Statistics

Match Editorial

SRM 147
Thursday, May 22, 2003

Archive
Normal view
Discuss this match

Match summary

This match provided some very impressive performances, perhaps **SnapDragon** and **Yarin** have some new competitors coming up the ranks. **bladerunner** continues to improve his rating while maintaining his 100% success rate in both challenges and submissions. **overwise**, a newcomer to competition, must have spent his 6 months of TopCoder membership practicing, because he managed to get spots on both the Highest Match Total and Impressive Debut lists with a score of 1846.62.

The Problems

CCipher Discuss it

Used as: Division-II, Level 1:

Value	250
Submission Rate	199 / 220 (90.45%)
Success Rate	179 / 199 (89.95%)
High Score	**overwise** for 248.71 points

Whenever you have a situation where you want to wrap around from the end of something to the beginning, you should use the modulus operator. For example, in this case, there are 26 things, and we want to move backwards, and wrap around if we get to the beginning. So, the first thing to do is convert the character into a number between 0 and 25. We can do this by subtracting 'A' from the actual character. 'B'-'A' = 1, and so on. Then, when we have converted it into a number, we subtract shift from that number. Now, this may give us a negative number, in which case we have to add 26. But, if it gives us a positive number, we don't want to add 26. One elegant way to deal with these two cases is to always add 26, and then take the result mod 26 at the end. If the subtraction gave a negative number, the modulus operation will not do anything. If the subtraction gave a non-negative number, then the modulus operation will have the same effect as subtracting 26. So, this leads us to the short solution below (this is in Java, but all languages are similar for this problem):

```
char[] c = cipherText.toCharArray();
for (int i = 0; i < c.length; i++) c[i] = (char)((((c[i]-'A')+26-shift)%26)+'A');
return new String(c);
```

PeopleCircle Discuss it

Used as: Division-II, Level 2:

Value	600
Submission Rate	98 / 220 (44.55%)
Success Rate	49 / 98 (50.00%)
High Score	**overwise** for 588.79 points

Used as: Division-I, Level 1:

Value	350
Submission Rate	93 / 110 (84.55%)
Success Rate	67 / 93 (72.04%)
High Score	**Yarin** for 341.15 points

This problem looks sort of hard, because it seems like we have to do some complicated working backwards. However, if you look carefully at the examples, you should get an idea of an easy way to solve the problem. First off, we know how many people there are total: `numFemales + numMales`. So, start with that many people. Now, it is not too hard to figure out which ones are removed. We simply implement the removal algorithm, marking all of the people removed as we go. Then, all of the removed people are females, and hence we know exactly which people are males and which are females. So, the tricky part then, is how do we implement the removal algorithm. The best way is to start with an array of characters, and initialize them all to 'M'. Then, as we remove people, mark them as 'F'. Then, convert the character array to a string, and return it. Now, to do the removal, we should basically just follow the instructions. As noted in the analysis for the problem above, we should use the modulus operator to wrap around. Then, we just keep track of where we are, and have two nested loops. Note that starting `pos` where we do here makes the rest of the code a little simpler.

```
   char[] ret = new char[numMales+numFemales];
   Arrays.fill(ret,'M');
   int pos = ret.length-1;
   for (int i = 0; i < numFemales; i++) {
      for (int c = 0; c < K;) {
         pos = (pos+1)%ret.length;
         if (ret[pos]=='M') c++;
      }
      ret[pos] = 'F';
   }
   return new String(ret);
```

GoldenChain Discuss it

Used as: Division-II, Level 3:

Value	950
Submission Rate	64 / 220 (29.09%)
Success Rate	41 / 64 (64.06%)
High Score	**overwise** for 909.12 points

The tricky part of this problem is to figure out the correct approach. Once you do that, coding it is relatively simple. First off, its clear that if we open up one link for each section, we will have enough to make a loop. However, in opening up this number of links, it might turn out that we can open up every link in a section, and thus have one less section. So, the idea then is to open up every single link from as many sections as possible, up to the point where we have enough links to put everything together. So, in pseudocode, that would go something like this:

```
while(there are not enough open links to form a loop)
    open up one loop from the shortest section
end while
```

We could implement this without too much trouble. First sort the elements of sections, and then start opening links, and keeping track of how many sections are completely opened. This is probably the most obvious and bug free way to do it. But, if you want to code it really fast, and make it really short, you can do a little better. Rather than opening one link at a time, we can explicitly figure out how many we should open from each section. There are only 2 cases here. In the first case, we open up every link in a section. We do this whenever the number of remaining sections after opening all of the links in one section is greater than or equal to the number of links we will have after doing so. That is the first case in the code below. Otherwise, we aren't going to open up every single link in the section. Instead, we open up a number of links equal to the number of remaining sections, including the current one. It takes a little tinkering to get everything exactly right - no off by 1 errors.

```
   Arrays.sort(s);
   int links = 0, i;
   for(i = 0;i<s.length; i++){
      if(s.length-i-1>=s[i]+links)
         links+=s[i];
      else
         links+=Math.max(s.length-links-i,0);

   }
   return links;
```

Dragons

Discuss it

Used as: Division-I, Level 2:

Value	500
Submission Rate	67 / 110 (60.91%)
Success Rate	32 / 67 (47.76%)
High Score	**dgarthur** for 458.86 points

This is a pretty straightforward simulation - nothing like last week's. There are some dragons, who are stealing food from each other. You are told how they steal food from each other, and how much each one starts with, and have to return how much one of them has after a given number of rounds. The first thing to do here is set up an adjacency matrix representing which dragons steal from which. For example (each 1 represents that the dragon in that row steals from the dragon in that column).

```
int[][] adj = {   {0,0,1,1,1,1},
      {0,0,1,1,1,1},
      {1,1,0,0,1,1},
      {1,1,0,0,1,1},
      {1,1,1,1,0,0},
      {1,1,1,1,0,0}};
```

This makes it pretty simple to do the stealing. For each dragon, we give him an amount of food equal to the sum of 1/4 of the food from each of his neighbors. We can do this with a couple of nested for loops, by using our adjacency matrix. The tricky part though, is that we can't simply use doubles, because the return requires that we use fractions (we actually could use long doubles, but only in C++, since 64 bit doubles are quite accurate enough). One simple way to handle this is to use a common denominator for all of the amounts of food and then just keep track of the numerators, and multiply the denominator by 4 each time. So, the numerator of the amount that a dragon gets will just be the sum of the numerators of the amounts that the adjacent dragons had, and the denominator can just be multiplied by 4. However, 4^45 won't fit in a long, so we have to do some reducing along the way. Now, if we think about it a little, its pretty obvious that, after each round, dragons on opposite sides will have the same amount of food, since they are both stealing the same amounts from the same dragons. Thus, after the first round, each dragon will get the same amount of food from 2 sets of 2 people. So, the up dragon will get the same amount of food from the left and right dragons, and the same amount of food from the front and back dragons. Thus, the if we add up all of the numerators, they must be divisible by 2. So, rather than multiplying the denominator by 4 each time, we can safely multiply it by 2, and halve the sum of the numerators. (We can't do this the first time, because dragons on opposite sides may have a different amount the first time.) 2^45 will fit in a 64 bit datatype with no problem, so that would work fine. Then at the end, we reduce the fraction and return it. Putting it all together:

```
long[] num = new long[6];
long de = 2;
for(int i = 0; i<6; i++){num[i] = 2*initialFood[i];}
for(int i = 0; i<rounds; i++){
   de*=2;
   long num2[] = new long[6];
   for(int j = 0; j<6; j++){
      for(int k = 0; k<6; k++){
         if(adj[j][k]==1)num2[j]+=num[k];
      }
      num2[j]/=2;
   }
   num = num2;
}
long rnum = num[2] / gcd(de,num[2]);
long rde = de / gcd(de,num[2]);
if(rde==1)return ""+rnum;
return rnum+"/"+rde;
```

A couple of notes on this. You don't need a proper GCD function, since the denominator is always a power of 2. You can just divide out all of the twos. Also, if you wanted to, you could use C++'s long doubles to solve this problem. Long doubles are precise when the denominator of the fraction is a power of 2, and isn't too big. The above solution can be modified pretty easily to work with long doubles, which have just barely enough precision.

Flags Discuss it

Used as: Division-I, Level 3:

Value	1000
Submission Rate	13 / 110 (11.82%)
Success Rate	12 / 13 (92.31%)
High Score	**tjq** for 772.00 points

At first glance, this problem looks like a pretty typical graph/dynamic programming problem. We can use dynamic programming to figure out how many flags there are ending with a certain color, with a certain number of stripes. We can use this information to figure out the numbers for all of the colors with one more stripe. For example, if color C may only come after color A or color B, then the number of flags with n+1 stripes, the last of which is color C, is just the sum of the number of flags with n stripes, the last of which is color A or color B. So, if the number of stripes is sufficiently small, this dynamic programming approach will work easily. However, the examples show that the number of stripes required could be very high. But, this only happens when no color can be followed with more than 1 other color. If any color can be followed by more than 1 other color, the number of flags grows exponentially, and the number of required stripes is small. Otherwise, if every color is followed by at most one other color, then for every amount of stripes greater than 1, the number of flags will be constant. So, we can use a little division to figure out exactly how many are required. You can look at pretty much any successful solution to see this implemented.

By **lbackstrom**
TopCoder Member