TCHS SRM 2
Monday, June 12, 2006

Archive
Normal view
Discuss this match

Match summary

The second TopCoder High School Single Round Match, first one for the Beta region, attracted 109 registrants (59 of them newcomers) and proved to be quite eventful. With challenge opportunities left in all 3 problems, the heat was rising as the end of the intermission approached. When the coding phase ended there were 209 problems open for challenge, 94 250-pointers, 77 500-pointers and 38 1000-pointers. The first few minutes of the challenge phase were a real bloodbath as 250-pointers failed all around to the dreaded **poison = elixir** case. All in all, during the challenge phase 24 250-pointers were brought down along with 10 500-pointers and 12 1000-pointers. This amounts to a grand total of 46 successful challenges (the systests took down only 21 solutions!).

The first place goes to **Burunduk1** from Russian Federation representing Saint-Petersburg Phys-Math Lyceum #30, the first ever TCHS red! Congratulations! Second place goes to **Weiqi** from China representing Mingzhu Middle School, and third place goes to a newcomer **dzhulgakov** from Ukraine representing National Technical University KhPI. A great thing about this SRM is that, even with quite hard problems, only a few people ended with a non-positive score. Congratulations to all competitors!

The Problems

FountainOfLife

Rate It Discuss it

Used as: Level One:

Value	250
Submission Rate	94 / 107 (87.85%)
Success Rate	69 / 94 (73.40%)
High Score	**sluga** for 248.85 points (1 mins 55 secs)
Average Score	209.93 (for 69 correct submissions)

The math solution

This problem was a good example of a pen and paper friendly problem. The best way to deal with this problem is to put your keyboard away and write down the problem on paper.
The mixture becomes deadly once the volume of the poison is at least 50%. This is just a complicated way of saying that the volume of the poison needs to be at least equal to the volume of elixir. At any given time *t* there is **poison** * *t* liters of poison and **elixir** * *t* + **pool** liters of elixir. We want the following to be true:

```
t >= pool / (poison - elixir)
```

pool will always be positive and (**poison** - **elixir**) can be either positive, negative or zero. If (**poison** - **elixir**) is negative or zero, than the mixture will never become deadly since **pool** will be at least 1. If (**poison** - **elixir**) is positive then we need to find the minimal *t* for which the above holds true. Because **pool** / (**poison** - **elixir**) is a positive number and *t* must be at least equal to it, the minimal *t* is in fact equal to **pool** / (**poison** - **elixir**). Once you have this writen on the paper, coding it is quite simple. The following code solves the task:

```
   public double elixirOfDeath(int elixir, int poison, int pool){
if ( poison <= elixir ) return -1.0;
return 1.0 * pool / (1.0 * (poison - elixir) );
   }
```

The most common error on this problem was the lack of '=' in the if, causing either wrong return value or a division by zero.

ApocalypseSomeday Rate It Discuss it

Used as: Level Two:

Value	500
Submission Rate	77 / 107 (71.96%)
Success Rate	61 / 77 (79.22%)
High Score	**Penguincode** for 496.83 points (2 mins 16 secs)
Average Score	373.68 (for 61 correct submissions)

The Brute-force solution

One way to approach this problem was to simply iterate number by number, starting from 666, and counting all beastly numbers until we find the **n**-th. As **n** was quite large, it wasn't obvious that this will work in time. One way a coder could have realized the code will work in time is by considering the number 1000666. It is easy to see that there are 1000 beastly numbers prior to 1000666 of the same type (namely 666, 1666, 2666.....999666). Now the same reasoning applies to the number 1006660, and to 1066600, and to 1666000. So we already have 4000 beastly numbers below 1666000. Multiply this by 3 and you have 12000 beastly numbers below 3666000. This can easily be iterated in time.

Note: This is just an approximation, but a good one to use in the heat of the contest. In fact there are 5500 beastly numbers below 1666000. Try to find the error and a more correct approximation.

The following code is an implementation of this method.

```
    public int getNth(int n){
        int x = 665;
        while ( n > 0 ){
            ++x;
            if ( Integer.toString(x).indexOf("666") != -1 ) --n;
        }
        return x;
}
```

The O(n) solution

Another way to approach this problem is to further expand the idea that we used to examine the runtime of the BF solution. Beastly numbers can be divided into patterns, like ****666, ***666*, etc. For every pattern let us remember the last number we used to fill in the * in an array *pat*. In order to find the next beastly number, we simply iterate the array *pat* and use the next number for that pattern to form a beastly number, constantly keeping the smallest number found thus far. Since there is a fixed number of patterns and you will iterate them **n** times, the complexity is O(**n**). The exact details and the code for this solution are left as an exercise to the reader.

For testing purposes the 100,000th beastly number is 22,230,666, the 1,000,000th is 177,966,663 and the 10,000,000th is 1,666,008,549.

Wizarding Rate It Discuss it

Used as: Level Three:

Value	1000
Submission Rate	38 / 107 (35.51%)
Success Rate	12 / 38 (31.58%)
High Score	**Weiqi** for 822.55 points (13 mins 49 secs)
Average Score	610.76 (for 12 correct submissions)

The O(3^n) recursive solution

One way to solve this problem is to try out all possible combinations of keeping, deleting or replacing the characters

of the spell. There are at most 3^{13} combinations so the code will run in time. This approach can be easily implemented with a recursive function. The parameters for our function will be an int, the character we are currently processing and a string, the counterspell we are currently forming. This is not the only way you could implement this solution, other coders used different designs, some with more success than others. I recommend checking their solutions in the arena. Here is a well commented code:

```
string s, r;
int bestpower;



public void solve(int depth, String c){
    if ( depth == s.length() ){
        // check if we have an empty incantation
        if ( c == "" ) return ;

        // we have a candidate for the counterspell so we check it's power

        int power = 1;
        for (int i = 0; i < c.length(); i++ ){
            power *= (c.charAt(i) - 'A' + 1);
            power %= 77077;
        }



        // compare the current counterspell and the best found thus far



        if ( power > bestpower ){
            bestpower = power;
            best = c;
        }
        else if ( power == bestpower ){
            if ( c.length() < best.length() ) best = c;
            else if ( c.length() == best.length() && c.compareTo(best) < 0 ) best = c;
        }
    }
    else {
        // first use the letter as-is
        solve(depth + 1, c + s.charAt(depth));

        // next try to delete it
        solve(depth + 1, c);

        // now try to replace it
        if ( r.charAt(s.charAt(depth) - 'A') != '-' )
            solve(depth + 1, c + r.charAt(s.charAt(depth) - 'A'));
    }
}



public String counterspell(String spell, String rules){
    bestpower = -1;
    s = spell;
    r = rules;
```

```
        // start from the first character with an empty counterspell
        solve(0, "");
        return best;
    }
```

The non-recursive solutions

Recursion was not necessary to solve this problem. Non recursive solutions exist. Check **nordom**'s, **kupicekic**'s, **marek**'s and **Burunduk1**'s contest solutions for examples. Those solutions are not all the same -- they implement different algorithms. Some of those are quite difficult to come up with and code, and it would be too much to examine in detail each and every one of them, but here are a few useful observations:

- Notice the usage of switch(t % 3) in **nordom**'s solution. Essentially, it is looking at every possible solution as a number in base 3, and than uses the % to determine the operation that is performed, and / to perform a base 3 analog to binary bitwise rightshift. However the % operation is quite slow, so this code is actually slower than the recursion.
- Instead of base 3 and the slow %, / operators, much faster bitwise & and >> operators may be used. First, you encode every possible keep-delete decision in a binary number, and than for all possible keep-delete decision you encode all possible keep-replace decision in a new binary number. This was used in **kupicekic**'s solution.

The most common error on this problem was returning an empty string.

By **ivankovic**
TopCoder Member