Kick Start 2017 - Round F

Analysis: Kicksort

Kicksort: Analysis

In Quicksort, the best pivots at any stage are the ones that are closest to the *median* of the current list (although we would have no way of knowing which ones those are without doing some additional work before choosing!) This problem hinges on a misinterpretation of that idea, in which we take pivots from near the middle *index* of the list instead. Unless our sequence happens to be already sorted or close to sorted, this is no better than choosing an arbitrary index!

A Small-only approach

The sequences in the Small dataset can have between 2 and 4 elements. There are only 2! + 3! + 4! = 32 different permutations that meet these criteria. This enables some strategies that do not usually work in Kickstart. For example, we can pre-solve each case by hand, or use nested conditional statements that take advantage of the small size of the sequence. But we recommend using one of the two approaches below instead.

A quadratic simulation approach

The most straightforward general approach is to actually run the Kicksort algorithm on the given sequence, and see whether it ever picks a pivot that is *not* worst-case. One possible complication is figuring out how to have this recursive algorithm pass information up the call chain if it does find a non-worst-case pivot; one option is to have it throw an exception that can then be caught. We must also be careful not to overflow the stack with recursive calls, since there can be **N** of them; one way to avoid this is to use our language's provided way to change the system's stack size. (Warning: This sort of system tweaking can be dangerous in real-world applications, which tend to avoid deep recursive strategies like this one.)

Quicksort is famously $O(\mathbf{N}^2)$ when it consistently picks worst-case pivots, and that is what many of our simulations will do, so this strategy takes $O(\mathbf{N}^2)$ time. This is still fast enough to solve the Large dataset, but can we do better?

A linear approach

First of all, let us consider different sequences of the same length that have "YES" answers — that is, the ones for which Kicksort will always pick worst-case pivots. Even though the pivots themselves may have different values, Kicksort will always use the same *indexes*, in the same order, as pivots. For a sequence of length 6, for example, it will pick index 2 as a pivot, and then divide the other values into one empty list (which is unimportant since no pivot is chosen from it) and one list of length 5 containing the remaining values. (It turns out not to matter for our purposes whether that list is the "low" or "high" one.) Then it will pick index 2 from that list of length 5, which corresponds to index 3 in the original list. If we continue to trace such a case, we find that the indexes chosen from the original list will be 2, 3, 1, 4, 0, 5. That is, we are starting at index floor(**N** - 1) / 2, then jumping 1 to the right, then 2 to the left, then 3 to the right, and so on. This does not depend at all on the values in the list!

A similar pattern holds for lists of odd length, although in that case, the first jump is 1 to the left. Knowing this, we can visit the indexes of our pivots in order, without doing any simulation. It is

not too hard to implement the pattern of changing direction and adding a distance of 1 with each new jump.

Each time we visit a new index, we check whether it holds either the lowest or highest value that has not already been used. If this continues all the way through the last index, we have a "YES" case. However, if we ever encounter a value that does not satisfy those conditions, then we have a "NO" case and we can stop.

At this point, we can take advantage of the fact that our sequences are permutations of numbers from 1 to **N**. (Of course, in real life, we could "sort" such sequences in constant time, without ever reading them, if we knew the length in advance!) We know at the outset that our lowest unused value is 1 and our highest unused value is **N**. When our current value matches this lower bound, we increment it by 1, because we already know that the next lowest value is exactly 1 more than the previous lowest; similarly, when our pivot matches the upper bound, we increment our upper bound by 1. We only need to keep track of these two values as we go.

Since we potentially have to bounce around the entire sequence, this strategy takes O(**N**) time. It is unusual for a problem about sorting to have a solution that takes less than O(**N** log **N**) time, but in this case, it is a consequence of having restricted our sequences to permutations.