15-295 Spring #14
April 27, 2016
http://www.codeforces.com/group/KIrM1Owd8u/contest/207044

This contest had particularly difficult-to-read problems.
I'll explain each problem (that I worked on) in my own words,
then give some idea of the algorithms. Thanks to those who
helped by entering solutions.

---DS

A. Three Seamarks
==========

You're given three (distinct -- although it does not say
this) points in the plane called M1, M2, and M3. You're
also given two angles A12 and A23. The angles are integral
in degrees and are in the range [1,179]. The goal is to
find a point P in the plane such that angle [M1,P,M2] = A12
and [M2,P,M3] = A23. You are told that such a point exists.

Consider the set of points:

S12 = {P | angle [M1,P,M2] = A12}

By basic geometry, it's not hard to see that this set of
points is a subset of the union of two circles. Call these
circles C12 and D12.

So the algorithm is to construct this pair of circles and
also C23 and D23, and construct all intersections Among the
"12" circles and the "23" circles (there are at most 8) and
try each such intersection to make sure it solves the
problem.

You need to be able to find the required circles. (Which I
won't explain without a whiteboard.) In addition the
primitives involved with this are CIRCLE_INTERSECTION and
COMPUTE_THE_ANGLES_OF_A_TRIANGLE. The former can be done
using some vector manipulation. The latter can be done via
the law of cosines.

There is one additional nasty case, which is best separated
out. If the three points are the corners of a square, and
the angles are 45 degrees, then two of these circles are the
same. And any point on them works. I first tried picking a

random point on this circle, but it did not work (I don't know why). I did successfully handle this by breaking out the case and generating the fourth point of the square as the solution.

B. High-Speed Pedestrian walkway 1.0

Here's a restatement of the problem. You're given an infinite supply of tiles of dimension 1*1, 1*2, 1*3, …, 1*m. The tiles of width i come in c_i different colors. The problem is to count (modulo 10^9+9) the number of different ways to tile a 2*n strip. Here n can be huge (up to 10^18) but m is at most 9.

This can be done with matrix exponentiation, and I'm not sure if there's a better way.

This is a variant of the problem of tiling a 2 * n grid with 1 * 2 dominoes. We can write a similar recurrence where we keep track the lengths of the two rows:

```
<------- n ------->
[.................]
[........]<-- d -->
```

f(n, d) = number of ways to tile the above structure. Then

```
f(n, d) =
   SUM {c_k * f(n-k, d-k)} for k <= d
+  SUM {c_k * f(n-d, k-d)} for k > d
+  c_2 * f(n-1, 0) if d == 0    // to account for vertical 1*2
tiles
```

where k ranges over all possible tiles of length 1...m. Note that d is always at most 9 because we are always putting a tile on the longer row. The answer is f(N,0). By doing this DP we get an O(nm^2) algorithm.

Now N is big, but there's a well-known way to speed up such recurrences, using a similar idea for computing huge Fibonacci numbers: matrix multiplication. Note that f(n,d) only calls f(n,*), f(n-1,*), ..., f(n-9,*). We can create a state vector

v_n = [f(n,0), f(n,1), ..., f(n,9), f(n-1,0), f(n-1,1), ..., f(n-1,9), ..., f(n-8,0), ..., f(n-8,9)]

and the idea is to compute a new state vector

v_{n+1} = [f(n+1,0), f(n+1,1), ..., f(n+1,9), f(n,0), f(n,1), ..., f(n,9), ..., f(n-7,0), ..., f(n-7,9)]

through a matrix multiplication (i.e. v_{n+1} = Mv_n). What will the matrix M look like? It will consist of the coefficients of the corresponding states in the above recurrence. For example, row 3 of M is supposed to compute f(n+1,2) from v_n, so it will have coefficient of f(n,1) be c_1, coefficient of f(n-1,0) be c_2, coefficient of f(n-2,1) be c_3, coefficient of f(n-2,2) be c_4, etc. There is a slight catch where we refer to f(n+1,*) in our recurrence: this can be avoided by trying all possible combinations of tiles to put at the longer and shorter side for that case (instead of just putting a tile at the longer side). We can generate the matrix in O(m^2).

The initial vector v_0 is just [1, 0, 0, ... 0]. The answer is the first coordinate of v_N = M^N v_0. We can exponentiate matrix M in O(log N * 90^3) time, so this runs in time.

C. Cubes
========

D. Camelogistics
================

Here's a note from William Xiao to Andy Liu about this problem. I'm quoting it without permission. I hope that is okay.

Off by one errors are quite annoying for this problem. :(

You probably already figured out that you can take as much apples you can one km at a time. This is because given any alternative approach, you can rearrange your steps so that you first travel between 0 to 1 km, then travel between 1 and 2 km, and so on.

Now, you have to figure out how many apples can be taken one km. If you have n apples on the (i)th km, then you will lose about 2n/k apples as you take all of them to the (i+1)th km. This is because every time you bring k apples 1 km, the camel has to eat 2 apples for going forward and backward (except for the last round, when the camel only has to eat 1

apple because it doesn't have to go back). There is also an edge case when n = 1 (mod k) because then you don't want to bring the last apple forward. I will be a bit more precise below.

I was stuck with off by one errors for quite a while. I found it helpful to make a chart of how many apples you'll lose after 1 km, then try to come up with a formula. In the end, I got the following: Case 1: If n = 1 (mod k), then you lose 2*(n-1)/k apples for travelling 1 km (except when n = 1, in which case you lose your single apple). Case 2: Otherwise, let n' be n rounded up to the nearest multiple of k. Then you lose 2*n'/k - 1 apples.

It is possible to implement this in O(l) time by iterating through each km and keeping track of how many apples we retain. However, this is too slow because l could be up to 10^9. But notice that the number of apples you subtract each rounds remains the same until you lose about k apples. In other words, we might stay in Case 2 for several rounds and the value of n' remains the same, until we no longer round up to the same multiple of k. So we can travel multiple km at once, subtracting off a proportional amount of apples.

So each iteration, we lose at least about 2n/k apples from what we said in the second or third paragraphs. However, you also lose at least k apples from the optimization we just made in the above paragraph. So we lose at least sqrt(n) apples each round. So the total number of rounds is at most O(log log n) when we lose all the apples. Of course, if l is small enough, we short circuit and return a positive number of apples. Each round takes constant computation, which was described above. Hence, the algorithm takes O(log log n) total time (note that this is an upper bound independent of both l and k).

I put my code below, but it might be hard to read:

```
#include <bits/stdc++.h>
using namespace std;

int l, n, k;

int main() {
     scanf("%d %d %d", &l, &n, &k);
```

```
      while (l > 0) {
      if (n <= 1) {n = 0; break;}
      if (n % k == 1) {n -= 2*(n/k); l--; continue;}
      int den = 2*((n+k-1)/k)-1;
      int num = (n-(k*((n-1)/k)+2));
      int l2 = min(l,num/den+1);
      n -= l2*den;
      l -= l2;
      }
      printf("%d\n", n);
      return 0;
}
```

E. The Carpet

This is another poorly worded, and quite difficult, geometry problem. You're given two points A and B, as well as a disk D. A and B can touch the boundary of disk D, but cannot be inside of it. You're supposed to lay out one or more pieces of string in the plane. The pieces should serve to connect A, B, and D, and it should be of minimal length. The string cannot enter the interior of D.

(I originally thought the string had to be a single piece. Because it is described as a "carpet that is rolled out". You know, like a red carpet for a dignitary. This interpretation is wrong as we'll see below. I only found out after getting "wrong answer". The carpet can have multiple pieces. It can also be curved, unlike a normal red carpet.)

Let's assume for a second that D is a point instead of a disk. If the triangle (A,B,D) has all angles under 120 degrees, then the solution is a little spanning tree (with leaves A, B, and D) in the plane consisting of three straight lines that connect A, B, and D to a center point C. C is called a Steiner point. The three angles where the lines meet at C are at 120 degrees. (It's kind of a minimal surface in three dimensions, like a soap bubble. These kinds of diagrams always have 120 degree angles.)

This problem can be generalized to more points. The resulting trees are called "optimal interconnection trees in the plane". It's a beautiful problem. There's a new book

(2015) about the problem with that title. The link below about the book has an animation with five points.
http://diku.dk/english/news/news2015/steiner-book/

Anyway, the 3-point problem can be solved using a circle intersection primitive (described for problem A above).

When D is an actual disk, there are several additional cases you have to deal with. Sometimes the solution will be two straight lines from A and B to a point on the boundary of the disk. (I had to use binary search to handle this case.) Sometimes it will involve a curved portion along the boundary of the disk. I won't go into all the cases.

Useful primitives for this problem include computing the distance between a line segment and a point. Given Side-Angle-Side of a triangle compute the other side.

F. GCD and LCM
==========

The problem is: given g the gcd and m the lcm of k numbers, how many different ordered sequences S of k numbers satisfy gcd(S) = g and lcm(S) = m.

The gcd must clearly divide the lcm, so that is a special case that needs to be removed right at the beginning. Then, you can realize that the problem is equivalent if you divide m by g and simply find sets of numbers whose LCM(S) = m / g.

The next step is to break m / g into its prime factorization. The main realization regards the distribution of primes. For each prime, there must be some number in S that is not divisible by p (or g would have been higher), and there must be some number in S that is divisible by p^k (or m would have been lower).

We can do principle of inclusion-exclusion to count up these possibilities.

Over each combination of states of each prime:
|S| = (number of ways to distribute the numbers)
    - (number of uses where prime used > 0 times)
    - (number of ways where prime used < k times)
    + (number of ways where prime used 0 < times < k)

G. Pots

=======

H. Messenger
============

The problem is the following. You're given a string of length up to 200,000 (watch out --- it can contain blanks and other junk, just not a newline). And you're given a sequence of a particular type of permutation to apply to the order of the characters of the string. Each permutation is described by a parameter k. It reverses the order of the first k characters of the string (in place) , and also reverses the rest of the string (in place).

The goal is to output the result of applying up to 200,000 of these operations.

Corwin did this with splay trees. The trees can be made to support maintaining a sequence of characters under range reversals. It's done by keeping a "reverse bit" in each node of the tree. If the reverse bit is 1, it indicates that everything in the subtree rooted here is reversed.

However it turns out that there is a much simpler way to solve the problem. Put it this way. The set of permutations you can achieve with the moves allowed in this problem is quite small. Play around with it and figure it out yourself.

I. The smallest fraction
========================