TopCoder Competitions

SRM 498

Added by mystic_tc , last edited by sasidharkasturi on Mar 08, 2011 (view change)
Labels: (None)

Single Round Match 498
Saturday, February 26th, 2011

Archive
Match Overview
Discuss this match

Match summary

In this match, coders were treated to several problems chronicling the exploits of Fox Ciel, written by first-time author **ir5**. Coders seemed to find the problems somewhat easier than usual, and the success rates were high.

The crown in Division 1 was taken by **tourist**, whose lightning-fast submissions and two successful challenges left him about 80 points ahead of **ACRush**, consigning the latter to second place. **rng_58** rounded off the top three. A total of 21 coders solved all three problems, and many scores were in the thousands.

If you think this is surprising for Division 1, wait till you hear about Division 2, where more than 150 coders solved all three problems. **xiaowuc1** scored a convincing win, finishing all three problems in around 19 minutes and scoring nearly 1600 points. In second place came **PS_(NULP)**, with 1530 points, followed by a near-tie between **PooPS** and **gmunkhbaatarmn**, who scored about 1490 points each. Of noteworthy mention is **s-n-e-e-r**, who came 15th despite having a 500-pointer that failed system tests - his seven successful challenges more than made up for it!

The Problems

AdditionGame

Rate It Discuss it

Used as: Division Two - Level One:

Value	250
Submission Rate	1098 / 1148 (95.64%)
Success Rate	1014 / 1098 (92.35%)
High Score	**everid** for 249.90 points (0 mins 35 secs)
Average Score	218.63 (for 1014 correct submissions)

There are three positive integers. In each of **N** steps, we can pick one of them (*x*, say), add *x* to a running total, and decrease *x* by one if it is still positive.

So at each step, we just need to figure out which integer to pick. Clearly, we might as well take the largest one, since there's no reason to do otherwise: picking a smaller number doesn't help to improve our score either immediately or later in the game. A simple implementation passes the system tests with plenty of time to spare:

```
int ans = 0;
for(int i = 0; i < N; i++)
{
   if(A >= B && A >= C)
   {
      ans += A;
      A -= (A > 0);              //Note that (A > 0) evaluates to 1 if and only if A > 0.
   }
   else if(B >= A && B >= C)
   {
      ans += B;
      B -= (B > 0);
   }
   else if(C >= A && C >= B)
   {
      ans += C;
      C -= (C > 0);
   }
}
return ans;
```

This problem begs to be generalized, so here are some extensions that you can think about:

- Suppose that **N** and the integers are allowed to be quite large (up to 10^8, say). Can you find a faster algorithm? (*Hint*: if we implement the above strategy, what can you say about the final values of the numbers after all **N** steps? You might need to consider several cases.)
- What if there are *M* numbers written on the blackboard?
- Finally, what if numbers are allowed to become negative? In other words, if Ciel picks the number *x*, she gains *x* points, and *x* is decreased by one, regardless of whether *x* is positive or not.

Alternative solutions and additional comments.

<PLACE YOUR COMMENTS HERE>

FoxSequence

Rate It Discuss it

Used as: Division Two - Level Two:

Value	500
Submission Rate	844 / 1148 (73.52%)
Success Rate	323 / 844 (38.27%)
High Score	**xiaowuc1** for 486.77 points (4 mins 42 secs)
Average Score	297.26 (for 323 correct submissions)

Used as: Division One - Level One:

Value	250

Submission Rate	755 / 764 (98.82%)
Success Rate	541 / 755 (71.66%)
High Score	**tourist** for 246.10 points (3 mins 35 secs)
Average Score	194.06 (for 541 correct submissions)

In order to solve this problem, it is necessary to read the problem statement carefully, and ensure that the algorithm handles all the possible cases that can occur.

We are given a sequence, and want to decompose it into five parts (where the middle part could be empty). One way to do this is simple brute force: try all possible ways to divide the sequence into five parts, and if one of them works, return "YES". This does fit within the time limit, and is reasonably simple to code. For the system tests, the code below runs in 6 ms in the worst case:

```
int N = seq.size();

for(int a = 1;   a < N-1; a++)
for(int b = a+1; b < N-1; b++)
for(int c = b;   c < N-1; c++)
for(int d = c+1; d < N-1; d++)
{
   int bad = 0;

   int d0a = seq[a]-seq[a-1];
   if(d0a <= 0) continue;
   for(int p = 1; p < a; p++)
   if(seq[p]-seq[p-1] != d0a)
   {
      bad = 1;
      break;
   }
   if(bad) continue;

   int dab = seq[b]-seq[b-1];
   if(dab >= 0) continue;
   for(int p = a+1; p < b; p++)
   if(seq[p]-seq[p-1] != dab)
   {
      bad = 1;
      break;
   }
   if(bad) continue;

   if(b != c)
   {
      int dbc = seq[c]-seq[c-1];
      if(dbc != 0) continue;
      for(int p = b+1; p < c; p++)
      if(seq[p]-seq[p-1] != dbc)
      {
         bad = 1;
         break;
      }
   }
   if(bad) continue;

   [Repeat the first two blocks above, with the appropriate modifications.]

   return "YES";                                                       // This partitioning works.
}
return "NO";                                                           // None of the partitionings are valid.
```

Alternatively, we could try to find a faster algorithm. This can be done by working through the sequence from left to right, implicitly breaking it up into its component sequences as we go along. In other words:

1. Start at the first term of the sequence.
2. As long as the difference between the next term and this one is positive, move one step right. If any two of the differences you've encountered are different, the answer is "NO". If you can't even move one step, the answer is also "NO".
3. (As in the previous step, but replace 'positive' with 'negative'.)
4. As long as the current term and the next one are equal, move one step right. It's OK if you don't move at all in this stage.
5. Repeat Step 2.
6. Repeat Step 3.
7. If we're now at the end of the sequence, this means that the sequence is a fox sequence; return "YES". Otherwise, return "NO".

This procedure is not too difficult, but requires a little care. For example, we should be careful not to go past the end of the sequence, since this may result in an out-of-bounds error. Also, several solutions started off by evaluating **seq**[1] - **seq**[0], which isn't well-defined if **seq** only has one element. A way to avoid this is to note that the length of a fox sequence must be ≥ 5, so we can return "NO" immediately if length(**seq**) ≤ 4.

Finally, note that it isn't necessary to code each step separately from scratch! Once you've written code for one of the steps, it's quite easy to copy-and-paste it five times, and make small modifications so that it performs as described above.

Alternative solutions and additional comments.

A slight variant of the solution described above. We compute the differences and defer the decision of correctness till the end. Inspired from Petr's solution.
In an arithmetic progression(AP), the invariant is its common difference(CD).

1. For the AP from 0 to a: CD_1 > 0
2. For the AP from a to b: CD_2 < 0
3. For the AP from b to c: CD_3 = 0
4. For the AP from c to d: CD_4 > 0
5. For the AP from d to N-1: CD_5 < 0

Some properties of the foxsequence:

1. If b == c, i.e; there are no elements in the sequence having equal elements, CD_3 wouldn't exists.
2. The sign of the CDs are fixed across problems and are described above.
3. Atleast 5 elements are required, for a valid sequence to exist.
4. Consider three elements e_i, e_j, e_k appearing one after another in the sequence. If **e_j - e_i != e_k - e_j**, then e_i, e_j belong to one AM and e_j, e_k should to the next AM. i.e; A different CD implies a different AP.
5. According to the given problem, either **4**(when b=c i.e; CD_3 doesn't exist) or **5**(b not equal to c i.e; CD_3 exists and is equal to 0) common differences can exist.

Scanning the sequence from left to right pairwise, all CDs are added to a list. The number of elements(CDs) in the list and the signs of the elements(CDs) gives the correctness of the solution.

The following code written in java solves the problem:

```
public String isValid(int[] seq) {
        ArrayList diffs = new ArrayList();
        int diff = 0;
        if (seq.length >= 5 ) {                                 // Atleast 5 elements are needed.
                diffs.add(seq[1] - seq[0]);
                for (int i = 1, n = seq.length; i < n; i++) {
                        diff = seq[i] - seq[i - 1];
                        if ((!diffs.isEmpty()) && diffs.get(diffs.size() - 1) != diff)
                                diffs.add(diff);
                }

                if (diffs.size() == 4 && diffs.get(0) > 0 && diffs.get(1) < 0
                                && diffs.get(2) > 0 && diffs.get(3) < 0)               // CD_3 doesn't exist.
                        return "YES";
                if (diffs.size() == 5 && diffs.get(0) > 0 && diffs.get(1) < 0
                                && diffs.get(2) == 0 && diffs.get(3) > 0
                                && diffs.get(4) < 0)                                  //  CD_3 exists and is equal to 0.
                        return "YES";
        }
        return "NO";
}
```

<PLACE YOUR COMMENTS HERE>

NinePuzzle

Rate It Discuss it

Used as: Division Two - Level Three:

Value	950
Submission Rate	364 / 1148 (31.71%)
Success Rate	311 / 364 (85.44%)
High Score	**bhavu** for 945.76 points (1 mins 54 secs)
Average Score	629.88 (for 311 correct submissions)

This was a surprisingly easy Div 2 Hard problem, even if one did not spot the 'greedy' solution (which I will explain later). We have a puzzle with 9 pieces and want to get from an initial state to a goal state. We want to find the minimum number of pieces that need to be changed in the initial state so that the goal state is reachable from the initial state.

To do this, we can work backwards. Find all states from which the goal is reachable, and, for each of these states, find the number of cells for which the current color differs from the initial one. This is the number of pieces that need to be repainted. Since each move is reversible, we can implement this as a breadth-first search starting from the goal position. There aren't that many states (in the worst case, the goal string contains 3 copies of one color, 2 copies of each of the remaining colors, and 1 '*', in some order, leading to 10!/(2!2!2!3!) = 75600 states). So we can simply check each of them, count how many repaintings each needs, and return the minimum of the results. No clever optimizations are needed.

In the following code, the states that have been encountered are stored in a set *S*, and the *i*-th element of the vector *V* stores the cells adjacent to cell *i*.

```
    #define PB push_back

    vector<vector<int> > V(10);

    V[0].PB(1), V[0].PB(2);
    V[1].PB(0), V[1].PB(2), V[1].PB(3), V[1].PB(4);
    V[2].PB(0), V[2].PB(1), V[2].PB(4), V[2].PB(5);
    V[3].PB(1), V[3].PB(4), V[3].PB(6), V[3].PB(7);
    V[4].PB(1), V[4].PB(2), V[4].PB(3), V[4].PB(5), V[4].PB(7), V[4].PB(8);
    V[5].PB(2), V[5].PB(4), V[5].PB(8), V[5].PB(9);
    V[6].PB(3), V[6].PB(7);
    V[7].PB(6), V[7].PB(3), V[7].PB(4), V[7].PB(8);
    V[8].PB(7), V[8].PB(4), V[8].PB(5), V[8].PB(9);
    V[9].PB(5), V[9].PB(8);

    set<string> S;
    S.clear();

    int ans = 999999;                                 // or any value larger than 10
    int initstar = 0;                                 // initstar stores the position of the empty cell
    for(int i = 0; i < 10; i++)                       //     in the initial layout
       if(init[i] == '*')
          initstar = i;

    deque<string> Q;
    Q.PB(goal);
    S.insert(goal);
    while(Q.empty() == 0)
    {
       string k = Q[0];
       Q.pop_front();

       if(k[initstar] == '*')                         // the empty cells for the goal and initial states coincide...
       {                                              // ...so this is a possible start state
          int a = 0;
          for(int p = 0; p < 10; p++)
             if(p != initstar && k[p] != init[p])
                a++;
          ans = min(ans,a);
       }

       int currstar = 0;                               // currstar stores the position of the empty cell
       for(int p = 0; p < 10; p++)                     //     in the current state
          if(k[p] == '*')
             currstar = p;

       for(int x = 0; x < V[currstar].size(); x++)
       {
          swap(k[currstar], k[V[currstar][x]]);
          if(S.find(k) == S.end())                     // we haven't seen this state...
          {                                            // ...so put it into Q
             S.insert(k);
             Q.PB(k);
          }
          swap(k[currstar], k[V[currstar][x]]);
       }
    }
    return ans;
 }
```

However, there is an even simpler solution. All that is necessary is to view the initial and goal states as unordered sets of pieces, and find the minimum number of pieces that need to be repainted in the initial set so that it matches the goal set. For example, consider the second sample case, in which the initial state is described by four 'R's and five 'B's, and the final state is described by 5 'R's and 4 'B's. Then one 'B' needs to be repainted with the color 'R', so the answer is 1.

Why does this work? Because, given an initial configuration of tiles, it's actually possible to get to any other configuration which uses the same set of tiles. So the order of the tiles doesn't matter.

To prove this, it's enough to demonstrate a systematic procedure for getting from a configuration *C* to another configuration *D* which uses the same set of tiles. Let's start by shifting pieces in both *C* and *D* so that the center cells (the ones numbered '4') in both configurations are empty. (At the end, we can reverse any moves we've performed on *D*.) We'll call the color of the piece in cell *i* in *D* the 'correct' color for that cell.

Now, consider cell 0. If it already has a piece of the correct color, we're happy. Otherwise, we want to get a piece of the correct color into that cell. So move one of the other pieces into cell 4, and shift pieces cyclically around the 'border' of the triangle until the correct one ends up in cell 0. This piece is now fixed, and we don't touch it from now on. Finally, move the piece in cell 4 back out to some cell on the border (shifting pieces along the bottom row if necessary).

Now let's look at position 6. If it already has a piece of the correct color, we're happy. Otherwise, we move one of the other pieces into cell 4, and shift pieces around the new 'border' (which consists of the cells 1 - 2 - 5 - 9 - 8 - 7 - 6 - 3 - 1) until the correct piece ends up in position 6. The piece at position 6 is fixed from now on, and we move the piece in cell 4 back out to some cell on the border, shifting pieces if necessary.

We can then fix the piece in position 9, and then the piece in position 1, using a similar method.

We now want to fix the piece in position 2, but we need to do something slightly different here. Start by moving the piece that needs to be in position 2 into the center cell, '4'. (Let's say this piece is currently in cell '7'.) This creates a hole in cell 7. Fill this hole by shifting pieces cyclically (in this case, move a piece from 8 to 7, then from 5 to 8, then from 2 to 5). Finally, move the desired piece from cell 4 to cell 2, and we've fixed cell 2. Then, work around the hexagon, fixing cells 5, 8, 7 and 3 in turn. And we're done!

Since this gives a procedure for getting from any arrangement to any other arrangement, it follows that the algorithm we described earlier works. This leads to a very short solution. (Notice that this code automatically handles the '*' character correctly, by getting rid of it from both strings.)

```
for(int i = 0; i < init.size(); i++)
   if(goal.find(init[i]) != -1)            // there's a match!
   {
      goal.erase(goal.find(init[i]), 1);
      init.erase(i, 1);
      i--;
   }

return init.size();                        // 'init' now contains characters that can't be matched in 'goal'.
```

Alternative solutions and additional comments.

<PLACE YOUR COMMENTS HERE>

FoxStones

Rate It Discuss it

Used as: Division One - Level Two:

Value	450
Submission Rate	568 / 764 (74.35%)
Success Rate	455 / 568 (80.11%)
High Score	**Michael_Levin** for 438.09 points (4 mins 42 secs)
Average Score	311.30 (for 455 correct submissions)

There is an **N** by **M** board containing stones and some marked cells. We want to find the number of layouts that are similar to the current layout.

Let (*i*, *j*) be the position of a stone in the original layout. Consider the position of this stone in the new layout - where can it go? It needs to be in a cell that is similar to (*i*, *j*), in the sense that the distances to each of the marked cells are the same.

Thus, it makes sense to store, for each cell, an array containing the distances to each of the marked cells, in order. Then we can call two cells *equivalent* if the arrays associated with them are identical. So, in the new configuration, the stone that was originally at (*i*, *j*) can go to any of the cells which are equivalent to (*i*, *j*).

Now consider the set of cells that are equivalent to (*i*, *j*) (including (*i*, *j*) itself). The stones in these cells can go only to these cells, and to no others. So, in the new configuration, the stones in these cells will be a permutation of the stones that used to be in these cells in the original configuration. If there are *k* such stones, there are *k*! such permutations.

This leads to a simple algorithm. Each set of equivalent cells can be considered independently, so we just need to multiply the number of permutations for each of the different equivalence-sets. We can simplify things a little by pre-calculating an array of factorials:

```
long long MOD = 1000000009;
long long f[50000];
f[0] = f[1] = 1;
for(int i = 2; i < 50000; i++)
   f[i] = (f[i-1]*i)%MOD;
```

Then, for each cell on the board, we compute the array associated with it, and store it in a map which maintains the count of cells associated with each array. Finally, for each array, we take its count *k*, and multiply the current answer by *k*!. There are at most 40000 cells, and each array has at most 50 elements, so this algorithm should run in time. A C++ implementation follows:

```
map<vector<int>, int> A;
for(int i = 1; i < N+1; i++)
for(int j = 1; j < M+1; j++)
{
   vector<int> V;
   for(int k = 0; k < sx.size(); k++)
      V.push_back( max(abs(sx[k]-i), abs(sy[k]-j)) );
   A[V]++;
}

long long ans = 1;
for(typeof(A.begin()) it = A.begin(); it != A.end(); it++)
   ans = (ans*f[it->second])%MOD;
return ans;
```

Alternative solutions and additional comments.

<PLACE YOUR COMMENTS HERE>

FoxJumping

Rate It Discuss it

Used as: Division One - Level Three:

Value	1000
Submission Rate	28 / 764 (3.66%)
Success Rate	21 / 28 (75.00%)
High Score	**rng_58** for 801.23 points (14 mins 56 secs)
Average Score	528.48 (for 21 correct submissions)

Fox Ciel now wants to do some jumping. She can make jumps (dx, dy), where $0 \le dx \le$ **Mx** and $0 \le dy \le$ **My**, but she can't make jumps (d, d) where d is an element of **bad**. Note that she also can't stay where she is (i.e., she can't make the jump (0, 0)), so let's start by adding 0 to **bad**. (0 is still a multiple of 10, so doing this shouldn't cause any issues later.)

One could start by trying a standard dynamic programming (DP) solution. Suppose Fox Ciel has jumped k times and is now at cell (cx,cy). How many ways are there for her to achieve this? This is easy to calculate: we just sum all contributions from k - 1 jumps and all cells in the rectangle defined by $cx - \mathbf{Mx} \le x \le cx$, $cy - \mathbf{My} \le y \le cy$, ignoring cells that contribute to bad jumps. But this is definitely going to time out, because k can be up to 1600, the coordinates can be up to 800 each, and when evaluating the answer for each cell we need to sum up to 800^2 values. Ugh!

Let's forget about the bad jumps for the moment. We could ask the following question: how many ways are there for Fox Ciel to get to (**Tx**,**Ty**) in **R** steps, assuming that all possible jumps are considered to be good? In other words, suppose we have two sequences of length **R**. Each element in the first sequence is in [0, **Mx**], and each element in the second sequence is in [0, **My**]. We want the elements in the first sequence to sum to **Tx**, and the elements in the second sequence to sum to **Ty**. How many ways are there to achieve this?

To solve this, note that the two sequences are independent. Thus, we can compute the answers for each separately, and multiply the two answers at the end.

So the problem we have to solve is: how many sequences of length **R**, each element of which is in [0, **Mx**], have elements that sum to **Tx**? **R** is at most 1600, and **Mx** and **Tx** are each at most 800.

This is a standard DP problem. Let $A_{i,\,j}$ be the number of sequences of length i whose elements sum to j. We want to compute $A_{\mathbf{R},\,\mathbf{Tx}}$. Then $A_{i,\,j} = A_{i-1,\,j-\mathbf{Mx}} + A_{i-1,\,j-\mathbf{Mx}+1} + \ldots + A_{i-1,\,j}$, and the base cases are $A_{0,\,0} = 1$ and $A_{0,\,\text{anything else}} = 0$. But then there are 1601 × 801 cells in our table, and computing each cell takes up to 801 steps, so this could still time out. How can we speed it up?

Note that each computation consists of summing a consecutive block of cells. So let's maintain partial sums: define

$$S_{i,\,j} = A_{i,\,0} + A_{i,\,1} + \ldots + A_{i,\,j}.$$

Then we can write

$$A_{i-1,\,j-\mathbf{Mx}} + A_{i-1,\,j-\mathbf{Mx}+1} + \ldots + A_{i-1,\,j} = S_{i-1,\,j} - S_{i-1,\,j-\mathbf{Mx}-1},$$

which now takes only $O(1)$ time to calculate. Note that $A_{i,\,j} = S_{i,\,j} - S_{i,\,j-1}$, so we can now get rid of the A-array entirely: the recurrence becomes $S_{i,\,j} = S_{i,\,j-1} + S_{i-1,\,j} - S_{i-1,\,j-\mathbf{Mx}-1}$ and we can fill the array up in $O(\mathbf{R} * \mathbf{Mx})$ time.

So we've solved the problem in the case where there are no bad moves. How do we take bad moves into account, and why do we have that funny constraint that the elements of **bad** need to be multiples of 10?

The answer to the first question comes from the inclusion-exclusion principle. Let B be a set of indices (from 0 to **R** - 1), and let X_B be the number of sequences for which the moves corresponding to positions in B are bad moves. (It's possible for there to be other moves, not in B, that are also bad.) Then the total number of jump sequences with no bad moves can be calculated as follows:

```
  sum of $X_B$ over all $B$ of size 0
- sum of $X_B$ over all $B$ of size 1
+ sum of $X_B$ over all $B$ of size 2
- sum of $X_B$ over all $B$ of size 3
+ ...
```

We now need to calculate each of these terms. Let's suppose we want to find the sum of the X_B over all B of size k. There are ${}^{\mathbf{R}}C_k$ terms in this sum, corresponding to the ${}^{\mathbf{R}}C_k$ possible positions for the k known bad jumps. If we knew what the bad elements at these positions summed to ($badsum$, say), we could just remove that sum from each of **Tx** and **Ty**, and then solve our previous problem (namely, how many jump sequences of length **R** - k, with each element in ([0,**Mx**], [0,**My**]), sum to (**Tx** - $badsum$, **Ty** - $badsum$)?).

But this is again a standard DP problem. Since each element of **bad** is a multiple of 10, we only need to consider the values {0, 10, ..., 800}. Let $B_{i,\,j}$ be the number of ways to pick a sequence of i bad values, such that their sum is 10*j. Then

$$B_{i,\,j} = B_{i-1,\,j\,-\,\mathbf{bad}[0]/10} + B_{i-1,\,j\,-\,\mathbf{bad}[1]/10} + \ldots$$

where the sum is taken over all elements of **bad**. There are up to 1601 × 81 cells in the table, and to compute the sum for each cell we need to sum at most 50 terms, which gives a total of about 6.5 million calculations. (Now we see why the problem author was kind enough to constrain the values in 'bad' to be multiples of 10 -- otherwise, there'll be around 65 million calculations, which is probably still OK, but a little risky.)

We can then use this to calculate the total number of good sequences using the inclusion-exclusion formula above. Putting all these steps together gives the following pseudocode (in the real thing, don't forget to do all calculations modulo 10007):

```
Define the following arrays:  Sx[1601][801], Sy[1601][801], B[1601][81] and comb[1601][1601].
Initialize all their contents to 0.

For each t, set comb[t][0] = comb[t][t] = 1.
Fill the rest of comb[][] using the recurrence: comb[t][u] = comb[t-1][u] + comb[t-1][u-1].

For each i from 0 to R, set Sx[i][0] = 1.
For each i from 1 to Tx, set Sx[0][i] = 1.
Fill the rest of Sx[][] using the recurrence: Sx[i][j] = (Sx[i][j-1] + Sx[i-1][j] - Sx[i-1][j-Mx-1].
Repeat these steps for Sy[][].

Put 0 in 'bad', and divide all elements of bad by 10.
Set B[0][0] = 1.
For each i from 1 to R
For each j from 0 to 81
   For each k from 0 to size(bad)-1
      Set B[i][j] = B[i][j] + B[i-1][j-bad[k]].

Set ans = 0 and sgn = -1.
For each i from 0 to R
   Set sgn to -sgn.
   For each j from 0 to 81
      Set Txleft = Tx - j*10 and Tyleft = Ty - j*10.
      Add sgn*B[i][j]*comb[R][i]*(Sx[R-i][txleft] - Sx[R-i][txleft-1])*(Sy[R-i][tyleft] - Sy[R-i][tyleft-1]) to ans
Return ans.
```

Let's end with a quickie. How would you solve this problem if **R** were allowed to take values up to 1000000? (Don't think too hard!)

Alternative solutions and additional comments.

<PLACE YOUR COMMENTS HERE>

By **sl2**
TopCoder Member