Statistics

Match Editorial

Archive
Printable view
Discuss this match

SRM 189
Wednesday, March 31, 2004

Match summary

Division 1 had a relatively simple easy problem, where rounding and double imprecision were finally not much of an issue. The medium problem turned out to be pretty tricky, and required a dynamic programming approach that stumped most coders. The hard problem was a bit more straightforward, but dealt with dates, which tend to be tricky. Division 2 had a rather rough time too, as coders faced a rather difficult easy problem, and had to write a binary search for their hard problem. When systests finished, bladerunner, Jan_Kuipers, and radeye took the top 3 division 1 spots. In division 2, first timer yujrvbym (say that 5 times fast) narrowly edged out therealmoose and newvision to catapult himself into division 1.

The Problems

CutoffRounder Rate It Discuss it

Used as: Division Two - Level One:

Value	300
Submission Rate	158 / 180 (87.78%)
Success Rate	109 / 158 (68.99%)
High Score	**therealmoose** for 298.69 points (1 mins 53 secs)
Average Score	223.16 (for 109 correct submissions)

There are two approaches to solving this problem. The first is simply to use doubles. The constraints ensure that the fractional part of the number will not be equal to the cutoff, so we don't have to worry about issues with double imprecision. Hence, we can parse the strings into two doubles and compare the fractional parts as follows (assuming n and c are doubles representing num and cutoff):

```
if(n-(int)n >= c)return (int)n+1;
else return (int)n;
```

If doubles make you nervous, you can solve the problem without them by looking at the parts of the strings after the decimal point as integers. In order to make them directly comparable, you should pad the fractional parts with zeros on the right so that they are the same length. Then, you can compare them as integers, which aren't subject to any risky imprecision issues. If the constraints allowed for the fractional parts to be equal, you would have to do something like this or you might run into double imprecision issues.

Display Rate It Discuss it

Used as: Division Two - Level Two:

Value	500
Submission Rate	102 / 180 (56.67%)
Success Rate	64 / 102 (62.75%)
High Score	**newvision** for 483.85 points (5 mins 13 secs)
Average Score	321.19 (for 64 correct submissions)

Used as: Division One - Level One:

Value	250
Submission Rate	168 / 171 (98.25%)
Success Rate	138 / 168 (82.14%)
High Score	**Eryx** for 243.15 points (4 mins 47 secs)
Average Score	197.82 (for 138 correct submissions)

The problem statement for this one made things seem a bit more complicated than they actually are. The most obvious way to do this is to use a series of translations and scalings to fulfill the requirements. It turns out that this is pretty simple to do. First, you should translate the points so that the minimum of the x coordinates is 0, as is the minimum of the y coordinates. To do this, you simply subtract the minimum x coordinate from all of the x coordinates, and do the same thing for the y coordinates. Hence, {10,20,30} becomes {0,10,20}. Next, find the maximum value of all the x coordinates, maxX. To scale all of the x coordinates, multiply them all by 1000/maxX. This will ensure that the largest one is exactly 1000, and that all of the other ones are between 0 and 1000, inclusive. We do the same thing for the y coordinates. Hence, by translating and then scaling, we get the right maxs and mins, and we preserve the relative distances in each direction. This is actually a little bit more work than we need to do though. We can condense the translation and scaling into a single step by first finding the min and the max in each direction. Then, we simply set x[i] = (x[i] - minX)/(maxX-minX) - ignoring the rounding step.

A closing note on rounding in this problem. We can do the rounding with or with out ints. If we want to be on the safe side and avoid doubles, we can use the following formula, assuming x[i], minX and maxX are ints:

```
x[i] = (x[i] - minX + (maxX-minX)/2)/(maxX-minX)
```

Alternatively, it is safe to use doubles in this problem because 0.5 can be represented exactly using doubles. Hence, if the actual value is `x.5`, we can be sure that we will round correctly.

Mortgage

Rate It Discuss it

Used as: Division Two - Level Three:

Value	1000
Submission Rate	21 / 180 (11.67%)
Success Rate	6 / 21 (28.57%)
High Score	**yujrvbym** for 821.20 points (13 mins 53 secs)
Average Score	630.08 (for 6 correct submissions)

If it weren't for all the rounding, we could solve this problem explicitly with a formula (which you can find online if you are interested). However, as it is stated, I don't know of any simpler way than to try a bunch of payment plans and find the smallest one that pays off the debt. The most obvious implementation of this is to start by trying to pay $1 a month, and see if that works. Then see if $2 a month works, and so on until you find one that works. However, with return values in the billions, this method will timeout by a large margin. Luckily, there is a faster way: binary search. Note that if paying $x a month is sufficient to repay the debt, then $x+1 clearly is also enough. This observation means that a binary search will work properly, requiring that we check only lg(loan) potential monthly payment. Each of these payment plans requires 12*terms iterations to simulate, so our runtime is O(lg(loan)*terms*12), which is plenty fast. The tricky thing about binary searches are the off by 1 errors. The simplest way to do this sort of binary search, in my opinion, is to keep a lower bound that is exclusive, and an upper bound that is inclusive. Then, in each iteration, we evaluate the value half way between the upper and lower bounds. We move the lower bound up to the midpoint if it is not enough, and the upper bound down if it is. At the end, hi = lo+1, so we just return hi.

```
while(hi>lo+1){
   long mid = (hi+lo)/2;
   boolean works = highEnough(mid,loan,interest,term);
   if(works)
      hi = mid;
   else
      lo = mid;
}
return (int)hi;
```

Then we just have to write the highEnough function, which is a straightforward simulation:

```
boolean highEnough(long monthly, long owed, int interest, int term){
   long start = owed;
   for(int i = 0; i<term*12; i++){
      owed-=monthly;
      if(owed<=0)return true;
      long acrued = (owed * interest + 11999) / 12000;
      owed = owed + acrued;
      if(owed > start)return false;
   }
   return false;
}
```

ScheduleResources

Rate It Discuss it

Used as: Division One - Level Two:

Value	600
Submission Rate	39 / 171 (22.81%)
Success Rate	11 / 39 (28.21%)
High Score	**Jan_Kuipers** for 529.82 points (10 mins 37 secs)
Average Score	379.93 (for 11 correct submissions)

There are a few simplifications to this problem which make it much easier. The first is to notice that once we schedule the elements of A, scheduling the elements of B in the same way is optimal. This follows from the fact that if there are multiple things in B to do, it doesn't matter which order we do them in, as the total time is the same. Also, there is never any reason to break up an element of A and do half of it, then something else, then the other half. Consider the case where we do this by doing half of A_1, some of A_2, and then the rest of A_1. If we move the part of A_2 before the first part of A_1, we will get done with A_1 at the same time, and get done with A_2 no later, so we are at least as well of. So, now we are just trying to find an optimal ordering on the elements of A. This is a fairly standard problem that comes up pretty often. The trick is to find the optimal ordering for each subset of A. Since the finish time for the input part of the problem is the same for every ordering of the subset, this amounts to finding ordering that gives the lowest output finishing time. Once we know this, we can consider extending the ordering by adding one more process to it, giving a larger subset. All we need to keep track of for a given subset if the best time found so far to complete the ouputs. So, basically, we have dynamic programming where our subproblem is to figure out the quickest we can complete some subset. It turns out that if we use bitmasks for our subsets this can be done very simply.

```
int[] best = new int[1<<A.length];
int[] sum =  new int[1<<A.length];
Arrays.fill(best,1000000000);
best[0] = 0;
for(int i = 0; i<best.length; i++){
   for(int j = 0; j<A.length; j++){
      if(((1<<j)&i) == 0){
         sum[i|(1<<j)] = sum[i] + A[j];
```

```
            best[i|(1<<j)] = Math.min(best[i|(1<<j)],Math.max(sum[i|(1<<j)],best[i]) + B[j]);
         }
      }
   }
   return best[best.length-1];
```

`Math.min(best[i|(1<<j)],Math.max(sum[i|(1<<j)],best[i]) + B[j])` bears some extra explanation, since it is the crux of the algorithm. Starting from the inside out, `sum[i|(1<<j)]` represents the sum of all the A's in the subset represented by `i|(1<<j)`. We can start the final B after all of the A's have finished, and after all of the proceeding B's have finished. The finish time of all the proceeding B's is represented by `best[i]`. Therefore, `Math.max(sum[i|(1<<j)],best[i])` represents the time at which the final B may start, and the whole second part of the Math.min call represents the time that the final B will finish. This, of course, is what we are trying to minimize, so we update `best[i|(1<<j)]` (which represents the optimal B finishing time for the subset) if the new value is small enough.

Finally, it turns out that there is a polynomial algorithm for this problem, which is extremely simply. I'll leave it up to the readers to figure it out, and post the answer in the round tables if no one else does.

BlueMoons

Rate It | Discuss it

Used as: Division One - Level Three:

Value	900
Submission Rate	68 / 171 (39.77%)
Success Rate	17 / 68 (25.00%)
High Score	**jms137** for 646.37 points (19 mins 28 secs)
Average Score	484.45 (for 17 correct submissions)

Calendar related problems are typically messy. This one is no exception, and there are a lot of different ways to go about it. See jms137's solution for a rather short, though complicated implementation. I'm going to descibe a solution that is a little more work, but a bit more straightforward. Basically, we will just add and subtract 29.35 days at a time, adjusting the date accordingly. The first thing to do is to parse the dates into 3 integers each: day, month and year. To avoid, rounding errors, we'll using integers for everything so instead of having an int for the day, we'll have an int for hundredths of a day. An important decision to make when doing the parsing is whether to use 0-based or 1-based days and months. In my solution, I did everything 0-based, since I typically find it a bit easier, but you can do 1-based with about the same amount of trouble, and it's sort of a tossup as to which is better. Once I have the date of some full moon parsed, I start going backwards, one full moon at a time, until I get to the last full moon before the start date. Then, I start counting up, one full moon at a time, until I get past the end date. If I have have the same month two full moons in a row, and I am within the interval, I've found a blue moon. Though this sounds simple, counting one full moon at a time is a bit tricky. When counting down, I subtract 2935 hundredths of a day at a time. If I cause the number of hundredths to be below 0, I add to it while decrementing the month until the number of hundredths is greater than or equal to 0. Counting up is pretty similar, except in reverse. One important thing to note is that, in the case of Februrary, we may need to go up or down 2 months to get to the next full moon.

```
int[] days = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
public int count(String interval, String fullMoon){
   String[] sp = interval.split(" to ");
   int sm = Integer.parseInt(sp[0].substring(0,2)) - 1;
   int sy = Integer.parseInt(sp[0].substring(3));
   int em = Integer.parseInt(sp[1].substring(0,2)) -1;
   int ey = Integer.parseInt(sp[1].substring(3));
   int md = Integer.parseInt(fullMoon.substring(0,2))*100
      +Integer.parseInt(fullMoon.substring(3,5)) - 100;
   int mm = Integer.parseInt(fullMoon.substring(6,8)) - 1;
   int my = Integer.parseInt(fullMoon.substring(9));
//this loop counts down until we get to the last full moon before the start date
   while(my > sy || my == sy && mm >= sm){
//count back 29.53 days
      md -= 2953;
      while(md < 0){
//md < 0 so go back a month
         mm--;
         if(mm < 0){
            mm = 11;
            my--;
         }
//add the number of days in the month to md.
//Hence if md was -x, it becomes daysInMonth - x, which is what we want
         md += days[mm] * 100;
         if(mm==1 && my%4==0 && (my%100!=0 || my%400==0))md+=100;
      }
   }
   int prevMonth = -1;
   int ret = 0;
//this loop counts up until it gets past the end date
   while(my < ey || my == ey && mm <= em){
      md += 2953;
      int daysInMonth = days[mm] * 100;
      if(mm==1 && my%4==0 && (my%100!=0 || my%400==0))daysInMonth+=100;
      while(md >= daysInMonth){
```

```
            md -= daysInMonth;
            mm++;
            if(mm == 12){
               mm = 0;
               my++;
            }
            daysInMonth = days[mm] * 100;
            if(mm==1 && my%4==0 && (my%100!=0 || my%400==0))daysInMonth+=100;
         }
//if there were two full moons this month, and we are within the interval, increment ret
         if(mm==prevMonth && (my > sy || my==sy && mm >= sm)){
            ret++;
         }
         prevMonth = mm;
      }
      return ret;
}
```

By **lbackstrom**
TopCoder Member