WOBURN CHALLENGE

2016-17 Online Round 4

Solutions

Problem J1: Anyone Can Be Anything

Let C_i be the number of animals with a preferred job category of i. We can tally up the values of C_1 and C_2 in $O(N)$ time by iterating over each of the N animals and adding onto the correct counts. The maximum number of animals who prefer job category 1 and can be assigned to it is then $\min(O_1, C_1)$. Similarly, the maximum number of animals who prefer job category 2 and can be assigned to it is $\min(O_2, C_2)$. Note that these two quantities don't conflict with one another, and that any remaining animals who can't be assigned to their preferred job category can simply be assigned to the other one, without worrying about taking away any other animal's preferred spot. As such, the answer is $\min(O_1, C_1) + \min(O_2, C_2)$.

Official Solution (C++)

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

int N, O[3], p, cnt[3] = {0};

int main() {
  cin >> N >> O[1] >> O[2];
  while (N--) {
    cin >> p;
    cnt[p]++;
  }
  cout << min(O[1], cnt[1]) + min(O[2], cnt[2]) << endl;
  return 0;
}
```

Problem J2: You're Dead!

Due to the constraint that the temperatures of challenges chosen on any given day may not differ by more than L°, it makes sense to try to group challenges with similar temperatures together into the same days. As such, let's start by sorting the challenges' temperatures in non-decreasing order (which can be done in $O(N \log N)$ time, or in $O(N)$ time if we take advantage of the small range of possible temperatures).

Then, we can iterate over the challenges in non-decreasing order of temperature, grouping them together into as few days as possible. As we go, if the next challenge can be validly grouped into the same day as the previous challenge, then it's optimal to greedily do so – it can't help to save it for a later day. With this in mind, we can keep track of the number of days used so far, the number of challenges already allocated to the current day, and the temperature of the current day's first challenge (which dictates the maximum allowable temperature of the day's final challenge). This information can then be used to determine whether the next challenge can be added onto the current day, or if it must instead be the first challenge on the following day. This greedy process can be implemented in $O(N)$ time.

Official Solution (C++)

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

int N, S, L, T[1000], strt = 0, ans = 0;

int main() {
  cin >> N >> S >> L;
  for (int i = 0; i < N; i++) cin >> T[i];
  sort(T, T + N);
  for (int i = 0; i < N; i++) {
    // Session i must be the final session on the current day?
    if ((i == N - 1) || (i - strt + 1 == S) || (T[i + 1] > T[strt] + L)) {
      ans++;
      strt = i + 1;
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem J3/S1: Parking Duty

If no meters were to be skipped, then Judy's day would consist of $N - 1$ "legs", with the i-th leg consisting of travelling from coordinates (X_i, Y_i) to (X_{i+1}, Y_{i+1}) in $T_{i+1} - T_i$ seconds. The minimum top speed required to complete such a leg is simply the Euclidean distance of the leg divided by the time allowed for it. As such, the minimum top speed required for the entire day would be equal to the maximum of the N - 1 legs' required top speeds.

In reality, the day will only have $N - 2$ legs, as one meter will be skipped. If some meter i is chosen to be skipped, aside from the first or last (such that $1 < i < N$), then legs i and $i + 1$ will be removed and replaced by a new leg that travels directly from meter i to $i + 2$ (whose required top speed can be computed in the same way). If the first or last meter is skipped, then the first or last leg will instead be removed and not replaced with any new one.

At any rate, it can be observed that at most 2 of the original $N - 1$ legs can be removed, with the others staying as they are. Furthermore, if the original leg with the largest required top speed is the i-th one (in the case of ties, i can be any leg with the maximum required top speed), then in order to even have a chance of reducing the answer, the i-th leg must be one of the ones removed. Therefore, once we determine the index i (which can be done in $O(N)$ time), we only need to consider skipping either meter i or $i + 1$, and for each of those two options, we can afford to simulate the entire day with the given meter skipped in $O(N)$ time, once again calculating the required top speeds of all $N - 2$ legs present and taking the largest of them. Finally, the answer will be the smaller of the two required top speeds yielded by the two possibly optimal options.

Official Solution (C++)

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

int N, T[200000], X[200000], Y[200000];

// Min velocity to drive from meter i to meter j in time.
long double velocity(int i, int j) {
  long double d = sqrt(pow(X[i] - X[j], 2.0) + pow(Y[i] - Y[j], 2.0));
  return d / (T[j] - T[i]);
}

// Min velocity for the whole day with the given meter being skipped.
long double simulate(int skip) {
  long double maxV = 0;
  for (int i = 0; i < N - 1; i++) {
    if (i == skip - 1) {
      if (i + 2 < N) {
        maxV = max(maxV, velocity(i, i + 2));
      }
    } else if (i != skip) {
      maxV = max(maxV, velocity(i, i + 1));
    }
  }
  return maxV;
}

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> T[i] >> X[i] >> Y[i];
  }
  // Find a leg with the largest required velocity.
  long double maxV = 0;
  int maxI = 0;
  for (int i = 0; i < N - 1; i++) {
    long double v = velocity(i, i + 1);
    if (v > maxV) {
      maxV = v;
      maxI = i;
    }
  }
  // Consider skipping either of the meters adjacent to that leg.
  cout.precision(9);
  cout << fixed << min(simulate(maxI), simulate(maxI + 1)) << endl;
  return 0;
}
```

Problem J4/S2: Pawpsicles

Consider an implicit graph with 5N nodes, with each node representing a possible state which Nick can be in. Each node can be described by two values – his current location i ($1 \le i \le N$), and the next location type t which he needs to visit ($1 \le t \le 5$, with $t = 5$ representing that he's already completed all 4 steps of his Pawpsicle operation). His initial state is $\{i = 1, t = 1\}$, and he wants to reach a state $\{i, t = 5\}$ (for any i) as quickly as possible. By travelling along roads and visiting the correct types of locations, he can move between these states in certain amounts of time. These transitions between states correspond to weighted, directed edges in the implicit graph.

At this point, we've reduced the problem to finding the shortest path on a graph, a well-known task which can be accomplished using Dijkstra's algorithm in O($E + V \log V$) time, where V and E are the number of nodes and edges in the implicit graph, respectively. In this case, $V = 5N$, and E is in $O(N + M)$. What remains is implementing Dijkstra's, and handling the implicit graph's edges properly. For example, the j-th road can be used to go from state $\{i = A_j, t\}$ to $\{B_j, t\}$ for any value of t, and for each node j such that $T_j > 0$, there exists a weight-0 edge from state $\{i = j, t = T_j\}$ to $\{i = j, t = T_j + 1\}$. We want to stop as soon as we reach any node with $t = 5$.

Official Solution (C++)

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

const int MAXN = 100001;
const int INF = 0x3f3f3f3f;

struct state {
  int i, t, d;
  state(int i, int t, int d) : i(i), t(t), d(d) {}

  bool operator<(const state &B) const {
    return d > B.d;
  }
};

int N, M, T[MAXN];
vector< pair<int, int> > adj[MAXN];
int dist[MAXN][5];

int main() {
  memset(dist, INF, sizeof dist);
  cin >> N >> M;
  for (int i = 1; i <= N; i++) {
    cin >> T[i];
  }
  while (M--) {
    int a, b, c;
    cin >> a >> b >> c;
    adj[a].push_back(make_pair(b, c));
    adj[b].push_back(make_pair(a, c));
  }
  priority_queue<state> q;
  q.push(state(1, 1, 0));
  dist[1][1] = 0;
  while (!q.empty()) {
    state s = q.top();
    q.pop();
    if (s.d != dist[s.i][s.t]) {
      continue;
    }
    if (s.t == T[s.i]) {  // Next required type encountered?
      s.t++;
    }
    if (s.t == 5) {  // Done?
      cout << s.d << endl;
      return 0;
    }
    for (int i = 0; i < (int)adj[s.i].size(); i++) {
      state s2 = state(adj[s.i][i].first, s.t, s.d + adj[s.i][i].second);
      if (s2.d < dist[s2.i][s2.t]) {
        q.push(s2);
        dist[s2.i][s2.t] = s2.d;
      }
    }
  }
  cout << -1 << endl;
  return 0;
}
```

Problem S3: Night Howlers

Let's consider binary searching for the answer – that is, the minimum valid howling radius *R*. All values of *R* smaller than the answer are invalid, while all values of *R* larger than or equal to the answer are valid, meaning that the binary search will work out if we can just find a way to determine the validity of a given value of *R*.

Given a fixed value of *R*, we need to determine the minimum number of initial wolves which must be made to howl in order for all *N* wolves to join in, and then compare this count to *K*. Rather than attempting to deal with chain reactions of wolves causing each other to howl, an insight can be made to simplify the problem greatly: Each wolf *i* must be made to howl if and only if there exists no wolf *j* which would directly cause wolf *i* to howl (in other words, there exists no *j* such that $A_j < A_i$, $P_j \geq P_i - R$, and $P_j \leq P_i + R$). If there's indeed no such wolf *j*, then no matter what set of other wolves are made to howl, wolf *i* will surely never start howling until Judy and Nick make him howl themselves. If there is such a wolf *j*, then Judy and Nick will need to ensure that wolf *j* will end up howling one way or another, at which point wolf *i* will also join in.

What remains is determining whether or not any valid wolf *j* exists for each wolf *i*. The simplest way to do so is to sort the wolves in increasing order of rank (which can be done once in O(*N* log *N*) time, before the start of the binary search), and then iterate over them in this order while maintaining an ordered set of the positions of wolves considered so far. For each wolf *i*, we only need to check whether any position in the current set is in the interval $[P_i - R, P_i + R]$ (as any such wolf would be guaranteed to also have a smaller rank), before proceeding to insert P_i itself into the set. If the set is implemented as a balanced binary search tree, this query and update operations can each be performed in *O*(log *N*) time. As such, the overall time complexity of this algorithm is O(*N* log(*N*) log(*M*)), where *M* is the maximum of the *P* values (which dictates the number of iterations required for the binary search).

Official Solution (C++)

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

int N, K;
pair<int, int> W[200000];

int main() {
  cin >> N >> K;
  for (int i = 0; i < N; i++) {
    cin >> W[i].second >> W[i].first;
  }
  sort(W, W + N);
  int low = 1, high = 1e9;
  while (low < high) {
    int R = (low + high) / 2;
    // Test validity of middle value of R.
    int cnt = 0;
    set<int> pos;
    for (int i = 0; i < N; i++) {
      // Check if a superior wolf is within distance R.
      int p = W[i].second;
      set<int>::iterator it = pos.lower_bound(p);
      int req = (it == pos.end() || *it > p + R) ? 1 : 0;
      if (it != pos.begin()) {
        if (*--it >= p - R) {
          req = 0;
        }
      }
      cnt += req;
      if (cnt > K) {
        break;
      }
      pos.insert(p);
    }
    if (cnt > K) {
      low = R + 1;
    } else {
      high = R;
    }
  }
  cout << low << endl;
  return 0;
}
```

Problem S4: Hopps and Wilde on the Case

The network of locations and roads forms a tree, rooted at its 1st node. Anytime Judy or Nick travel down an edge from a node to one of its children, they'll need to travel back up it later, so for convenience we can only count the downwards edge traversals and then multiply the answer by 2 at the end.

This problem can be approached with dynamic programming, split into two portions. First, let *DP1*[*i*] be the minimum amount of time required for a single cop to visit all of the required nodes in node *i*'s subtree (starting and ending at *i*), with the other cop never entering said subtree. For convenience, let *DP1*[*i*] = –1 if *i*'s subtree contains no nodes with clues. We can initialize *DP1*[*i*] to 0, and for each child *c* of *i* such that *DP1*[*c*] > –1, we can add *DP1*[*c*] + 1 onto *DP1*[*i*], as the cop will need to traverse the edge down from *i* to *c* and then proceed to visit all of *c*'s subtree's required nodes. Finally, we should remember to set *DP1*[*i*] back to –1 if C_i = 0 and *DP1*[*c*] was –1 for each child *c*.

Next, let *DP2*[*i*][*j*] be the minimum amount of time which must be spent by the first cop within *i*'s subtree (starting and ending at *i*), given that the second cop is also contributing by spending *j* minutes within *i*'s subtree (also starting and ending at *i*), and such that all required nodes in node *i*'s subtree end up getting visited by at least one of the cops. Note that the answer we're looking for is the minimum value of max(*j*, *DP2*[1][*j*]) across all possible values of *j* between 0 and *N*.

The trickiest part of the algorithm is actually computing the values of DP2[*i*][0..*N*]. We can start by initializing *DP2*[*i*][0..*N*] = 0, and then consider each of i's children c in turn such that *DP1*[*c*] > –1. We'll want to compute a temporary updated copy of the DP array *DP2'*[*i*][0..*N*] (whose values should first be initialized to infinity), and then copy *DP2'*[*i*][0..*N*] back into *DP2*[*i*][0..*N*]. Now, there are 3 types of options for this child - either the first cop should go down and handle its subtree alone, or the second cop should do so, or both cops should visit it. The first option corresponds to a recurrence of the form *DP2'*[*i*][*j*] = min(*DP2'*[*i*][*j*], *DP2*[*i*][*j*] + *DP1*[*c*] + 1), for all *O*(*N*) values of *j*. The second option indicates *DP2'*[*i*][*j* + *DP1*[*c*] + 1] = min(*DP2'*[*i*][*j* + *DP1*[*c*] + 1], *DP2*[*i*][*j*]). The third option indicates *DP2'*[*i*][*j* + *k* + 1] = min(*DP2'*[*i*][*j* + *k* + 1], *DP2*[*i*][*j*] + *DP2*[*c*][*k*] + 1 (note that, for each *j*, we need to also consider all *O*(*N*) possible values of *k*, corresponding to how the cops may split their time in *c*'s subtree).

The values of *DP1*[*i*] and DP2[*i*][0..*N*] all depend only on the DP values of *i*'s children, meaning that all of the DP values may be computed recursively starting from node 1. For each node *i* visited during the recursion, updating the value of *DP1*[*i*] based on each of *i*'s children takes *O*(1) time, and updating the values of *DP2*[*i*][0..*N*] based on each of *i*'s children takes $O(N^2)$ time. Each node is the child of at most one other node, meaning that the overall time complexity of this algorithm is $O(N^3)$.

Official Solution (C++)

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

const int MAXN = 201;
const int INF = 0x3f3f3f3f;

int N, C[MAXN];
vector<int> adj[MAXN];

/* DP1[i] = min time for cop 1 to complete i's subtree alone (or -1 if it
 *          contains no clues).
 * DP2[i][j] = min time for cop 1 to complete i's subtree, given that cop 2 is
 *             spending j time in it.
 */
int DP1[MAXN];
int DP2[MAXN][MAXN];
int tmp[MAXN];
```

```
void solve(int i, int p) {
  DP1[i] = C[i] ? 0 : -1;
  for (int j = 0; j < (int)adj[i].size(); j++) {  // Consider each neighbour.
    int c = adj[i][j];
    if (c == p) {
      continue;  // Don't recurse back up to the parent.
    }
    solve(c, i);
    if (DP1[c] < 0) {  // No clues in c's subtree?
      continue;
    }
    DP1[i] = max(DP1[i], 0) + DP1[c] + 1;  // Update DP1.
    memset(tmp, INF, sizeof tmp);
    for (int a = 0; a < N; a++) {  // Update DP2.
      if (DP2[i][a] < INF) {
        tmp[a] = min(tmp[a], DP2[i][a] + DP1[c] + 1);               // Cop 1.
        tmp[a + DP1[c] + 1] = min(tmp[a + DP1[c] + 1], DP2[i][a]);  // Cop 2.
        for (int b = 0; b < N - a; b++) {                           // Both cops.
          tmp[a + b + 1] = min(tmp[a + b + 1], DP2[i][a] + DP2[c][b] + 1);
        }
      }
    }
    memcpy(DP2[i], tmp, sizeof tmp);
  }
}

int main() {
  cin >> N;
  for (int i = 1; i <= N; i++) {
    cin >> C[i];
  }
  for (int i = 0; i < N - 1; i++) {
    int a, b;
    cin >> a >> b;
    adj[a].push_back(b);
    adj[b].push_back(a);
  }
  solve(1, -1);  // Recurse from node 1 to compute DP values.
  int ans = INF;
  for (int i = 0; i < N; i++) {  // Consider all possible time allocations.
    ans = min(ans, max(i, DP2[1][i]));
  }
  cout << (ans * 2) << endl;
  return 0;
}
```