Statistics

Match Editorial

Archive
Printable view
Discuss this match

SRM 223
Saturday, December 18, 2004

Match summary

Coder **marian** finished in first place in Division One, with the fastest time on the 1000-point problem. In the final seconds of the challenge round, **Yarin** was within 50 points of the top spot. But a successful defense by **Jan_Kuipers** knocked **Yarin** down to third, and **Jan_Kuipers** finished second.

The fastest time on the 1000-point problem combined with a successful challenge was enough to earn **herbert_deuarte** Division Two's top spot in his third SRM. Coder **tc-chung** finished closely behind in second place, with **ibonaci** in third. All three of these coders moved up to Division One.

The Problems

MostProfitable Rate It Discuss it

Used as: Division Two - Level One:

Value	250
Submission Rate	190 / 199 (95.48%)
Success Rate	166 / 190 (87.37%)
High Score	**Fabi** for 247.73 points (2 mins 43 secs)
Average Score	215.81 (for 166 correct submissions)

For each item you are given the cost, the price, and the number of sales. The total profit for each item is:

(price - cost) * sales

To find the most profitable, you simply loop over all the items, compute the profit (using the formula above), and keep track of the index of the item with the highest positive profit. At the end of your method, you return the element of items with the index of the most profitable item, or the empty string if no items were profitable.

Two things you must watch out for are to ignore any items with zero or negative profit, and if there is a tie for the most profitable item you are to return the item with the lowest index. Tiebreaking rules such as this are common in TopCoder problems, in order to guarantee that a problem has a well-defined unique answer.

See **tc-chung**'s solution for a clear, straightforward implementation.

BlackAndRed Rate It Discuss it

Used as: Division Two - Level Two:

Value	500
Submission Rate	184 / 199 (92.46%)
Success Rate	171 / 184 (92.93%)
High Score	**ahri** for 489.65 points (4 mins 8 secs)
Average Score	388.06 (for 171 correct submissions)

The simplest way to solve this problem is to try cutting the deck in every possible location, and then simulate the game for each one to see if you win. If you start your search with a cut of zero, and work your way up to N-1, you can return the size of the cut as soon as you find one where you win the game. This solution has a running time of O(N^2), which is perfectly fine since N is at most 50.

However, an O(N) solution exists. This solution also makes it clear why you can always find a place to cut the cards so that you win. (You may have noticed that the problem did not tell you what to do if no such cut could be found.)

To visualize the O(N) solution, make a graph where the value at position *x* is the number of black cards in the first *x* cards, minus the number of red cards in the first *x* cards. *x* increases as each new card is turned over, and the graph goes up one unit if it is black, and down one unit if it is red.

Here is graph for the "RBBRRRRBBBRBBRRRBRBBRRRRBBBBBRRRB":

Notice that this graph ends up at the same level at which it starts, because there are an equal number of red and black cards. If the value ever dips below the starting value (as it does in this graph), that means you lose the game. Cutting the deck is equivalent to moving the same number of line segments from the front of the graph to the end. So finding the right place to cut the deck is equivalent to finding the right place to cut the graph such that the value never dips below the starting value. This point is, obviously, at the minimum of the graph. If there are more than one, you select the left-most minimum (corresponding to the cut with the fewest cards.) In this example, that point is seven units from the left, so the answer is 7.

Here is the graph of the cut deck "BBBRBBRRBRBBRRRRBBBBBRRRBRBBRRRR":

See **writer**'s solution in the practice room for a simple implementation of this algorithm.

PrimeAnagrams Rate It Discuss it

Used as: Division Two - Level Three:

Value	1000
Submission Rate	34 / 199 (17.09%)
Success Rate	11 / 34 (32.35%)
High Score	**herbert_duarte** for 622.61 points (25 mins 39 secs)
Average Score	482.14 (for 11 correct submissions)

Used as: Division One - Level Two:

Value	500
Submission Rate	132 / 182 (72.53%)
Success Rate	87 / 132 (65.91%)
High Score	**overwise** for 455.64 points (9 mins 2 secs)
Average Score	295.40 (for 87 correct submissions)

There are 8 factorial (40320) ways to rearrange the order of 8 digits. There are 7 choose 2 (21) ways to break each of those up into 3 groups of digits. That gives only 846720 sets of primes to test, and that even overcounts by a factor of 6 because it tests each set in all 6 orders. So this problem can easily be solved by brute force.

The first step is to loop over all permutations of the input order. See **marian**'s solution for an example of how to do this using STL in C++, or my solution (**writer**'s, in the practice room) for an example of how to do this with a recursive function. For each of these permutations, you can separate them into all possible groups of 3 numbers, as demonstrated by the following pseudocode:

```
given: p (one permutation of the input string)

for i = 1 to len(p)-3
    for j = i+1 to len(p)-2
    {
        // 1st number is digits 0 through i-1
        // 2nd number is digits i through j-1
        // 3rd number is digits j through len(p)-1
    }
```

Then, for each group of 3 numbers, you test that none of them have a leading zero, and then test if all three are prime. Many coders generated a list of prime numbers with up to 6 digits at the beginning of their method for use in this step, most likely drawing from their library of pre-written code.

Once you find a set of 3 prime numbers, you compare their product to the product of the primes in the best set you have found so far. No further tiebreaker is necessary, because every number has a unique prime factorization, and therefore no two sets of prime numbers can have the same product.

QuizShow Rate It Discuss it

Used as: Division One - Level One:

Value	250
Submission Rate	174 / 182 (95.60%)
Success Rate	121 / 174 (69.54%)
High Score	**antimatter** for 247.53 points (2 mins 50 secs)
Average Score	207.01 (for 121 correct submissions)

This problem may have seemed like a daunting probability problem at first, but it doesn't take long to realize that you can simply test each possible wager, and compute your probability of winning. Computing this probability is straightforward, as there are only 8 possible outcomes (you and your two opponents can each independently win or lose, each with a probability of 50%).

See **writer's** solution in the practice room for an simple, commented solution to this problem.

If this were a real quiz show, the situation would be much more difficult, as you would probably not know your opponents' wagers at the time you had to make your own, nor would you be able to have such a simple estimate of each person's chance of answering the question correctly.

RevolvingDoors Rate It Discuss it

Used as: Division One - Level Three:

Value	1000
Submission Rate	22 / 182 (12.09%)
Success Rate	10 / 22 (45.45%)
High Score	**marian** for 600.14 points (27 mins 23 secs)
Average Score	497.32 (for 10 correct submissions)

RevolvingDoors is a breadth-first search problem. The state-space you are searching over is the space of all positions on the map (up to 50x50, minus the space taken up by door centers and any walls), times each of the possible orientations of all of the doors (up to 2^10). From any state, you can either move into an adjacent space (which does not increase your distance), or you can toggle the orientation of a door you are standing next to (which increments your distance by one). Once the search is complete, the answer is the minimum of the distances of all 2^N states corresponding to the end square. (The final orientations of the doors do not matter.)

The main difficulty of this problem turned out to be coding an efficient implementation. Several solutions failed due to challenges or the system tests on a 50x50 map that had all 10 doors serving no purpose in the middle, and the end square completely blocked off by walls. This foiled several inefficient solutions by forcing them to explore the maximum number of states. One common optimization used by coders to reduce the number of states was to only consider the "interesting" positions of the map: the start square, the end square, and the four squares from which you can interact with the doors.

In a post-contest discussion, it was hypothesized that the highest possible number of turns necessary for a map with N doors is 2^(N+1)-1. For example, here are maps with 1 and 2 doors, that requires 3 and 7 turns, respectively:

```
#####              ######
#   ###            #S | #
#-O- E#            #  O #
#  ####           ## #| #
#S##              ## |# ##
###              #E O  #
                 ###|  #
                   #####
```

SnapDragon came up with this astonishing example of a map with 10 doors that require 2047 total turns:

```
#####################################################
###   ##   ##   ##   ##   ##   ##   ##   ##   ##   #
## |# # |# # |# # |# # |# # |# # |# # |# # |# # |# #
#E O    0    0    0    0    0    0    0    0    0  #
###| ###| ###| ###| ###| ###| ###| ###| ###| ###|  #
#   ##   ##   ##   ##   ##   ##   ##   ##   ##   # #
#                                                 S#
#####################################################
```

The door on the far right must be turned 1025 times! The sequence of door turns in the shortest path through this map is reminiscent of the Chinese Rings puzzle and the Towers of Hanoi.

By **legakis**
TopCoder Member