WOBURN CHALLENGE

2018-19 Online Round 1

Solutions

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

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

Problem J1: Homework

The total time required to complete the assignments is $A \times M$ minutes. So, upon inputting the values A, M, and T, we should check if $A \times M \leq T$, output "Y" if so, and output "N" otherwise.

Official Solution (C++)

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

int main() {
  int A, M, T;
  cin >> A >> M >> T;
  if (A * M <= T) {
    cout << "Y" << endl;
  } else {
    cout << "N" << endl;
  }
  return 0;
}
```

Problem J2: Making the Cut

We can create a loop which runs 5 times. On each iteration, we should input a name, and if it's equal to N, then we should output "Y" and stop. If the loop finishes and we still haven't found a matching name by the end, we should then output "N" instead.

Official Solution (C++)

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

int main() {
  string N, S;
  cin >> N;
  for (int i = 0; i < 5; i++) {
    cin >> S;
    if (S == N) {
      cout << "Y" << endl;
      return 0;
    }
  }
  cout << "N" << endl;
  return 0;
}
```

Problem J3/I1: Comparing Grades

Let's create a function Grade(g) which takes a percentage grade g as its argument, and returns the letter grade which corresponds to g. This function may be implemented with a series of if-statements which check which of the letter grades' ranges g falls into. We can then check whether Grade(A) is equal to Grade(B), outputting "`Same`" if so, and "`Different`" otherwise.

Official Solution (C++)

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

char Grade(int g) {
  if (g >= 90) {
    return 'A';
  } else if (g >= 80) {
    return 'B';
  } else if (g >= 70) {
    return 'C';
  } else if (g >= 60) {
    return 'D';
  }
  return 'F';
}

int main() {
  int A, B;
  cin >> A >> B;
  if (Grade(A) == Grade(B)) {
    cout << "Same" << endl;
  } else {
    cout << "Different" << endl;
  }
  return 0;
}
```

Problem J4/I2: Germaphobia

Let's define MinDoors(x, y) to be the minimum number of doors which Bob must pass through to get from a classroom x to another classroom y. The only options he should consider are to avoid the hallways and pass exclusively through classrooms (which requires $|y - x|$ doors), or to go through the hallway (which always requires 2 doors). Therefore, MinDoors(x, y) = min($|y - x|$, 2). The total answer is then the sum of MinDoors(C_i, C_{i+1}) for each i from 1 to $M - 1$, plus an extra 2 (for one door required to get from the hallway to the first class, and another door required to get from the final class to the hallway).

Official Solution (C++)

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

int N, M;
int C[100];

int main() {
  cin >> N >> M;
  int ans = 2;  // Always 2 doors (before first class and after last class).
  for (int i = 0; i < M; i++) {
    cin >> C[i];
    if (i > 0) {
      // At most 2 doors to move from the previous class.
      ans += min(2, abs(C[i] - C[i - 1]));
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem I3/S1: Inspiration

The most direct solution involves iterating over all desks in the grid, and for each type-2 student encountered, iterating over all desks that they can see and checking if a type-1 student is sitting at any of them. Unfortunately, this approach is too slow to receive full marks. There can be up to $R \times C$ type-2 students, and the average number of desks within view of each one is on the order of K, meaning that the time complexity of this solution is $O(R \times C \times K)$.

If we assume that $K = R - 1$, then we can come up with a faster solution as follows. For each column, we can iterate through its desks in order from front to back, while maintaining a Boolean variable b indicating if there have been any type-1 students so far (initially set to false). When we encounter a type-1 student, we should set b to true, and when we encounter a type-2 student, we know that they can be inspired if and only if b is currently true. The time complexity of this solution is $O(R \times C)$, which is fast enough.

This approach can then be generalized to working for any value of K as follows. Let's swap out b for a variable p which represents the row of the most recent (furthest-back) type-1 student encountered so far (initially set to a very negative value). When we encounter a type-1 student at row r, we should set p to be equal to r, and when we encounter a type-2 student at row r, we know that they can be inspired if and only if $p \geq r - K$.

Official Solution (C++)

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

int R, C, K;
int D[50000][10];

int main() {
  cin >> R >> C >> K;
  for (int r = 0; r < R; r++) {
    for (int c = 0; c < C; c++) {
      cin >> D[r][c];
    }
  }
  // Consider each column.
  int ans = 0;
  for (int c = 0; c < C; c++) {
    // Iterate through the rows, maintaining the most recent type-1 student's row.
    int p = -1e9;
    for (int r = 0; r < R; r++) {
      if (D[r][c] == 1) {
        // Found a type-1 student.
        p = r;
      } else if (D[r][c] == 2 && p >= r - K) {
        // Found a type-2 student who's close enough to the most recent type-1 student.
        ans++;
      }
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem I4/S2: Essay Generator

The essay should include as many distinct length-1 words as possible (of which there are 26). If Alice runs out of those, then the essay should be filled in with as many distinct length-2 words as possible (of which there are 26^2 = 676). If Alice also runs out of those, then she should pad out the remainder with length-3 words. She'll never need to resort to length-4 words, as the total number of words with lengths 1..3 is $26 + 26^2 + 26^3$ = 18,278, which is greater than the maximum value of W (10,000).

With that in mind, there are various possible ways to generate an optimal essay in $O(W)$ time. We might consider each length L from 1 to 3 in turn, and generate all words of length L in any order (either with recursion or with L nested loops), appending them to the essay and stopping early if the essay's length reaches W. It's also possible to get it done with just three nested loops representing the word's first, second, and third characters, with the first two loops including placeholder values which represent words with shorter lengths.

Official Solution (C++)

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

int main() {
  int W;
  cin >> W;
  // Construct words in non-decreasing order of length
  for (char a = 'a' - 1; a <= 'z'; a++) {
    for (char b = 'a' - (a < 'a' ? 1 : 0); b <= 'z'; b++) {
      for (char c = 'a'; c <= 'z'; c++) {
        if (a >= 'a') {
          cout << a;
        }
        if (b >= 'a') {
          cout << b;
        }
        cout << c;
        if (--W) {
          cout << " ";
        } else {
          cout << endl;
          return 0;
        }
      }
    }
  }
  return 0;
}
```

Problem S3: Reach for the Top

This problem may be solved with two distinct approaches.

Graph Theory Approach

Let's represent the rope as a directed graph with $2H$ nodes (numbered from 0 to $2H - 1$), with each node h representing Bob holding onto the rope at a height of h. Each action which would take Bob from a height h_1 to another height h_2 then corresponds to an edge from node h_1 to node h_2. We're then looking for the shortest distance (minimum sum of edge weights) from node 0 to any node from H upwards.

BFS (Breadth First Search) is a well-known algorithm which can be used to compute shortest distances on unweighted graphs such as this one. The time complexity of BFS is $O(V + E)$, where V is the number of nodes in the graph, and E is the number of edges. In this case V is in $O(H)$, but E is in $O(H^2)$, as each node has $O(H)$ outgoing edges corresponding to possible drop actions from it. This gives us an overall time complexity of $O(H^2 + N)$, which is too slow to earn full marks.

It's possible to optimize this approach by making the graph weighted, and adding an extra set of $2H$ nodes to it. Let's refer to the original nodes as $X_{0..(2H-1)}$, and the new nodes as $Y_{0..(2H-1)}$, with each new node Y_h representing Bob being mid-drop at a height of h. We'll want to have a weight-0 edge from each node Y_h to Y_{h-1}, representing the continuation of a drop. The start of a drop can then be represented by a weight-1 edge from each node X_h to Y_h, while the end of a drop can be represented by a weight-0 edge from each node Y_h back to X_h. This removes the need for $O(H)$ outgoing drop-related edges from each node!

We can no longer use BFS due to the graph being weighted, but in this case, we can use a variant of it which works when all edge weights are equal to either 0 or 1. This variant uses a deque rather than a queue, and pushes nodes coming from weight-0 edges onto the front rather than the back of the deque. It similarly runs in $O(V + E)$. Since V and E are now in $O(H)$, this gives us an overall time complexity of $O(H + N)$, which is fast enough. Note that Dijkstra's algorithm could be used instead, though it's slower and slightly more complex.

Dynamic Programming Approach

Let $DP[h]$ be the minimum number of jumps required for Bob to hold onto the rope at a height of h, and let $M[d]$ be the maximum height h such that $DP[h] = d$. Initially, we have $DP[0] = 0$ and $M[0] = 0$, and we'll let $M[-1] = -1$ for convenience. We'll proceed to fill in the DP values in a series of phases corresponding to increasing d values, starting with $d = 0$. For each d, we'll iterate over heights h from $M[d - 1] + 1$ up to $M[d]$. For each such h, either $DP[h] = d$, or $DP[h]$ has yet to be filled in and we can set it to $d + 1$ (due to dropping down from a height of $M[d]$). Either way, we'll end up with this phase of DP values all filled in, and from each h we can fill in $DP[h + J]$ as well, while updating $M[DP[h + J]]$ as appropriate. As soon as we arrive at $M[d] \geq H$, d must be the answer, and if we instead run out of new phases to explore, then the answer must be –1.

Official Solution – Graph Theory (C++)

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

const int MAXH = 1000000;

int H, J, N;
int dist[MAXH + 1][2];
bool B[MAXH + 1], vis[MAXH + 1][2];
deque< pair<int, int> > Q;

// Consider moving to state (i, j) along a w-weight edge from distance d.
void Go(int i, int j, int d, int w) {
  i = min(i, H);  // Cap at a height of H.
  if (i >= 0 &&           // Within the rope?
      (j || !B[i]) &&     // Not trying to hold onto itching powder?
      d + w < dist[i][j]  // Best distance to that state thus far?
  ) {
    dist[i][j] = d + w;
    if (w) {
      Q.push_back(make_pair(i, j));
```

```
    } else {
      Q.push_front(make_pair(i, j));
    }
  }
}

int main() {
  cin >> H >> J >> N;
  for (int i = 0; i < N; i++) {
    int a, b;
    cin >> a >> b;
    for (int j = a; j <= b; j++) {
      B[j] = true;
    }
  }
  // BFS.
  memset(dist, 60, sizeof dist);
  Q.push_back(make_pair(0, 0));
  dist[0][0] = 0;
  while (!Q.empty()) {
    int i = Q.front().first, j = Q.front().second;  // Get next state.
    Q.pop_front();
    if (vis[i][j]) {
      continue;  // Already visited this state.
    }
    vis[i][j] = true;
    int d = dist[i][j];
    if (i == H) {  // Done?
      cout << d << endl;
      return 0;
    }
    // Currently dropping?
    if (j) {
      Go(i - 1, 1, d, 0);  // Continue dropping.
      Go(i, 0, d, 0);  // Stop dropping.
    } else {
      Go(i + J, 0, d, 1);  // Jump.
      Go(i, 1, d, 1);  // Start dropping.
    }
  }
  cout << -1 << endl;  // Impossible.
  return 0;
}
```

Official Solution – Dynamic Programming (C++)

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

const int MAXH = 1000000;

int H, J, N;
int DP[MAXH + 1], M[MAXH + 2];
bool B[MAXH + 1];

int main() {
  cin >> H >> J >> N;
  for (int i = 0; i < N; i++) {
    int a, b;
    cin >> a >> b;
    for (int j = a; j <= b; j++) {
      B[j] = true;
    }
  }
```

```
  // DP in phases.
  memset(DP, 60, sizeof DP);
  DP[0] = M[0] = 0;
  for (int d = 0; d <= H; d++) {
    if (M[d] >= H) {  // This phase reaches the top?
      cout << d << endl;
      return 0;
    }
    // Consider all heights in this phase.
    for (int i = (d ? M[d - 1] + 1 : 0); i <= M[d]; i++) {
      if (!B[i]) {
        // Consider dropping down from the top of this phase.
        DP[i] = min(DP[i], d + 1);
        // Consider jumping up from here.
        int h = min(i + J, H);
        if (!B[h]) {
          DP[h] = DP[i] + 1;
          M[DP[i] + 1] = h;
        }
      }
    }
  }
  cout << -1 << endl;  // Impossible.
  return 0;
}
```

Problem S4: Bad Influence

Let V_i be the volatility of students 1..i. The volatility of a set of students is simply the number of students i who cannot be coerced into using their phones by anybody else (in other words, the number of students i for which there exists no student j such $S_i < S_j$ and $D_j - R_j \leq D_i \leq D_j + R_j$). We can thus think of each student as contributing either 0 or 1 volatility to each V value. Let M_i be the smallest value of j which satisfies the above conditions (or $M_i = N + 1$ if there's no such j). If $M_i \leq i$, then student i contributes no volatility at all, and otherwise, they contribute 1 volatility to each of the values $V_{i..(M\_i-1)}$. So, if we had the values $M_{1..N}$, we could use them to compute the volatilities $V_{1..N}$ with a simple $O(N)$ sweep.

What remains is computing the values $M_{1..N}$ efficiently. Let's process the N students in decreasing order of S value, either by explicitly sorting them by their S values, or by keying them by their S values and then iterating over all possible S values from 10^6 down to 1. Let X_d be the minimum index (in the original ordered student list) of any student processed so far whose range of influence includes desk d (or X_d = infinity if there's been no such student so far). When processing student i, we can observe that M_i must be equal to X_{Pi}. And upon finishing with student i, we should update X_d to min(X_d, i) for each desk d in the inclusive interval $[D_i - R_i, D_i + R_i]$.

Let's maintain the values $X_{1..K}$ (where K is the maximum possible D value) using a segment tree with lazy propagation, in which each node stores the minimum value of any X value in its interval, as well as the minimum new value which all X values in its interval should lazily be updated with. This allows us to both query an X value and update a range of X values as necessary in $O(\log K)$ time each, resulting in an overall time complexity of $O((N \log K) + Z)$ (where K is the maximum possible D value and Z is the maximum possible S value).

Official Solution (C++)

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

const int MAXN = 300000, MAXS = 1000000, MAXD = 1000000, MAXT = 2100000;

int N;
int I[MAXS + 1], D[MAXS + 1], R[MAXS + 1], delta[MAXN + 1];
int sz, treeMin[MAXT], treeLazy[MAXT];

void Prop(int i) {
  treeMin[i] = min(treeMin[i], treeLazy[i]);
  if (i < sz) {
    for (int c = i * 2; c <= i * 2 + 1; c++) {
      treeLazy[c] = min(treeLazy[c], treeLazy[i]);
    }
  }
  treeLazy[i] = 1e9;
}

void Update(int i, int r1, int r2, int a, int b, int v) {
  Prop(i);
  if (a <= r1 && r2 <= b) {
    treeLazy[i] = v;
    Prop(i);
    return;
  }
  int c = i * 2, m = (r1 + r2) / 2;
  if (a <= m) {
    Update(c, r1, m, a, b, v);
  }
  if (b > m) {
    Update(c + 1, m + 1, r2, a, b, v);
  }
  Prop(c);
  Prop(c + 1);
  treeMin[i] = min(treeMin[c], treeMin[c + 1]);
}

int Query(int i, int r1, int r2, int a, int b) {
  Prop(i);
  if (a <= r1 && r2 <= b) {
    return treeMin[i];
  }
  int c = i * 2, m = (r1 + r2) / 2, ret = 1e9;
  if (a <= m) {
    ret = min(ret, Query(c, r1, m, a, b));
  }
  if (b > m) {
    ret = min(ret, Query(c + 1, m + 1, r2, a, b));
  }
  return ret;
}

int main() {
  cin >> N;
  for (int i = 1; i <= N; i++) {
    int s, d, r;
    cin >> s >> d >> r;
    I[s] = i;
    D[s] = d;
    R[s] = r;
  }
```

```
  // Initialize tree.
  sz = 1;
  while (sz <= MAXS) {
    sz *= 2;
  }
  memset(treeMin, 60, sizeof treeMin);
  memset(treeLazy, 60, sizeof treeLazy);
  // Process students in decreasing order of social standing.
  for (int s = MAXS; s >= 0; s--) {
    int i = I[s];
    if (!i) {
      continue;
    }
    int d = D[s], r = R[s];
    int m = min(N + 1, Query(1, 0, sz - 1, d, d));
    if (m > i) {
      delta[i]++;
      delta[m]--;
    }
    Update(1, 0, sz - 1, max(d - r, 1), min(d + r, MAXD), i);
  }
  // Compute answers, and output.
  int v = 0;
  for (int i = 1; i <= N; i++) {
    v += delta[i];
    cout << v << endl;
  }
  return 0;
}
```