Statistics

Match Editorial

Archive
Printable view
Discuss this match

SRM 216
Monday, October 18, 2004

Match summary

It was not an easy night for Division 1 participants as all three problems proved both unusual and challenging, each in their own way. Many strong competitors were thrown off by this and ended up with scores of 0, which makes **tomek**'s intimidating 1641.97 points all the more impressive - congratulations! He was followed by fellow Pole **Eryx** in second, and **kalinov** in third. It was also nice to see a new face among the top scorers with **ivan_metelsky** finishing in fourth place in his fourth TopCoder competition.

There were many more submissions in Division 2, but the system tests were anything but easy. It was a tight race near the top as **AnonnymousT** took first place with three successful challenges, followed by newcomers **zjq** and **tesh11**.

The Problems

CultureShock

Rate It Discuss it

Used as: Division Two - Level One:

Value	250
Submission Rate	188 / 210 (89.52%)
Success Rate	127 / 188 (67.55%)
High Score	**Sleeve** for 249.19 points (1 mins 37 secs)
Average Score	214.81 (for 127 correct submissions)

The most natural approach to this problem is to split the given text into an array of words, and to then check if each word is "ZEE". You can then reconstruct the string by concatenating the words, adding spaces as necessary. In C++, this can be done as follows:

```
string result, temp;
istringstream in(text);
while (in >> temp) {
    if (temp == "ZEE")
        result += " ZED";
    else
        result += " " + temp;
}
return temp.substr(1);
```

In Java, you can use StringTokenizers, following **Sleeve**'s solution, or you can use the very handy split function. If you do not use built-in parsing tools, you have to be careful that you only replace complete words, an oversight that accounted for many of the incorrect submissions for this problem.

If you a real guru, you can actually solve the problem in a single line. Here is one such solution:

```
return (" "+text+" ").replaceAll(" ZEE "," ZEE ").replaceAll(" ZEE "," ZED ").replaceAll("  "," ").trim();
```

Spaces are added at the beginning and end of text as advised here. I leave it as a little puzzle to figure out why the first and third `replaceAll` calls are needed. By the way, for those wondering who Bob and Doug are, I refer you to Google.

RockStar

Rate It Discuss it

Used as: Division Two - Level Two:

Value	500
Submission Rate	112 / 210 (53.33%)
Success Rate	56 / 112 (50.00%)
High Score	**ged** for 470.75 points (7 mins 10 secs)
Average Score	309.92 (for 56 correct submissions)

Used as: Division One - Level One:

Value	250
Submission Rate	141 / 165 (85.45%)
Success Rate	96 / 141 (68.09%)
High Score	**Eryx** for 247.02 points (3 mins 7 secs)
Average Score	196.71 (for 96 correct submissions)

This problem is as much about logic as it as about coding, a fact that threw off many competitors. For such a problem, the first thing you need to do is just organize your thoughts. In this case, a few observations turn out to be key:

- If you can ever play a song that starts fast, you might as well immediately play all ff songs that start fast and end fast. This is because playing one of these songs does not in any way affect what you can do in the future.
- Similarly, if you can ever play a song that starts slow, you might as well immediately play all ss songs that start slow and end slow.

- After this, you essentially have no choices. If you have a song that starts fast and ends slow, you must play it first. You can only follow this with a song that starts slow and ends fast, which can only be followed by a song that starts fast and ends slow, etc.

Based on these observations, you can break the problem into several cases, each of which can be dealt with easily:

```
if (fs+ff > 0) {
    if (fs == 0) return ff;
    else if (sf >= fs) return ff + ss + 2*fs;
    else return ff + ss + 2*sf+1;
} else
    return ss + min(sf, 1);
```

Of course, it is easy to make a mistake on one or more of these cases, even with the fairly broad range of examples. I did not really see any pattern in the mistakes here - you just had to be careful.

Although it is probably not very helpful for solving the problem, the scenario can also be related to the question of Eulerian paths on a multi-graph with two vertices (fast and slow), where each song corresponds to a directed edge. Using this insight, you can actually efficiently solve a more general version of the problem where there are more than two possible song speeds.

TournamentRanker Rate It Discuss it

Used as: Division Two - Level Three:

Value	1000
Submission Rate	46 / 210 (21.90%)
Success Rate	26 / 46 (56.52%)
High Score	**kmd-10** for 784.06 points (15 mins 50 secs)
Average Score	570.62 (for 26 correct submissions)

Not surprisingly, the key to solving this problem is being able to determine which of two competitors should be ranked higher. There are a couple ways of approaching this, but I think the cleanest is to follow the problem directions precisely as given. First of all, make sure you know the number of wins that each competitor has, and to whom each competitor lost. Then, you can translate the ranking description almost directly into a recursive function. In C++, for example, it might look like this:

```
map<string, int> wins;
map<string, string> lostTo;
bool isRankedHigher(const string &s1, const string &s2) {
    if (wins[s1] != wins[s2])
        return wins[s1] > wins[s2];
    else
        return isRankedHigher(lostTo[s1], lostTo[s2]);
}
```

You could then sort the competitors in one line: `sort(names.begin(), names.end(), isRankedHigher)`. This can also be done in Java, although the syntax is a little more cumbersome. See, for example, **Rakot**'s solution.

If you are not comfortable with recursion, one other approach is to build the sorted list one competitor at a time. You only consider adding competitors with a maximal number of wins, and then the competitors they lost to will have already been sorted, so they can be compared directly without recursion. See **theMadhouse**'s solution for a variation on this method.

Refactoring Rate It Discuss it

Used as: Division One - Level Two:

Value	500
Submission Rate	36 / 165 (21.82%)
Success Rate	29 / 36 (80.56%)
High Score	**battyone** for 483.19 points (5 mins 20 secs)
Average Score	351.68 (for 29 correct submissions)

It only takes one quick look at the statistics to see that this was an unusual problem. With only 29 correct solutions, it was clearly not easy. On the other hand, **battyone** solved it really fast. So how do you do it?

I think there are three things you have to realize. First of all, you have to see that this is a level 2 problem, not a level 3 problem. You are not going to need crazy math, so you should not even be trying that.

The next thing you should realize is that factorizations and recursion go very well together. Specifically, if $n = a * m$, then $a * m$ is a factorization of n, as is a multiplied by any factorization of m. Indeed, we can find all factorizations of n by iterating over all of its divisors a, and then recursively finding all factorizations of n/a. Unfortunately, this will find certain factorizations more than once. For example, 2*3*4 and 4*3*2 might both be counted, even though they are the same factorization. To solve this, it suffices to generate the factors in non-decreasing order. This can be done as follows:

```
int count(int n, int lastFactor) {
    int result = 0;
    for (int a = lastFactor; a*a <= n; a++)
    if (n % a == 0)
        result += count(n/a, a) + 1;
    return result;
}
```

So what is the final thing you have to realize? Just that this solution is already fast enough! This is not at all obvious, but the examples give you the worst case, so you can easily check. Indeed, written in Java, this code never takes more than about 4.5 seconds.

If, for some reason, your program is not quite fast enough, there are still a couple optimizations you can try. If you precompute the factors of n, and then use memoization with a hash table, for example, you can make everything run in under a second.

Roxor Rate It Discuss it

Used as: Division One - Level Three:

Value	1000
Submission Rate	8 / 165 (4.85%)
Success Rate	6 / 8 (75.00%)
High Score	**tomek** for 827.08 points (13 mins 35 secs)
Average Score	662.01 (for 6 correct submissions)

I was not sure if anybody would solve this problem, and if I saw a way to make it easier, I gladly would have taken it. Although one competitor confided that "I can do little more than cry in the face of this problem", it actually ended up being a little easier than I expected.

For two-player games like this, it is helpful to think in terms of winning and losing *positions*. Specifically, if two perfect players are playing, then either the first player wins or he loses. If he wins, call the starting position *winning*; otherwise call it *losing*. It is then easy to check that (a) all moves from a losing position change the game to a winning position, and (b) winning moves are precisely those moves that change the game from a winning position to a losing position.

Thus, it is helpful here to find winning and losing positions, and to worry about winning moves from there. So how do you do that? It turns out that example 1 is a big, big hint, and the key is reducing modulo 2 the number of stones in each pile. Specifically, given a position P, we let P' denote the same position with the number of stones in each pile reduced modulo 2 to either 0 or 1. Then, call P *bad* if P' is a losing position. We claim that if player 1 is in a bad position P, then player 2 can ensure that player 1 will always be in a bad position. To see this, consider any move that player 1 could make:

Case 1: Suppose he does not remove the last stone from a pile. Then, player 2 can simply copy his move to get to a new position Q that also reduces to P'. Since P' is a losing position, Q is a bad position, and player 1 is still stuck.
Case 2: Suppose he removes the last stone from some pile to get to position Q. Then, that pile must have had exactly one stone in it, so he could have done the same move from position P' to get to position Q'. Now, we assumed P' was a losing position, so Q' must be a winning position, and thus there must exist a move from there that takes the board to a losing position, R'. Finally, this same move can always be done from position Q to get to position R. Since R' is a losing position, R is a bad position, and again, player 1 is still stuck.

In particular, note that player 2 always has a legal move after any of player 1's moves, so player 1 cannot possibly make the last move, and thus, cannot win. It follows that every bad position is, in fact, a losing position. From here, it is easy to see that, in fact, P is a winning position if and only if P' is a winning position!

This turns out to be a very important observation, as it reduces the game to just 2^{15} states (since there are 2 meaningful possibilities for the number of stones in each pile). At this point, one can just do a complete search over all positions using memoization. If we represent each state as a 15-bit integer, for example, we could check if a position is a winnable like this:

```
map<int,bool> isWinnable;
boolean getIsWinnable(int pos, int n) {
    if (isWinnable.count(pos))
        return isWinnable[pos];

    boolean result = false;
    for (int i = 0; i < n; i++)
    if ( (pos & (1<<i)) != 0)
    for (int j = i+1; j < n; j++)
    for (int k = j; k < n; k++)
        result |= !getIsWinnable(pos ^ (1<<i) ^ (1<<j) ^ (1<<k), n);

    isWinnable[pos] = result;
    return result;
}
```

Finding the appropriate winning move with this information is fairly straightforward. See **kalinov**'s solution for a very clean implementation.

As a random caveat, I did not have a fully mathematical solution to this problem (ie without the memoized search at the end), and I did not really expect there to be one. Although five of the six correct solutions to the problem used the method described here, it looks like **ivan_metelsky**'s solution is purely mathematical. I have not had a chance to look at it yet, but it should give another interesting approach.

By **dgarthur**
TopCoder Member

Apr 22 @ClintonBon – Clinton Bonner We know what you're thinking. Great, another 'puff piece' on the 'wisdom of crowds' and how all we need to do is post...Read More

View More

About Us

Get Connected

Your email address

Submit