Statistics — Match Editorial

SRM 148
Wednesday, May 28, 2003

Archive
Normal view
Discuss this match

Match summary

SRM 148 went off without a hitch. The division one set featured a deceivingly hard 1100 point problem that stumped most of the coders. **SnapDragon**, who won the match, finished the first two problems before most coders finished the first. He successfully finished every problem quicker than all other coders. **SnapDragon** was on pace to complete the entire round in 30 minutes until he ran into the 1100 point problem. Numerous other coders also sped through the first two, later to be stopped in their tracks by the hard. Once the challenge phase ended very few 1100 submissions still remained. In division two, a new coder by the name **bogklug** scored 1733.52 winning his division by a wide margin. Division two coders did remarkably well on a relatively difficult set lending credence to the theory that their average ability is increasing steadily.

The Problems

DivDigits Discuss it

Used as: Division Two - Level One:

Value	250
Submission Rate	197 / 211 (93.36%)
Success Rate	165 / 197 (83.76%)
High Score	**Sleeve** for 249.49 points

The key to this problem is to first represent the input integer as a string. Once that is complete, loop through each character and test for divisibility. Just be sure not to try 0, since your divisibility check may throw an exception.

CeyKaps Discuss it

Used as: Division Two - Level Two:

Value	600
Submission Rate	161 / 211 (76.30%)
Success Rate	143 / 161 (88.82%)
High Score	**MadProgrammer** for 589.08 points

This problem resembles a topic in algebra called permutation groups. The permutation in this problem has been represented as a series of key swaps. There is actually a theorem of algebra that says any permutation can be represented as a list of swaps. Anyways, lets look at one possible set of swaps, and use that analysis to come up with an algorithm. Lets say the key that was produced as output was an 'A' and the swaps were "A:B","C:D","B:C","C:D","A:E". This means the key that produces the letter 'A' initially had an 'A' cap. That 'A'-cap became a 'B'-cap after the first swap. The second swap did not affect it. The third swap changed that 'B'-cap to a 'C'-cap. The fourth changed the 'C'-cap to a 'D'-cap. T he last didn't affect it. Thus, the cap that ended on the 'A'-key was the 'D'-cap so the user hit 'D' to produce the 'A'. Repeating this process on each output letter produces the return value. Java code to do this looks like:

```
public String decipher(String typed, String[] switches) {
   String hmm = "";
```

```
   for (int i =0; i <typed.length(); i++) {
      char curr = typed.charAt(i);
      for (int k = 0; k < switches.length; k++) {
         if (switches[k].charAt(0)==curr)
            curr=switches[k].charAt(2);
         else if (switches[k].charAt(2)==curr)
            curr=switches[k].charAt(0);
      } hmm+=curr;
   } return hmm;
}
```

MNS Discuss it

Used as: Division Two - Level Three:

Value	1000
Submission Rate	52 / 211 (24.64%)
Success Rate	36 / 52 (69.23%)
High Score	**bogklug** for 940.69 points

Used as: Division One - Level Two:

Value	450
Submission Rate	114 / 151 (75.50%)
Success Rate	95 / 114 (83.33%)
High Score	**SnapDragon** for 447.54 points

Given the numbers, simply try all permutations and check each to see if it is valid. Validity amounts to checking whether the row sums and column sums are equivalent. To remove duplicates you can either use a set data structure that will check automatically, or you can divide out to remove potential duplicates. For example, if the number 3 occurred 4 times, dividing by 4! removes all duplicates associated with swapped 3's. To actually generate the permutations, a recursive algorithm that tries each order does the trick. C++ coders can use the built in next_permutation function for a similar effect. As an alternate method, Java users can implement an iterative next permutation function much like the one found in the C++ library. Such an implementation follows:

```
//Members of vals must be distinct
//Based on C++ next_permutation function
int[] nextperm(int[] vals) {
   int i =  vals.length-1;
   while (true) {
      int ii =  i;
      i--;
      if (vals[i] < vals[ii]) {
         int j = vals.length;
         while (vals[i] >= vals[--j]);
            int temp = vals[i];  //Swap
            vals[i] = vals[j];
            vals[j] = temp;
         int begin = ii, end = vals.length-1;
         while (end>begin) {
               int stemp = vals[end];   //Swap
               vals[end] = vals[begin];
```

```
               vals[begin] = stemp;
            end--; begin++;
         }
         return vals;
      } else if (vals[i] == vals[0]) {
         int begin = 0, end = vals.length-1;
         while (end>begin) {
               int stemp = vals[end];   //Swap
               vals[end] = vals[begin];
               vals[begin] = stemp;
            end--; begin++;
         }
         return vals;
      }
   }
}
```

Since nextperm only works on lists of numbers without duplicates we will use a sort of hack to get it to work. Generate an array containing the numbers 0-8. Permute that array to get all permutations. Use the permuted list of numbers at any one point as the ordering to impose on the actual list that may contain duplicates.

Circlegame Discuss it

Used as: Division One - Level One:

Value	250
Submission Rate	143 / 151 (94.70%)
Success Rate	104 / 143 (72.73%)
High Score	**SnapDragon** for 243.96 points

Given a set of cards, and two operations, your code should repeat those operations until a state has been reached where the operations cannot be performed. You could try to perform as many operations as possible on each iteration, but that may complicate the code. Easier is to do perform 1 operation each time, and just start all over again. This basically consists of: Search for a 'K', if you find it remove it and start all over, if not search for a consecutive 13 pair, remove it, and start all over. Repeat till you can't find any more.

NumberGuessing Discuss it

Used as: Division One - Level Three:

Value	1100
Submission Rate	12 / 151 (7.95%)
Success Rate	5 / 12 (41.67%)
High Score	**SnapDragon** for 496.57 points

At first glance, you realize the problem is going to be some sort of recursion. Each player that follows you is going to employ a similar strategy on a smaller data set. Even the constraints sort of screamed recursion (range^numLeft<1000000). The way I see it, there were two tricks that drastically increase the difficulty of this problem. The first is, running time. Even with the constraints, a pure bf solution will probably time out. In addition, data must be passed backwards and forwards through the recursion adding another element of difficulty. That said, there are also many rounding issues to deal with when comparing various guesses.

The recursion splits into two cases: Someone guesses after you, or you are the last one to guess. If you are the last one to guess, then you should have a list of everyone else's guesses. Lets say the following line represents the range of numbers, with X's representing previous guesses:

```
------X--------X-------X-----X-X-----
```

The key insight is that the number of places you have to check is very limited. I will mark these places with O's:

```
-----OXO-------XO------XO----XOXO----
```

In other words, the only places to check are right after all previous guesses, or right before the first guess. Simply take the one that produces that best score. As for the other case, you will guess all possible places that haven't been guessed already, and then recursively call the next guy. Once that call returns, you will have a list of future guesses parameterized by your current guess. Evaluate your score on each possibility and return the best, lowest one.

Now comes the score evaluation part. Lets you are looking at a range between two guesses like:

```
-------X-------X----------
         here
```

The score of any choice in that range is the number of spots between the X's plus 1 divided by 2. Here it would come out to (7+1)/2 = 4. On the other hand lets say your guess falls on an end:

```
-------X-------X----------
 here
```

In this case compute the same calculation as before from where you are, to the next closest guess. Then add to that all of the spots before you. For example:

```
------O----X--------...
```

Lets say your guess is the O and the X represents the closest guess. Including where you guessed, there are 5 spots from the O to the the X that should be calculated as above ((5+1)/2 = 3). Then there are the 6 spots before you that just get added on. (3+6=9). So the score for this guess is 9.

By **brett1479**
TopCoder Member