COP 4516 Spring 2018 Week 14 Final Team Contest Solution Sketches

Company Board 2

Consider characterizing the problem recursively. Unfortunately, when solving the problem for a given root of the tree, it's not enough to know the answers for the children of that root. The reason for this is that we have no way of knowing, given the answers for each of the child nodes, whether or not we are allowed to place the person at the root node into the set, in addition. We don't know this because we don't know if any of the people represented by any of the child nodes are in those respective sets. Thus, we must change the recursive characterization of the problem to be: solve the problem for the root, given that the root IS chosen. Also, solve the problem for the root, given that the root IS NOT chosen. So, the input to our recursive function contains two variables:

1. The root node of the tree in consideration.
2. A boolean or 0/1 value indicating whether or not that root node is included in the set.

With this characterization of the problem, we have enough information from our recursive calls to solve the problem. In particular:

f(node, true) = val[node] + $\sum_{x \in child(node)} f(x, false)$

This indicates that if we take the node at the root, then we add a value of val[node] (number of connections of that employee) to the sum of all the recursive calls on each child where none of those children themselves are allowed to be in the tree, which is indicated by the false in the corresponding recursive call.

We must solve differently for the other input:

f(node, false) = $\sum_{x \in child(node)} \max(f(x, true), f(x, false))$

If our node isn't in the set, then we don't get to add val[node]. But, we then have NO restriction on our recursive calls. This means, what we have to do is try both options for each child: take the child node and DON'T take the child node. Of these two options, naturally, since both are possible, we want to take the maximum of the two. Then, we sum this over each child.

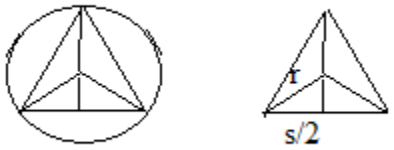
If we write this recursively, we'll get time limit exceeded. The reason for this is that although there are at most 2n recursive calls, where the input has n employees, due to the flip-flopping nature of the true/false parameter, the same exact recursive call gets executed many times. As we learned in class, the antidote to this is creating a memo table and storing results to recursive calls in it. If a call was previously done, immediately return the answer in a base case.

For the small set, one can try each combination of nodes and take the best one that satisfies the restrictions. The run time of this would be roughly along the lines of $O(n2^n)$ with $n \leq 10$ for the small case so this would easily run in time.

Illuminati

There are just two processes to make calculations for: circumscribing a circle around an equilateral triangle and circumscribing an equilateral triangle around a circle. Let's work both out in detail:

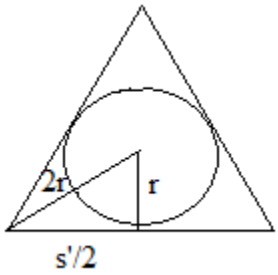
Start with an equilateral triangle with side length s. It has area $\frac{s^2\sqrt{3}}{4}$. We can get this by drawing a perpendicular down to a side and calculating the length of that perpendicular using the induced 30-60-90 triangle. Now, consider the circumscribed circle. Its center will be at the point equidistant from each of the vertices of the triangle. Thus, we can calculate the radius using the appropriate 30-60-90 triangle as well:



To solve for r, we can solve for the short side of the 30-60-90, which is $\frac{s}{2\sqrt{3}}$.

This means that $r = \frac{s}{\sqrt{3}}$. It follows that the area of the corresponding circle is $\frac{\pi s^2}{3}$. The ratio of the area of the circle to the equilateral triangle is $\frac{\frac{\pi s^2}{3}}{\frac{s^2\sqrt{3}}{4}} = \frac{4\pi}{3\sqrt{3}}$.

Now, let's handle the other ratio. Given a circle with radius r (which has area πr^2) let's circumscribe an equilateral triangle about it and find its area:



Let the side of the equilateral triangle be s'. Using the 30-60-90 triangle shown, we get the following equation: $\frac{s'}{2} = r\sqrt{3}$. It follows that $s' = 2\sqrt{3}r$. The area of the triangle is

$$A = \frac{s'^2\sqrt{3}}{4} = \frac{(2\sqrt{3}r)^2\sqrt{3}}{4} = \frac{12\sqrt{3}r^2}{4} = 3\sqrt{3}r^2$$

Thus, the ratio of the area of the triangle to the circle is $\frac{3\sqrt{3}r^2}{\pi r^2} = \frac{3\sqrt{3}}{\pi}$.

The key to avoiding rounding errors is to note that the product of these two ratios is:

$$\frac{4\pi}{3\sqrt{3}} \times \frac{3\sqrt{3}}{\pi} = 4$$

Finally, figure out the areas of both base figures (C = π, T = $\frac{\sqrt{3}}{4}$) and then multiply this by the appropriate factor. If we iterate the process an even number of times, say 2k, then this factor is just 4^k. If we iterate it an odd number of times, say 2k+1, get the result for 2k times and then multiply by one of the two factors derived above depending on which case is being handled.

For the small data, no matter how you do the math, rounding issues shouldn't be a problem since we're only doing a couple iterations. But, for the most part, this is one of those problems where if you figure out how to solve the small data, the large data shouldn't be much more difficult.

Who Made the Problem

This is the banger in the set. The hardest part is remembering how to read in a whole line of text and not mixing line reading with tokenizing. Read the whole line in, convert it to lower case, and then search for "corgi" using a built in search function. (You can write your own, but this shouldn't be necessary.) There aren't significant differences between the small and large input. If you tokenize, the small input is easier. Also, since it's all lowercase letters, there's no need to turn the input into all lower case.

Somewhat Out of Sync

The period for two pairs of wipers is just the least common multiple (LCM) of the two periods of those pairs. Since n ≤ 100, we have plenty of time to just calculate the LCM of every pair of input numbers and store them in a set (preferably a TreeSet due to the sorting restriction). Recall that LCM(a, b) = ab/GCD(a, b). The only remaining observation for this problem is to note that with a and b up to 1,000,000 in the large input, ab can overflow int, as can the LCM. Thus, longs must be used in the calculation (or long long in C++). Also, if you don't use a set, in your initial creation of values, some values could be duplicated and these need to be removed before outputting them. Naturally, using a built in set which already removes duplicates is the better solution. For the small input, longs are not necessary but the rest of the pieces of the solution are.

Corgi Toys

A binary index tree allows us efficiently to query the sum of a table from index 1 to index k. In this problem, we are given the inverse query: Given some value X, find the maximum k for which sum(1, k) ≤ X. It's easy for us to find the sum given k, but harder for us to find k given the sum. In addition, as k increases the sum increases since the table only has non-negative values in it. This precisely fits all of the characteristics of a binary search! Thus, we will store the binary index tree the natural way (each toy has its index and we add for buying that toy and add the negative quantity for selling), and for our queries, we will binary search the value of k. Thus, answering a query can be done in $O(\lg^2 n)$ time. Thus, the total run time for a whole input case will be $O(n\lg^2 n)$ with $n \leq 10^5$, which runs in time.

A more clever and efficient solution is to use the "binary-lifting" idea. The idea is as follows: Say find the sum from 1 to 2^{19} and this sum is under our value X (note that this ISN'T a query because this sum is stored DIRECTLY in the bit). There's no need to "add that up" again. Instead, we should just add to it the sum from $2^{19}+1$ to $2^{19}+2^{18}$ (also a direct entry in the BIT) and see where that gets our sum. If it's too much, our k is smaller than $2^{19}+2^{18}$. You can iterate where each iteration decreases the corresponding range by a factor of 2. See Charles's posted solution - the code is incredibly short and its run time is O(nlgn) because each look up in the binary lifting code runs in O(1) time!!!

The final idea which passes the time limit is an idea called square root decomposition. The idea here is to make the table size k^2 for the smallest $k > n$. Then, we store two arrays, one array of size k (sqrttable) and another of size k^2 (table). table is a regular frequency array where table[i] just stores how many of toy i are in stock. sqrttable is somewhat of a cumulative frequency array. sqrttable[i] stores the sum of toys from index ik to i(k+1)-1. Whenever buying or selling toy i, we must add/subtract the appropriate amount from both table[i] and sqrttable[i/k]. This is an O(1) operation for adding and selling. To do a query, what we will do is see if sqrttable[0] > X. If it is, then our answer is in between 0 and k-1. If not, then we know our answer is k or greater. Basically, we keep on adding the values sqrttable[i] until we exceed X. We then subtract 1 from i indicating that we should not have added that last one in. Then, we can successively add in the last few values from table until we exceed X and can correctly answer with the appropriate value of k. Note that we ever iterate at most $\sqrt{n}$ times in sqrttable, and then once we get to table, we know we can't iterate more than $\sqrt{n}$ times because we know that however many toys we can add it, we can't add in$\sqrt{n}$ of them. Thus, a query runs in $O(\sqrt{n})$. For $n = 10^5$ and the given time limit, this runs fast enough.

For the small data, it's enough to store a regular frequency table and just run a for loop for each query, since there will only be 10 of them, the worst case for these is looping $10 \times 10^5 = 10^6$ times, which runs quickly.

Word Flip

The most straight-forward way to view the hard version of this problem is to see that we can run a breadth first search. The number of possible states is $26^4 < 10^6$ and for each position we have about 14 possible moves (move up or down each of the four letters or exchange any of the six possible pairs). Thus, the worst case run-time of a BFS is roughly 26^4 x 14, which is well within the given time limit.

For the easy version of the problem, since each letter must stay put, we want to minimize the number of moves to transform each letter to the letter it must transform to. The absolute value of the different in Ascii values will give the answer for one direction of rotation while 26 minus that value will give the answer for rotating in the opposite direction. Thus, we just take the minimum of these two and sum over all four letters.

There is an alternate solution based on what was taught in the class for the hard version. The key observation is that for any set of moves, we may as well first do all the swaps we want up front and then just advance letters, like we do in the easy solution. So, what we can do is try all permutations of the first set of letters, and for each permutation, calculate the cost for the easy version of the problem. But, we then have to calculate the cost of arriving at the permutation to add to this. For any permutation, the cost of arriving to it is based on cyclic breakdown of the permutation. Consider this following permutation of the values 0, 1, 2, 3, 4, 5, 6, 7:

3, 2, 5, 6, 1, 7, 0, 4

Its cyclic breakdown is:

$0 \rightarrow 3 \rightarrow 6 \rightarrow 0$
$1 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rightarrow 1$

We obtain this by starting with some index i and moving to perm[i] and repeating the process until we get back to the original i.

Here the cycle lengths are 3 and 5, respectively. The fewest number of swap necessary to obtain this permutation from 0, 1, 2, 3, 4, 5, 6, 7 is going to be 2 + 4, since for each cycle of size k, we must perform k-1 swaps in succession.

With only 4 letters, there really aren't too many cyclic breakdowns. Nonetheless, it's probably easier just to write the code generally (my alternate solution does this). So, for the alternate solution, we try each permutation. For each permutation we calculate the number of swaps it takes to get there as described above, and then add to it the solution to the easy version of the problem. We take the minimum over all permutations. For this specific input size, this alternate algorithm is much faster than the BFS.