WOBURN CHALLENGE

2017-18 Online Round 2

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 Rings of Power

Upon inputting the 3 integers E, D, and M, we need to output the sum $E + D + M + 1$.

Official Solution (C++)

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

int E, D, M;

int main() {
  cin >> E >> D >> M;
  cout << E + D + M + 1 << endl;
  return 0;
}
```

Problem J2: Breeding an Army

Saruman can create no more than M_s / M_u Uruk-hai (rounded down to the nearest integer) before running out of men. Similarly, he can create no more than O_s / O_u Uruk-hai due to orc restrictions, and no more than L_s / L_u Uruk-hai due to mud restrictions. Putting these facts together, the maximum number of Uruk-hai he can create is the minimum of the three floored quotients M_s / M_u, O_s / O_u, and L_s / L_u.

Official Solution (C++)

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

int Mu, Ou, Lu, Ms, Os, Ls;

int main() {
  cin >> Mu >> Ou >> Lu >> Ms >> Os >> Ls;
  cout << min(Ms / Mu, min(Os / Ou, Ls / Lu)) << endl;
  return 0;
}
```

Problem J3/I1: Escaping the Mines

Let's start by looping over the values $J_{1..N}$ and determining which members of the Fellowship are able to clear the chasm by themselves (all members i such that $J_i \geq M$). Let Y be the number of members who can do so, and let X be the number of members who cannot. Each member of the former group can help out at most one member of the latter group, meaning that min(X, Y) extra members can get carried across in addition to the Y members who can jump across regardless. This gives us an answer of Y + min(X, Y).

Official Solution (C++)

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

int N, M, J, canJump = 0, needHelp = 0;

int main() {
  cin >> N >> M;
  for (int i = 0; i < N; i++) {
    cin >> J;
    if (J >= M) {
      canJump++;
    } else {
      needHelp++;
    }
  }
  cout << canJump + min(canJump, needHelp) << endl;
  return 0;
}
```

Problem J4/I2: Entish Translation

One approach to this problem involves taking two separate passes over the text, one for each step of the translation process – first collapsing all contiguous sequences of consonants, and then parsing the string into tokens and collapsing all contiguous sequences of equal ones. A simple way of performing both types of reductions is to iterate over the characters/tokens, keep track of the previous one, and use it to determine whether or not the current one should be removed. It's possible to either modify the string in place by deleting characters, or create a new string from the new characters which should be kept.

In the official solution, we instead combine everything into a single pass over the characters of the text, while keeping track of several pieces of information - the answer string (which is gradually built up from the appropriate characters/tokens), the current token string, the previous token, and the previous character. When a letter is processed, it should be added onto the current token unless both this letter and the previous character are consonants. When a dash is processed or the end of the text is reached, the current token should be appended to the answer string unless it's equal to the previous token (if any), and then the current token should be reset to an empty string.

Official Solution (C++)

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

string ans = "", prevToken = "", curToken = "";
char prevChar = '-';

bool IsConsonant(char c) {
  return "aeiou-".find(c) == string::npos;
}

void EndCurToken() {
  if (curToken != prevToken) {
    if (ans.size() > 0) {
      ans += "-";
    }
    ans += curToken;
  }
  prevToken = curToken;
  curToken = "";
}

void ProcessChar(char c) {
  if (c == '-') {
    EndCurToken();
  } else if (!IsConsonant(prevChar) || !IsConsonant(c)) {
    curToken += c;
  }
  prevChar = c;
}

int main() {
  string S;
  cin >> S;
  for (int i = 0; i < S.size(); i++) {
    ProcessChar(S[i]);
  }
  EndCurToken();
  cout << ans << endl;
  return 0;
}
```

Problem I3/S1: Keeping Score

At first glance, it may seem that there are 10^9 possible values of X to consider. However, we only need to bother considering at most G of them, the ones which are equal to SG_i (for some i), as it can't help Gimli to choose some other arbitrary value of X. This immediately gives us a solution with a time complexity of $O(LG)$, in which we try each such value of X, and tally up Legolas and Gimli's scores for it (by iterating over all of their killed enemies) to determine whether it's valid.

However, the above solution isn't efficient enough to receive full marks. To do better, let's start by sorting the values $SL_{1..L}$ in non-increasing order (which can be done in $O(L \log L)$ time), and do the same with the values $SG_{1..G}$ (in $O(G \log G)$ time).

Let's now consider trying each possible value of X in this order, with the largest ones first. Let's skip indices i such that $SG_i = SG_{i+1}$. Now, when we consider $X = SG_i$, we can determine Gimli's score without iterating over all of his killed enemies - his score is simply i (as the values $SG_{1..i}$ are larger than or equal to X while the values $SG_{(i+1)..G}$ are smaller than it).

Determining Legolas's score is trickier, but we can observe that there must be some corresponding index j such that the values $SL_{1..j}$ are larger than or equal to X while the values $SL_{(j+1)..L}$ are smaller than it, meaning that Legolas's score is j. Furthermore, as i increases, X cannot increase and so j cannot decrease. Therefore, as we iterate over values of i, we can update j as necessary (incrementing it as long as $SL_{j+1} \geq X$), and stop if we find a value of X for which $i > j$.

The overall time complexity of this algorithm is $O(L \log L)$.

Official Solution (C++)

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

int L, G, SL[200005], SG[200005];

int main() {
  cin >> L >> G;
  for (int i = 0; i < L; i++) cin >> SL[i];
  for (int i = 0; i < G; i++) cin >> SG[i];
  // Sort each list of enemies by non-increasing strength level.
  sort(SL, SL + L, greater<int>());
  sort(SG, SG + G, greater<int>());
  // Consider each possible cutoff point X from Gimli's list.
  int cntL = 0;
  for (int i = 0; i < G; i++) {
    int X = SG[i];
    // Update the Legolas's count for this X.
    while (cntL < L && SL[cntL] >= X) {
      cntL++;
    }
    // Is this X valid?
    if (i + 1 > cntL) {
      cout << X << endl;
      return 0;
    }
  }
  // No solution found.
  cout << -1 << endl;
  return 0;
}
```

Problem I4/S2: Don't Follow the Lights

We can represent the grid as a graph, with one node per cell, and an unweighted, directed edge for each valid move between pairs of cells. We then need to find the length of the shortest path from one node to another on this graph, which can be done with a standard application of Breadth First Search (BFS). There are $O(RC)$ nodes in the graph, and at most four outgoing edges from each node, meaning that there are also $O(RC)$ edges in total. The time complexity of BFS is then $O(RC)$, which is fast enough.

Assuming familiarity with BFS, the trickiest part of the solution is efficiently determining the set of edges which exist in the graph in the first place. For a given cell (r, c) (in row r and column c), it's possible to move upwards to cell $(r - 1, c)$ if $r > 1$, neither of those two cells contain a torch, and there are fewer than two torches in all of cells $(1..(r - 2), c)$ (or, equivalently, in cells $(1..r, c)$). Similar conditions apply to moving downwards, leftwards, and rightwards from cell (r, c).

The most direct method of finding all valid moves which satisfy the above conditions it to consider each cell (r, c) independently. A simple check for whether there are fewer than two torches in cells $(1..r, c)$ takes $O(R)$ time. The corresponding check for torches below cell (r, c) also takes $O(R)$ time, while the checks for torches to the left and right each take $O(C)$ time. Therefore, this approach for constructing the graph takes $O(RC \times (R+C))$ time in total, which isn't efficient enough to receive full marks.

The time complexity of the above process can be improved to $O(RC)$ by observing that, for example, the number of torches in cells $(1..r, c)$ can be computed in $O(1)$ if we already have the number of torches in cells $(1..(r - 1), c)$. If we fix c and iterate r upwards from 1 to R, while maintaining the number of torches encountered so far in that column (in other words, the number of torches in cells $(1..r, c)$), then we can find all of the valid upwards moves in that column in just $O(R)$ time. This can then be repeated for all C columns, and a similar process can be repeated for the other three directions.

Official Solution (C++)

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

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

string G[1505];
vector<State> moves[1505][1505];
int dist[1505][1505];

int main() {
  // Input the grid.
  int R, C;
  cin >> R >> C;
  for (int r = 0; r < R; r++) {
    cin >> G[r];
  }

  // Precompute all valid moves.
  for (int r = 0; r < R; r++) {
    // Left
    for (int c = 0, t = 0; c < C; c++) {
      if (G[r][c] == '*') t++;
      if (c - 1 >= 0 && G[r][c - 1] != '*' && t < 2) {
        moves[r][c].push_back(State(r, c - 1));
      }
    }
```

```
    // Right.
    for (int c = C - 1, t = 0; c >= 0; c--) {
      if (G[r][c] == '*') t++;
      if (c + 1 < C && G[r][c + 1] != '*' && t < 2) {
        moves[r][c].push_back(State(r, c + 1));
      }
    }
  }
  for (int c = 0; c < C; c++) {
    // Left.
    for (int r = 0, t = 0; r < R; r++) {
      if (G[r][c] == '*') t++;
      if (r - 1 >= 0 && G[r - 1][c] != '*' && t < 2) {
        moves[r][c].push_back(State(r - 1, c));
      }
    }
    // Right.
    for (int r = R - 1, t = 0; r >= 0; r--) {
      if (G[r][c] == '*') t++;
      if (r + 1 < R && G[r + 1][c] != '*' && t < 2) {
        moves[r][c].push_back(State(r + 1, c));
      }
    }
  }

  // Breadth-first search.
  memset(dist, -1, sizeof dist); // Initialize to all cells unreachable.
  queue<State> Q;
  for (int r = 0; r < R; r++) {
    for (int c = 0; c < C; c++) {
      if (G[r][c] == 'S') {
        Q.push(State(r, c)), dist[r][c] = 0;
      }
    }
  }
  while (!Q.empty()) {
    // Get next state.
    State s = Q.front();
    Q.pop();
    int d = dist[s.r][s.c];
    // Reached the destination?
    if (G[s.r][s.c] == 'D') {
      cout << d << endl;
      return 0;
    }
    // Try each valid move from this state.
    for (int i = 0; i < moves[s.r][s.c].size(); i++) {
      State s2 = moves[s.r][s.c][i];
      if (dist[s2.r][s2.c] < 0) {
        Q.push(s2), dist[s2.r][s2.c] = d + 1;
      }
    }
  }
  // No solution found.
  cout << -1 << endl;
  return 0;
}
```

Problem S3: Battle of the Pelennor Fields

We can process the events one by one while maintaining information about the state of the battlefield in the form of two data structures. The first is a set O of points (integers) at which there are non-vulnerable orcs (note that the answer at any point is simply the size of O). The second is a set A of disjoint intervals (pairs of integers) of points in range of archers. Note that the intervals in A are closed (inclusive of its endpoints). Pre-populating A with a pair of dummy intervals $[-\infty, -\infty]$ and $[\infty, \infty]$ can make the following implementation more convenient. We'll want to implement both of these sets with balanced binary trees (e.g. C++ sets) to allow for efficient insertion, deletion, searching, and ordered iteration.

It's important that the intervals in A are kept disjoint, such that no points on the number line is contained in multiple of them. For example, if there are two intervals $[a, b]$ and $[c, d]$ such that the latter is contained within the former (with $a \leq c$ and $d \leq b$), then $[c, d]$ should be removed completely. On the other hand, if they partially overlap with one another (with $a \leq c$, $c \leq b$, and $b \leq d$), then they should be combined into a single interval $[a, d]$.

Let's now consider how we can process the arrival of an orc at position o. We'll need to check whether or not this orc is already vulnerable (contained within an interval in A). This can be done in $O(\log N)$ time by searching for the interval $[x, y]$ in A with the largest x value such that $x \leq o$, and then checking whether $o \leq y$. If not, then o should be inserted into O, also in $O(\log N)$ time.

What remains is being able to process the arrival of an archer at position a and with a bow range of r. This archer corresponds to an interval $[x, y]$, where $x = a - r$ and $y = a + r$. If $[x, y]$ is already contained within an interval in A, then it can be completely disregarded. This can be checked in $O(\log N)$ time, very similarly to the above check for orcs.

Otherwise, the next step is to remove all orcs in O which are within $[x, y]$. To do so, we can search for the smallest point o in O such that $o \geq x$ (if any), and repeatedly erase it and iterate to the next larger point in O until it's either greater than y or the end of O is reached. This process doesn't necessarily take $O(\log N)$ time for any given archer, as $O(N)$ orcs might be removed all at once, but it does take what's known as $O(\log N)$ amortized time, as each of the $O(N)$ orcs will be removed at most once in total. The result is that this step will take a total of $O(N \log N)$ time across all archers.

Finally, we need to update A according to the new interval. We should first iterate over existing intervals $[x', y']$ in A which overlap with $[x, y]$ to the left (such that $x' \leq x$ and $y' \geq x$). For each such interval, we can erase it from A while combining it into the new interval by setting x = min(x, x'). Similarly to the previous step, this process takes $O(\log N)$ amortized time per archer. The same approach should then be repeated for existing intervals overlapping $[x, y]$ to the right, and then finally, the new combined interval $[x, y]$ can be inserted into A, satisfying the guarantee that it's disjoint from all other intervals still in A.

The total complexity of the algorithm described above is $O(N \log N)$.

Official Solution (C++)

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

const int INF = 2000000002;

typedef set<int> IntSet;
typedef set< pair<int,int> > PairSet;
typedef IntSet::iterator IntIter;
typedef PairSet::iterator PairIter;

IntSet O;   // Set of non-vulnerable orc points.
PairSet A;  // Set of disjoint intervals of points covered by archers.

void ProcessOrc(int o) {
  // Insert this point into O if not inside an existing interval in A.
  PairIter I = A.lower_bound(make_pair(o + 1, -INF));
  I--;
  if (o < I->first || o > I->second) {
    O.insert(o);
  }
}
```

```
void ProcessArcher(int a, int r) {
  // Check if the new interval is covered by an existing one in A.
  int x = a - r, y = a + r;
  PairIter I = A.lower_bound(make_pair(x + 1, -INF));
  I--;
  if (I->first <= x && y <= I->second) return;

  // Erase any points in O which are inside the new interval.
  IntIter OI = O.lower_bound(x);
  while (OI != O.end() && *OI <= y) {
    IntIter tmp = OI;
    OI++;
    O.erase(tmp);
  }

  // Merge/erase intervals to the right.
  PairIter J = I;
  J++;
  while (J->first <= y) {
    y = max(y, J->second);
    PairIter tmp = J;
    J++;
    A.erase(tmp);
  }

  // Merge/erase intervals to the left.
  while (I->second >= x) {
    x = min(x, I->first);
    PairIter tmp = I;
    I--;
    A.erase(tmp);
  }

  // Insert the new merged interval into A.
  A.insert(make_pair(x, y));
}

int main() {
  // Initialize A with dummy intervals for convenience.
  A.insert(make_pair(-INF, -INF));
  A.insert(make_pair(INF, INF));
  // Input and process the events.
  int N;
  cin >> N;
  while (N--) {
    int e;
    cin >> e;
    if (e == 1) {
      int o;
      cin >> o;
      ProcessOrc(o);
    } else {
      int a, r;
      cin >> a >> r;
      ProcessArcher(a, r);
    }
    cout << O.size() << endl;
  }
  return 0;
}
```

Problem S4: One Does Not Simply Walk Into Mordor

We will first consider the case in which the line segments may intersect. Let's define a "region" as a contiguous sequence of equal L values (making sure to count L_N and L_1 as being consecutive). Let r be the number of regions (which can be easily counted in $O(N)$ time), and note that r is guaranteed to be at least 2. There are then r points on the circle which are immediately surrounded by two different L values, meaning that at least one line segment must pass through the circle there. This suggests that there must be at least r line segment endpoints, and so at least c = ceiling(r / 2) line segments. Furthermore, c line segments are always enough to get the job done. Imagine sorting the r points mentioned above, adding one more arbitrary point if r is odd, and then connecting the first point to the c-th one, the second one to the (c+1)-st one, and so on. This will divide up the circle such that each region is segmented off entirely from the others.

The non-intersecting case isn't so simple, but can be solved with dynamic programming. Let $DP[i][s]$ be the minimum number of line segments required to divide up the portion of the circle starting at index i and spanning s markings, with L_i being the portion's "representative marking". We'd like to consider various ways of cutting this portion into smaller sub-portions while saving on cuts when those sub-portions have markings matching L_i. Let's consider all possible splits of this subarray into two subarrays, with the first including the first k markings and the second including the remaining $s - k$ markings ($1 \le k < s$). Note that the second subarray starts at marking $i + k$ (wrapping around if it exceeds N).

If $k > 1$, we'll make one cut to separate index i from the remainder of the first subarray, which will then require $DP[i + 1][k - 1]$ more cuts itself. Similarly, we'll make one more cut to separate out the second subarray, which will then required $DP[i + k][s - k]$ more cuts itself. However, each of these two main cuts can be omitted if that portion's representative marking (L_{i+1} or L_{i+k}) is equal to L_i. We can calculate $DP[i][s]$ by determining the above value for all possible values of k and taking the minimum.

Finally, any arbitrary marking can be used as the "representative marking" of the entire circle, meaning that the answer is for example $DP[1][N]$. The approach described above involves $O(N^2)$ DP states and $O(N)$ transitions per state, resulting in a time complexity of $O(N^3)$.

Official Solution (C++)

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

int N, L[400], DP[400][401];

// Returns the minimum number of line segments required
// for the subarray starting at index i with size s.
int Solve(int i, int s) {
  if (s == 1) return 0; // Base case
  if (DP[i][s] >= 0) return DP[i][s]; // Already computed.
  // Consider each possible splitting point.
  int d = N;
  for (int j = 0; j < s - 1; j++) {
    int d1 = j ? (Solve((i + 1) % N, j) + 1) : 0;
    int i2 = (i + j + 1) % N;
    int d2 = Solve(i2, s - j - 1) + (L[i] == L[i2] ? 0 :
1);
    d = min(d, d1 + d2);
  }
  return DP[i][s] = d; // Memoize result.
}

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) cin >> L[i];

  // Intersecting
  int cnt = 0;
  for (int i = 0; i < N; i++) {
    if (L[i] != L[(i + 1) % N]) {
      cnt++;
    }
  }
  int ans1 = (cnt + 1) / 2;

  // Non-intersecting.
  memset(DP, -1, sizeof(DP));
  int ans2 = N;
  for (int i = 0; i < N; i++) {
    ans2 = min(ans2, Solve(i, N));
  }

  cout << ans1 << ' ' << ans2 << endl;
  return 0;
}
```