TopCoder Competitions

SRM 607

Added by [[rng_58]] , last edited by vexorian on Feb 08, 2014 (view change)
Labels: (None) EDIT

Single Round Match 607
Monday, February 3rd, 2013

Archive
Match Overview
Discuss this match

Match summary

SRM 607 was the first contribution of **Wheeler**. A problem set that focused on dynamic programming and clever optimization. The division 1 coders began their journey dealing with a classical string problem. Then they had to face the reality of an complicated problem that could become much simpler through observation. Finally, the beautiful division 1 hard that was a fractal of deeper and deeper algorithmic challenges. This problem wasn't solved by any of the coders, and **lyrically** was the only coder that was able to submit a solution at all. The division winners would be decided by the first two problems. Through marvellous speed at those problems, **tomek** got the first place with a relevant lead over other big names like **K.A.D.R** (2nd place), **Petr** (3rd place) and **tourist** (4th place). Congratulations also to division 2 winner: **enrevol**.

The Problems

BoundingBox | PalindromicSubstringsDiv2 | CombinationLockDiv2 | PalindromicSubstringsDiv1 | CombinationLockDiv1 | PulleyTautLine

CombinationLockDiv2

Rate It | Discuss it

Used as: Division Two - Level Three:

Value	1000
Submission Rate	49 / 1103 (4.44%)
Success Rate	3 / 49 (6.12%)
High Score	**zhangchenxu** for 449.89 points (45 mins 26 secs)
Average Score	423.49 (for 3 correct submissions)

Beware

This (specially the division 1 version) is a problem in which it is easy to make assumptions that are wrong. It is best not to take advantage of properties/limitations that we cannot prove.

Always to 0

An idea that is not necessary but will simplify the code is to notice that we do not really care about the specific values of the digits in **s** or **t**, but just about the distance between each **s**[i] and **t**[i]. We can turn the problem into one in which we should convert all digits to 0. For example, the steps that can take $s_i = 4$ to $t_i = 9$, are the same steps that can take $d_i = 5$ to 0. Similar to finding the distance between angles in a circle, this needs modular

arithmetic: $d_i \equiv (s_i - t_i) \bmod 10$. The problem becomes about turning those d_i values into 0.

Always in the same direction

There *is* a property that we can prove and can greatly simplify the problem. Imagine a sequence of moves in which we moved all numbers in many intervals up or down. If two intervals overlap and the directions of the intervals are different, they would cancel out:

```
      d = 13749920384994
(move)      ++++++
      d = 13750031484994
(move)         -----
      d = 13750030373883
```

That sequence of moves is equivalent to:

```
      d = 13749920384994
(move)      ++++
      d = 13750030384994
(move)           ---
      d = 13750030373883
```

Which do not overlap. It may at first appear that there are cases in which an overlap might be useful. Like Splitting a large interval in two halves:

```
      d = 13749920384994
(move)    ++++++++++++
      d = 14850031493004
(move)         -
      d = 14850021493004
```

But in reality, we can still reach the same state in two moves that don't overlap. What about a more formal proof? Whenever two intervals A and B of different direction overlap, it costs 2 moves. Using set operations, the two moves can be replaced by other two moves: $A - \left(A \bigcap B\right)$ (in same direction as A) and $B - \left(A \bigcap B\right)$ (in same direction as B. The resulting digits are the same and the cost is also the same. If there are any more overlaps, we can repeat this simplification until there are no overlaps.

Opening and closing intervals

If we wanted to make a dynamic programming solution, we should first find an *"optimal substructure"* in this problem. The issue is that all those overlap those intervals make the problem too tangled. This is why we need to think in terms of opening and closing intervals.

Let us say we move from the lowest indexed digit to the highest, and decide, for each of them, the direction and the number of times to rotate the dial. We can only choose direction/number combinations that make the digit 0. Imagine the decision is like this: $(+22, +12, +1, -7, -6, -11)$, meaning first digit moves 22 up, second digit 12 times up, fourth digit 7 times down and so and so. Note that we are moving some digits more than 9 times. It is tempting to make the assumption that you only need to move each digit at most 9 times, but that assumption is wrong (*Very* wrong, there are test cases in which the number of times we need to move a single digit is $2n$). Let's avoid any risk and just use $9n$ as the maximum number of times a single digit may move (This is a very safe bound, because all test cases can be solved in at most $9n$ steps).

With moves like $(+22, +12, +1)$, what is the minimum number of intervals that cover all those moves? We can make a single interval over the 3 digits, then 11 intervals that cover the first two digits and 9 more intervals for the first digit. A way to interpret this is that we decided that 22 intervals must go through the first digit, 12 intervals through the second digit and only 1 through the third digit. How about we changed the decision from picking the

number of intervals in each digit to picking the number of new intervals added for that digit? (Possibly, we could also close intervals), then the cost only increases by the number of intervals we open in that digit.

Another example: $(+1, +2, +3, +4, +3, +2, +1)$: First we open an interval for the first digit. Second digit needs 2 intervals, which means we open a new interval. And so and so, eventually we move from $+4$ to $+3$, this means *closing* one interval, so it doesn't add to the cost. In total, we open 4 intervals, which means it can be solved with 4 intervals:

```
   +
  +++
 +++++
+++++++
```

When the intervals are negative, they behave in the same way, but this time the intervals make the digits decrease.

Dynamic programming

Let us decide the direction and number of intervals for each digit in increasing order of index. We need to remember the number of open intervals x, and the direction u. If $u = 1$, the direction is up, if $u = 0$ the direction is down. $f(p, x, u)$ returns the minimum cost to zero the digits with index greater than or equal to p if the number of open intervals after deciding the first p digits is x and their direction is u.

- Base case: $p = n$, we have set all the digits to 0. Nothing to do, the minimum cost is 0.
- Else we pick a new direction n_u and a new number of intervals y:
  - If the directions are different, we are forced to open y new intervals. Cost is $y + f(p + 1, y, n_u)$.
  - Else the number of new intervals is either $y - x$ (if $y > x$) or 0 (otherwise). Cost is: $\max(y - x, 0) + f(p + 1, y, u)$

  Pick the minimum cost out of all the options.

We use dynamic programming so that each state of the function is evaluated at most once. There are $n + 1$ values for p and the maximum number of moves per digit is $9n$. There are $2 \cdot (n + 1) \cdot 9 \cdot n$ states in the function and each needs to try $2 \cdot 9n$ options. The complexity order is $O\left(n^3\right)$, but there are some sizable factors (9*9*2*2)

Solution

```
static const int INF = 1000000000;
static const int MAX_N = 50;
static const int MAX_OP = 450;
vector<int> d;
int n;
int dp[MAX_N + 1][MAX_OP + 1][2];

int rec(int p, int x, int up )
{
    int & res = dp[p][x][up];
    if (res == -1) {
        if (p == n) {
            res = 0;
        } else {
            res = INF;
            // pick direction and number of changes:
            for (int nu = 0; nu <= 1; nu++) {
                for (int y = 0; y <= MAX_OP; y++) {
                    // verify the new direction and mumber of moves will se
                    // the digit to 0:
                    if ( nu == 0 ) {
                        if ( (d[p] + 9*y) % 10 != 0) {
                            //invalid
                            continue;
                        }
                    } else {
                        if ( (d[p] + y) % 10 != 0) {
                            //invalid
                            continue;
                        }
                    }
                    int z = 0;
                    if (nu != up) {
```

Alternative solutions and additional comments.

<Place your comments here>

Next problem: PalindromicSubstringsDiv1

By **vexorian**

TopCoder Member

Editorial feedback	Choose
I liked it.	
I didn't like it.	

18 Comments | Add Comment