15295 Spring 2017 #11 -- Problem Discussion
April 5, 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. Translation

From the examples, you can get a mapping of 51 characters. Then, you have to find which character is missing. When you have the mapping, you just apply it to the input. - Xiao

B. Islands

C. Postman

You separate the array of houses into 2. 1 for positive, the other 1 for negative. Sort the houses according to their absolute distance from post office. Every time, you try to deliver to the furthest house. If you have capacity left, allocate it the second, third…. furthest houses until you are full. - Xiao

D. Revolutionary Roads

Note that n*m is at most 20 million, so that is an acceptable big-oh bound to strive for. Here's my way of describing an algorithm that achieves this. I believe several people in the class came up with this or similar ideas.

(1) ∀ vertices v compute R(v), the set of vertices reachable from v. (This can be done by doing a dfs starting from v.)
(2) ∀ vertices v compute R*(v) the set of vertices that can reach v. (Same as (1) except use the reversed graph.) Note that the size of the strongly connected component (SSC) containing v is | R(v) ∩ R*(v) |.
(3) ∀ edge (u,v) do this:

> We need to compute the size of the SCC containing v that results when we add the reverse edge (v,u). Let R'(v) be the set of vertices reachable from v when this edge is added. It's easy to see that
>
> $$R'(v) = R(v) \cup R(u)$$
> $$R^{*'}(v) = R^*(v)$$
>
> So our answer for the size of the newly created SSC that results when edge (v,u) is added is | R'(v) ∩ R*'(v) |.

This algorithm is easy to code. No fancy data structures are needed. For example you can use a boolean array to represent the sets. Operations such as union and intersection take O(n) time. The result is an O(nm) algorithm that is fast enough.

The algorithm can be made 64 times as fast by representing these sets by bitmaps in the form of an array of 64 bit integers. Some languages (i.e. C++ and Java) include standard implementations of sets based on this. So you can implement this algorithm to run in time O(n*m / 64). To achieve this you probably need to construct the sets R() and R*() by a topological order traversal of the DAG of SSCs instead of the dfs method described above. ---Danny

E. Explode 'Em All

Suppose you've selected a subset of rows that you're going to put bombs in; then, for each row that you're not going to bomb, you get a set of columns in which there must be bombs (i.e. every column that has a rock in any of those rows). You can then easily compute the minimum number of bombs you need under this selection of rows that have bombs - it's simply the maximum of the number of rows that need bombs and the number of columns that need bombs.

If we can iterate over all selections of rows and do the above computation in constant time, then we get a O(2^n) algorithm, which will just squeak under the 1s time limit (my Java solution took 935ms). The approach to doing this is to do a bitset DP. First, for each row, construct a bitset corresponding to the columns in that row that have rocks. Then we can begin our DP: for each selection of rows to bomb (represented by a bitset - this will fit in an integer, since n <= 25), we will store a bitset representing the columns to bomb (again, an integer suffices). Process the subsets in descending order of the number of rows you are selecting to bomb. For the initial case of bombing all rows, we don't need to bomb any columns, so we can simply store 0. For all other cases, we have already done most of the work: let the set of rows we are processing be R, and let r be an element of R; since we are going in reverse order of the number of rows we select, we have already processed R; and thus, to get the new set of columns we must bomb when selecting R-{r}, we can simply bitwise-or the value we previously computed for R with the bitset corresponding to row r. As we process these, we can keep a running minimum value of max(|R|, setBitCount(DP[R])), and then output this minimum at the end.

To make this sufficiently efficient, we need to employ some bit hacks. The ones I used were:

- Constant-time subset iteration: a snippet of code here shows how you can iterate
- Select an element from a subset: to pick an element r from the set R, we can do (R & -R). This will give you the lowest bit of the mask.
- Counting set bits: I'm not sure about clever ways to do this, but Java provides Integer.bitCount(), which was fast enough for this problem.

-- Matthew

F. Exploring Pyramids

This problem can be solved with a DP over all odd-length substrings. For each substring A[i..j] (inclusive), we store a pair of values DP[i][j] = (total number of trees described by A[i..j], total number of trees described by A[i..j] in which the root has at most one neighbor). In the base case of i=j, we store DP[i][j] = (1, 1). If i != j and A[i] = A[j], then DP[i][j] = (sum of k = i to j-1 of DP[i+1][k].l * DP[k][j-1].r, DP[i][j].l). In all other cases, DP[i][j] = (0, 0). We can fill this table by processing the substrings in increasing order of length, and at the end we output DP[i][j].l.

-- Matthew