WOBURN CHALLENGE

2017-18 Online Round 4

Solutions

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

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

Problem J1: Fight or Flight

Upon inputting the two strings C and T, we should output either of them if they're equal to one another, or the string "`Undecided`" if they're unequal.

Official Solution (C++)

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

int main() {
  string C, T;
  cin >> C >> T;
  if (C == T) {
    cout << C << endl;
  } else {
    cout << "Undecided" << endl;
  }
  return 0;
}
```

Problem J2: Anger Management

We can iterate over the sequence of N events while maintaining two pieces of information – Bruce Banner's current anger level a, and the number of times h that he's transformed into the Hulk so far (both of which are initially 0). Upon inputting each A_i, we should check if both $a < 10$ and $a + A_i \geq 10$ – if so, it's time to increment h by 1. Either way, we should then increment a by A_i. At the end, we can output the final value of h.

Official Solution (C++)

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

int main() {
  int N, ang = 0, ans = 0;
  cin >> N;
  for (int i = 0; i < N; i++) {
    int A;
    cin >> A;
    if (ang < 10 && ang + A >= 10) {
      ans++;
    }
    ang += A;
  }
  cout << ans << endl;
  return 0;
}
```

Problem J3/I1: The Infinity Stones

We'll need to iterate over the 6 Infinity Stone names in alphabetical order and output each one that's not present in the given list of N names. To determine whether a certain Infinity Stone name s should be outputted, we can iterate over all N of the given names while keeping track of whether or not we've found one which is equal to s.

Official Solution (C++)

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

const string STONES[6] = {
  "Mind",
  "Power",
  "Reality",
  "Soul",
  "Space",
  "Time"
};
```

```
int main() {
  int N;
  string S[6];
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> S[i];
  }
  for (int i = 0; i < 6; i++) {
    bool has = false;
    for (int j = 0; j < N; j++) {
      if (STONES[i] == S[j]) {
        has = true;
      }
    }
    if (!has) {
      cout << STONES[i] << endl;
    }
  }
  return 0;
}
```

Problem J4/I2: Efficiency

Let (r, c) denote the cell in the row r and column c of the grid. We'll first need to find Hawkeye's position (R_H, C_H). To do so, we can iterate over all $R \times C$ cells in the grid, and when we encounter a cell (r, c) with the value "`H`", we can set R_H to r and C_H to c.

From there, we'll need to consider all four possible directions for Hawkeye's shot, simulating how many dead soldiers each of them would result in and then using the maximum of the four values as our answer. To simulate Hawkeye shooting North, we can iterate a variable r between 1 and $R_H - 1$ (inclusive), while counting the number of cells (r, C_H) with the value "`C`". The other three directions can be handled similarly.

Official Solution (C++)

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

int main() {
  int R, C, Hr, Hc;
  char G[50][50];
  cin >> R >> C;
  for (int r = 0; r < R; r++) {
    for (int c = 0; c < C; c++) {
      cin >> G[r][c];
      if (G[r][c] == 'H') {
        Hr = r;
        Hc = c;
      }
    }
  }
  int ucnt = 0, dcnt = 0, lcnt = 0, rcnt = 0;
  for (int r = Hr; r >= 0; r--) ucnt += G[r][Hc] == 'C';  // Up
  for (int r = Hr; r < R; r++) dcnt += G[r][Hc] == 'C';   // Down
  for (int c = Hc; c >= 0; c--) lcnt += G[Hr][c] == 'C';  // Left
  for (int c = Hc; c < C; c++) rcnt += G[Hr][c] == 'C';   // Right
  cout << max(max(ucnt, dcnt), max(lcnt, rcnt)) << endl;
  return 0;
}
```

Problem I3/S1: Wakandan Sabotage

Let's start by exploring some cases with small K values by hand to determine which roads Loki should optimally destroy, and what the resulting maximum shortest distance will be. When $K = 0$, no roads are destroyed, and the maximum shortest distance will be $(N – 1) + (M – 1)$, for example between the top-right and bottom-left cities. When $K = 1$, destroying a single horizontal road (for example the leftmost one in the top row) is the best that Loki can do, as it increases the answer by a whole $N – 1$ (for example, the shortest distance between the top-left and top-right cities will be $2(N – 1) + (M – 1)$). When $K = 2$, assuming that $M > 2$, the answer can be increased by a whole additional $N – 1$ if Loki destroys one horizontal road in the top row and one in the bottom row, as long as they're not in the same column (for example, the shortest distance between the top-left city and the bottom-right city can be $3(N – 1) + (M – 1)$).

This pattern continues for the first $M – 1$ roads destroyed by Loki – if he destroys horizontal roads alternating between the top and bottom rows, he can force the maximum shortest path to zigzag between the top and bottom rows on its way from the left column to the right one, resulting in a distance of $(K + 1)(N – 1) + (M – 1)$.

If even more roads need to be destroyed, the maximum shortest distance can no longer be increased. It's clear that the maximum possible shortest distance can never be larger than $M(N – 1) + 2(M – 1) – K$, as that's the total number of non-destroyed roads remaining. In fact, exactly this distance can always be achieved once at least $M – 1$ roads are destroyed – if more roads must be destroyed, they can be repeatedly destroyed from one "end" of the maximum shortest path, with each one decreasing its distance by just 1.

The above observations can be put together to form the following closed-form answer:

$$\min\begin{Bmatrix}(K+1)(N-1)+(M-1)\\ M(N-1)+2(M-1)-K\end{Bmatrix}$$

Official Solution (C++)

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

int main() {
  int N, M, K;
  cin >> N >> M >> K;
  cout << min((K + 1)*(N - 1) + (M - 1), M*(N - 1) + 2*(M - 1) - K) << endl;
  return 0;
}
```

Problem I4/S2: Strange Travels

The sanctums and portals form a directed, unweighted graph with N nodes and M edges. For each artifact i, we'll need to compute the shortest distance $D1_i$ from node 1 to node S_i, as well as the shortest distance $D2_i$ from node S_i to node 1. If $D1_i$ or $D2_i$ are infinite for any i (in other words, if no paths exist connecting those nodes), then we should output –1. Otherwise, the answer will be the sum of $D1_{1..K}$ and $D2_{1..K}$.

Computing all of the shortest distances $D1_{1..K}$ can be done in $O(N + M)$ time with a standard application of breadth-first search (BFS), starting from node 1.

However, computing the shortest distances $D2_{1..K}$ may be more problematic. We could initiate a separate breadth-first search starting from each node $S_{1..K}$, but that approach would be too slow to receive full marks, having a time complexity of $O(K(N + M))$. We can instead imagine an alternate version of the graph in which the directions of all edges are reversed (known as the transpose graph). Performing a single breadth-first search on the transpose graph starting from node 1 can allow us to compute all of the shortest distances $D2_{1..K}$ in just $O(N + M)$ time as well.

Official Solution (C++)

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

const int MAXN = 100005;

int N, M, dist[2][MAXN];
vector<int> adj[2][MAXN];

int main() {
  cin >> N >> M;
  for (int i = 0; i < M; i++) {
    int A, B;
    cin >> A >> B;
    adj[0][A].push_back(B);
    adj[1][B].push_back(A);
  }
  // Initialize to all unreachable.
  memset(dist, -1, sizeof dist);
  // BFS on regular/transpose graphs.
  for (int g = 0; g < 2; g++) {
    queue<int> Q;
    Q.push(1);
    dist[g][1] = 0;
    while (!Q.empty()) {
      int i = Q.front();
      Q.pop();
      for (int j = 0; j < adj[g][i].size(); j++) {
        int k = adj[g][i][j];
        if (dist[g][k] < 0) {
          Q.push(k);
          dist[g][k] = dist[g][i] + 1;
        }
      }
    }
  }
  // Input artifacts, and compute answer.
  int K, ans = 0;
  cin >> K;
  for (int i = 0; i < K; i++) {
    int C;
    cin >> C;
    for (int g = 0; g < 2; g++) {
      if (dist[g][C] < 0) {
        cout << -1 << endl;
        return 0;
      }
      ans += dist[g][C];
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem S3: Guardians of the Cash

Let $A_{1..N}$ be the Southern skyline, and $B_{1..N}$ be the Western skyline. Let $M_i = \min\{A_i, B_i\}$. All $2N - 1$ stacks in row i and column i must be pruned down to a height of at most M_i. Let's refer to this reduction of stacks as "processing index i". Note that it's not a simple matter of processing each index between 1 and N. When an index is processed, it may affect the heights of stacks in other rows/columns, which may affect other A, B, and M values.

This may suggest the following approach: Process all of the indices, then process them all again, and so on until a full iteration of processing all N indices results in no further changes to the collection of coins. Processing a single index takes $O(N)$ time, and it can be shown that this algorithm will terminate within $O(N)$ iterations of processing all N indices, resulting in a time complexity of $O(N^3)$. Processing the indices in random orders may be tempting, but the time complexity will remain at $O(N^3)$, and sufficiently strong test data can be constructed such that the algorithm will require roughly $N/2$ iterations.

A key observation is that, when an index i is processed, it can only affect the M values of other indices j when $M_j > M_i$. Furthermore, when such an index j is affected, its M value may be reduced down to M_i, but not lower than that. This suggests that, if we process all N indices in non-decreasing order of M, the M values of already-processed indices will never be affected, and so a single iteration through all N indices will be sufficient.

That being said, it's insufficient to sort all N indices by their initial M values and process them in that static order, as their M values can change during the algorithm, which may change the order in which they should be processed. Therefore, we'll need to dynamically keep track of the A, B, and M values. At each step of the algorithm, we'll choose to process the remaining unprocessed index with the minimum M value (in a fashion reminiscent of Dijkstra's shortest path algorithm). While processing an index, we'll then need to update other A, B, and M values more efficiently than iterating over all $O(N^2)$ stacks in the grid. Overall, it's possible to implement this approach in time $O(N^2 \log N)$, for example by maintaining a set (balanced binary search tree) of pairs $\{M_i, i\}$ as well as a multiset of C values for each row and each column.

Official Solution (C++)

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

const int MAXN = 1005;

int N, C[MAXN][MAXN], M[MAXN];
bool done[MAXN];
multiset<int> A[MAXN], B[MAXN];
set< pair<int, int> > S;

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      cin >> C[i][j];
      A[i].insert(C[i][j]);
      B[j].insert(C[i][j]);
    }
  }
  // Initialize set of M values.
  for (int i = 0; i < N; i++) {
    M[i] = min(*A[i].rbegin(), *B[i].rbegin());
    S.insert(make_pair(M[i], i));
  }
  // Simulate the greedy process.
  long long ans = 0;
  while (!S.empty()) {
    // Find the smallest M value.
    int i = S.begin()->second;
    S.erase(S.begin());
    int m = M[i];
    // Flatten the ith row/column down to m.
    done[i] = true;
    if (C[i][i] > m) {
      ans += C[i][i] - m;
    }
    for (int j = 0; j < N; j++) {
      if (done[j]) continue;
      if (C[i][j] > m) {
        ans += C[i][j] - m;
        B[j].erase(B[j].find(C[i][j]));
        C[i][j] = m;
        B[j].insert(C[i][j]);
      }
      if (C[j][i] > m) {
        ans += C[j][i] - m;
        A[j].erase(A[j].find(C[j][i]));
        C[j][i] = m;
        A[j].insert(C[j][i]);
      }
      int newM = min(*A[j].rbegin(), *B[j].rbegin());
      if (newM < M[j]) {
        S.erase(make_pair(M[j], j));
        M[j] = newM;
        S.insert(make_pair(M[j], j));
      }
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem S4: Alpha Nerd

The optimal strategy is to make the weight of the graph's actual minimum spanning tree equal to 0, and then force Spiderman's algorithm to get "stuck" as many times as possible, such that it's forced to traverse a weight-1 edge to the next node on its path (thus increasing the answer by 1).

The algorithm can't be stuck on its first iteration, as if the actual minimum spanning tree's weight is 0, then there must be at least one weight-0 edge incident to the first node. After that, it can be made to be stuck at any given node whenever all nodes adjacent to that node along fixed-weight edges have already been visited. So, our goal is to maximize the number of times this occurs.

The fixed-weight edges form a forest of trees on the graph, which can be handled independently. For now, let's ignore which node will be chosen as the algorithm's starting node, and only focus on which nodes we can force the algorithm to get stuck at. If there's ever an "isolated" node a (such that all its children have already been visited, and either its parent has been visited or it has no parent), we can get the algorithm to jump to it and immediately be stuck again. Otherwise, we should get the algorithm to jump to a node b one level up, and then traverse the weight-0 edge down to a now-isolated child c before being stuck again. Decomposing each of the trees into an optimal set of single nodes a and pairs of nodes (b, c) as described above can be done in $O(N)$ time by finding each tree and recursively iterating over it, while greedily simulating how the nodes will be split up.

Upon counting the number of nodes at which the algorithm will get stuck in each tree, we can sum these values up, and then subtract 1 for the very last node visited by the algorithm (as we will have overcounted it as being stuck after that point). At this point, one thing remains addressed – it matters which node the algorithm starts at, so which one should it be? As mentioned above, the only special thing about the first node is that the algorithm can't get stuck at it. Therefore, if we choose to have started at any node which we weren't forcing the algorithm to get stuck at anyway, then our answer will be unaffected. There will always be at least one such node unless $M = 0$, in which case we must subtract 1 more from our answer.

Note: *For those looking for an additional challenge, a more difficult version of this problem is available at: https://wcipeg.com/problem/wc174s4h*

Official Solution (C++)

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

struct Subtree {
  int c;
  bool u;
  Subtree(int c, bool u) : c(c), u(u) {}
};

const int MAXN = 300005;
vector<int> adj[MAXN];
bool visit[MAXN];

Subtree rec(int i, int p) {
  visit[i] = true;
  // Loop over all children in this graph section.
  Subtree res(1, true);
  for (int j = 0; j < adj[i].size(); j++) {
    int c = adj[i][j];
    if (c == p) continue;
    Subtree s = rec(c, i);
    res.c += s.c;
    if (s.u && res.u) {
      res.c--;
      res.u = false;
    }
  }
  return res;
}

int main() {
  int N, M;
  cin >> N >> M;
  for (int i = 0; i < M; i++) {
    int A, B;
    cin >> A >> B;
    adj[A].push_back(B);
    adj[B].push_back(A);
  }
  // Find and process each component of the graph.
  memset(visit, false, sizeof visit);
  int ans = 0;
  for (int i = 1; i <= N; i++) {
    if (!visit[i]) {
      ans += rec(i, 0).c;
    }
  }
  // Decrease ans for the last and/or first steps.
  cout << max(0, ans - (M == 0 ? 2 : 1)) << endl;
  return 0;
}
```