WOBURN CHALLENGE

2018-19 Online Round 2

Solutions

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

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

Problem J1: Alternate Access

If $A < B$, then Ethan must climb upwards $B - A$ times, for a total energy cost of $U \times (B - A)$ Joules.
Otherwise (if $A > B$), he must climb downwards $A - B$ times, for a total energy cost of $D \times (A - B)$ Joules.

Official Solution (C++)

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

int main() {
  int A, B, U, D;
  cin >> A >> B >> U >> D;
  if (A < B) {
    cout << U*(B - A) << endl;
  } else {
    cout << D*(A - B) << endl;
  }
  return 0;
}
```

Problem J2: This Message will Self-Destruct

Let m be the numerical value (0..9) of the first character of S. We can obtain this value by subtracting the ASCII value of `"0"` from the ASCII value of that character. Similarly, let s_1 and s_2 be the numerical values of the third and fourth characters, respectively. Then, m corresponds to minutes, s_1 to tens of seconds, and s_2 to single seconds. The total number of seconds therefore comes out to $60m + 10s_1 + s_2$.

Official Solution (C++)

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

int main() {
  string S;
  cin >> S;
  cout << 60*(S[0] - '0') + 10*(S[2] - '0') + (S[3] - '0') << endl;
  return 0;
}
```

Problem J3/I1: Seeing Double

One approach we can take is to iterate over Benji's mask names B_i (for $i = 1..M$), and for each one, compare it against each of Ethan's mask names A_j (for $j = 1..N$), incrementing our running answer if we find a match (such that $B_i = A_j$). Over the course of this algorithm, we may compare each of the strings $A_{1..N}$ against each of the strings $B_{1..M}$, resulting in a time complexity of $O(N \times M)$.

Another possible approach is to start by inserting Ethan's mask names $A_{1..N}$ into a data structure such as a set (a balanced binary search tree) or hash table, and then iterate over each of Benji's mask names B_i (for $i = 1..M$), querying the data structure for the existence of B_j rather than iterating over all of the strings $A_{1..N}$ each time. The time complexity of this algorithm is either $O((N + M) \times \log(N))$ or $O(N + M)$, depending on the data structure used.

Official Solution (C++)

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

int main() {
  // Input first list, and store its names.
  int N;
  cin >> N;
  set<string> A;
  for (int i = 0; i < N; i++) {
    string s;
    cin >> s;
    A.insert(s);
  }
  // Input second list, and count names present in first list.
  int M;
  cin >> M;
  int ans = 0;
  for (int i = 0; i < M; i++) {
    string s;
    cin >> s;
    ans += A.count(s);
  }
  cout << ans << endl;
  return 0;
}
```

Problem J4/I2: Ammunition

Let's refer to guards whose B values are 0 as zero-guards, and to the remaining guards as positive-guards. A few useful observations can be made:

- If Ethan has at least 2 bullets, and he tranquilizes a zero-guard, he'll still have at least 1 bullet afterwards. There's no harm in doing this whenever possible.
- If Ethan has exactly 1 bullet, and he tranquilizes a zero-guard, he'll have 0 bullets afterwards and will need to stop. This should only be done when no other options remain.
- If Ethan has at least 1 bullet, and he tranquilizes a positive-guard, he'll still have at least 1 bullet afterwards. The only harm in doing this is based on how many excess bullets are wasted due to the gun's capacity being reached (which is worse the more bullets Ethan has). Therefore, this should only be done when Ethan has exactly 1 bullet, unless no other options remain.

This suggests an optimal greedy strategy for Ethan to follow at each point in time:

- If Ethan has 0 bullets remaining or there are 0 guards left, stop.
- If all remaining guards are positive-guards, tranquilize all of them in any order.
- If Ethan has 1 bullet remaining and there's at least 1 positive-guard, tranquilize any positive-guard.
- Otherwise, tranquilize any zero-guard.

It's possible to simulate this strategy in $O(N^2)$ time.

It's also possible to consider what exactly will occur over the course of the strategy and come up with a closed-form answer without needing to simulate it. Firstly, if $S = 0$, then the answer is 0. Otherwise, the answer is at most N, and

at most the total number of bullets which Ethan loads into his gun (including the initial S bullets). For each positive-guard i, he can tranquilize them when he has exactly 1 bullet and thus load $\min(B_i, M)$ bullets into his gun (unless he has so many bullets that he's forced to shoot a positive-guard while having 2 or more bullets, in which case the answer will come out to N either way). This gives us an answer of:

$$\min\left(S + \sum_{i=1}^{N} \min(B_i, M), N\right)$$

when $S > 0$, which may be evaluated in $O(N)$ time.

Official Solution (C++)

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

int main() {
  int N, M, S;
  cin >> N >> M >> S;
  if (S > 0) {
    for (int i = 0; i < N; i++) {
      int B;
      cin >> B;
      S += min(B, M);
    }
  }
  cout << min(S, N) << endl;
  return 0;
}
```

Problem I3/S1: Laser Grid

For convenience, let's pretend that there are two extra vertical lasers with V values 0 and 1,000,000, and similarly two extra horizontal lasers with H values 0 and 1,000,000. Then, let x_1 be the x-coordinate of the nearest vertical laser to the left of Ethan (in other words, the largest V value smaller than X_E). Similarly, let x_2 be the x-coordinate of the nearest vertical laser to the right of Ethan (the smallest V value larger than X_E), and let y_1 and y_2 be the y-coordinates of the nearest horizontal lasers below and above Ethan, respectively. The values x_1, x_2, y_1, and y_2 may be computed by iterating over all of the lasers' H and V values, in $O(N + M)$ time.

We can then observe that the i-th microchip is reachable if and only if $x_1 < X_i < x_2$ and $y_1 < Y_i < y_2$. This check can be performed in $O(1)$ time per microchip, resulting in a total time complexity of $O(N + M + K)$.

Official Solution (C++)

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

int E[2], L[2], C;
int low[2], high[2];

int main() {
  cin >> E[0] >> E[1] >> L[0] >> L[1] >> C;
```

```
  // For each dimension, handle lasers to narrow down reachable interval.
  for (int d = 0; d < 2; d++) {
    low[d] = 0, high[d] = 1e6;
    for (int i = 0; i < L[d]; i++) {
      int c;
      cin >> c;
      if (c < E[d]) {
        low[d] = max(low[d], c);
      } else {
        high[d] = min(high[d], c);
      }
    }
  }
  // Handle microchips.
  for (int i = 0; i < C; i++) {
    bool canReach = true;
    for (int d = 0; d < 2; d++) {
      int c;
      cin >> c;
      canReach &= low[d] < c && c < high[d];
    }
    cout << (canReach ? "Y" : "N") << endl;
  }
  return 0;
}
```

Problem I4/S2: Plutonium

Let's iterate over the days in reverse order, from N down to 1, while filling in their actual P values. We'll use $P_i = 0$ to represent P_i being allowed to take on any value based on $P_{i..N}$, and we'll set $P_{N+1} = 0$ for convenience.

When processing a day i, we should first consider whether or not there's a conflict between it and day $i + 1$, such that there's no way to assign both of them valid P values. This is only the case when $O_i \geq 1$, $P_{i+1} \geq 2$, and $O_i \neq P_{i+1} - 1$. If we encounter any instances of this, we should immediately output –1.

We should then fill in day i's P value. If O_i is positive, then we must have $P_i = O_i$. Otherwise, if $P_{i+1} \geq 2$, then we must have $O_i = P_{i+1} - 1$, and in the remaining case (when $P_{i+1} \leq 1$), we can set $O_i = 0$.

Having filled in the P values, we can examine them to determine the minimum and maximum possible number of withdrawals. Let's consider each day i such that $2 \leq i \leq N$. If $P_i = 1$, then there must have been a withdrawal on that day. And if $P_i = 0$, it's possible that there either was or wasn't a withdrawal on that day.

Official Solution (C++)

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

int N, O[200001], P[200001];
int ans1 = 0, ans2 = 0;

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> O[i];
  }
  // Add dummy extra day.
  P[N] = 0;
```

```
  // Iterate over days in reverse.
  for (int i = N - 1; i >= 0; i--) {
    // Conflict between this day and the following one?
    if (O[i] && P[i + 1] > 1 && O[i] != P[i + 1] - 1) {
      cout << -1 << endl;
      return 0;
    }
    // Determine this day's P value.
    P[i] = O[i] ? O[i] : max(0, P[i + 1] - 1);
    // This day must have had a withdrawal?
    if (i && P[i] == 1) {
      ans1++, ans2++;
    }
    // This day may have had an optional withdrawal?
    if (i && P[i] == 0) {
      ans2++;
    }
  }
  if (P[0] > 1) {
    cout << -1 << endl;
  } else {
    cout << ans1 << " " << ans2 << endl;
  }
  return 0;
}
```

Problem S3: Multitasking

At any given point in the bomb defusing process, Ilsa has cut i wires, completely defused a bombs, and partially defused b bombs. A partially-defused bomb has exactly one of its wires cut.

We can do dynamic programming where DP[a][b] is the probability that there are a completely-defused bombs and b partially-defused after Ilsa has cut $2a + b$ wires. Necessarily, the sum of all DP values for the same number of wires cut must equal 1. Note that all numbers of cut wires i are equally likely before Ethan walks into the room.

We start with DP[0][0] = 1, and with all other DP values initialized to 0. When 0 wires have been cut, it must be the case that no bombs are either completely or partially defused. We can then iteratively compute the DP value for all states where 1 wire is cut, then 2 wires, and so on as follows:

For a given number of wires i, such that we've already computed all DP values that correspond to i wires being cut, we can compute the values for $i + 1$ wires being cut by iterating over all possible states (a, b) such that $2a + b = i$. For each such state, we compute the probability p that the next wire cut will completely defuse a partially-defused bomb, and the probability q that the next wire cut will be the first wire cut on a new bomb:

- There are b wires that would complete a partially-defused bomb if cut, and $2N - i$ uncut wires. Therefore, $p = b / (2N - i)$.
- p and q must sum to 1. Therefore, $q = 1 - p$.

Then, we update two of the states for $i + 1$ wires as follows:

- With probability p, there will be one more completely-defused bomb and one fewer partially-defused bombs. Therefore, we can increase DP[$a + 1$][$b - 1$] by DP[a][b] $\times$ p.
- With probability q, there will be one more partially-defused bomb. Therefore, we can increase DP[a][$b + 1$] by DP[a][b] $\times$ q.

Let U be the random variable of the number of uncut wires (and let u be a specific number of uncut wires). The answer we actually want to compute is the expected number of uncut wires, given that we know that there are K completely-defused bombs:

$$E[U \mid a = K] = 1 \times P(U = 1 \mid a = K) + \ldots + 2N \times P(U = 2N \mid a = K)$$

Let S[K] be the sum of dp[K][b] over all possible values of b. We can then state that:

$$P(U = u \mid a = K) = DP[K][2N - 2K - u] / S[K]$$

The probability of being in a given state (a, b) that corresponds to u uncut wires is proportional to DP[a][b]. Since all values of u are equally likely without any information, the probability of being in state (a, b) is simply DP[a][b] / S[K] after you know that $a = K$.

Computing the table of DP values takes $O(N^2)$ time, and combining them together takes $O(N)$ time, so the total time complexity is $O(N^2)$.

Taking the second sample case as an example, there are three states in which Ethan might walk in on Ilsa: (0, 0), (0, 1), and (0, 2), since a is known to be 0. It must be the case that Ilsa has 2, 3, or 4 wires remaining since with any fewer wires remaining there would have been at least 1 completely-defused bomb. All of these values of u are equally-likely when Ethan walks into the room, but when he observes that no bombs have been completely-defused, the chance that exactly 2 wires remain uncut is decreased since there's a chance that one bomb would have been completely defused by that point.

The chance that 2 cut wires belong to the same bomb (out of 2 total bombs) is 1 / 3, so the chance of having 2 wires uncut and 0 completely-defused bombs is only 2/3 the chance of having 3 wires uncut or 4 wires uncut. Observe that DP[0][2] = 2/3, while DP[0][0] = DP[0][1] = 1.

$$P(U = 4 \mid a = 0) = DP[0][0] / S[K] = 1 / (8/3) = 3/8$$
$$P(U = 3 \mid a = 0) = DP[0][1] / S[K] = 1 / (8/3) = 3/8$$
$$P(U = 2 \mid a = 0) = DP[0][2] / S[K] = 2/3 / (8/3) = 2/8$$

Dropping terms that equal 0, the answer we want to compute is

$$E[U \mid a = 2] = 2 \times P(U = 2 \mid a = 0) + 3 \times P(U = 3 \mid a = 0) + 4 \times P(U = 4 \mid a = 0)$$
$$E[U \mid a = 2] = (2 \times 2/8) + (3 \times 3/8) + (4 \times 3/8)$$
$$E[U \mid a = 2] = 25 / 8 = 3.125$$

Official Solution (C++)

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

int N, M, K;
double DP[4001][2002];  // DP[i][k] = P(exactly k bombs have been defused after i cuts).

int main() {
  cin >> N >> K;
  M = N * 2;
  DP[0][0] = 1;
  for (int i = 0; i < M; i++) {
    for (int k = 0; k <= K; k++) {
      int remainingBombs = M - i;
      int halfCompleteBombs = i - 2 * k;
      double bombCompletionProb = (double)halfCompleteBombs / remainingBombs;
      DP[i + 1][k + 1] += DP[i][k] * bombCompletionProb;
      DP[i + 1][k] += DP[i][k] * (1 - bombCompletionProb);
```

```
    }
  }
  double sum = 0.0, weight = 0.0;
  for (int i = 0; i <= M; i++) {
    sum += (2 * N - i) * DP[i][K];
    weight += DP[i][K];
  }
  cout << fixed << setprecision(9) << sum / weight << endl;
  return 0;
}
```

Problem S4: Car Convergence Chaos

Given a pair of starting intersections E and S, with $M = (E + S) / 2$, let K be the number of intersections which each car will visit ($K = M - S + 1$). Let $X_{1..K}$ be the times at which Ethan would arrive at each of the intersections on his path (such that $X_1 = 0$, $X_2 = T_S$, $X_3 = T_S + T_{S+1}$, and so on). Similarly, let $Y_{1..K}$ be Solomon's sequence of arrival times. Then, let $D_{1..K}$ be a sequence of differences between these arrival times (such that $D_i = X_i - Y_i$). If we were to remove all 0's from $D_{1..K}$, then the CCCC would be equal to the number of elements in D which either have no preceding element, or have a different sign than their preceding element. If we consider each (E, S) pair independently, computing its D sequence and resulting CCCC value in this fashion would take $O(K)$ time, which would result in an $O(N^3)$ algorithm overall. We'll need to do better than that to obtain full marks.

Let's consider each possible bomb intersection M, and let's imagine for the moment that Ethan and Solomon are each driving away from it rather than towards it. We'll define a similar set of sequences to the ones described above. Let X'_i be the time at which Ethan would arrive at the i-th intersection when driving left from M (such that $X'_1 = 0$, $X'_2 = T_M$, $X'_3 = T_M + T_{M-1}$, and so on). Similarly, let Y'_i be the time at which Solomon would arrive at the i-th intersection when driving right from M, and let $D'_i = X'_i - Y'_i$. We can now make the key insight that, for any K, $D_{1..K}$ is similar to $D'_{1..K}$ — just reversed (which is irrelevant), and shifted by subtracting D'_K from each of its values (such that D_1 ends up being equal to 0).

One way of thinking about the CCCC of a given sequence $D_{1..K}$ is that it's equal to the number of times that it "crosses" 0 (the number of times which there's a positive value followed by zero or more 0's followed by a negative value, or vice versa), plus 1 if the sequence has at least 1 non-zero value. And by the same token, the CCCC of a given sequence $D'_{1..K}$ is then the number of times that it "crosses" D'_K. For example:

- The CCCC of D' = [3, 3, 3] is 0 (as its values are all equal to 3)
- The CCCC of D' = [1, 3, 2] is 2 (as 2 is crossed by 1 → 3, and not all of its values are equal to 2)
- The CCCC of D' = [1, 2, 2, 3, 2] is 2 (as 2 is crossed by 1 → 2 → 2 → 3, and not all of its values are equal to 2)
- The CCCC of D' = [1, 2, 2, 1, 2] is 1 (as not all of its values are equal to 2)

We can represent a D' sequence as a set of intervals of crossed values, based on pairs of adjacent elements. For example, if D' = [5, 10, 10, 15, 12], then all values in the *exclusive* intervals (5, 10), (10, 10), (10, 15), and (12, 15) are crossed (in other words, in the *inclusive* intervals [6, 9], [11, 14], and [13, 14]). Furthermore, if we exclude intervals with both endpoints equal (of the form (x, x)), then if two intervals in a row go in the same direction (either both are increasing or both are decreasing), then their joining value is also crossed. In this example, 5 → 10 and 10 → 15 satisfy this property, meaning that 10 is also crossed. However, 10 → 15 and 15 → 12 don't, meaning that 15 is not crossed.

With this representation, we're ready to compute all of the CCCCs for a given bomb intersection M efficiently. We'll start by iterating outwards from it in both direction to compute its full D' sequence in $O(N)$ time. We'll then compress the original values in D' down to values 1..C, where C is the number of distinct values in D' (such that C is in $O(N)$), in O(N log N) time. We'll proceed to iterate i upwards from 1 to $|D'|$ while maintaining information about intervals of crossed values — specifically, we'll use a binary indexed tree to maintain an array A in which A_x is the difference between the number of intervals starting at and ending at x (such that the sum of $A_{1..x}$ is then the number of intervals spanning x). For each i, we'll query for the number of intervals so far which encompass D'_i in order to compute the CCCC of $D'_{1..i}$ (in other words, the CCCC of the (S, E) pair (M – (i – 1), M + (i – 1))), and we'll then insert at most 2 intervals based on D'_i. In total, this process takes $O(N \log N)$ time per bomb intersection M, resulting in an overall time complexity of $O(N^2 \log N)$.

Official Solution (C++)

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

int N, K;
int T[3000];
int BIT[1505];  // Binary indexed tree storing the number of intervals spanning each index i.

// Update the BIT by adding v onto index i.
void Update(int i, int v) {
  for (i++; i <= K; i += (i & -i)) {
    BIT[i] += v;
  }
}

// Update the BIT by inserting the interval (min(i, j), max(i, j)).
void InsertExclusiveInterval(int i, int j) {
  int a = min(i, j) + 1;
  int b = max(i, j) - 1;
  if (a <= b) {
    Update(a, 1);
    Update(b + 1, -1);
  }
}

// Query the BIT for the sum of indices 0..i.
int Query(int i) {
  int v = 0;
  for (i++; i > 0; i -= (i & -i)) {
    v += BIT[i];
  }
  return v;
}

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> T[i];
  }
  // Consider each midpoint m.
  long long ans = 0;
  for (int m = 1; m < N - 1; m++) {
    // Expand outwards from m, compute sequence of differences.
    int a = m, b = m, d = 0;
    vector<int> D;
```

```
    while (a >= 0 && b < N) {
      D.push_back(d += T[a--] - T[b++]);
    }
    // Coordinate-compress differences.
    vector<int> comp(D);
    sort(comp.begin(), comp.end());
    comp.resize(K = unique(comp.begin(), comp.end()) - comp.begin());
    // Consider each prefix of differences, maintaining BIT of difference intervals.
    memset(BIT, 0, sizeof BIT);
    int prevD, prevOffset = 0;
    for (int i = 0; i < D.size(); i++) {
      // Get the current compressed difference
      int d = lower_bound(comp.begin(), comp.end(), D[i]) - comp.begin();
      if (i) {
        // Add on the number of times this difference has previously been crossed.
        ans += Query(d);
        // Insert the interval between the previous difference and this one.
        InsertExclusiveInterval(prevD, d);
        // Get the offset between the previous difference and this one.
        int offset = d - prevD;
        if (offset && prevOffset && (offset > 0) == (prevOffset > 0)) {
          // The previous difference has now been crossed, so insert an interval just for it.
          InsertExclusiveInterval(prevD - 1, prevD + 1);
        }
        if (offset) {
          prevOffset = offset;
        }
        // Add 1 for this self-crossing difference, unless all differences have been 0 so far.
        if (prevOffset) {
          ans++;
        }
      }
      prevD = d;
    }
  }
  cout << ans << endl;
  return 0;
}
```