Problem Tutorial: "Alternating Serves"

This problem is fairly straightforward. The score itself is somewhat irrelevant. All we care about is the sum of Darcy's and Rajko's score. Every time the sum reaches a multiple of 5, the server switches. Simulating the game will work for this problem. However, a single calculation of

$$\left\lfloor \frac{d+r}{5} \right\rfloor$$

will be even if Darcy is serving and odd if Rajko is serving, where $\lfloor \cdot \rfloor$ is the "floor" function (rounds down).

Total Complexity: $O(1)$

Problem Tutorial: "Break From Training"

This is a pure simulation problem. You are told exactly what to do and you must code up all of the details correctly. Because there are only 108 tiles (and only 56 tiles left after dealing the tiles), code efficiency is not very important. However, there are several places that you could easily make a mistake, so make sure that you write your code nicely so that it is easy to follow!

Some easy places you may fail:

- Does your code say this hand is a winner? (It is)

  1B 2B 3B 1B 2B 3B 1B 2B 3B 1B 2B 3B 4B 4B

- Are you discarding the correct card for Daniel/Darcy? If Daniel's hand is

  1B 2B 3B 1B 2B 3B 1B 2B 3B 1B 2B 3B 4B 4C

  make sure he is discarding 4B and not 4C.

Total Complexity: $O(n \cdot k^2)$, where n is the number of tiles and k is the size of the hand. Note that there are many ways to solve this problem, each with slightly different complexities.

Problem Tutorial: "Crazy Email Chains"

It will be helpful to think about this problem as a directed graph. Each person is a vertex and there is an edge from person A to person B if person A tells you to contact person B. In this graph, we are asking how many vertices eventually lead you to a vertex with no out edge. Note that starting from each vertex and going until you find a cycle or an endpoint takes $O(n^2)$ time and is too slow.

However, note that any vertex that is reachable from vertex 1 has the same answer as vertex 1 (that is, they either both lead to a cycle or neither of them do). Thus, you should determine if you can find a cycle from vertex 1. Then you should mark every vertex reachable from vertex 1 with this answer. You then move on to vertex 2. If the result has already been computed, you move on to vertex 3. If it hasn't been computed, you compute the answer (and set every vertex reachable from vertex 2 to the same answer as vertex 2), then move on to vertex 3. Etc.

How do you determine if you reach a cycle from vertex v? There is at most one outgoing edge for each vertex, so you have no choice but to follow that edge if it is there. When you arrive at a vertex, you should place a marker to signify that you have visited that vertex. If you ever visit a vertex that has previously been visited, you have reached a cycle. If you ever reach a vertex that you already know the answer to, just take its answer!

The only tricky small item left is to deal with the names. To deal with the strings, you should familiarise yourself with `map` (for C++), `dict` (for Python) and `TreeMap` (for Java).

Total complexity: $O(nM \log n)$, where M is the maximum length of name in the input.

Comments from Darcy after the contest

I added in a couple test cases that I hoped would catch a team or two. As it turns out, it ended up catching almost 100 submissions! Here is one of those test cases:

```
2
Darcy -> available
available -> Darcy
```

Note that Darcy is **not** available! He is simply forwarding his emails to a person named `available`.

Problem Tutorial: "Deconstructed Password"

Here is a naive algorithm:

> Go through the string and find all locations where $a_i = n + 1$. Each of these locations need different characters. Since Bob's password only consists of lowercase letters, there must be at most 26 of these (if there are more than 26, then Alice must have made a mistake). Then, go from right-to-left in the string and fill in s_i with s_{a_i}.

The above idea is *almost* right. There are a couple places that you could go wrong: You must ensure that a_i is the index of the *next* occurrence of the letter, not just *some* occurrence of the letter. Secondly, you must ensure that a_i is strictly larger than i.

If you ensure that both of these are satisfied, then the naive algorithm above will work. To ensure that you are pointing to the *next* occurrence of the letter, you could mark that letter as used once some index points to it. Alternatively, you can just check that the only duplicates in the sequence is $n + 1$. If there is another integer i that is duplicated, then it is guaranteed that one of the indices is not pointing at the *next* occurrence of the letter.

Total Complexity: $O(n)$

Problem Tutorial: "Enumerating Trees"

This is definitely the hardest problem in the set. The experienced coders of the crowd will likely jump to the assumption that this is a Dynamic Programming problem, and they'd be right!

Note: I will be ignoring the modulo p everywhere in this explanation. Don't forget to do it!

Let T_k be the number of (non-isomorphic) rooted trees on k vertices. Obviously, $T_1 = 1$.

Imagine you have (somehow) generated a list of all (non-isomorphic) rooted trees:

$$\begin{aligned}
\text{Rooted trees on 1 vertex} &= \left[t_1^{(1)}\right] \\
\text{Rooted trees on 2 vertices} &= \left[t_1^{(2)}\right] \\
\text{Rooted trees on 3 vertices} &= \left[t_1^{(3)}, t_2^{(3)}\right] \\
\text{Rooted trees on 4 vertices} &= \left[t_1^{(4)}, \ldots, t_4^{(4)}\right] \\
&\vdots \\
\text{Rooted trees on 1000 vertices} &= \left[t_1^{(1000)}, \ldots, t_{T_{1000}}^{(1000)}\right]
\end{aligned}$$

Now imagine we are trying to compute T_k. If we had magically generated the lists above, we would know the answer already — it is just the length of the appropriate list! Unfortunately, the lists are way too big to actually generate. The first thing to notice is that each child of r (the root of the tree) is a root of a smaller tree. To ensure that we are not double counting, we will sort the root's children by the number of nodes in their sub-tree. If two children have the same number of nodes in their subtree, order them by which occurs first in the appropriate list above. If we do this for every node in the tree (sort their children in this fashion), then the tree is in *canonical form*. You can show that two trees have the same canonical form if and only if they are isomorphic.

So now the problem has been reduced to counting the number of canonical trees on k vertices. Assume we have already calculated $T_1, T_2, \ldots, T_{k-1}$. To calculate T_k, we need to determine how many of the root's children are in a subtree with 1 vertex, how many of the root's children are in a subtree with 2 vertices, ..., and how many of the root's children are in a subtree with $k-1$ vertices. (Note that a subtree of a child cannot have k vertices since the whole tree has k vertices and the root is one of them). Say you have decided that you want p subtrees of size i; then there are exactly

$$\binom{p + T_i - 1}{p}$$

choices for which p subtrees you could choose in canonical form (this is the choice with replacement formula or can be determined via Stars-and-Bars).

At this point, we almost have enough information to write our DP! What we will do is construct the trees from left-to-right by determining exactly the number of children of the root with subtrees of size i. Once we have determined that, we have a smaller subproblem. For example, if we want all trees of size 100 and we have decided that we will have 4 children with 1 vertex in their subtree, we are now in a smaller problem where we are counting the number of trees with 96 vertices ($100 - 4$), and all of the children of the root have subtrees of size at least 2.

So we finally arrive at our solution. I will not bother putting the Dynamic Programming into the code below. See the model solutions for an example of that.

```
// Counting number of trees on k vertices where every child of the
//   root is the root of a subtree of size at least "smallest"
int T(int k,int smallest){
  if(k == 1) return 1;

  // Number of subtrees of size "smallest" (T_i in the explanation above)
  int num_trees = T(smallest,1);

  int answer = 0;

  // num is the number subtrees of size "smallest" to use (possibly 0)
  for(int num = 0 ; num*smallest <= k-1 ; num++ )
    answer += choose(num_trees+num-1,num) * T(k - num*smallest , smallest+1);

  return answer;
}
```

(Assume that `choose` can be calculated in $O(1)$ — see below for a comment on that)

Naively looking at the code above, it seems that there are $O(n^2)$ states with a $O(n)$ loop inside giving a total complexity of $O(n^3)$. However, imagine if we fix k. The total number of times the `for` loop is executed in `T(k,1)`, `T(k,2)`, ..., `T(k,k-1)` is

$$\sum_{i=1}^{k-1} k/i = k \sum_{i=1}^{k-1} 1/i = O(k \log k).$$

Minor note: Above, I said to assume that `choose` can be calculated in $O(1)$. This can be accomplished as follows. We need to compute:

$$\binom{p+0-1}{0}, \binom{p+1-1}{1}, \binom{p+2-1}{2}, \dots$$

The first binomial is equal to 1. From there, we can calculate the next binomial using this formula:

$$\binom{n+1}{k+1} = \frac{n+1}{k+1}\binom{n}{k}.$$

Total Complexity: $O(n^2 \log n)$

Problem Tutorial: "Fine-Tuned Resistance"

Note that we do not need to get exactly the correct value, just *close enough*. There are several solutions to this problem, here is one of them:

Note that even though there are n different resistors, our solution must work with a single resistor ($n = 1$). Thus, we will assume that $n = 1$ always and use r_1 as our only resistor, even if we have more available to us.

Step 1 First, we will deal with the whole number part. Note that r_1 copies of r_1 will give us a resistance of 1.

$$\frac{1}{R'} = \sum_{i=1}^{r_1} \frac{1}{r_1} \implies R' = 1.$$

Use this parallel network as many times as you can until your target number is less than 1.

Step 2 Now our target number is between 0 and 1. If we can build parallel networks for

$$\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64} \text{ and } \frac{1}{128},$$

then we can simply use those! *How?* If the number is in the range $\frac{1}{2} \leq R < 1$, then we will use the $\frac{1}{2}$ network. If the number is not in that range (i.e., it is in the range $0 \leq R < \frac{1}{2}$), then we do nothing. Now note that the target value is less than $\frac{1}{2}$. We will repeat the same process: if the target is in the range $\frac{1}{4} \leq R < \frac{1}{2}$, we will use the $\frac{1}{4}$ network. Now R is in the range $0 \leq R < \frac{1}{4}$. If we do the same with $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{32}$, $\frac{1}{64}$ and $\frac{1}{128}$, then the remaining target will be in the range $0 \leq R < \frac{1}{128}$. Since $\frac{1}{128} < 0.01$, we are in the allowable range!

Now how do we actually build those networks? In a similar way to how we built the networks in step 1:

- The $\frac{1}{2}$ network is $2r_1$ copies of the r_1 resistor.
- The $\frac{1}{4}$ network is $4r_1$ copies of the r_1 resistor.

  $\vdots$
- The $\frac{1}{128}$ network is $128r_1$ copies of the r_1 resistor.

How many resistors have we used? In Step 1, we used at most $100r_1$ resistors (which is 1000 in the worst case). In Step 2, we used at most $2r_1 + 4r_1 + 8r_1 + \cdots + 128r_1 = 254r_1$ resistors (which is 2540 in the worst case). Thus, we have used at most 3540 resistors.

Total complexity: $O(r_1(R - log(maxerror)))$ Note that $-log(maxerror)$ is an approximation to the number of different powers of 2 you need to get the desired accuracy. $maxerror$ is 0.01 for this problem.