resulting in a total extra time of $DP[i - 1] + \max(2(P_j - P_i), \max(W_i, \ldots, W_j))$. As long as we compute and re-use running maximums of W values rather than computing $\max(W_i, \ldots, W_j)$ from scratch each time, the time complexity of this algorithm will be $O(N^2)$.

Official Solution (C++)

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

const int MAXN = 3000;

pair<int,int> P[MAXN];
int DP[MAXN + 1];  // DP[i] = min extra time to finish first i holes and end at the ith one.

int main() {
  int N, H;
  cin >> N >> H;
  for (int i = 0; i < N; i++) {
    cin >> P[i].first >> P[i].second;
  }
  sort(P, P + N);  // Sort holes by position.
  DP[0] = 0;
  for (int i = 0; i < N; i++) {
    DP[i + 1] = 2e9;
    // Consider each initial hole j for the current interval.
    for (int j = i, maxW = 0; j >= 0; j--) {
      maxW = max(maxW, P[j].second);  // Update max W value present in the interval j..i.
      int t = max(2*(P[i].first - P[j].first), maxW);  // Extra time to handle the interval.
      DP[i + 1] = min(DP[i + 1], DP[j] + t);  // Update DP value.
    }
  }
  cout << DP[N] + H << endl;  // Output, adding on base walking time.
  return 0;
}
```

Problem S3: Dance Royale

We'll model *Dance Royale* as a directed graph with N nodes (one per location) and up to N edges (one per transition from a node to its destination node, if any).

Each component of the graph may be processed independently of the other components, and contains either exactly one cycle or no cycles. If the component contains a cycle, then the remainder of the component consists of trees rooted at nodes on that cycle (such that, if there's an edge from node i to node j, then node j is node i's parent). Otherwise, if the component is acyclic, then the entire component is a single tree rooted at a node with out-degree 0.

We'll iterate over all N nodes in any order, and each time we come across one whose component has not yet been processed, we'll process it by repeatedly following its edges forward until either arriving back at a previously-visited node (thus indicating a cycle with some length C, which we can find easily), or arriving at a node with out-degree 0 (thus indicating an acyclic component).

In either case, we'll perform either a DFS or BFS throughout the component on the transpose graph (with all edges reversed), starting from the node arrived at in the above process (that is, any node on the cycle if there is one, or otherwise the 0-degree node). Let $F(x)$ be the total number of players at nodes which are at a distance of x away from the starting node (these totals can be stored in a hash map to allow them to be efficiently reset to 0 between different components).

Now, if the component has a length-C cycle, then all players at distances x, $x + C$, $x + 2C$, and so on will end up at the same position along the cycle after an infinite amount of time (for $0 \le x < C$). Letting $G(x)$ be this sum, this will result in $G(x) \times (G(x) - 1) / 2$ dance battles occurring (for each possible x). Otherwise, if the component is acyclic, then the situation is similar but without any periodicity by C — instead, there will simply be $F(x) \times (F(x) - 1) / 2$ dance battles (for each possible x).

The time complexity of this algorithm is $O(N + M)$.

Official Solution (C++)

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

const int MAXN = 300000;

int D[MAXN], C[MAXN];
vector<int> R[MAXN];
bool vis[MAXN];
int b, cycL;
unordered_map<int, int> cnt;

void DFS(int i, int d) {
  vis[i] = true;
  cnt[d] += C[i];
  for (int j = 0; j < R[i].size(); j++) {
    int k = R[i][j];
    if (k != b) {
      DFS(k, (d + 1) % cycL);
    }
  }
}
```

```
int main() {
  int N, M;
  cin >> N >> M;
  for (int i = 0; i < N; i++) {
    cin >> D[i];
    D[i]--;
    if (D[i] >= 0) {
      R[D[i]].push_back(i);
    }
  }
  for (int i = 0; i < M; i++) {
    int L;
    cin >> L;
    L--;
    C[L]++;
  }
  // Find and process each connected component.
  long long ans = 0;
  for (int i = 0; i < N; i++) {
    if (!vis[i]) {
      // Find the component's base node (any node on cycle, or node with out-degree 0).
      b = i;
      for (; D[b] >= 0 && !vis[b]; b = D[b]) {
        vis[b] = true;
      }
      // Find length of cycle, if any.
      if (D[b] < 0) {
        cycL = 1e9;
      } else {
        cycL = 1;
        for (int j = D[b]; j != b; j = D[j]) {
          cycL++;
        }
      }
      // Traverse entire component, and compute counts by distance to base node.
      cnt.clear();
      DFS(b, 0);
      // Update answer for each distance.
      for (unordered_map<int, int>::iterator I = cnt.begin(); I != cnt.end(); I++) {
        long long c = I->second;
        ans += c * (c - 1) / 2;
      }
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem S4: Super Luigi Odyssey

Our approach will consist of two overall steps. In the first step, we'll only concern ourselves with determining which platform Luigi will arrive at after each event. In the second step, we'll then compute the actual amount of energy used along the way.

To handle the first step, we'll begin by sorting the platforms in increasing order of position, and then constructing a 2D segment tree of platform heights in $O(N \log N)$ time. This will allow us to efficiently simulate the sequence of events — for each type-2 or type-3 event i, we can first determine the index c of Luigi's current platform in the ordered list of non-submerged platforms (in other words, the number of platforms to the left of it whose heights exceed the lava's current height), and then find either the $(c - X_i)$th or $(c + X_i)$th non-submerged platform from the left (if any). Each of these may be queried using the segment tree in $O(\log^2 N)$ time. Over the course of this

simulation, we'll assemble a list of jumping intervals (h, a, b), such that we know that Luigi will at some point jump platform-by-platform between platforms a and b while the lava height is h.

In the second step, our task will be to compute the sum of these jumping intervals' energy costs. We'll perform a line sweep over lava heights in decreasing order, with an event for each jumping interval (at its given lava height), as well as an event for each platform (at its height, such that it will be non-submerged for all subsequent lava heights). Along the way, we'll maintain an ordered set of non-submerged platforms (represented as a Balanced Binary Search Tree), as well as a Fenwick Tree (also known as a Binary Indexed Tree) with the cost of jumping from each pillar to its next non-submerged pillar to the right (or 0 if there's no such next pillar).

Each time a platform becomes unsubmerged, we can update the set in $O(\log N)$ time, and then consider the neighbouring unsubmerged platforms to its left/right to also update the Fenwick Tree in O(log N) time. Finally, each time we consider a jumping interval, we can query the Fenwick Tree for that interval of pillars in $O(\log N)$ time to determine the sum of energy costs of the jumps within it. These energy cost sums should then all be added together to yield the final answer.

Overall, this gives us an algorithm with time complexity $O(N \log N + M \log^2 N)$.

Official Solution (C++)

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

const int MOD = 1000000007;
const int MAXN = 250001;
const int SZ = 530000;

int N, M, K, NE;
int bot, lava;
pair<int, int> P[MAXN];
pair<int, pair<int, int> > E[SZ];
set<int> S;
int CostBIT[MAXN];
vector<int> TreeH[SZ];

int CalcCost(int a, int b) {
  int d = abs(P[b].first - P[a].first);
  return K == 1 ? d : (long long)d*d % MOD;
}

void UpdateCost(int i, int c) {
  for (i++; i <= N; i += (i & -i)) {
    CostBIT[i] = (CostBIT[i] + c) % MOD;
  }
}

int QueryCost(int i) {
  int c = 0;
  for (i++; i > 0; i -= (i & -i)) {
    c = (c + CostBIT[i]) % MOD;
  }
  return c;
}

int QueryP(int i, int r1, int r2, int p) {
  if (r1 == r2) {
```

```
    return 0;
  }
  i <<= 1;
  int m = (r1 + r2) >> 1;
  if (p <= m) {
    return QueryP(i, r1, m, p);
  }
  int c = TreeH[i].size();
  c -= lower_bound(TreeH[i].begin(), TreeH[i].end(), lava + 1) - TreeH[i].begin();
  return c + QueryP(i + 1, m + 1, r2, p);
}

int FindP(int i, int r1, int r2, int p) {
  if (r1 == r2) {
    return p == 0 ? r1 : -1;
  }
  i <<= 1;
  int m = (r1 + r2) >> 1;
  int c = TreeH[i].size();
  c -= lower_bound(TreeH[i].begin(), TreeH[i].end(), lava + 1) - TreeH[i].begin();
  if (p < c) {
    return FindP(i, r1, m, p);
  }
  return FindP(i + 1, m + 1, r2, p - c);
}

int main() {
  // Input platforms.
  cin >> N >> M >> K;
  for (int i = 0; i < N; i++) {
    cin >> P[i].first >> P[i].second;
  }
  // Sort platforms by position, and determine initial one.
  int strtP = P[0].first;
  sort(P, P + N);
  int curP = 0;
  for (int i = 0; i < N; i++) {
    if (P[i].first == strtP) {
      curP = i;
    }
  }
  // Construct 2D segment tree.
  bot = 1;
  while (bot < N) {
    bot <<= 1;
  }
  for (int i = 0; i < N; i++) {
    TreeH[bot + i].push_back(P[i].second);
  }
  for (int i = bot - 1; i > 0; i--) {
    int a = i*2, b = i*2 + 1;
    int aN = TreeH[a].size(), bN = TreeH[b].size();
    int aI = 0, bI = 0;
    while (aI < aN || bI < bN) {
      if (bI == bN || (aI < aN && TreeH[a][aI] < TreeH[b][bI])) {
        TreeH[i].push_back(TreeH[a][aI++]);
      } else {
        TreeH[i].push_back(TreeH[b][bI++]);
      }
    }
  }
  // Input events, and simulate them to compute sequence of platform intervals.
  for (int i = 0; i < M; i++) {
    int e, x;
    cin >> e >> x;
```

```
    if (e == 1) {
      // Update lava height.
      lava += x;
      if (lava >= P[curP].second) {
        cout << -1 << endl;
        return 0;
      }
    } else {
      // Jump left/right.
      int p = QueryP(1, 0, bot - 1, curP) + (e == 2 ? -x : x);
      int tot = TreeH[1].size();
      tot -= lower_bound(TreeH[1].begin(), TreeH[1].end(), lava + 1) - TreeH[1].begin();
      if (p < 0 || p >= tot) {
        cout << -1 << endl;
        return 0;
      }
      int nxtP = FindP(1, 0, bot - 1, p);
      if (nxtP < 0) {
        cout << -1 << endl;
        return 0;
      }
      // Create event for platform interval.
      E[NE++] = make_pair(lava, make_pair(curP, nxtP));
      curP = nxtP;
    }
  }
  // Create events for platforms.
  for (int i = 0; i < N; i++) {
    E[NE++] = make_pair(P[i].second, make_pair(-1, i));
  }
  // Sort events in non-increasing order of lava level, and sweep through them.
  int ans = 0;
  S.insert(-1);
  S.insert(N);
  sort(E, E + NE);
  reverse(E, E + NE);
  for (int i = 0; i < NE; i++) {
    int a = E[i].second.first, b = E[i].second.second;
    if (a < 0) {
      // Insert platform b.
      S.insert(b);
      set<int>::iterator I = S.find(b);
      set<int>::iterator J = I;
      I--, J++;
      int prv = *I, nxt = *J;
      if (prv >= 0) {
        if (nxt < N) {
          UpdateCost(prv, (MOD - CalcCost(prv, nxt)) % MOD);
        }
        UpdateCost(prv, CalcCost(prv, b));
      }
      if (nxt < N) {
        UpdateCost(b, CalcCost(b, nxt));
      }
    } else {
      // Process platform interval a..b.
      if (a > b) {
        swap(a, b);
      }
      ans = ((long long)ans + QueryCost(b - 1) - QueryCost(a - 1) + MOD) % MOD;
    }
  }
  cout << ans << endl;
  return 0;
}
```