October 2010 Contest Problem Editorials

1) POST(written by David Stolp)

As suggested by the problem constraints, there are 2 cases to be dealt with.

Case 1: W≥N and H≥N.

In this case, a O(N^2) solution is possible. We note that Dave will never have to move south or west, except possibly when he's within N of (0,0) or (W,H). In particular, this means that every shortest path will pass through exactly one point on the diagonal consisting of points of the form (i, N-i) (call this Diagonal A), and exactly one point on the diagonal consisting of points of the form (W-i, H+i-N) (call this Diagonal B). Call a path "clean" if it does not pass through any bad intersections. Call a path "dirty" if it may or may not pass through any bad intersections. A path from P1 to P2 is still considered clean if the only bad intersection is P2 itself. We begin by calculating the minimal distance and number of clean paths from (0,0) to each point on Diagonal A, and from (W,H) to each point on Diagonal B. This can be done using a breadth-first search. We can discard points on a diagonal if there is another point on the same diagonal that can be reached in fewer moves. Now for any point P on or between the two diagonals, we can compute the number of clean paths from (0,0) to P as follows:

For every point on PA on Diagonal A, if P is north/east of PA, calculate the number of dirty paths from PA to P as Binom(dx+dy, dx), where Binom is the binomial coefficient, and dx and dy are the differences in x and y values, respectively. Note that 1000000037 = 421*563*4219, so the binomial coefficient can be efficiently calculated using Lucas' Theorem in conjunction with the Chinese Remainder Theorem. Multiply the number of clean paths from (0,0) to PA by the number of dirty paths from PA to P, and add this to our total.

For every bad intersection Pbad south/west of P but north/west of Diagonal A, calculate the number of dirty paths from Pbad to P. Multiply this number by the number of clean paths from (0,0) to Pbad, and subtract this from our total.

Finally, for each point PB on Diagonal B, we multiply the number of clean paths from (0,0) to PB by the number of clean paths from PB to (W,H) and add this to our grand total.

Case 2: W<N or H<N

In this case a O(N^3) algorithm will suffice, although asymtotically faster algorithms do exist. A line sweeping algorithm gets the job done. Assume, without loss of generality, that H<N. We keep track of the shortest distance and number of paths for all points on a vertical line. The line begins at x=0, then we move the line east until it reaches x=W. We proceed using "short jumps" and "long jumps". A short jump is used to move the line past any bad intersections, and a long jump is used to move the line through a section containing no bad intersections. Short jumps are achieved using a breadth-first search in a small area. For long jumps, we have two lines, L1 and L2 for which no bad intersections are located on or between them, and the shortest distance and number of paths is known for every point on L1. Note that the minimal distances for points on L2 are just the corresponding minimal distances for points on L1 plus the distance between L1 and L2. For each point on P2 on L2, and for each point P1 on L1, we calculate the number of paths from P1 to P2 using binomial coefficients (our restrictions guarantee that all such paths are clean). If it's possible for a shortest path from (0,0) to P2 to pass through P1, we multiply the number of paths from (0,0) to P1 by the number of paths from P1 to P2 and add this to our total for P2. This process can be sped up with fast multiplication techniques such as Karatsuba multiplication or Number-theoretic Transform, but these are not necessary to pass within the time limit.

2) STREDUC(written by AnhDQ)

This problem requires a very good skill of dynamic programming (DP) and organizing your program.

At first, we call:

- L_i = the minimum length of string s' which we can reduce from the substring $\overline{s_1..s_i}$.
- $L_i = min\begin{cases} L_{i-1}+1 \\ min\left(F_{i,j,|S_j|}\right) \end{cases}$
  - ➢ The first case means that we add the character s_i to the minimum string s'' which we can reduce from the substring $\overline{s_1..s_{i-1}}$.
  - ➢ The second one means that we (may) delete a substring ending at the position of s_i using the sample string S_j.

Then we declare the DP function F:

- $F_{i,j,k}$ = the minimum length of string s' which we can reduce from the substring $\overline{s_1..s_i}$, do delete a substring ending at the position of s_i using the substring $\overline{S_{j,1}..S_{j,k}}$.
- $F_{i,j,k} = \begin{cases} L_{i-1}\ whether\ k=1 \\ min(F_{t,j,k-1})\ whether\ (s_t == S_{j,k-1}\ \&\&\ G_{t+1,i-1} == true) \end{cases}$

where:

- $G_{i,j} = true/false$ whether we can reduce the substring $\overline{s_i..s_j}$ to an empty string or not.
- $G_{i,j} = true \Leftrightarrow \exists\ H_{i,j,k,|S_k|} == true$

where:

- $H_{i,j,k,t} = true/false$ whether we can reduce the substring $\overline{s_i..s_j}$ to an empty string or not, using the substring $\overline{S_{k,1}..S_{k,t}}$.

- $H_{i,j,k,t} = true\ \Leftrightarrow s_v == S_{k,t-1}\ \&\&\ H_{i,v,k,t-1} == true\ \&\&\ G_{v+1,j-1} == true$

As you see, we must use 4 DP functions to calculate the result, and some of them are nested, so you may have to use forward declaration if needed. And using recursions with memorization may be a good way to simplify your code.

Complexity: $O(l^2 \times n \times max|S_i|)$

3) INTCOMB(written by Zac Friggstad)

The problem is more commonly known as the "Shortest Vector in a Lattice" problem and is not very well understood from the perspective of polynomial-time approximations. One result is that it can not be approximated within 2^(log^c d) for any constant c < 1/2 unless any problem in NP can be solved by a randomized algorithm that almost always answers correctly whose running time is much faster than exponential (see [1] for details). On the other hand, essentially the best know approximation finds a solution whose cost is within a factor 2^{d/2} of the optimum [2]. There have been minor improvements since [2], but the best algorithm is still exponential in its approximation guarantee. A nice description of this algorithm can be found in the book [3].

In the special case of d = 1, elementary number theory tells us that the shortest vector has length equal to the greatest common divisor of the lengths of the input vectors. If d = 2, then there is a polynomial-time exact algorithm that proceeds much like generalization Euclid's algorithm. To find the shortest integer combination of two vectors a,b with d=2, the algorithm proceeds as follows:

```
1) if |a| > |b| then swap(a,b)
2) let x := <a,b>/|b| (where <a,b> is the dot product of a,b and |b| is the length of b)
3) if the absolute value of x is > 1/2 then set b := b-m*a where m is the closest integer to x and go to step 1
4) output a
```

To recover the coefficients, one simply has to do a little additional bookkeeping. This is reminiscent of Euclid's algorithm for gcd since it involves subtracting copies of the shorter vector from the longer vector until some terminating condition is met. Proof of correctness of this approach can be found in [3]. For those who are curious, the basic idea is to prove that if the angle between a,b is between 60 and 120 degrees, then the shorter of the two is a shortest integer combination and to prove that if |a| <= |b| with |x| <= 1/2 (where x is as in the algorithm above), then the angle between a and b is between 60 and 120. Perhaps the most annoying part of the analysis of this algorithm is proving it runs in polynomial time in the number of bits used to represent the vectors. Unfortunately, translating these ideas to higher dimensions doesn't yield an exact algorithm in general but this is the starting point for the ideas in [2].

For interest's sake, while the best approximation guarantee known for poly-time algorithms is exponential, this is still sufficient for the following application. Given a polynomial p[x] over the integers Z or the rationals Q, there is a polynomial-time (polynomial in the degree of p[x] and the number of bits used to represent the coefficients) that will completely factor p[x] into factors that are irreducible over Z[x] or, respectively, Q[x]. An essential subroutine involves finding short vectors in a lattice and the algorithm in [2] finds sufficiently short vectors for this application.

[1] S. Khot, Hardness of approximating the shortest vector problem in
lattices, Journal of the ACM, 52 (5), 2005, pp 789-808.

[2] A.K. Lenstra, H.W. Lenstra, L. Lov sz. Factoring polynomials with
rational coefficients. Mathematische Ann., 261, 1982, pp 513-534.

[3] V. V. Vazirani, "Approximation Algorithms", Springer-Verlag, 2003.

4) COUNTPAL(written by AnhDQ)

The problem can be solved easily by dynamic programming:
F(i) = CountPal(s') for s' is the subtring of s[1]..s[i]
so F(i) = sum of F(j) with j < i whether the substring of s[j+1]..s[i] is a palindrome.
To verify a substring is a palindrome or not, we can pre-compute to save time of DP.
Complexity: O(|s|^2)

5) CLUE(written by David Stolp)

Given the sums of the cards of each logician, we create a list of every possible game configuration with the same sums. Then one turn at a time, we determine if the logician whose turn it is can win, and if not we remove from the list all configurations in which that logician would win on that turn. A configuration is a winning configuration if in every other configuration where the current logician's cards are the same, the secret cards are also the same. A game will never end if every logician has a turn without the size of the list decreasing. In this case there is no winner. That being said, the following game lasts 49 turns before being won!

```
9 10 11
6 7 14
1 2 4
3 8 18
13 15 16
```

6) LOGGERS(written by David Stolp)

(The notation [x] is used for the floor function: [x] is the greatest integer not exceeding x.)
First off, non-integers are problematic, so consider that for any real numbers x, y, z with x+y=z, either
[x]+[y]=[z] or
[x]+[y]=[z]+1
Conversely, given a real number z and integers x′ and y′ satisfying either
x′+y′=[z] or
x′+y′=[z]+1,
then set x=(z+x′-y′)/2 and y=(z-x′+y′)/2 and we have [x]=x′, [y]=y′, and x+y=z.
Therefore we can restrict cuts to integer lengths, where the resulting logs from a cut sum to either the length of the original log, or the length of the original log minus one. Now the problem is solvable using Grundy numbers. For any length L, we calculate G(L)=mex({G(A) xor G(B) : (A+B=L or A+B=L-1) and A,B>0}). Then for a game with initial lengths of L_1,...,L_N if G(L_1) xor ... xor G(L_N) is zero, Bob has a winning strategy. Otherwise Alice has a winning strategy.

Login or Register to post a comment.