WOBURN CHALLENGE

2016-17 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: The Elite *N*

There are $M + 1$ points at which we know that we must have Aeroxis active – just before the first battle (essentially "trainer 0"), and for each cloud-type trainer T_i. What remains is determining which Pokemon should be used for the battles in between those points (and after the last of them). For each of those unknown intervals, it's optimal to either use Aeroxis or Brinoble for its entire duration – it doesn't make sense to swap at some point in the middle.

Let's iterate over the M cloud-type trainers while maintain a running total of the time taken. For each such trainer, we can first of all add A onto the total time. Then, let d be the number of non-cloud-type trainers between the current trainer and the previous point at which we must have had Aeroxis active. The time required to beat them using only Aeroxis is simply $A \times d$. On the other hand, the time required to switch to Brinoble, use it to beat those trainers, and then switch back to Aeroxis is $2S + d \times B$. We should greedily use whichever of those two strategies is faster, and add that value onto the total time. Either way, we'll end up with Aeroxis as required.

The remaining interval is the one after the last cloud-type trainer, and the same approach can be applied to it. Let d be the number of non-cloud-type trainers after trainer T_M. The time required to beat them using only Aeroxis is again $A \times d$, while the time required to switch to Brinoble and use it to beat those trainers is $S + d \times B$ (note that we don't need to switch back to Aeroxis after this interval).

The time complexity of this greedy algorithm is $O(M)$.

Official Solution (C++)

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

int N, M, A, B, S, T, p = 0;
long long ans = 0;

int main() {
  cin >> N >> M >> A >> B >> S;
  for (int i = 0; i < M; i++) {
    cin >> T;
    long long d = T - p - 1;
    ans += A + min(d*A, S*2 + d*B);
    p = T;
  }
  long long d = N - p;
  ans += min(d*A, d*B + S);
  cout << ans << endl;
  return 0;
}
```

Problem J2: Pokéwarehouses

The most important thing to observe about this problem is that no more than 2 intermediate warehouses are ever required. To show this, we can consider the following simple algorithm which always successfully completes the shipment given 2 intermediate warehouses: repeatedly find the next item i which must be delivered to the destination warehouse, move each of the items above i to another warehouse (there will always be at least one other warehouse which they can be moved to), and then move i to the destination warehouse.

Given this insight, we've reduced the problem to just determining whether 0 intermediate warehouses are sufficient, or if not, then whether 1 intermediate warehouse is sufficient. Otherwise, the answer must be 2.

Determining whether 0 intermediate warehouses are sufficient is trivial. In this situation, there are no choices regarding how the items may be moved, and the initial order of the items must be $S = [N, N-1, \ldots, 1]$ in order for the shipment to work out.

Determining whether 1 intermediate warehouse is sufficient can be done by simulating the process, as there are similarly very

Official Solution (C++)

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

int N, S[100005], W[100005];

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> S[i];
  }
  int i = 0, j = 0, next = N, ans = 0;
  while (next && ans < 2) {
    if (i < N && S[i] == next) {
      i++;
      next--;
    } else if (j && W[j - 1] == next) {
      j--;
      next--;
    } else if (i < N) {
      ans = 1;
      W[j++] = S[i++];
    } else {
      ans = 2;
    }
  }
  cout << ans << endl;
  return 0;
}
```

few choices to be made regarding how the items should be moved. We can repeatedly apply the following greedy logic: if the next item *i* which must be delivered to the destination warehouse is currently on top of either of the other stacks, then move it to the destination warehouse, otherwise the only valid move is to move the top item from the initial warehouse to the intermediate warehouse (and if the initial warehouse is already empty, then there are no valid moves and the shipment can't be completed successfully).

The time complexity of this simulation is $O(N)$, as an item will be moved at each step, and each item will be moved at most twice in total.

Problem J3/S1: Cutting Edge

Let's assume the grid is arbitrarily large for now. The optimal solution when $K = 0$ involves just 2 trees, one to the right of the top-left cell, and the other below it. When $K = 1$, we can add just 2 more trees to block off the bottom-right cell in the same fashion. For $K = 2$, we'll need to add a diagonal line of 3 additional trees to increase the "width" of one of those barricades from 1 to 2, and so on. As can be seen, the differences between the subsequent answers for all possible values of K (assuming an initial answer of 0) are 2, 2, 3, 3, 4, 4, etc.

So far, this is all independent of the size of the grid. So exactly what effect do the grid's dimensions have? Firstly, the shortest path from the top-left to the bottom-right cell passes through $N + M - 3$ other cells, meaning that if $K \geq N + M - 3$, the answer is –1, even if every cell on such a path contained a tree, they could all be cut down. Otherwise, if we assume that the grid is wider than it is tall ($N \leq M$), then the lengths of our diagonal barricades added at each step will eventually cap out at N, yielding a sequence of answer differences 2, 2, 3, 3, ..., N, N, N.

What remains is computing the sum of the first $K + 1$ terms of this sequence to yield the total answer. This can be done trivially in $O(N + M)$ time, but that's too slow to get full marks, so a closed form expression will be required. We should split the sequence into two portions – the first $2(N - 1)$ terms of the sequence, which are increasing in pairs, and the remaining terms, each of which is N. Computing the sum of the required terms in the second portion is then trivial. The first portion can once more be split into two separate arithmetic sequences 2, 3, ..., N. We can then determine how much of each of those sequences we need (by dividing $(K + 1)$ by 2, rounded up and down), and compute their sums in standard fashion in $O(1)$ time.

Official Solution (C++)

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

long long calc_exp(long long c) {
  return c*(c + 3)/2;
}

long long N, M, K;

int main() {
  cin >> N >> M >> K;
  K++;
  if (K >= N + M - 2) {
    cout << -1 << endl;
    return 0;
  }
  long long inc = min(K, 2*(min(N, M) - 1));
  long long full = max(K - inc, 0LL);
  cout << calc_exp(inc/2) + calc_exp((inc + 1)/2) + full*min(N, M) << endl;
  return 0;
}
```

Problem J4/S2: Training Regimen

The optimal approach is a greedy one as follows: at each point, train in the town with the smallest T_i value that can be reached, until town N can be reached. We can simulate this approach by maintaining the set of towns which are currently reachable (this set starts with only town 1, and then only expands), the smallest T_i value of any town in this set (this value can similarly only decrease), Pyglion's current level (initially 1), and the total time spent training so far (initially 0). Once the set of reachable towns includes town N, we can stop and output the total time. If that never occurs and we're unable to expand the set any further, then town N must be unreachable from town 1, meaning that the answer is –1 instead.

One way to simulate this algorithm would be one level at a time – repeatedly iterating over all routes incident to all reachable towns, using ones with sufficiently low level requirements in order to expand the set of reachable towns, and then training to increase Pyglion's level by 1 if the set can't be expanded any more otherwise. If K is the maximum possible C_i value, then such an approach would take $O(M \times (N + K))$ time, which is too slow for full marks.

One thought is to modify this approach by repeatedly determining the minimum amount by which Pyglion's level must be increased in order to allow at least one new town to become reachable, and increasing it directly by that amount instead of by 1 level at a time. However, this implementation would still require $O(NM)$ time, which is also too slow.

For full marks, we must improve this implementation by avoiding repeatedly scanning over all routes incident to all reachable towns. This can be done in a manner very similar to Prim's algorithm for finding a graph's minimum spanning tree, by instead maintaining a priority queue of towns, ordered by the minimum level required for them to become reachable. At each step, we'll want to pop off the town which can be reached with the minimum possible level, and all of its adjacent towns can then be added to this priority queue. As we go, as before, we'll level up Pyglion as necessary to make each next town reachable, training at the already-reachable town with the smallest T_i value. The time complexity of this algorithm is $O(N + M \log N)$.

Official Solution (C++)

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

const int MAXN = 200005;
const int INF = 1000000000;

int N, M, T[MAXN], minC[MAXN];
int p = 1, t = INF;
vector<pair<int, int> > adj[MAXN];
long long ans = 0;

int main() {
  cin >> N >> M;
  for (int i = 0; i < N; i++) {
    cin >> T[i];
  }
  for (int i = 0; i < M; i++) {
    int a, b, c;
    cin >> a >> b >> c;
    a--;
    b--;
    adj[a].push_back(make_pair(b, c));
    adj[b].push_back(make_pair(a, c));
  }
  priority_queue<pair<int, int> > Q;
  fill(minC, minC + MAXN, INF);
  Q.push(make_pair(0, 0));
  minC[0] = 0;
  while (!Q.empty()) {
    int c = -Q.top().first;
    int u = Q.top().second;
    Q.pop();
    if (minC[u] != c) {
      continue;
    }
    if (c > p) {
      ans += (long long)(c - p)*t;
      p = c;
    }
    if (u == N - 1) {
      cout << ans << endl;
      return 0;
    }
    t = min(t, T[u]);
    for (int j = 0; j < adj[u].size(); j++) {
      int v = adj[u][j].first;
      int c2 = adj[u][j].second;
      if (c2 < minC[v]) {
        Q.push(make_pair(-c2, v));
        minC[v] = c2;
      }
    }
  }
  cout << -1 << endl;
  return 0;
}
```

Problem S3: Puzzle Rooms

This is the sort of problem which generally can't be solved efficiently, but given its structure, it should be a safe bet that some pattern to the answers will emerge for relatively small values of C, and extend to arbitrarily large values of C. This proves to indeed be the case, and the pattern can be visually seen fairly easily if all of the answers up to around $C = 16$ are found (though guessing the pattern earlier, around $C = 13$, is also reasonable).

For such small values of C, a brute force algorithm will suffice, though not just any one. The most natural approach is to try all 2^{4C} possible sets of walls on the grid, and for each one, find the furthest-away pair of empty cells using BFS in another $O(C^2)$ time. This process is incredibly inefficient, but with a few optimizations, it can be made to successfully yield the answers up around $C = 13$ in a reasonable amount of time – most likely not within the problem's 5-second time limit, but at least fast enough to run at home for a while, look at the answers, determine their pattern, and put together a solution with hardcoding.

A more elegant brute force algorithm exists which is both simpler, and can also produce the required answers within a few seconds. We can instead consider directly generating the path from the starting cell to the destination cell. We'll start from each possible starting cell, and repeatedly recurse to adjacent cells, while maintaining the cells which the path has already visited. The only requirement is that the path never visits a cell adjacent to an earlier cell on the path (aside from its previous cell). All cells which aren't on the path can then be filled in with walls to yield a valid grid.

In any case, it can then be seen that once C becomes large, optimal grids contain repeated instances of the following pattern of walls in the middle:

```
.#....
...###
###...
....#.
```

In particular, the answer for $C = 14$ is equivalent to the answer for $C = 8$ but with this pattern inserted once for a certain interval of its interior columns. In the same way, the answer for every $C > 14$ can be determined from the answer for a grid with 6 fewer columns, meaning that only the answers for $C = 1..13$ must be computed from scratch.

Official Solution (C++)

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

const int dr[4] = {-1, 1, 0, 0};
const int dc[4] = {0, 0, -1, 1};

int C, ansL[14], ansSr[14], ansSc[14], ansEr[14], ansEc[14];
bool P[4][100], ansP[14][4][100];

bool valid(int r, int c, int pr, int pc) {
  if (P[r][c])
    return false;
  for (int d = 0; d < 4; d++) {
    int r2 = r + dr[d];
    int c2 = c + dc[d];
    if (r2 < 0 || r2 >= 4 || c2 < 0 || c2 >= C || (r2 == pr && c2 == pc))
      continue;
    if (P[r2][c2])
      return false;
  }
  return true;
}
```

```
void rec(int sr, int sc, int r, int c, int len) {
  P[r][c] = true;
  if (len > ansL[C]) {
    ansL[C] = len;
    ansSr[C] = sr;
    ansSc[C] = sc;
    ansEr[C] = r;
    ansEc[C] = c;
    memcpy(ansP[C], P, sizeof P);
  }
  for (int d = 0; d < 4; d++) {
    int r2 = r + dr[d];
    int c2 = c + dc[d];
    if (r2 < 0 || r2 >= 4 || c2 < 0 || c2 >= C || !valid(r2, c2, r, c))
      continue;
    rec(sr, sc, r2, c2, len + 1);
  }
  P[r][c] = false;
}

const int rStrt[3] = {0, 4, 5};
const char rPat[4][7] = {
  ".#....",
  "...###",
  "###...",
  "....#."
};

int N;

int main() {
  cin >> N;
  while (N--) {
    cin >> C;
    int rep = max((C - 8)/6, 0);
    C -= rep*6;
    if (!ansL[C]) {
      memset(P, false, sizeof P);
      for (int i = 0; i < 4; i++) {
        for (int j = 0; j < C; j++) {
          rec(i, j, i, j, 0);
        }
      }
    }
    cout << (ansL[C] + rep*16) << endl;
    for (int i = 0; i < 4; i++) {
      for (int j = 0; j < C; j++) {
        if (i == ansSr[C] && j == ansSc[C]) {
          cout << "S";
        } else if (i == ansEr[C] && j == ansEc[C]) {
          cout << "E";
        } else
          cout << (ansP[C][i][j] ? '.' : '#');
        if (j == rStrt[C % 3]) {
          for (int k = 0; k < rep; k++) {
            cout << rPat[i];
          }
        }
      }
      cout << endl;
    }
  }
  return 0;
}
```

Problem S4: Replay Value

The graph formed by the towns and routes forms a tree structure. Upon rooting the tree at an arbitrary node (such as node 1), this problem can be solved with dynamic programming. For each node i, let $Up[i]$ be the number of different non-decreasing paths contained within i's subtree and ending at i, and similarly let $Down[i]$ be the number of different non-increasing paths contained within i's subtree and ending at i. Paths of length 0 (spanning only 1 node) are included.

Let's consider how to compute these *Up* and *Down* values for each node i. Firstly, we can set $Up[i] = Down[i] = 1$ to include the path starting and ending at i (which is both non-decreasing and non-increasing). Next, consider each of i's children c which paths might be coming from. If $D[i] > D[c]$, then all paths coming from c into i must non-decreasing, meaning that we can increment $Up[i]$ by $Up[c]$. Similarly, if $D[i] < D[c]$, then we can increment $Down[i]$ by $Down[c]$. Finally, if $D[i] = D[c]$, then both types of paths may occur, so we should both increment $Up[i]$ by $Up[c]$, and increment $Down[i]$ by $Down[c]$. Note that node i's Up and Down values only depend on those of its children, meaning that we can perform these computations recursively and populate these arrays for all nodes in $O(N)$ time.

What remains is computing the actual answer based on these Up and Down values. For each node i, let's consider how many paths are contained within i's subtree and include i itself (in other words, paths which have i as their "topmost" node). If we can determine this value for every node, and add these N values together for all nodes, then every valid path will be counted exactly once. For the most part, this value for node i is simply $Up[i] \times Down[i]$. However, some invalid paths also end up being counted in this fashion, which must then be subtracted. One such path is the path which starts and ends at i, as the starting town is required to be different than the destination town, so we should subtract 1 for that. Aside from that, for each child c such that $D[i] = D[c]$, this approach will include paths which come up from c to i, and then back down to c, which are also disallowed. As such, for each such child, we should simply subtract $Up[c] \times Down[c]$ from the answer. These computations to tally up the total answer can be done during the dynamic programming process in an additional $O(N)$ time.

Official Solution (C++)

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

const int MAXN = 500000;

int N, D[MAXN], par[MAXN];
long long DP[MAXN][2];
long long ans = 0;
vector<int> ord, adj[MAXN];

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> D[i];
  }
  for (int i = 0; i < N - 1; i++) {
    int a, b;
    cin >> a >> b;
    a--;
    b--;
    adj[a].push back(b);
    adj[b].push back(a);
  }
  // Traverse tree to get parents and BFS ordering.
  queue<int> Q;
  for (Q.push(0); !Q.empty(); Q.pop()) {
    int i = Q.front();
    ord.push back(i);  // Add node i to ordering.
    // Iterate over node i's children.
    for (int j = 0; j < adj[i].size(); j++) {
      int c = adj[i][j];
      if (c != par[i]) {
        Q.push(c);
        par[c] = i;
      }
    }
  }
  // Traverse tree bottom-up to compute DP and ans.
  for (int o = N - 1; o >= 0; o--) {
    int i = ord[o];
    DP[i][0] = DP[i][1] = 1;
    // Go through node i's children.
    for (int j = 0; j < adj[i].size(); j++) {
      int c = adj[i][j];
      if (c != par[i]) {
        // Child has equal value?
        if (D[i] == D[c]) {
          // Allow both directions.
          for (int d = 0; d < 2; d++) {
            DP[i][d] += DP[c][d];
          }
          // Subtract invalid paths from answer.
          ans -= DP[c][0]*DP[c][1];
        } else {
          // Allow only one direction.
          int d = D[i] > D[c] ? 0 : 1;
          DP[i][d] += DP[c][d];
        }
      }
    }
    // Add all paths peaking at node i to answer.
    ans += DP[i][0]*DP[i][1] - 1;
  }
  cout << ans << endl;
  return 0;
}
```