Statistics

Match Editorial

Archive
Printable view
Discuss this match

SRM 187
Tuesday, March 16, 2004

Match summary

Well, SRM 187 was somewhat of an upset. Number 2 ranked **tomek**, number 3 ranked **John Dethridge** and number 4 ranked **ZorbaTHut** were all competing and expected to finish 1, 2, 3, but ended up placing 4th, 9th and 11th. All lost points off their 3000+ target ratings. Any little mistake or hesitation is very costly to your rating when your are so far up in the standings. Meanwhile, **haha** wins the SRM on the strength of the fastest 900 point solution more than 50 points ahead of the next competitor, **venco**, in second and **Ryan** only 12 points back in third. Twenty-two coders got the hard problem correct this time.

The top five Division-II finishers kraju, nikal, ggoprea, kolodrub, and Ragnarok all made the jump to Division-I as did thirteen other Division-II coders. Nine coders got the hard problem correct this time in Division-II this time, which is a little better than last SRM's zero. There would have probably been several more good solutions, but there was an unfortunately timed system security modification which disabled java.awt.* including the Polygon class (which happens to have a built in point-in-polygon method). Several people wasted a lot of time before it was clear that Polygon was not going to work. Had I understood what was going on, I would have put a warning in the problem statement, but I was clueless.

The Problems

OfficeParking

Rate It Discuss it

Used as: Division Two - Level One:

Value	300
Submission Rate	214 / 239 (89.54%)
Success Rate	173 / 214 (80.84%)
High Score	**jwjchap** for 296.96 points (2 mins 52 secs)
Average Score	235.24 (for 173 correct submissions)

This is an example of getting too much information in a problem statement which then seems to make the problem more difficult that it really is (a characteristic of many real world problem statements). We are asked to find the total number of parking spaces used on a given day given a sequence of arrivals and departures complete with the handle of the arriving or departing TopCoder. At first glance it might seem that the way to go is a simulation keeping track of who is in each parking space and always putting the next arrival in the lowest numbered free parking space. And this clearly works and isn't terribly complex, but it is much more complex than it needs to be.

Because TopCoders always park as close to the building as they can, the total number of used parking spaces is the same as the maximum number of parking spaces ever used at the same time. When a coder parks in the farthest used parking space, it can only be because all the closer spaces were full. Similarly when a coder departs, it does not matter which parking space is emptied, at least for the purposes of determining the maximum or total number of spaces used. So you can completely ignore the handle information and only use the event "arrives" or "departs".

Here is all it takes:

```
{
int max = 0 , c = 0 ;
for ( i=0; i < events.length; i++ )
   {
   if ( events[i].endsWith("arrives") ) c ++ ;
   else                                 c -- ;
   if ( max < c ) max = c ;
   }
return max ;
}
```

jwjchap's high score of 296.96 points in only 2 mins 52 secs shows that solution can programmed very quickly if you recognize the trick right away. The high average score of 235.24 out of 300 shows most coders were on top of this problem.

Note: If you use `.index()>-1` instead of `.endsWith()`, be sure to test for `" arrives"` to insure you do not match "arrives" with part of the handle.

DNASingleMatcher

Rate It Discuss it

Used as: Division Two - Level Two:

Value	500

Submission Rate	170 / 239 (71.13%)
Success Rate	128 / 170 (75.29%)
High Score	**jwjchap** for 496.73 points (2 mins 18 secs)
Average Score	360.53 (for 128 correct submissions)

This problem is remarkably similar to the Div-I 900 point problem. What makes them so different in difficulty? Well, for one, this one uses two strings while the other uses three strings. But that is not the big difference. The biggest difference is that the strings in this problem are only from 1 to 50 characters long. With such short strings you can afford to use some slow brute force algorithms that would take way too long if the strings were much longer. The most direct brute force algorithm I could think of is O(n4), which is fast enough for n = 50.

```
int max = 0 ;
for( substringLength = 1 ; substringLength <= sequence1.length() ; substringLength ++ )
   for ( startPos = 0 ; startPos < sequence1.length()-substringLength ; startPos ++ )
      if ( sequence2.indexOf(sequence1.substring(startPos,substringLength)) > -1 )
         max = substringLength ;
return max ;
}
```

The simple built in `index()` method/function in all languages has a linear expected running time for random inputs, but can actually take O(n2) time in the worst case. Of course we have the worst case in the systests, but with n <= 50 it really doesn't matter.

If you have a pre-written linear-time string index method in your bag-of-tricks (and shouldn't everyone?), then you can plug that into the above code and reduce the complexity to O(n3) with almost no effort. But those algorithms are notoriously difficult to debug, so typing one in from memory during the SRM is difficult. There are several to choose from, but probably the simplest is "Algorithm Z" in Gusfield's *Algorithms on Strings, Trees, and Sequences* book.

Another simple optimization you can make to the above algorithms is to replace the outer loop with a binary search for the longest substring length. This yields algorithms with O(n3log n) and O(n2log n) time complexity.

A whole different approach yields an O(n2) algorithm. Imagine placing the two strings beside each other. Then in one pass iterate down both strings, incrementing a counter if the corresponding characters match or setting it to zero if they do not match. The maximum value of this counter gives you the longest common substrings which start in the same relative positions in the two strings. Then do the same thing with the two strings offset one character position, then two, then 3, etc. (in both directions). Each time you get the longest common substrings which start at a particular offset from each other. Viola! n2 and you didn't even have to do any substring() calls or use more than about four temporary variables!

In the Division-I 900 point problem discussion below, even faster algorithms will be discussed.

PointInPolygon

Rate It Discuss it

Used as: Division Two - Level Three:

Value	1000
Submission Rate	42 / 239 (17.57%)
Success Rate	11 / 42 (26.19%)
High Score	**ggoprea** for 705.77 points (20 mins 12 secs)
Average Score	478.32 (for 11 correct submissions)

Point in polygon is a classic problem, with many known algorithms from which to choose. Some are specialized for convex polygons, or stars, or other shapes, or use lots of preprocessing so that many point tests can later be made rapidly for the same polygon. These are of little interest in the SRM environment. Some algorithms are simple and brute force and would not be considered for any real application requiring lots of testing. Those are just the ticket for throwing together in the arena.

One of the simplest, theoretically, is based on the Jordan Curve Theorem. This theorem states that given a simple closed curve (of which our polygon qualifies), if you cast a ray from a point out to infinity, you will cross the curve an odd number of times if you started in the curve's interior and you will cross the curve an even number of times if you started in the exterior, provided you do not hit points on the curve tangent to the ray or cusps. These special points can be handled as special cases with different counting rules (0 for a normal tangent, 1 for an inflection point, 0 for a cusp unless your are tangent to the cusp then 1, it's a mess). It is usually simpler just to pick another direction until you get one without any special points, or move your test point slightly if you know you don't move it across an edge.

Because the points in this problem are specified on an integer grid, a simple trick can be used to avoid all the special cases which complicate the ray casting. First test to see if you are on the boundary. If so, you are done. If not, then you know you are at least 1 unit of distance from the nearest polygon edge, so you can safely move your test point one half unit in any direction. To keep all your arithmetic in integers, multiply all the coordinates by two and move the test point 1 unit in X and 1 unit in Y. Now it is completely safe to cast a ray in any of the orthogonal directions without hitting a vertex or being tangent to any edge.

Let's pick the positive X direction. Since all the edges are either exactly horizontal or exactly vertical the test in the first loop below is sufficient to test for the on-the-boundary case, and the test in the second loop is sufficient to count vertical edges to the right of the testPoint which intersect the horizontal line, X = testPointX.

```
{
int n = verticies.length ;
for ( int i = 0 ; i < n ; i ++ )
   {
   int x1 = Integer.parseInt(vertices[i].split(" ")[0]);
   int y1 = Integer.parseInt(vertices[i].split(" ")[1]);
   int x2 = Integer.parseInt(vertices[(i+1)%n].split(" ")[0]);
   int y2 = Integer.parseInt(vertices[(i+1)%n].split(" ")[1]);

   if ( Math.max(x1,x2) >= testPointX && Math.min(x1,x2) <= testPointX  &&
        Math.max(y1,y2) >= testPointY && Math.min(y1,y2) <= testPointY )
      return "BOUNDARY" ;
   }

testPointX = testPointx * 2 + 1 ;
testPointY = testPointY * 2 + 1 ;
for ( int i = 0 ; i < n ; i ++ )
    {
    int x1 = 2 * Integer.parseInt(vertices[i].split(" ")[0]);
    int y1 = 2 * Integer.parseInt(vertices[i].split(" ")[1]);
    int x2 = 2 * Integer.parseInt(vertices[(i+1)%n].split(" ")[0]);
    int y2 = 2 * Integer.parseInt(vertices[(i+1)%n].split(" ")[1]);

    if ( testPointX < x1 && x1 == x2  &&
       Math.max(y1,y2) > testPointY && Math.min(y1,y2) < testPointY )
       c ++ ;
    }
if ( c % 2 == 0 ) return "EXTERIOR" ;
return "INTERIOR" ;
}
```

Cyberline

Rate It Discuss it

Used as: Division One - Level One:

Value	300
Submission Rate	175 / 178 (98.31%)
Success Rate	156 / 175 (89.14%)
High Score	**tomek** for 293.57 points (4 mins 13 secs)
Average Score	257.81 (for 156 correct submissions)

Our first task in division I is to extract the last Cyberword from a Cyberline of Cyberpoetry. Cyberpoets like lots of punctuation and using numbers, '@', and '-' inside words. Who can understand what motivates these Cyberartists? Not I. But does it rhyme? Our simplified rules state that hyphens don't affect rhyming and should be removed, joining the parts of words they connect saving the letters, numbers and '@' characters. The punctuation is treated like spaces, causing word breaks, but otherwise ignored.

The fastest score took a whole 4 mins and 13 seconds, by **tomek** for 293.57 points which probably indicates that the problem was harder to read and understand quickly than I thought. But the submission rate (98.31%), success rate (89.14%), and average score of 257.81 out of 300 sounds like people nailed this problem pretty easily. And rightly so, as it can be done in about 1.1 lines in Java. Just remember to take out the hyphens BEFORE you do the split on spaces. I think the trim() is unnecessary because of the way the Java split() works, but you might need the equivalent of trim() in other languages.

```
{
String[]w=cyberline.replaceAll("-","")
                   .replaceAll("[~!@#$%^&*()_-=+{}|/.,;:]"," ")
                   .trim().split(" ") ;
return w[w.length-1];
}
```

NumericalIntegral

Rate It Discuss it

Used as: Division One - Level Two:

Value	500
Submission Rate	154 / 178 (86.52%)
Success Rate	128 / 154 (83.12%)
High Score	**antimatter** for 487.82 points (4 mins 30 secs)
Average Score	381.09 (for 128 correct submissions)

Ah numerical integration. Lots of the dreaded doubles all over the place, lots of worrying about accuracy. Too bad Pops missed this contest, he would have loved it. Numerical analysis as a branch of Computer Science is a big field, and it has lots of applications and employs lots of programmers. The solution of systems of coupled partial differential equations (PDEs), and the Navier-Stokes equations specifically (which govern turbulent flow), account for almost all of the computer time spent in the big supercomputer centers. If you come up with some improvement in the speed or accuracy of these simulation codes it can mean really big money. Everybody should know at least a little about the basic numerical computation problems and algorithms. They are nothing to be feared, even if they are a little underrepresented in TopCoder competitions. Next week we can solve Maxwell's equations for electromagnetism numerically!

Numerically approximating an integral, sometimes called numerical quadrature, is one of the most basic and fundamental problems. The constrains on this problem were set loose so that any halfway reasonable method should be plenty accurate, especially with a whole eight seconds to calculate one value. Typically something like this would be in a tight loop and be called millions of times. Under those circumstances you can not afford to take lots of iterations with a slowly converging approximation. But here you should have no problem taking 100,000 or a 1,000,000 iterations with a simple method to get plenty of accuracy. All you had to do here was get the basic loop structure right and do a summation.

Calculating the integral, or the area of a region in this case, consists of modeling the region as a series of small pieces. Each piece has a simple to calculate area, but the pieces only approximate the desired area. As the area is divided into more and more smaller and smaller pieces, the approximation gets better (assuming convergence). For this problem it is sufficient to model the area with rectangles (because an extremely smooth and well behaved function was chosen to integrate). Each rectangle has a width of dx and a height of e^{-x^2} the product is the area of one rectangle. The sum of these products over a range of x values is the approximate integral (sound familiar? if we take the limit as dx goes to zero analytically we have the exact integral).

Different numerical integration "rules" differ on how they model the end of those little "rectangles" (differential areas). The rectangular rule takes one value and assumes the height is constant across the differential area. The trapezoidal rule takes the value at both edges and connects them with a line segment forming a trapezoid as the differential area. Simpson's rule models the top of the differential area as quadratic curve. It does this by taking three samples, both edges and the center, and weighting them in the ratio 1:4:1. Then there is Simpson's 3/8ths rule with 1:3:3:1. It goes on and on. More complex rules take more samples and more weights and sometimes samples of derivatives and combining them into more complex models of the differential area in the hope that this will make the approximation converge faster.

This is all it takes to do the integral with the rectangular rule.

```
{
int n = 1000000 ;
double dx = (x2-x1)/n ;
double sum = 0 ;
for ( int i = 0 ; i < n ; i ++ )
   {
   double x = x1 + dx * (i+0.5) ;
   sum += dx * Math.exp(-x*x) ;
   }
return new DecimalFormat("0.00000").format(sum)  ;
}
```

The function we integrated, the Gaussian, shows up in lots of places, especially in statistics where it is the most common probability distribution. The integral of this function from -infinity to x is called the error function, ERF(x), and is the cumulative probability distribution for the Gaussian. This integral varies from our integral only by adding a constant term. (Aren't you glad I didn't ask you to integrate from -infinity? That is a lot of steps!).

DNAMultiMatcher

Rate It | Discuss it

Used as: Division One - Level Three:

Value	900
Submission Rate	50 / 178 (28.09%)
Success Rate	22 / 50 (44.00%)
High Score	**haha** for 661.01 points (18 mins 33 secs)
Average Score	477.80 (for 22 correct submissions)

As I mentioned above, this problem is very similar to the Division-II 500 point problem, DNASingleMatcher, where I described algorithms for $O(n^4)$, $O(n^3 \log n)$, $O(n^3)$, $O(n^2 \log n)$, and $O(n^2)$. But here we have three strings instead of 2 and they can be 2500 characters long instead of 50. This pretty much makes it a completely different problem. While $O(n^4)$ was fine for the easier problem, $O(n^2 \log n)$ is probably as slow as you can go without timing out here. It is perhaps surprising that it is possible to do the longest common substring problem for any number of strings in time which is O(n) where n is the sum of the lengths of all the strings. This result was not discovered until 1973!

BradAustin was the only one that I saw who *did* have a fancy pre-coded linear time string search routine in his bag-of-tricks, and used it with binary search on the substring length, to build a successful solution to this problem which runs in $O(n^2 \log n)$.

Two $O(n^2)$ dynamic programming solutions were used by various coders. One uses a two dimensional array of n^2 elements. While this was sufficient for this particular problem size, clearly this technique could not scale much larger. **haha**'s solution uses only a couple of size n arrays making it much more scaleable. The idea of the dynamic programming solutions is that you can perform essentially the same iteration and calculations as in the "shifting and counting" algorithm for the Division II 500, except you store all the count values in an array. Specifically, as you iterate, each cell of the array holds the length of the best known substring match (in string b) that begins at that location in string a. You calculate this twice, once for the pair (sequence1, sequence2) and once for the pair (sequence1, sequence3). Then if you pairwise min all the elements of the two arrays, the maximum value in the resulting array is the length of the longest common substring to all three.

radeye coded a very interesting algorithm based on hash tables. The outside loop is a binary search on the length of the longest substring. For a given length, all substrings of that length from the three strings are put into three hash tables. There are O(n) substrings of length O(n) so all this hashing is $O(n^2)$. The hash tables are then intersected using a java builtin method which is (I assume) O(n). if there is anything in the resulting hash table then there is a common substring of that length. Now throw away the hash tables and start over for the next length in the binary search. The whole process is $O(n^2 \log n)$ in time and $O(n^2)$ in space as I see it. Very clever radeye.

tomek does some very clever modular arithmetic which, if I follow it correctly, allows him to calculate the O(n) hash codes for the substrings for a given length in O(n) time. Amazing. But he has to do an extra sorting step, combined with the outside loop binary search on the substring lengths which appears to give him an overall $O(n \log^2 n)$ time complexity with something like a $1-2^{-50}$ probability of getting the right answer. This took only 25 mins and 40 seconds.

This problem can be solved asymptotically faster than tomek's solution but at a cost of great complexity. The key is to build a data structure known as a "suffix tree". A suffix tree holds all the suffixes of a length n string in only 2n nodes and can take only linear time to construct. Straightforward methods of constructing the suffix tree, or something similar, can be done in $O(n^2)$ time, and it is also simpler to make it with $O(n^2)$ space. **John Dethridge** does something equivalent to this with arrays of strings (a suffix array), and then does a sorting and matching procedure similar to tomek. **ZorbaTHut** builds an $O(n^2)$ trie equivalent to the suffix tree. Once you have the trie, sorted suffix array or suffix tree, you can traverse it in time linear in the size of the data structure and find the longest substrings. Look in the practice room for my linear time implementation for this problem using Ukkonen's algorithm to build the suffix tree. It uses the technique of concatenating all three strings with unique end of string markers and putting them in one suffix tree. Then one bottom up traversal gives you the answer.

By **Rustyoldman**
TopCoder Member