WOBURN CHALLENGE

2018-19 On-Site Finals

Solutions

Problem J1: Conditional Contracts

Let's ignore the requirement that the two film widths must be distinct (as using the same film width twice cannot result in employing more actors anyway). Then, each chosen film width might as well be equal to at least one actor's required film width (as it cannot be better to choose a film width required by no actors). This gives us at most N distinct film widths worth considering (W_1, …, W_N). We can consider all N^2 pairs of these widths, count how many actors may be employed for each such pair (by iterating over all N actors and counting ones whose requirements have been satisfied), and output the largest of these employment counts.

The time complexity of the above approach is $O(N^3)$, which is fast enough given the constraint that $N \le 100$. It's also possible to solve this problem more efficiently (in as little as $O(N)$ time, by using a hash map to tally up the number of actors requiring each film width and then adding together the largest two such actor counts).

Official Solution (C++)

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

int main() {
  int N, W[100], ans = 0;
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> W[i];
  }
  // Consider all possible pairs of widths.
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      // Count number of employable actors.
      int c = 0;
      for (int k = 0; k < N; k++) {
        if (W[k] == W[i] || W[k] == W[j]) {
          c++;
        }
      }
      ans = max(ans, c);
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem J2: Script Doctor

For each occurrence of "`bull`", removing the "`b`" may not be sufficient – for example, the string "`bbull`" would be reduced to "`bull`", which still contains "`bull`". Similarly, removing an "`l`" may not be sufficient.

However, removing the "`u`" is always sufficient for wiping out the "`bull`" occurrence. Therefore, we'll want to replace each occurrence of "`bull`" with "`bll`".

Official Solution (C++)

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

int main() {
  string S;
  cin >> S;
  // Output while stripping out the "u" from each "bull".
  for (int i = 0; i < S.size(); i++) {
    if (i > 0 && S.substr(i - 1, 4) == "bull") {
      continue;
    }
    cout << S[i];
  }
  cout << endl;
  return 0;
}
```

Problem J3/S1: Behind the Scenes

Any piece of filmmaking equipment should only be moved when necessary, immediately before a shot will be filmed on its current stage (the total cost cannot be reduced by instead moving it earlier). Furthermore, whenever multiple pieces of equipment are to be moved away from a given stage, they should all be moved together to a single other stage. Therefore, when simulating the sequence of N shots in order, right before each shot i, we should move all equipment currently on stage S_i to one of the other two stages.

The only remaining question for each shot i is: Which of the other two stages should stage S_i's equipment be moved to? Let the two possible destination stages be a and b, and let $Next[i][s]$ be the earliest shot j such that $j \geq i$ and $S_j = s$ (or ∞ if there's no such shot). $Next[i][s]$ indicates the next time at which equipment would need to be moved again if moved to stage s right before shot i, which is the only relevant consequence of the choice of destination stage. Therefore, the equipment should be moved to stage a if $Next[i][a] <$ Next$[i][b]$, or to stage b otherwise.

A simple implementation of the above approach, in which each required *Next* value is computed by iterating over the remaining shots, has a time complexity of $O(N^2)$ and is fast enough. An $O(N)$ solution is also possible, by first iterating over the shots in reverse to pre-compute all of the *Next* values.

Official Solution (C++)

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

int main() {
  int E[3], N, S[1000], ans = 0;
  cin >> E[0] >> E[1] >> E[2] >> N;
  for (int i = 0; i < N; i++) {
    cin >> S[i];
    S[i]--;
  }
  for (int i = 0; i < N; i++) {  // Simulate.
    // Find next upcoming section different than S[i] (if any).
    int s = S[i];
    int nxt = (s + 1) % 3;
    for (int j = i + 1; j < N; j++) {
      if (S[j] != s) {
        nxt = S[j];
        break;
      }
    }
    // Greedily move equipment from S[i] to whichever section isn't next.
    ans += E[s];
    E[3 - s - nxt] += E[s];
    E[s] = 0;
  }
  cout << ans << endl;
  return 0;
}
```

Problem J4/S2: Top Billing

Consider the layout for a grid with $R = C = 5$ in figure 1. Tom Cows will reach the ending cell in 5 minutes (for example by following the sequence of moves →→→→↑). On the other hand, Monkey Freeman will move as follows:

- Up (the cells above and to the right both have equal Manhattan distances of 4 to the ending cell, so he'll choose the former)
- Up (similarly choosing it over moving down)

- Right (similarly choosing it over moving down)
- Up (similarly choosing it over moving left)
- Right (similarly choosing it over moving down) etc…

```
.....
...#.
..#..
.#..E
S....
```

Fig 1.

```
.E     ...     ....
S.     .#E     ..#.
       S..     .#.E
               S...
```

Fig 2.

This pattern will repeat until Monkey reaches the grid's top-right cell, at which point he'll move straight downwards to the ending cell. For this particular grid, Monkey will require 11 minutes, which is slower than Tom by the minimum required amount of $2C - 4 = 6$ minutes. More generally, this pattern can be extended to any square grid, as in the examples in figure 2.

For any such grid, Tom will need C minutes while Monkey will need $3C - 4$ minutes, yielding the minimum required difference of $2C - 4$. If we need to fill a non-square grid (such that $R > C$), then $R - C$ additional rows of vacant cells may simply be added below this pattern without affecting anything.

Official Solution (C++)

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

int main() {
  int R, C;
  cin >> R >> C;
  for (int r = 1; r <= R; r++) {
    for (int c = 1; c <= C; c++) {
      if (r == C) {
        cout << (c == 1 ? 'S' : '.');
      } else if (r == C - 1) {
        cout << (c == C ? 'E' : c == 2 ? '#' : '.');
      } else if (2 <= r && r <= C - 2) {
        cout << (c == C - r + 1 ? '#' : '.');
      } else {
        cout << '.';
      }
    }
    cout << endl;
  }
  return 0;
}
```

Problem J5/S3: Screen Time

We'll define a type-i scene as one excluding actor i ($1 \le i \le 3$). Let T_i be the total number of type-i scenes, and C_i be the number of type-i scenes included in the chosen subset. A subset is valid if and only if the following hold:

- $0 \le C_i \le T_i$, for each i,
- $C_1 < C_2$ (equivalent to $S_1 > S_2$), and
- $C_1 < C_3$ (equivalent to $S_1 > S_3$).

The number of different subsets with a given set of C values is equal to the product $\prod_{i=1}^{3} \binom{T_i}{C_i}$. Before going further, let's consider how we can efficiently evaluate choose values in general (modulo a prime P such as 1,000,000,007). We'll first let ModInv(v, p) be the *modular inverse*[1] of v for prime modulus p, which is equal to $v^{(p-2)}$ mod p.

ModInv(v, p) may be evaluated in $O(\log p)$ time using either *exponentiation by squaring*[2] or the *extended Euclidean algorithm*[3]. Then, we have $\binom{n}{k} \bmod P = \frac{n!}{k!(n-k)!} \bmod P = \left[(n! \bmod P) \times \text{ModInv}(k!, P) \times \text{ModInv}\left((n-k)!, P\right)\right] \bmod P$.

[1] https://en.wikipedia.org/wiki/Modular_multiplicative_inverse
[2] https://en.wikipedia.org/wiki/Exponentiation_by_squaring
[3] https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

If we precompute $x!$ mod P and ModInv($x!$, P) for each x between 0 and N, inclusive, then we'll be able to evaluate any necessary choose value in constant time. Now, let $Ways[i][j] = \left[\sum_{k=j}^{T_i} \binom{T_i}{k}\right]$ mod P – in other words, the number of size-j-or-greater subsets of type-i scenes.

Note that $Ways[i][j] = \left[Ways[i][j+1] + \binom{T_i}{j}\right]$ mod P, meaning that we can precompute $Ways[i][j]$ for all possible pairs (i, j) in $O(N)$ by iterating downwards through the j values.

Let's consider fixing the number of type-1 scenes in the chosen subset. For a given C_1, there are $Ways[2][C_1 + 1]$ subsets of type-2 scenes such that $C_2 > C_1$ is satisfied, and similarly $Ways[3][C_1 + 1]$ subsets of type-3 scenes. Therefore, the total number of valid subsets which can be chosen with a given C_1 is $\left[\binom{T_1}{C_1} \times Ways[2][C_1 + 1] \times Ways[3][C_1 + 1]\right]$ mod P, which can be added up over all $O(N)$ possible choices of C_1 to yield the final answer.

Official Solution (C++)

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

const int MAXN = 200002, MOD = 1000000007;

long long fact[MAXN], inv[MAXN], combM[2][MAXN];

int Pow(long long a, int b) {  // Returns a^b.
  int v = 1;
  while (b) {
    if (b & 1) {
      v = v * a % MOD;
    }
    a = a * a % MOD;
    b >>= 1;
  }
  return v;
}

long long Ch(int n, int k) {  // Returns n choose k.
  return fact[n] * inv[k] % MOD * inv[n - k] % MOD;
}

int main() {
  int N, A, B, C[3] = {0};
  cin >> N;
  while (N--) {
    cin >> A >> B;
    C[A + B - 3]++;
  }
  // Precompute factorials and their modular inverses.
  for (int i = 0; i < MAXN; i++) {
    fact[i] = i ? (fact[i - 1] * i % MOD) : 1;
    inv[i] = Pow(fact[i], MOD - 2);
  }
  // Precompute ways to choose at least j scenes of type i for each (i, j).
  for (int i = 0; i < 2; i++) {
    combM[i][C[i]] = 1;
    for (int j = C[i] - 1; j >= 0; j--) {
      combM[i][j] = (combM[i][j + 1] + Ch(C[i], j)) % MOD;
    }
  }
  // Consider all possible counts of type-3 scenes.
  int ans = 0;
  for (int i = 0; i <= C[2]; i++) {
    ans = (ans + Ch(C[2], i) * combM[0][i + 1] % MOD * combM[1][i + 1]) % MOD;
  }
  cout << ans << endl;
  return 0;
}
```

Problem S4: Posters

Let's think of the network of cities and highways as a tree rooted at node A. We'll then approach the problem with dynamic programming on this tree. Let $DP[i][a][b][x]$ be the minimum number of distinct nodes which must be visited by Bo Vine in node i's subtree such that:

- All movie theatres in node i's subtree will be visited by somebody
- If $a = 1$, the Head Monkey will visit node i, and otherwise if $a = 0$, she will not
- If $b = 1$, Bo Vine will visit node i, and otherwise if $b = 0$, he will not
- The Head Monkey will visit x distinct nodes in node i's subtree

Starting from some initial node, visiting k distinct nodes (including the initial one), and then returning to that initial node requires $2×(k – 1)$ hours. Therefore, for a given b value representing whether or not Bo Vine will visit the root (node A), and a given x value representing how many distinct nodes the Head Monkey will visit in total, $2×(\max(x, DP[A][1][b][x]) – 1)$ is the total number of hours that will be required. If these DP values could be computed, then we could consider all $O(N)$ possible pairs of (b, x) and take the smallest of their corresponding times.

What remains is computing the DP values, which we'll do recursively starting from the root. The general approach will be a standard one – for each child c of a given node i, we'll consider all possible triples of (a_1, b_1, x_1) for node i and all possible triples of (a_2, b_2, x_2) for node c, and update node i's DP values based on sums of the form $DP[i][a_1][b_1][x_1] + DP[c][a_2][b_2][x_2]$. The details of interest concern exactly which states and transitions are actually valid, and are as follows:

- If C_i = "1" and $a = b = 0$, then the state is invalid (node i's movie theatre must be visited)
- If $i = B$ and $b = 0$, then the state is invalid (Bo Vine must visit node B)
- If $a_1 = 0$ and $a_2 = 1$, then the transition is invalid (the Head Monkey can't reach node c without passing downwards through node i)
- If B is within c's subtree, $b_1 = 1$, and $b_2 = 0$, then the transition is invalid (Bo Vine can't reach node i without passing upwards through node c)
- If B is not within c's subtree, $b_1 = 0$, and $b_2 = 1$, then the transition is invalid (Bo Vine can't reach node c without passing downwards through node i)

To implement the above checks, we can pre-compute whether or not B is within each node i's subtree in a total of $O(N)$ time. Given that, the overall time complexity of this dynamic programming approach comes out to $O(N^3)$.

Official Solution (C++)

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

const int MAXN = 303;

int N, A, B;
vector<int> adj[MAXN];
string C;
bool hasB[MAXN];

// DP[i][a][b][x] = min. nodes visited by B in i's subtree such that:
//  - A will visit node i if a=1
//  - B will visit node i if b=1
//  - A will visit x nodes in i's subtree
int DP[MAXN][2][2][MAXN];
```

```
bool FindB(int i, int p) {
  if (i == B) {
    return hasB[i] = 1;  // Found B.
  }
  // Iterate over children?
  for (int j = 0; j < adj[i].size(); j++) {
    int c = adj[i][j];
    if (c == p) {
      continue;
    }
    // If child's subtree contains B, this subtree also contains B.
    hasB[i] |= FindB(c, i);
  }
  return hasB[i];
}

void DFS(int i, int p, int a, int b) {
  if (DP[i][a][b][0] >= 0) {
    return;  // Already computed.
  }
  memset(DP[i][a][b], 60, sizeof(DP[i][a][b]));  // Init to infinity.
  if ((C[i]=='1' && !a && !b) || (i==B && !b)) {
    return;  // Invalid state.
  }
  // Iterate over children.
  int tmp[MAXN];
  DP[i][a][b][a] = b;
  for (int j = 0; j < adj[i].size(); j++) {
    int c = adj[i][j];
    if (c == p) {
      continue;
    }
    // Consider all states for child.
    memset(tmp, 60, sizeof(tmp));
    for (int a2 = 0; a2 < 2; a2++) {
      for (int b2 = 0; b2 < 2; b2++) {
        // If A didn't visit here, can't visit child.
        if (!a && a2) {
          continue;
        }
        // This subtree contains B?
        if (hasB[c]) {
          // If B visited here, must also visit child.
          if (b && !b2) {
            continue;
          }
        } else {
          // If B didn't visit here, can't visit child.
          if (!b && b2) {
            continue;
          }
        }
        // Compute child's DP values.
        DFS(c, i, a2, b2);
        // Incorporate into ongoing DP.
        for (int x = 0; x <= N; x++) {
          for (int x2 = 0; x2 <= x; x2++) {
            tmp[x] = min(tmp[x], DP[i][a][b][x - x2] + DP[c][a2][b2][x2]);
          }
        }
      }
    }
    memcpy(DP[i][a][b], tmp, sizeof(tmp));
  }
}
```

```
int main() {
  cin >> N >> A >> B;
  A--, B--;
  cin >> C;
  for (int i = 0; i < N - 1; i++) {
    int x, y;
    cin >> x >> y;
    x--, y--;
    adj[x].push_back(y);
    adj[y].push_back(x);
  }
  // Record all subtrees which contain B.
  FindB(A, -1);
  // Consider all possible root states.
  memset(DP, -1, sizeof(DP));
  int ans = 1e9;
  for (int b = 0; b < 2; b++) {
    DFS(A, -1, 1, b);
    for (int x = 1; x <= N; x++) {
      ans = min(ans, max(x, DP[A][1][b][x]));
    }
  }
  cout << 2 * (ans - 1) << endl;
  return 0;
}
```

Problem S5: Opening Weekend

We'll represent the network of cities and roads as a tree, rooted at an arbitrary node (node 1). Our overall approach will be to consider vehicle heights in decreasing order, in a line-sweep-like manner. Given the current vehicle height, we'll let $Next[i]$ be the node to which vehicles of that height will drive to next from node i (equal to i itself if vehicles would remain there). Initially, for arbitrarily large vehicle heights, $Next[i] = i$ for each i (as no roads can be used). As the vehicle height decreases, for each i, $Next[i]$ may increase as roads leading to higher-numbered cities become usable. For each node i, if we sort its incident roads in non-increasing order of tunnel height ($O(N \log N)$ time), we can precompute the sequence of vehicle heights at which $Next[i]$ will increase (and what values it will increase to), each of which represents an event of interest in the overall line sweep. On top of these $O(N)$ events, we'll have K more events, one per moviegoer.

Assuming we maintain all nodes' *Next* values as we proceed through the line sweep, we'll have the appropriate values when processing each moviegoer i, at which point we'll want to determine the final node they'll arrive at if repeatedly moving from their current node c (initially C_i) to $Next[c]$ (until $Next[c] = c$). However, each moviegoer could perform $O(N)$ such moves, meaning that naive simulation yields a time complexity of $O(NK)$ and is too slow for full marks.

In order to optimize this approach, we'll begin by applying *heavy-light decomposition*[4] to the tree. We'll still maintain the individual *Next* values of nodes not on heavy paths. However, for each heavy path, we'll instead maintain a set of intervals of contiguous subsequences of nodes along the path which all lead to a single node. For example, consider the following heavy path (with dashes representing currently-usable edges and underscores representing unusable ones): 1–3–5–2_7–4–6. It can be reduced into intervals: [1..2] → 5, [7..4] → 7, [6..6] → 6.

Each time a line sweep event causes a node's *Next* value to change, we can update the above representation as necessary in $O(\log N)$ time (with care taken when updating a heavy path's interval set to merge/split intervals appropriately). And each time we need to simulate a moviegoer's route, they'll only move along $O(\log N)$ heavy paths (each of which may be processed in $O(\log N)$ time) and $O(\log N)$ other nodes (each requiring $O(1)$ time), for a total of $O(\log^2 N)$. This brings us to an overall time complexity of $O(K \log^2 N + (N + K)\times\log(N + K))$.

[4] https://en.wikipedia.org/wiki/Heavy_path_decomposition

Official Solution (C++)

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

#define PR pair<int,int>
#define mp make_pair
#define x first
#define y second

const int MOD = 1000000007, LIM = 400005, LIM2 = 800005;

vector<PR> con[LIM];
vector<int> ch[LIM];
pair<PR,PR> ev[LIM2];
int sz[LIM], ans[LIM];
int ii, in[LIM], rin[LIM], out[LIM], top[LIM];
set<pair<PR,PR> > S;

// Set a's next node to b
void SetNext(int a, int b) {
  // Get containing interval.
  a = in[a], b = in[b];
  set<pair<PR,PR> >::iterator I = S.lower_bound(mp(mp(a, 1e9), mp(0, 0)));
  I--;
  int x = I->x.x, y = I->x.y;
  PR e = I->y;
  S.erase(I);
  // b within same heavy path as a?
  if (top[rin[a]] == top[rin[b]]) {
    assert(abs(a - b) == 1);
    I = S.lower_bound(mp(mp(b, 1e9), mp(0, 0)));
    I--;
    int x2 = I->x.x, y2 = I->x.y;
    PR e2 = I->y;
    S.erase(I);
    if (a == e.x) {
      S.insert(mp(mp(min(x, x2), max(y, y2)), e2));
    } else if (a < e.x) {
      assert(a == x);
      S.insert(mp(mp(x2, x), e2));
      if (x < y) {
        S.insert(mp(mp(x + 1, y), e));
      }
    } else {
      assert(a == y);
      S.insert(mp(mp(y, y2), e2));
      if (x < y) {
        S.insert(mp(mp(x, y - 1), e));
      }
    }
    return;
  }
  // b not within same heavy path as a.
  if (a == e.x) {
    S.insert(mp(mp(x, y), mp(e.x, b)));
  } else if (a < e.x) {
    assert(a == x || rin[a - 1] < rin[a]);
    S.insert(mp(mp(x, a), mp(a, b)));
    S.insert(mp(mp(a + 1, y), e));
  } else {
    assert(a == y || rin[a + 1] < rin[a]);
    S.insert(mp(mp(a, y), mp(a, b)));
    S.insert(mp(mp(x, a - 1), e));
  }
}
```

```
// Compute final node when starting from node a
int Query(int a) {
  // Repeatedly follow containing intervals to their next nodes.
  a = in[a];
  while (true) {
    set<pair<PR,PR> >::iterator I = S.lower_bound(mp(mp(a, 1e9), mp(0, 0)));
    I--;
    if (a == I->y.y) {
      break;
    }
    a = I->y.y;
  }
  return rin[a];
}

// Compute subtree sizes and swap largest subtrees to be first.
void RecSz(int i, int p) {
  sz[i] = 1;
  for (int j = 0; j < con[i].size(); j++) {
    int c = con[i][j].y;
    if (c == p) {
      continue;
    }
    ch[i].push_back(c);
    RecSz(c, i);
    sz[i] += sz[c];
    if (sz[c] > sz[ch[i][0]]) {
      swap(ch[i][0], ch[i][ch[i].size() - 1]);
    }
  }
}

// Compute heavy paths.
void RecHLD(int i) {
  in[i] = ii++;
  rin[in[i]] = i;
  for (int j = 0; j < ch[i].size(); j++) {
    int c = ch[i][j];
    top[c] = j ? c : top[i];
    RecHLD(c);
  }
  out[i] = ii;
}

int main() {
  // Input tree.
  int N, K;
  cin >> N >> K;
  for (int i = 0; i < N - 1; i++) {
    int A, B, L;
    cin >> A >> B >> L;
    A--, B--;
    con[A].push_back(mp(L, B));
    con[B].push_back(mp(L, A));
  }
  // Perform heavy-light decomposition on tree.
  RecSz(0, -1);
  RecHLD(0);
  // Generate events for "next" transitions from each node.
  int E = 0;
  for (int i = 0; i < N; i++) {
    sort(con[i].begin(), con[i].end());
    int m = i;
    for (int j = con[i].size() - 1; j >= 0; j--) {
      int k = con[i][j].y;
      if (k > m) {
        m = k;
        ev[E++] = mp(mp(-con[i][j].x, -1), mp(i, k));
      }
    }
    S.insert(mp(mp(i, i), mp(i, i)));
  }
  // Input queries, and generate events for them.
  for (int i = 0; i < K; i++) {
    int C, H;
    cin >> C >> H;
    C--;
    ev[E++] = mp(mp(-H, i), mp(C, 0));
  }
  // Process line sweep events.
  sort(ev, ev + E);
  for (int i = 0; i < E; i++) {
    int k = ev[i].x.y;
    int a = ev[i].y.x;
    int b = ev[i].y.y;
    if (k < 0) {
      SetNext(a, b);
    } else {
      ans[k] = Query(a);
    }
  }
  for (int i = 0; i < K; i++) {
    cout << ans[i] + 1 << endl;
  }
  return 0;
}
```