WOBURN CHALLENGE

2016-17 Online Round 1

Solutions

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

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

Problem J1: A Spooky Season

Output "sp", then have a loop which runs S times and outputs one "o" each time, and then finally output "ky". Remember to not output newlines in between these characters!

Official Solution (C++)

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

int S;

int main() {
  cin >> S;
  cout << "sp";
  for (int i = 0; i < S; i++) {
    cout << "o";
  }
  cout << "ky" << endl;
  return 0;
}
```

Problem J2: Frankenstein's Monster

An important skill in solving problems like this one is interpreting the statement. The most important part of the statement is this:

> "*A 'word' is a maximal consecutive sequence of non-period characters in the string. That is, each word is either preceded by a period or is at the start of the string. Similarly, each word is followed by a period or is at the end of the string.*"

This means that we should replace all substrings "Frankenstein" with "Frankenstein's.Monster" if and only if one of the follow is true:

- it's at the beginning of *S* (always followed by a period, except when it's also at the end of *S*)
- it's both preceded and followed by a period
- it's at the end of *S* (always preceded by a period, except when it's also at the beginning of *S*)

We can solve this by finding all substrings Frankenstein and checking all these cases. Alternatively, this problem becomes a bit simpler if we start by augmenting the given string, adding one "." to the start and another "." to the end (we just have to remember to remove these at the end before outputting the answer). We then want to replace every occurrence of ".Frankenstein." with ".Frankenstein's.monster.". To find each occurrence, some programming languages provide a built-in function to search for a substring within a string. Alternatively, this can be done by looping over every character in the string and checking if the 14-character substring starting there is equal to ".Frankenstein.". Once an occurrence is found starting at index *i*, one way to replace it is to replace the whole string *S* with the concatenation of substrings *S*[1…(*i* – 1)] + ".Frankenstein's.monster." + *S*[(*i* + 14)…|*S*|] where |*S*| is the length of *S*.

Official Solution (C++)

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

string s;

int main() {
  cin >> s;
  s = "." + s + ".";  // Pad with periods for convenience.
  while (true) {
    int i = s.find(".Frankenstein.");
    if (i == string::npos) break;  // No more occurences found.
    s = s.substr(0, i) + ".Frankenstein's.monster." + s.substr(i + 14);
  }
  // Output the string with the padded periods removed.
  cout << s.substr(1, s.length() - 2) << endl;
  return 0;
}
```

Problem J3/S1: Hide and Seek

This problem can be approached greedily. If we consider the leftmost room *a*, Michael's leftmost chosen room *b* might as well be the rightmost possible room such that *a* and *b* are within *D* units of each other, since that'll cover not only room *a*, but as many more rooms to the right of a as possible. In particular, if room c is the rightmost room which is within *D* units of room *b*, then room b will cover all rooms between *a* and *c* (inclusive).

Therefore, if we can determine the locations of these 3 rooms *a*, *b*, and *c*, then we can add room b to Michael's list of chosen rooms, and henceforth ignore room *c* and all rooms left of it, thereby reducing the problem to only the section of the hallway to the right of room *c*. At that point, we can repeat this process until there are no more rooms remaining to the right of room *c*.

The first step is to find room *a*. Let's define *a1* to be the leftmost character of room *a*, and *a*2 to be its rightmost character (and similarly for rooms *b* and *c*). *a1* is simply the first "." in the floor plan. *a2* is then the character before the first "#" after *a1*.

The second step is to find room b. The furthest character in range of room a is *a2 + D*. If that character is a ".", then it's inside room b, and so *b2* is the character before the first "#" after *a2 + D*. Otherwise, room *b* must be to the left, so *b2* is the last "." before *a2 + D*. We don't need to find *b1*.

The final step is to find room *c*. The furthest character in range of room *b* is *b2 + D*. We can repeat exactly the same process as above to find *c2*. Once again, at that point, we can add 1 to the answer (since Michael will need to visit room *b*), and repeat the process with the remainder of the string to the right of *c2*.

Official Solution (C++)

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

int N, D, ans = 0;
string S;

int main() {
  cin >> N >> D >> S;
  int i = 0;
  // Loop until we break due to reaching the end of the string.
  while (i < N) {
    // Find start of next room (a1).
    while (S[i] == '#') i++;
    if (i >= N) break;
    ans++;
    // Find first wall past this room (a2 + 1).
    while (S[i] == '.') i++;
    // Find first wall past last room in range of this room (b2 + 1).
    i += D - 1;
    if (i >= N) break;
    while (S[i] == '#') i--;
    while (S[i] == '.') i++;
    // Find first wall past last room in range of that room (c2 + 1).
    i += D - 1;
    if (i >= N) break;
    while (S[i] == '#') i--;
    while (S[i] == '.') i++;
  }
  cout << ans << endl;
  return 0;
}
```

Problem J4/S2: Alucard's Quest

Alucard actually has very little choice. In order to reach each chamber that contains an item, Alucard must necessarily pass through all of the passageways between that chamber and the 1st chamber at some point. Furthermore, he never has a reason to pass through any of the remaining passageways (those which are not between the 1st chamber and any chamber containing an item). Therefore, it's clear exactly which set of chambers and passageways Alucard should pass through at least once, regardless of the order in which he collects the K items.

For a given chamber i, let's try to determine if Alucard will have to visit it. Let $A(i)$ be true if he will, and false otherwise. If chamber i contains an item itself, then certainly $A(i)$ = true. Otherwise, if we model the castle as a tree rooted at the 1st chamber, then $A(i)$ = true if and only if $A(c)$ = true for at least one child c of chamber i. This is because, in order to reach chamber c, Alucard will have to pass through chamber i.

How does this translate into the amount of magic power required in total?

For each chamber i such that $i > 1$ and $A(i)$ = true, Alucard will need to use the passageway between i and its parent at some point. Therefore, assuming we can compute the A values, we can add up then add up the monster counts in these passageways connecting chambers which must be visited to yield the answer.

What remains is computing the A values. $A(i)$ is computed based on chamber i and the A values of i's children, meaning we can recursively compute it starting from the 1st chamber. For convenience, we can pass in the index of i's parent in the recursive call so that we can determine which of its neighbours are its children when iterating over them.

We can also tally up the total answer during this process, rather than iterating over all nodes with true A values afterwards.

Since each of the N chambers is visited only once in the recursion, the algorithm has a time complexity of $O(N)$. The official solution given on the right implements this idea.

Official Solution (C++)

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

int N, K, a, b, m, c, ans = 0;
bool item[200005] = {0};
vector<pair<int, int> > adj[200005];

// Returns true iff chamber i must be visited.
bool A(int i, int parent) {
  bool ret = item[i];  // Must be visited if it has an item.
  for (int j = 0; j < adj[i].size(); j++) {
    int c = adj[i][j].first;
    if (c != parent && A(c, i)) { // Must this child be visited?
      // Then chamber i must also be visited,
      ret = true;
      // and this passageway must be cleared.
      ans += adj[i][j].second;
    }
  }
  return ret;
}

int main() {
  cin >> N >> K;
  for (int i = 0; i < N - 1; i++) {
    cin >> a >> b >> m;
    adj[a].push_back(make_pair(b, m));
    adj[b].push_back(make_pair(a, m));
  }
  for (int i = 0; i < K; i++) {
    cin >> c;
    item[c] = true;
  }
  A(1, -1);  // Recurse starting from the 1st chamber.
  cout << ans << endl;
  return 0;
}
```

Problem S3: Tricky's Treats

Tricky's time can be spent either walking between houses or visiting them for treats. The trade-off between spending more time walking to visit further-away houses versus having more time to visit more houses is difficult to manage. It will be much more manageable if we can fix the time spent on one of these two activities, and optimize for the other one.

After sorting the houses in increasing order of position, let's imagine that we'll decide that some house i will be the furthest house that Tricky will visit. This immediately fixes the amount of time that he'll spend on walking at $2P_i$ (he'll need to walk to position P_i and then back to position 0, and on the way he'll be able to freely visit any other houses closer than house i). This leaves $M - 2P_i$ time for visiting some subset of houses with indices from 1 to i.

This means that $h_i = \text{floor}((M - 2P_i) / T)$ houses can be visited. Clearly then, of the first i houses, the h_i of them with the largest C values should be chosen (or all i houses if $h_i \geq i$). If we consider each possible house i separately, calculate h_i, and find the largest $\min(i, h_i)$ C values out of $C_{1..i}$, we can come up with an answer in $O(N^2 \log N)$ time.

This approach can be improved to $O(N \log N)$ time by noticing that the optimal set of houses for a given i can be computed more efficiently based on the optimal set of houses for $i - 1$, rather than from scratch. We can iterate i upwards from 1 to N, while maintaining a priority queue of the current optimal C values as well as storing the sum of the values in this queue. When we reach a new house i, we can add C_i to the priority queue (and add C_i to its sum), and then we'll need to repeatedly pop the smallest value from the priority queue until it contains no more than h_i values (while similarly keeping the sum of these values up to date).

For convenience of implementation, we can negate all values stored in the priority queue to allow easy removal of its smallest value rather than its largest one. In this way, the priority queue will always store the largest $\min(i, hi)$ C values out of the first i values (as well as their sum), with their sum being the optimal number of treats which can be collected assuming that house i is the furthest house visited, and as such being a candidate for the final answer.

Official Solution (C++)

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

int N, M, T, prev = 0, sum = 0, ans = 0;
pair<int, int> H[100005];

int main() {
  priority_queue<int> Q;
  cin >> N >> M >> T;
  for (int i = 0; i < N; i++) {
    cin >> H[i].first >> H[i].second;
  }
  // Sort houses by increasing position.
  sort(H, H + N);
  // Consider each furthest house i to visit.
  for (int i = 0; i < N; i++) {
    int pos = H[i].first;
    int val = H[i].second;
    // Update time left to visit houses besides walking.
    M -= 2*(pos - prev);
    if (M < T) {
      break;
    }
    prev = pos;
    // Insert the new value and update the sum.
    Q.push(-val);
    sum += val;
    // Remove values until few enough to visit them all.
    while (T * Q.size() > M) {
      sum += Q.top();
      Q.pop();
    }
    ans = max(ans, sum);
  }
  cout << ans << endl;
  return 0;
}
```

Problem S4: TP

Computing expected values directly can be problematic, but they can often be computed based on probabilities of events instead. For example, in the case of this problem, if we let P_i be the probability that exactly i different houses will have been TP-ed after all M passes, then the expected number of different houses to be TP-ed is $\sum_{i=0}^{N} i \cdot P_i$.

Now, how can we compute these P values? We can use dynamic programming. Let $DP[p][h]$ be the probability that exactly h houses will have been TP-ed after p passes. For starters, we know that $DP[0][0] = 1$, and $DP[0][1..N] = 0$. From a given state (p, h), there are only 3 possible transitions to consider: whether the $(p + 1)$-th pass results in 0, 1, or 2 additional houses getting TP-ed (leading to states $(p + 1, h)$, $(p + 1, h + 1)$, or $(p + 1, h + 2)$). We'll need to compute the probability of each of these 3 transitions occurring. For starters, there are $t = N(N – 1)/2$ total possible pairs of houses, and if h houses have been TP-ed so far, then $h_2 = N – h$ houses have not.

In order for zero new houses to be TP-ed, both targeted houses must have already been TP-ed. The number of such houses is $c_0 = h(h – 1)/2$. Therefore, the probability of this transition is c_0 / t. As a result, we add $DP[p][h] \times c_0 / t$ onto $DP[p + 1][h]$.

Similarly, there are $c_1 = h \times h_2$ pairs of targeted houses which would result in 1 new house getting TP-ed, and there are $c_2 = h_2(h_2 – 1)/2$ pairs which result in 2 new houses getting TP-ed. In the same way, we can add $DP[p][h] \times c_1 / t$ onto $DP[p + 1][h + 1]$, and $DP[p][h] \times c_2 / t$ onto $DP[p + 1][h + 2]$.

Finally, once we've considered all states for $p = 0...(M – 1)$ and $h = 0..N$, we'll have populated the whole DP array. There are $O(NM)$ states and only a constant number (3) of transitions from each state, meaning that the time complexity of this algorithm is $O(NM)$. For each i, $P_i = DP[M][i]$, so we can proceed to calculate the expected value as described initially. This idea is implemented here in the official solution.

This problem can also be solved in $O(1)$ with some more advanced math. For any given house, $N – 1$ pairs of houses result in it getting TP-ed on any given pass. Therefore, the probability of it getting TP-ed on any given pass is $(N – 1) / t = (N – 1) / (N(N – 1)/2) = 2/N$, and the probability of it not getting TP-ed is therefore $1 – 2/N$. As such, the probability of any given house not getting TP-ed at all across all M passes is $(1 – 2/N)^M$. Due to the property of linear expectation, the expected number of houses which won't get TP-ed at all is simply N times that value. Thus, the expected number of houses which will get TP-ed is $N – N(1 – 2/N)^M$.

Official Solution (C++)

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

int N, M;
long double DP[2005][4005];

int main() {
  cin >> N >> M;
  long long tot = (long long)N*(N - 1)/2;
  DP[0][0] = 1.0;
  for (int i = 0; i < M; i++) {  // i passes done
    // j houses TP-d
    for (int j = 0; j <= min(i * 2, N); j++) {
      // k houses not TP-ed.
      int k = N - j;

      // TP 0 new houses.
      long long c = (long long)j*(j - 1)/2;
      DP[i + 1][j] += DP[i][j]*c / tot;

      // TP 1 new house.
      c = (long long)j * k;
      DP[i + 1][j + 1] += DP[i][j]*c / tot;

      // TP 2 new houses
      c = (long long)k*(k - 1)/2;
      DP[i + 1][j + 2] += DP[i][j]*c / tot;
    }
  }
  // Add up expected value.
  long double ans = 0;
  for (int i = 0; i <= min(M * 2, N); i++) {
    ans += i * DP[M][i];
  }
  cout << fixed << setprecision(6) << ans << endl;
  return 0;
}
```