WOBURN CHALLENGE

2017-18 Online Round 3

Solutions

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

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

Problem J1: Like, Comment, and Subscribe

With S limited to the interval [1, 100], there are only three possible next milestone counts:

- If $1 \leq S \leq 9$, then the next milestone is 10.
- If $10 \leq S \leq 99$, then the next milestone is 100.
- If $S = 100$, then the next milestone is 1000.

We can use a few if-statements to determine which of the three above cases we're in, and then subtract S from the resulting milestone count to obtain the answer.

Official Solution (C++)

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

int main() {
  int S;
  cin >> S;
  if (S < 10) {
    cout << 10 - S << endl;
  } else if (S < 100) {
    cout << 100 - S << endl;
  } else {
    cout << 1000 - S << endl;
  }
  return 0;
}
```

Problem J2: Certified Fresh

We can process the N reviews one by one, while maintaining three pieces of information – the number of positive reviews so far, the total number of reviews so far, and the maximum Tomatometer score achieved so far (all initialized to 0).

When processing a review, we can update the positive/total review counts, compute the current Tomatometer score, and update the maximum Tomatometer score if it's now been surpassed. After processing all N reviews, we can output the maximum Tomatometer score achieved.

Official Solution (C++)

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

int N, numP = 0;
double maxS = 0;

int main() {
  cin >> N;
  for (int i = 1; i <= N; i++) {
    char R;
    cin >> R;
    if (R == 'P') {
      numP++;
    }
    maxS = max(maxS, (double)numP / i);
  }
  cout << fixed << setprecision(9) << maxS << endl;
  return 0;
}
```

Problem J3/I1: Uncrackable

Let's start by iterating over the characters in the password one by one, while tallying up the counts of lowercase letters, uppercase letters, and digits encountered. What remains is to put together an if statement which checks whether all of the conditions described in the problem statement have been satisfied, to determine whether we should output "Valid" or "Invalid":

- password length ≥ 8
- password length ≤ 12
- lowercase letter count ≥ 3
- uppercase letter count ≥ 2
- digit count ≥ 1

Official Solution (C++)

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

string P;
int lower = 0, upper = 0, digit = 0;

int main() {
  cin >> P;
  for (int i = 0; i < (int)P.length(); i++) {
    if (P[i] >= 'a' && P[i] <= 'z') {
      lower++;
    } else if (P[i] >= 'A' && P[i] <= 'Z') {
      upper++;
    } else {
      digit++;
    }
  }
  if (
    P.length() >= 8 &&
    P.length() <= 12 &&
    lower >= 3 &&
    upper >= 2 &&
    digit >= 1
  ) {
    cout << "Valid" << endl;
  } else {
    cout << "Invalid" << endl;
  }
  return 0;
}
```

Problem J4/I2: Meme Generator

Upon reading in the input, the trickiest part is determining where exactly the *T* and *B* strings should be overlaid onto the image grid, such that they end up centered as required.

Let $|T|$ be the number of characters in *T*. Then, the first character of *T* should replace character $s = \lfloor(C - |T|) / 2\rfloor + 1$ in the second row of the image grid (1-indexed). At that point, we can iterate over each character T_i (for $i = 1..|T|$), and if T_i isn't an underscore, we should replace character $s + i - 1$ in the second row of the image grid with T_i.

The same process can be repeated to overlay *B* onto the second-last row of the image grid. Following that, we can proceed to output the entire updated image grid, one row at a time.

Official Solution (C++)

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

int R, C;
string G[105], T, B;

int main() {
  cin >> R >> C;
  for (int i = 0; i < R; i++) cin >> G[i];
  cin >> T >> B;
  int c = (C - T.length()) / 2;
  for (int i = 0; i < T.length(); i++) {
    if (T[i] != '_') {
      G[1][c + i] = T[i]; // Overlay T.
    }
  }
  c = (C - B.length()) / 2;
  for (int i = 0; i < B.length(); i++) {
    if (B[i] != '_') {
      G[R - 2][c + i] = B[i]; // Overlay B.
    }
  }
  for (int i = 0; i < R; i++) {
    cout << G[i] << endl;
  }
  return 0;
}
```

Problem I3/S1: Mutual Friends

Let F be the number of different possible friendships. There's one for each unordered pair of users, meaning that $F = N(N - 1) / 2$. F is small enough (only 15 when $N = 6$) that it's viable to consider each possible subset of these friendships. There are 2^F such subsets, and we can consider all of them using depth first search (recursion), or by iterating over all bitmasks from 0 to $2^F - 1$.

For each considered set of friendships, we can then compute its resulting grid of mutual friend counts by iterating over all $O(N^2)$ pairs of users (i, j), and counting the number of other users k such that the set of friendships includes both unordered pairs (i, k) and (j, k). This process takes $O(N^3)$ time. If the resulting grid of mutual friend counts exactly matches the required grid M, then we've found a valid set of friendships, meaning that we can output it and stop the depth first search. If we finish considering all 2^F possible sets of friendships without coming across a valid one, then the answer must be `Impossible`.

The time complexity of the algorithm described above is $O(2^F \times N^3)$.

Official Solution (C++)

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

int N, M[7][7];
bool F[7][7], solved;

void Rec(int i, int j) {
  if (solved) return;  // Already found a solution.
  if (i > N) {  // All friendships filled in.
    // Check whether this set of friendships is valid.
    int numF = 0;
    for (int i = 1; i <= N; i++) {
      for (int j = 1; j < i; j++) {
        if (F[i][j]) numF++;
        int m = 0;
        for (int k = 1; k <= N; k++) {
          if (k != i && k != j && F[i][k] && F[j][k]) {
            m++;
          }
        }
        if (m != M[i][j]) return;  // It's invalid.
      }
    }
    // It's valid.
    cout << numF << endl;
    for (int i = 1; i <= N; i++) {
      for (int j = 1; j < i; j++) {
        if (F[i][j]) cout << i << " " << j << endl;
      }
    }
    solved = true;
    return;
  }
  if (j >= i) {  // Time to move to the next i.
    Rec(i + 1, 1);
    return;
  }
  // Consider i and j being friends.
  F[i][j] = F[j][i] = true;
  Rec(i, j + 1);
  // Consider i and j not being friends.
  F[i][j] = F[j][i] = false;
  Rec(i, j + 1);
}

int main() {
  cin >> N;
  for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
      cin >> M[i][j];
    }
  }
  Rec(1, 1);
  if (!solved) {
    cout << "Impossible" << endl;
  }
  return 0;
}
```

Problem I4/S2: GleamingProudChickenFunRun

This problem can be solved with a greedy approach, by generally considering the clips from earliest to latest (though note that this is not a well-defined ordering itself, as clips have two potential endpoints to sort by).

Let clip i be the clip which ends the earliest (the one with the smallest B value). Assuming that we'd like to exclude clip i from S in an effort to minimize $|S|$, we'll need to include in S some clip j which overlaps with clip i (such that $A_j < B_i$). If there are multiple valid clips like this, we should choose the one which ends the latest (the one with the largest B value), so that it's more likely to take care of overlapping with more later-occurring clips. Note that the chosen clip j may end up being equal to clip i. Having included clip j in S, we can effectively ignore all other clips overlapping with it (clips k such that $A_k < B_j$). This leaves us with just the set of clips k such that $A_k \geq B_j$, on which the above process can be repeated – that is, we'll next look for the clip which ends the earliest of the remaining ones. Once there are no remaining clips, we can stop, as all clips will have been accounted for (by either having been included in S, or by being known to overlap with a clip in S).

If we sort the clips in non-decreasing order of B (in $O(N \log(N))$ time), the above greedy process can be implemented naturally in $O(N^2)$ time. We can maintain an index i of the next clip which we'll try to omit from S, scan through all N clips each time to find the optimal clip j to include in S, and then move i forwards to the next earliest clip which doesn't overlap with clip j.

Improving the time complexity to $O(N \log(N))$ is required to earn full marks, and there are several different approaches which accomplish this. One possibility is to create a reduced list of clips in which all clips which are contained entirely within other clips have been omitted. This new list has two useful properties: both the A values and the B values of its clips are sorted, and any optimal clip j which we might want to include in S must still be part of it. We can then maintain an index j into this new list, and move it forwards as necessary when searching for each clip j to include in S, without needing to scan through the entire list each time.

Official Solution (C++)

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

struct Clip {
  int a, b;
  Clip() {}
  Clip(int a, int b) : a(a), b(b) {}
};

bool operator<(const Clip &A, const Clip &B) {
  return make_pair(A.b, -A.a) < make_pair(B.b, -B.a);
}

int N, N2, ans = 0;
Clip C[300000], C2[300000];

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) cin >> C[i].a >> C[i].b;
  sort(C, C + N);  // Sort clips by B, breaking ties by reverse A.
  // Create list of potential clips for S (ones not contained within other clips).
  N2 = 0;
  for (int i = 0; i < N; i++) {
    while (N2 > 0 && C[i].a <= C2[N2 - 1].a) N2--;
    C2[N2++] = C[i];
  }
  for (int i = 0, j = 0; i < N; ) {
    // Try to exclude clip C[i] from S: find the last clip C2[j] which still overlaps with it.
    while (j + 1 < N2 && C2[j + 1].a < C[i].b) j++;
    // Include clip C2[j] in S and skip to the next clip C[i] which doesn't overlap with it.
    ans++;
    while (i < N && C[i].a < C2[j].b) i++;
  }
  cout << ans << endl;
  return 0;
}
```

Problem S3: Down for Maintenance

Let X be the minimum possible number of inactive intervals which the site's maintenance schedule could consist of. Our first order of business will need to be calculating X.

Let's start by sorting all $N + M$ observed times together in a single list (in $O((N + M) \log(N + M))$ time). Then, X is simply the number of occurrences of an inactive time directly preceding an active time. Keep in mind that the last time in the list directly precedes the first time as well.

Now, the answer is `Impossible` if and only if $I < X$.

Assuming that it's possible, the site's status is known for at least each of the $N + M$ observed times. The interesting question is, when can the statuses of other times also be inferred? If $I > X$, then in fact no other statuses can be inferred at all – with even 1 extra interval of leeway, the inactive intervals can always be shuffled around to make any other time either active or inactive.

This leaves the case in which $I = X$. In this situation, each inactive interval is forced to span a certain consecutive interval of observed inactive times, but past that, its starting and ending times may vary up to the previous/next observed active times. The result is that all times between consecutive pairs of observed inactive times are also inactive. Similarly, all times between consecutive pairs of observed active times are also active. However, the statuses of times sandwiched between an observed inactive time and an observed active time remain unknown.

From there, there are several approaches to efficiently processing the queries, in $O(\log(N + M))$ time each. For example, we can binary search for the queried point in the list of $N + M$ observed times. If it's present, then its status is already known. Otherwise, if $I = X$, we can look at the previous and next observed times surrounding it and see whether its status can be inferred from theirs.

Official Solution (C++)

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

struct Data {
  long long t;
  bool up;
  Data() {}
  Data(long long t, bool up) : t(t), up(up) {}
};

bool operator<(const Data &A, const Data &B) {
  return A.t < B.t;
}

struct Range {
  long long a, b;
  string ans;
  Range() {}
  Range(long long a, long long b, string ans) : a(a), b(b), ans(ans) {}
};

bool operator<(const Range &A, const Range &B) {
  return make_pair(A.a, A.b) < make_pair(B.a, B.b);
}
```

```
const long long MAXT = 86400000000LL;
int ND, I, N, M, K, minInactive = 0;
long long t;
Data D[200000];
set<Range> S;

void Ins(long long a, long long b, string ans) {
  if (a <= b) S.insert(Range(a, b, ans));
}

int main() {
  cin >> I >> N;
  for (int i = 0; i < N; i++) {
    cin >> t;
    D[ND++] = Data(t, true);
    Ins(t, t, "Up");
  }
  cin >> M;
  for (int i = 0; i < M; i++) {
    cin >> t;
    D[ND++] = Data(t, false);
    Ins(t, t, "Down");
  }
  sort(D, D + ND);  // Sort data points by time.
  // Compute min number of inactive intervals.
  for (int i = 0; i < ND; i++) {
    if (!D[i].up && D[(i + 1) % ND].up) minInactive++;
  }
  if (I < minInactive) {
    cout << "Impossible" << endl;
    return 0;
  }
  if (I == minInactive) {  // Can any inferences be made?
    // All times between 2 adjacent data points of the same type must share that type.
    for (int i = 0; i < ND; i++) {
      if (D[i].up == D[(i + 1) % ND].up) {
        string ans = D[i].up ? "Up" : "Down";
        if (i == ND - 1) {
          Ins(D[i].t + 1, MAXT - 1, ans);
          Ins(0, D[0].t - 1, ans);
        } else {
          Ins(D[i].t + 1, D[i + 1].t - 1, ans);
        }
      }
    }
  }
  // Process queries.
  cin >> K;
  for (int i = 0; i < K; i++) {
    cin >> t;
    set<Range>::iterator r = S.lower_bound(Range(t + 1, -1, ""));
    if (r == S.begin()) {
      cout << "Unknown" << endl;
    } else {
      r--;
      if (r->a <= t && t <= r->b) {
        cout << r->ans << endl;
      } else {
        cout << "Unknown" << endl;
      }
    }
  }
  return 0;
}
```

Problem S4: Relevant Results

The pages can be modelled as a directed graph, with an edge corresponding to each link. Let's start by breaking the graph down into its strongly connected components (SCCs), which can be done in $O(N + E)$ time using Kosaraju's algorithm (where E is the total number of edges). Let's then define an "important" SCC as one with no incoming edges from other SCCs. We can now observe that, in order to satisfy the i-th question, it's sufficient to increase the relevancy score of a single page in each important SCC to at least Q_i. Within each important SCC, each other page in it must be reachable from the chosen page, and each unimportant SCC must be reachable from at least one important SCC.

These insights suggest an $O(MN + E)$ algorithm in which the M questions are answered independently. For each question i, we can consider each important SCC. Within that SCC, we can compute the cost of raising each of its pages' relevancy scores to at least Q_i, and then take the cheapest one. The answers for all of the important SCCs can then be added together to yield the final answer.

However, the above approach is only efficient enough when $M = 1$. A significantly more complex approach is required for full marks. The idea is that each page p corresponds to a linear function $f_p(x)$ of cost required to increase that page's relevancy score to a given score x. We'd like to construct a similar function g_s for each important SCC s, such that $g_s(x) = \min\{f_p(x)\}$ over all pages p in SCC s. The function g_s corresponds to the lower convex hull of the individual pages' linear functions, and its set of segments can be assembled in linear time after sorting the pages by their C values (which correspond to their lines' slopes).

From there, the answer to the i-th question is equal to the sum of $g_s(Q_i)$ over all important SCCs s. In order to compute these answers efficiently for all questions, we can perform a unified line sweep over all interesting relevancy scores in non-decreasing order, with events for scores which are being asked about as well as scores at which convex hull segments begin. During the line sweep, we'll need to maintain both the current sum of g_s function values, and the sum of their current segments' slopes. These values can then be updated as necessary and used to fill in the answers to questions as they're encountered.

The time complexity of the above algorithm is $O((N + M) \log(N + M) + E)$.

Official Solution (C++)

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

struct Page {
  int r, c;
  Page() {}
  Page(int r, int c) : r(r), c(c) {}
};

bool operator<(const Page &A, const Page &B) {
  return make_pair(-A.r, A.c) < make_pair(-B.r, B.c);
}

struct Event {
  long long r, deltaC, deltaT;
  int q;
  Event() {}
  Event(long long r, int q, long long deltaC, long long deltaT)
    : r(r), q(q), deltaC(deltaC), deltaT(deltaT) {}
};
```

```
bool operator<(const Event &A, const Event &B) {
  return A.r < B.r || A.r == B.r && A.q < B.q;
}

const int MAXN = 400001, MAXM = 400001;

int N, M, NC, NE, R[MAXN], C[MAXN], compInd[MAXN];
bool vis[MAXN], inc[MAXN];
vector<int> con[MAXN], rcon[MAXN], comp[MAXN];
long long ans[MAXM];
stack<int> S;
Page P[MAXN];
Event E[MAXN + MAXM];

void DFS(int x) {
  vis[x] = true;
  for (int i = 0; i < con[x].size(); i++) {
    if (!vis[con[x][i]]) {
      DFS(con[x][i]);
    }
  }
  S.push(x);
}

void RevDFS(int x) {
  vis[x] = false;
  compInd[x] = NC;
  comp[NC].push_back(x);
  for (int i = 0; i < rcon[x].size(); i++) {
    if (vis[rcon[x][i]]) {
      RevDFS(rcon[x][i]);
    }
  }
}

long double Inter(Page A, Page B) {
  return (B.r * (long double)B.c / A.c - A.r) / ((long double)B.c / A.c - 1);
}

void ProcessComp(int c) {
  // Sort component's pages by non-increasing R, breaking ties by non-decreasing C
  int np = 0;
  for (int i = 0; i < comp[c].size(); i++) {
    int j = comp[c][i];
    P[np++] = Page(R[j], C[j]);
  }
  sort(P, P + np);
  // Remove pages which are strictly less useful than others
  int n = 0;
  for (int i = 0; i < np; i++) {
    if (n == 0 || P[i].c < P[n - 1].c) {
      while (n >= 2) {
        if (Inter(P[n - 1], P[i]) > Inter(P[n - 2], P[n - 1])) {
          break;
        }
        n--;
      }
      P[n++] = P[i];
    }
  }
  np = n;
  // Iterate over pages
  E[NE++] = Event(P[0].r, -1, P[0].c, 0);
  for (int i = 1; i < np; i++) {
    // Calculate score at which these 2 pages are equally optimal
    Page A = P[i - 1];
    Page B = P[i];
    long long r = Inter(A, B) + 1 - 1e-12;
    long long t1 = (r - A.r) * A.c;
    long long t2 = (r - B.r) * B.c;
    E[NE++] = Event(r, -1, B.c - A.c, t2 - t1);
  }
}
```

```
int main() {
  // Input
  cin >> N;
  for (int i = 1; i <= N; i++) {
    int K;
    cin >> R[i] >> C[i] >> K;
    while (K--) {
      int L;
      cin >> L;
      con[i].push_back(L);
      rcon[L].push_back(i);
    }
  }
  cin >> M;
  for (int i = 0; i < M; i++) {
    int q;
    cin >> q;
    E[NE++] = Event(q, i, 0, 0);
  }
  // Divide graph into SCCs
  for (int i = 1; i <= N; i++) {
    if (!vis[i]) {
      DFS(i);
    }
  }
  while (!S.empty()) {
    if (vis[S.top()]) {
      RevDFS(S.top());
      NC++;
    }
    S.pop();
  }
  // Find and process all SCCs with no incoming edges
  for (int i = 1; i <= N; i++) {
    int a = compInd[i];
    for (int j = 0; j < con[i].size(); j++) {
      int b = compInd[con[i][j]];
      if (a != b) {
        inc[b] = true;
      }
    }
  }
  for (int i = 0; i < NC; i++) {
    if (!inc[i]) {
      ProcessComp(i);
    }
  }
  // Sweep over events in increasing order of score
  long long tot = 0, c = 0;
  sort(E, E + NE);
  for (int i = 0; i < NE; i++) {
    if (i > 0) {
      tot += c * (E[i].r - E[i-1].r);
    }
    if (E[i].q >= 0) {
      ans[E[i].q] = tot;
    } else {
      c += E[i].deltaC;
      tot += E[i].deltaT;
    }
  }
  // Output
  for (int i = 0; i < M; i++) {
    cout << ans[i] << endl;
  }
  return 0;
}
```