Statistics **Match Editorial**

SRM 146
Wednesday, May 14, 2003

Archive
Normal view
Discuss this match

Match summary

I don't know when it happened the last time, but in this SRM the top-scorer was not a "red", but a high ranked yellow, **along**, who scored **1285.02**. Another unique thing in Division-I was the presence of a problem worth 800 points, as well as the fact that the highest rating gainer in Division-I (**cnettel**) was the same as after the last SRM! In Division-II, first-timer **tomek** didn't give the other competitors a chance by scoring **1597.39**, more than 300 points ahead of the runner-up. This shouldn't come as a surprise though, as **tomek** was part of the team that won this years ACM ICPC in Hollywood!

The Problems

Yahtzee Discuss it

Used as: Division-II, Level 1:

Value	250
Submission Rate	178 / 190 (93.68%)
Success Rate	157 / 178 (88.20%)
High Score	**Sleeve** for 248.84 points

Reference implementation: **Yarin** *(practice room)*

Implementation

This one is pretty straightforward. We can test selecting all values between 1 and 6 on the dice, and then see how much this would score. We then select the maximum of these scores. This can be done with two nested for-loops, like this:

```
int max=0;
for(int value=1;value<=6;value++) {
    int sum=0;
    for(int i=0;i<5;i++)
        if (toss[i]==value)
          sum=sum+value;
    if (sum>max) max=sum;
}
```

RectangularGrid Discuss it

Used as: Division-II, Level 2:

Value	500
Submission Rate	91 / 190 (47.89%)
Success Rate	72 / 91 (79.12%)
High Score	**tomek** for 487.33 points

Used as: Division-I, Level 1:

Value	300
Submission Rate	97 / 106 (91.51%)
Success Rate	93 / 97 (95.88%)
High Score	**Yarin** for 298.92 points

Reference implementation: **Yarin** *(practice room)*

Implementation

There are several ways one can solve this problem; the difference between the solutions is basically the number of nested for-loops you have! You can write solutions with either 4, 3, 2, 1 or no for-loops at all! Having 4 or 3 for-loops will be too slow though, as explained below.

The most common solution is the one with 2 for loops, and it's probably the easiest to code as well. We loop over all possible sizes of the rectangles:

```
for(int rect_width=1;rect_width<=width;rect_width++)
    for(int rect_height=1;rect_height<=height;rect_height++) {
        if (rect_width==rect_height) continue; // Don't count squares!
```

If rect_width equals rect_height, this is a square, and we don't want to count those, so make sure we skip to the next step in the iteration (using the instruction continue for instance). Now, we want to count how many rectangles of this size there are. One could do this by looping through all valid positions of one of the corners. For each such position, we have one rectangle:

```
        for(int left=0;left<=width-rect_width;left++)
            for(int top=0;top<=height-rect_height;top++)
                count++;
```

But this is just too slow (four nested for-loops), as we have to count each rectangle one by one, and as could be seen from the examples, there can be *many* rectangles, so we have to do it in a smarter way. The first observation is that the innermost loop (top) will iterate exactly height-rect_height+1 times, and each time we will increase count by one. So we can replace the last two lines above with

```
        for(int left=0;left<=width-rect_width;left++)
            count+=(height-rect_height+1);
```

Now we have a total of three nested for-loops, which almost works. So let us try to get rid of the left for-loop: This one iterates width-rect_width+1 times, each time increasing count with the same amount. Again, we can replace the last two lines with:

```
        count+=(width-rect_width+1)*(height-rect_height+1);
```

This is enough to get the program fast enough. But we can do it even faster! We can't easily continue in the same way though, because of the if statement and that the increase of count is dependent on the for-loop variable. The version with one for-loop works like this: we count the number of rectangles (*including* squares) and then subtract the number of squares (using a for-loop for the latter). If we take the same four for-loops above (and ignore the square check) and reorder them like this

```
for(int rect_width=1;rect_width<=width;rect_width++)
    for(int left=0;left<=width-rect_width;left++)
```

```
        for(int rect_height=1;rect_height<=height;rect_height++)
            for(int top=0;top<=height-rect_height;top++)
                count++;
```

we can, by some careful inspection, see that the left loop will iterate first 1 time (when rect_width is 1), then 2 times, then 3 times, etc up till width. That is, the left loop will iterate 1+2+3+...+width = width*(width+1)/2 times (well known math formula). The third and fourth loop are the same, so the number of times the count++ will be executed is (width*(width+1)/2)*(height*(height+1)/2). Thus we have:

```
rects = width*(width+1)*height*(height+1)/4; // Number of rectangles including squares
```

Now we just have to subtract from this number, the number of squares. We can loop through all sizes of the squares (from 1 to the minimum value of width and height) and check how many places they can be on:

```
for(int size=1;size<=min(width,height);size++)
    squares+=(width-size+1)*(height-size+1);
```

So we're now down to only one for-loop. We can get rid of that one as well by using some more math formulas, which I won't explain though. The number of squares can be calculated with this formula:

```
min=min_of(width,height);
dif=max_of(width,height)-min;
squares=dif*min*(min+1)/2+min*(min+1)*(2*min+1)/6;
```

Check out the solution by **lars2520** in the practice room for a really short implementation of this...

BridgeCrossing Discuss it

Used as: Division-II, Level 3:

Value	1000
Submission Rate	48 / 190 (25.26%)
Success Rate	9 / 48 (18.75%)
High Score	**tomek** for 763.03 points

Reference implementation: **Yarin** *(practice room)*

Implementation

This is a problem which appears in many problem solving books, where you're supposed to solve a specific case "manually" (ie, not with a computer program). Writing a computer program so solve the general case if of course harder for most people. Still, there are several different approaches that will work, because the constraints (the maximum number of people) is so small, at most 6.

The hint in the problem tells us that the best solution will be for two people two cross the bridge, one walk back, two cross, one walk back etc, until everyone has crossed. One thing that should always be remembered is that when the input is so low (always check the size of the input before you start trying to solve the problem!), always consider whether the most simple brute force solution would work. In this problem, I would consider this the most naive of all brute force solutions, which uses the hint:

```
Select any two people still on the start side, and make them cross.
Select any one people on the other side, and make him go back.
Repeat this procedure until everyone has crossed (using recursion)
```

Before we start to code this, we should make a quick check if this will be fast enough. We only check the worst case, ie when we have 6 people. Selecting any two people among 6 can be done in 15 ways (since order doesn't matter). Then we have two people on the other side, one of them (2 choices possible) go back. Now we have 5 people on the start side. We repeat the procedure: 2 of 5 cross (10 ways), 1 back (3 ways, since we have three people on the other side), 2 of 4 cross (6 ways), 1 back (4 ways), 2 of 3 cross (3 ways), 1 back (5 ways), 2 of 2 cross (1 way) - and now everyone will have crossed the bridge! To get the total number of ways we have to check, we use the multiplication principle in combinatorics: 15*2*10*3*6*4*3*5*1 = 324,000, which is a small number for a computer. The implementation can look like this: (note, we need a special case if there is only one people to cross the bridge!)

```
int best,n,side[6]; // side[x]=0 if on the start side, =1 if on the other side
vector<int> time;

void go(int tm, int left)
{
  for(int i=0;i<n;i++)
    for(int j=i+1;j<n;j++)
      if (side[i]==0 && side[j]==0) {
        // i and j will cross
        side[i]=side[j]=1;
        int new_tm=tm+max_of(time[i],time[j]);
        if (left==2) {
          if (new_tm&lt;best) best=new_tm;
        } else {
          for(int k=0;k&lt;n;k++)
            // k will go back
            if (side[k]==1) {
              side[k]=0;
              go(new_tm+time[k],left-1);
              side[k]=1;
            }
        }
        side[i]=side[j]=0;
      }
}
```

One way to improve this solution is to apply memoization. That is, we solve the problem from each given *state* (ie, which side of the bridge everyone is one). The total number of states are 2^n where n is the number of people (one could argue that it's 2 times because you also need to keep track on which side the flashlight currently is - but this is not necessarily so, if you consider crossing and going back as one move). See **tomeks** solution in the SRM for a nice implementation of this using bitmasks. I will not elaborate on this any more because there is another superior solution:

It's obvious that we want the slowest people to cross the bridge as little as possible, since that will affect the total time a lot. It makes sense to let to two slowest (call them Y and Z) cross the bridge at the same time. To do so, we need help by the two fastest (call them A and B): Let A and B cross, make A go back, let Y and Z cross, make B go back. We have then managed to get the two slowest over the bridge, and now have two less to worry about. Actually, using this method alone would work on all example cases (and is probably one reason that so many failed system tests)! However, assume that A is much faster than B (and thus everyone else, since B is the second fastest). Then it would make more sense to use A all the time. That is, A and Z cross, A go back, A and Y cross, A go back etc. This is needed for the case {1,98,99,100} for instance.

So do we just compare how well these two methods do and select the best? Not quite. Consider the case {1,25,26,98,99,100}. Using method one will take 325 minutes, and using method two will take 352 minutes. But the best solution takes only 302 minutes [(1,25) cross, 1 go back, (99,100) cross, 25 go back, (1,98) cross, 1 go back, (1,26) cross, 1 go back, (1,25) cross]! Here we mix the two methods above, so we need to be more clever. What we do is to consider "How do we get the two slowest people left across the bridge?" We try both methods, and select the fastest method. Now we got two people less. We repeat the check with the two now slowest people etc.

The only thing that's missing now is when there are three or less people left. We need a special case for since that won't require 4 crossing (both methods above use 4 crossings to get 2 people less on the original side), but at most 3. These three cases can easily be handled separately:

- 1 or 2 people - just make them cross
- 3 people - let A and B cross, A go back, A and C cross.

See my solution in the practice room for a short implementation of this solution. Notice that this solution is very fast. The bottleneck in the solution is the sorting of times in order (so we can find who is fastest and slowest instantly), and as we all know, we can sort pretty fast.

Masterbrain Discuss it

Used as: Division-I, Level 2:

Value	600
Submission Rate	54 / 106 (50.94%)
Success Rate	51 / 54 (94.44%)
High Score	**radeye** for 579.33 points

Reference implementation: **Yarin** *(practice room)*

Implementation

This problem is basically just Mastermind, except that the one who makes the secret combination is allowed to lie with his response exactly once. This sure makes the task of breaking the code harder for a human, but luckily it doesn't really make solving the problem with a computer any harder!

We begin by looping through all possible codes (only $7^4 = 2401$) - for instance using for nested for-loops from 1 to 7 - and for each such code check if it's a possible code. How do we know if the code can possibly be the secret code? *If* it would be the correct code, the *correct* response from the codemaker (ie if he does not lie) on the guesses we have made would match with the responses we got in all except exactly one case. So, for each code, we loop through all guesses, check what the proper reply from the codemaker (if he does not lie) would be for this guess on the code we're currently checking. If this match with the response we got in all cases except one, this could be the secret code, so then we increase a counter.

To check the proper reply of a guess, we should count how many digits are in the correct position (black pegs) and the incorrect position (white pegs). Checking the number of black pegs is easy, loop through all 4 positions and check if it's the same digit in both the code and the guess. If this is not the case, we update how many unmatched digits we have of each digit 1 to 7. For instance, if the code is 2432 and the guess is 4222, we have one black peg (the rightmost 2) and one unmatched 2, 3 and 4 in the code and two unmatched 2:s and one unmatched 4 in the guess. By summing the minimum of these unmatched values for the result and the code, we get the number of white pegs: that is min(0,0)+min(1,2)+min(1,0)+min(1,1)+min(0,0)+min(0,0)+min(0,0) = 2 white pegs.

A more common method to check white pegs is probably to loop over all pair of positions and check if they match. This requires that positions that have been matched are marked (so we don't count them more than once), plus it's slower since we have two nested loops.

Roundabout Discuss it

Used as: Division-I, Level 3:

Value	800

Submission Rate	23 / 106 (21.70%)
Success Rate	6 / 23 (26.09%)
High Score	**cnettel** for 460.63 points

Reference implementation: **Yarin** *(practice room)*

Implementation

This is the first time in a SRM I've seen a Divison-I, Level 3 problem worth only 800 points. The low score was quite misleading though, as can seen by the submission and acceptance rate as well as the highest score. The reason the problem was only worth 800 points was most likely because of the nature of the problem: simulation. Or, if you want, do what the problem states *exactly*, don't try to be clever. I've said it before. Sadly, I didn't listen to my own advice this time. Or maybe it was just that the problem was harder to interpret than most other simulation problems on TopCoder.

The first thing we need to do is to find a suitable datastructure to represent the state of the roundabout at a given time. We don't need to keep track of which car is which once it's in the roundabout; we only need to know where it's heading and it's location. Thus a string of 4 characters can describe the actual roundabout. Also belonging to the state is the queue of cars at each direction, waiting to enter the roundabout. One can use a proper queue datastructure for each direction, but it's probably easier to just use an array of strings, one element in the array for each direction, and the first character in each string the car which is in the front of that queue.

Next is the matter of how long we should simulate. Obviously until no car is left, but when is that? We can't just check that the roundabout is empty and that no car is waiting, because maybe some car will enter the queues (and thus the roundabout) later! One easy way is to first count the total number of cars that we should simulate (the total number of letters in north, east, south, west) and for every car that leaves the roundabout we decrease this number. Once it's zero, we're done. Even easier is to simulate sufficiently many steps, but only update the return value once something actually happens (ie a car leaves the roundabout). During challenge phase, I saw some really horrendous while-loop termination condition, involving disjunctions of 8 or more expressions, and some of the solutions that failed was due to incorrect terminating conditions in the while-loop.

Now to the actual simulation: At the beginning of each time step, we check in the four directions whether or not a new car arrives. If so, we add this car to the end of the queue for this direction. We then check if there are cars waiting from all four directions at the same time. As mentioned in the problem statement, this is a special case and must be considered below.

Before updating the positions of the cars in the roundabout, we should make a backup copy of the state. This is crucial, as all cars act on how the situation was in the previous second. If we start writing to state variable at the same time as we read it, some car (because of the circular structure) will inevitably act on the cars position in *this* second instead of the previous one.

For each position in the roundabout, we should now determine which car, if any, should occupy the position. If there was a car the previous second in the previous position (that is, if the current position is east, the previous is south etc) in the roundabout, that car will always occupy the current position (a car never stops in the roundabout!) *unless* it exited the roundabout in the previous position - in that case it's simply removed. Otherwise this position is free to be occupied by the first car in the corresponding queue *if* no car is about to the enter the roundabout from the previous position (here, as always, we should check the backup state, not the one we're writing to). Here we have the special case - if cars try to enter from all four directions at once, the car that comes from the north will have precedence.

And that's basically it. The error that I, and several others, made was that if a car was about to exit the roundabout, a car that was trying to enter the roundabout in the next position was allowed to do so, even though this is not allowed.

By **Yarin**

TopCoder Member