\\15295 Fall 2017 #3 -- Problem Discussion

September 13, 2017

This is where we collectively describe algorithms for these problems. To see the problem statements follow this link. To see the scoreboard, go to this page and select this contest.

A. Numbers Painting

Consider the following greedy algorithm. Color the numbers in increasing order. For each number x, compute all of its divisors, and give x the minimum positive color that is not used among its divisors.

Theorem: *For any N, the greedy algorithm colors [1 … N] with*$\lfloor \log N \rfloor + 1$ *colors, which is optimum.*

Proof: Let c =$\lfloor \log N \rfloor$, and let n = $2^c \le N$. Note that 1, 2, 4, … 2^c forms a clique of size c+1 in the divisor graph, and therefore requires at least c+1 colors. Let d be the color used by the greedy algorithm for N. Since the greedy algorithm chose d, there must be a divisor of N, call it N', such that N' is colored with color d-1. Because N' divides N, it must be the case that N' ≤ N/2. Furthermore there must be a divisor of N' (call it N'') that is colored with color d-2. Etc. At most c repetitions of this process gives us 1, which has color 1. Therefore d ≤ c+1. Q.E.D.

The trial division algorithm tries all potential divisors in {2, 3, … $\lfloor\sqrt{x}\rfloor$}. If d is a divisor of x, then the algorithm outputs d and x/d as divisors of x. This algorithm finds all divisors of x and runs in time O(√x). This is fast enough since N ≤ 10000.

[[If you're following this class you've seen the trick for generating all the divisors of 1, 2, … n in time O(n log n). For each i in [1, 2, 3, … n/2] we compute 2i, 3i, 4i, … and add i as a divisor for each of these numbers. It's easy to see that the total work is O(n log n).]]

--DS

B. Join the Strings

The natural solution is to simply sort the strings in lexicographical order, and then concatenate them. However this is insufficient. The problem is with prefixes; we cannot just sort them in increasing length. A counterexample is {"bc", "bca"}: swapping the order is then better ("bcabc" < "bcbca").

Trying a few examples, we see that an intuitive solution is to sort with the comparison function (a < b ⇔ a^b < b^a), where ^ is concatenation. It is not clear that this is even a valid total ordering. In fact it is, and this has a cute proof (found online by Tom).

Treat each string as a base-26 integer. Then

$a\text{^}b < b\text{^}a \Leftrightarrow a * 26\text{^}|b| + b < b * 26\text{^}|a| + a \Leftrightarrow a(26\text{^}|b| - 1) < b(26\text{^}|a| - 1)$
$\Leftrightarrow a / (26\text{^}|a| - 1) < b / (26\text{^}|b| - 1)$
So if we encode a string x as a real number (x / (26^|x| - 1)), we are simply sorting some real numbers. So transitivity holds and < is a valid total ordering. Hence it is well-defined to sort by <.

Sorting by this comparison function gives us s_1 <= s_2 <= … <= s_N. We claim concatenation in this order is optimal. Indeed suppose there were an optimal solution with s_1 in some other position, with string s_i right before s_1. Then since s_1^s_i <= s_i^s_1, we can swap s_1 with s_i and the new solution will be equally optimal. Continuing in this way, we can bubble s_1 to the front and retain optimality. But now by induction s_2 <= … <= s_N is the optimal ordering for the remaining strings. Hence this order of concatenation is optimal. --Raymond

C. Remote Control

D. Just Matrix

So, we have an unknown matrix of numbers. Let's start by giving a variable for each element we need to fill in. So for example in the 3x3 case we'd have 9 variables, arranged as follows.

x00	x01	x02
x10	x11	x12
x20	x21	x22

The two ordering constraints, *left* and *top* give an ordering constraint among the variables on each row and each column. To be specific, suppose that the top row of the *left* matrix is 0 0 2. The second number indicates that x01 is less than 0 of x00. In other words it's greater than x00. So we have x00 < x01. The third digit, 2, means that x02 is less than both x00 and x01. But since x00 is the smallest of these we can just write this as x02 < x00. Summarizing we have:

$$x02 < x00 < x01$$

So for each row and each column there is a sequence of n-1 inequalities like these among the variables of that row or column. (We'll come back to how to generate the necessary ordering later.) Our goal is to assign each variable with a unique number in $\{1,2,3, \dots ,n^2\}$ that satisfies all these constraints.

Here's how to do this. Construct a directed graph, where each vertex is one of the variables. Put a directed edge from V ➡ W if the inequality V < W occurs. Now use depth first search to compute a topological ordering on these vertices. (If the graph has a cycle, then there is no solution.) Now assign the variables 1, 2, 3, … in the topological order. This assignment will satisfy all the constraints.

It only remains to compute the necessary ordering for each row and each column. Since a row is up to 600 in length it will be too slow to use an $O(n^2)$ algorithm. There are several ways to do this in O(n log n). You can use splay trees (or any other BST), or you can use SegTrees. I'll describe the SegTree method that I came up with. Raymond has a different way.

First I'll show how to construct a permutation of the numbers 0, 1, … n-1 that satisfies the constraints of a row. As a working example, say the constraint row is 0 0 2 (as above). The 2 means that the number in that position (the last position) has two numbers greater than it to its left. This is equivalent to saying that it has 0 numbers less than it to its left. If the sequence is c0, c1, c2, then we replace it by (0-c0, 1-c1, 2-c2). In our working example this is 0 1 0.

The algorithm works from right to left. Initially we have a set {0,1,2} and we are working on 0 (the third element of the sequence 0 1 0). We know that among the current choices {0,1,2} the only number that is greater than 0 of them is 0. So we output the 0. Now our set is {1,2} and we're seeking a number that is greater than 1 of them, namely 2. So we output the 2. Now we're left with the set {1}, and the last output is 1. This generates the permutation 1, 2, 0. And this satisfies its requirements.

To implement this process efficiently we maintain a SegTree to store a sequence of bits b0, b1, … b(n-1). All the bits are initially 1, which represents the set {0,1,...,n-1}. We're going to need to support the following operations:

Assign(i,x) : Execute the assignment bi $\leftarrow$ x (where x is 0 or 1)
Find_rank(r): return the index i such that bi=1 and there are r ones among {b0, b1, … b(i-1)}

SegTrees support this very easily. Each internal node stores the number of 1 bits in the subtree rooted there. Assign() is just standard, and Find_rank is a simple walk down the segtree to find the desired index.

With this in hand we can now find the desired permutation by repeatedly finding the index using find_rank() outputting that index, and then setting that bit to 0 via Assign().

Now to convert the permutation perm = [1,2,0] into the desired index list ilist do the following:

```
Ilist[perm[0]] ←0
Ilist[perm[1]] ←1
Ilist[perm[2]] ←2
```

Which results in ilist = [2, 0, 1]. Thus we have generated our sequence x02 < x00 < x01. (note the subscripts)

--DS

E. XOR-omania

Take the sequence of As and xor each A_i by y. Notice that this does not change the Bs at all because B_{ij} = A_i xor y xor A_j xor y = A_i xor A_j. Using this fact, we assume WLOG that A_0 = 0. Then, all we need to do is figure out which Bs are A_0 xor A_i.

Next, note that if x = A_i xor A_j and y = A_i xor A_k then x xor y = A_j xor A_k so x xor y is a B value. In this way we can discover all Bs that are either A_0 xor A_i or A_1 xor A_i for any i.

The final step is to determine whether each B is is A_0 or A_1. WLOG we can choose the first other selected to be B_1 = A_0 xor A_2. Then, if we xor against any other B_k that is A_0 xor A_i, we will find that B_0 xor B_k \in Bs and B_1 xor B_k \in Bs. However, this also includes A_1 xor A_2. We can exclude A_1 xor A_2 by excluding B_1 xor B_0.

This gives us the list of A_0 xor A_i for all i. Since A_0 = 0, we simply add 0 to the list to recover all of the A_i.

-- Corwin

F. Wolves and Sheep