December 2017 USACO Gold/Platinum Review

Justin Zhang

January 12, 2018

1 Gold - A pie for a pie

1.1 Problem

You're given two lists of "pies," each of length n ($n \leq 10^5$), where the first one is Bessie's pies and the second one Elsie's. Each pie is assigned a value by both Bessie (b_i) and Elsie (e_i) ($b_i, e_i \leq 10^9$).

You're also given an integer D ($0 \leq D \leq 10^9$).

For each of Bessie's pies, the cows play a game: Bessie gives the pie p to Elsie. Then, Elsie must give a pie q in return such that the value of q, her eyes, is between her value of p and her value of p plus D, inclusive. The cycle repeats until either no such pie satisfies the condition, or until a cow receives a pie which she sees as having 0 worth.

Find, for each pie, the minimum number of cycles in this game that results in obtaining a 0 worth pie, or print -1 if this is not possible.

1.2 Example

They both own two pies. Bessie owns pies 1-2, and Elsie owns 3-4.

Bessie sees pie 1 as worth 1, Elsie 1. Bessie sees pie 2 as worth 5, Elsie 0. Bessie sees pie 3 as worth 4, Elsie 2. Bessie sees pie 4 as worth 1, Elsie 4.

For pie 1, an optimal answer is pie 1 -> pie 3 -> pie 2. Thus the answer is 3.

For pie 2, Elsie immediately receives pie 2, which is valued at 0. Thus the answer is 1.

1.3 Solution

Note that the pies can be represented by a directed (bipartite) graph, each pie p connected to q if it's possible to give q in response to p.

We want to find the shortest paths from a given pie to any of designated value 0 pies.

Because each edge is value 1, we can simply use a BFS (note that all-pairs shortest paths is much too slow). Initialize the BFS queue with the pies of value zero, and BFS outward.

To find the neighbors of a given pie (with values between v and $v + D$), we need to perform binary search on the (presorted) list of pies. This allows us to push them to the BFS queue. This can be done with a `multiset`.

The DFS is O(n). With the binary search, the runtime becomes $O(n \log n)$.

2 Gold - Barn Painting

2.1 Problem

Given a tree $n \leq 10^5$, some of which have nodes that are colored one of three colors, how many ways are there to color the remaining (uncolored) nodes such that no neighbors have the same color (mod $10^9 + 7$)?

2.2 Example

Given a tree defined with edges:

```
1-2
1-3
1-4
```

and node 4 set to color 3, the number of ways to color nodes 1, 2, and 3 total to 8.

2.3 Solution

Arbitrarily root the tree.

Notice that, for any given node n, the number of colorings for a subtree rooted at a child of n only depends on the color of n. We can thus obtain a solution for the subtree rooted at n by multiplying the solution for the subtrees rooted at the children of n for each of the three colors. Note that if the parent or a child of n is already defined, we must exclude these possibilities for n.

This is a DP solution, in which the state is a node, color pair. We can thus memoize our DFS to obtain a O(n) solution.

3 Platinum - Standing Out from the Herd

3.1 Problem

You're given n strings ($1 \leq n \leq 10^5$), such that the total length of the n strings doesn't exceed 10^5. Let's define the *uniqueness factor* to be the number of substrings in a given string that's not present in any other string.

For each string, find its uniqueness factor.

3.2 Example

Given the input

```
amy
tommy
bessie
```

The uniqueness factors are 3, 11, and 19, respectively.

3.3 Solution

Let's first construct a suffix array and LCP (longest common prefix) array of all the strings concatenated by a dividing character (e.g. "#"). This can be done in $O(n^2 \log n)$ by a rolling hash, binary search, and sorting (see previous SCT lecture on suffix arrays).

This gives us a sorted list of prefixes, along with the length of the longest matching prefix between neighbors.

For our example, it would be as follows:

```
#bessie             1   (-1)
#tommy#bessie       0   (-1)
amy#tommy#bessie    0   (0)
bessie              0   (2)
e                   1   (2)
essie               0   (2)
ie                  0   (2)
mmy#bessie          1   (1)
my#bessie           3   (1)
my#tommy#bessie     0   (0)
ommy#bessie         0   (1)
sie                 1   (2)
ssie                0   (2)
tommy#bessie        0   (1)
y#bessie            2   (1)
y#tommy#bessie      -   (0)
```

The suffixes are labelled with the LCP, and the string number they start with in parenthesis ("starting string").

Let's iterate through the array, top-down, skipping the strings that start with our divider (#). We start with `amy#tommy#bessie`.

We know that `amy#tommy#bessie` contributes three to the uniqueness factor of string 0 (there are three prefixes of `amy#tommy#bessie` that aren't shared with other strings – namely, [`amy`,`my`, `y`]). How? There are a few things we need to account for.

Notice that the first string before `amy#tommy#bessie` that doesn't share a starting string equal to that of `amy#tommy#bessie` (0) (specifically, `#tommy#bessie`, which has a starting string of -1) has an LCP of 0 with `amy#tommy#bessie`. Note that any string before `#tommy#bessie` must have a smaller LCP with `amy#tommy#bessie`. As such, the *most* prefixes of `amy#tommy#bessie` shared by any other string's substrings such that the substring is lexicographically less than `amy#tommy#bessie` is 0.

We must check the other way, as well. The first string after `amy#tommy#bessie` that doesn't share a starting string equal to that of `amy#tommy#bessie` is `bessie`. The LCP between the two is also 0. We thus know that there are no substrings of other strings such that the substring is greater than `amy#tommy#bessie` and has an LCP greater than 0.

If any of the two above LCP's were greater than 0, we would have to subtract the length of the suffix (to the divider; 3 in this case) by the maximum of the two, since the maximum of the two represents the point up to which the prefix is not in any other string. I'll refer to this as M.

For example, consider the string `mmy#bessie`. The LCP with the first previous non-matching starting string, `ie`, is 0. However, the LCP of the first non-matching starting string after, `my#tommy#bessie`,

is 1. This means that $M = 1$, and that the `m` is in another string, whereas `mm` and `mmy` are not. In this case, we'd add $3 - M = 2$ to the uniqueness factor of string 2.

Running this process on `amy#tommy#bessie` tells us that no other string's substrings are *any* of [`amy`,`my`, `y`]; $M = 0$. We thus add three to the uniqueness factor of `amy` and go to the next string.

For `bessie`, this procedure tells us to add six to the uniqueness factor of string 2. Moving on to `e`, we add one to the uniqueness factor of string 2.

However, we encounter a problem when considering `essie`. If we did the same process, we'd be double-counting `e` as a substring not shared in any other string. As such, we also have to take into consideration the LCP of the string we're on and the previous string: $M := max(M, LCP[i-1])$. This disallows us from even considering the "e" in `essie`, and the maximum we can add is 4.

If we repeat this for each suffix, collating the results, we obtain the final uniqueness values. The implementation, however, is tricky, so please refer to USACO's official solution for details. The runtime of this loop is $O(n)$. The suffix array, thus, will be the bottleneck in performance – likely either $O(n \log n)$ or $O(n \log^2 n)$. These are both, however, fast enough to obtain full credit.