Code Jam 2021 - Qualification Round

Analysis: Median Sort

This problem is about information theory. There are $\mathbf{N}!/2$ essentially different outputs, which means we need at least $\log_2(\mathbf{N}!/2)$ bits of information. For $\mathbf{N} = 50$ that is slightly over 213. That means that for Test Set 2, extracting less than a bit per query (on average) can work, but for Test Set 3 it will not. In Test Set 1, on the other hand, this informational analysis is not necessary.

Test Set 1

In this test set we have 300 queries per case, which is more than the number of different triples $\binom{\mathbf{N}}{3} = 120$. This means that we can query every possible subset of 3 elements and memoize the results. Freed from the limitations of the number of queries, any solution to sort based on medians works. One simple way is to notice that the only elements that are never the median are the first and last elements, so we can identify those, and arbitrarily choose which one is the first and which one is the last. Then, we can eliminate those two and find the candidates for second and next-to-last as the ones that are never median in queries with only remaining elements. To know which one to assign second, we can check an additional query between the element we chose to go first and the two newfound elements: the one who is the median of that subset should be second, and the other one should be next-to-last. We can iterate this to find and place pairs of elements moving inwards until we place them all.

See below for other sorting algorithms that work on medians. While the other test sets have the additional burden of having to make them work in an online way, we can use the pre-query tactic for this one to make our lives easier, even if we implement the same basic algorithm. It also allows for simpler implementations of some of the algorithms that use extra queries.

Test Set 2

Since just 1 bit of information per query is enough for this test set, we can use one of our known comparison-based optimal sorting algorithms like Merge Sort or Heap Sort. Even Insertion Sort can work if we use binary search to look for the insertion point. While Insertion Sort + Binary Search requires a suboptimal $O(\mathbf{N}^2)$ number of operations, it uses an optimal $O(\mathbf{N} \log \mathbf{N})$ number of comparisons.

For any of these algorithms we need a way to simulate a binary "less than" operation between two arbitrary items i and j. One way to do that is to ask for the median of i, j and k while knowing k is not the median. So, we could try to find an overall minimum or maximum and then use that as k for every other query. Notice that there are 2 elements that can be minimum and/or maximum and neither would be the median of any query. So, we can do a query for the median of the first three elements. We can discard the median of those and keep the other 2 as candidates. We add any element we have not checked and do another median query, discarding the median and keeping 2 candidates again. After $\mathbf{N} - 2$ queries, we have discarded $\mathbf{N} - 2$ elements, and the 2 that remain are the minimum and maximum.

Test Set 3

To solve Test Set 3 we can observe that our implementation of Insertion Sort can actually extract more than 1 bit per query and have a smaller constant. Instead of looking for the minimum/maximum first, we simply use the minimum of our current range in the median query as k. This means it is not guaranteed to never be the median. However, if the "current minimum"

is the median of our query, we know exactly where to insert and can stop searching. This effectively makes our query a binary comparison with a little bit of extra information, and that little bit (plus being careful about never overspending) can be enough to pass Test Set 3. More importantly, it enables us to not find for the minimum first, which is a significant overexpenditure.

We can also find solutions that have more wiggle room by extracting a lot more than 1 bit per query. Specifically, we want to extract something closer to the optimal $\log_2 3 \approx 1.58$ bits per query. That means considering all 3 possible outcomes of the query, and have them happen with approximately equal probability (see the information theory article for details on the link between the probability distribution of outcomes and the information content of the query response).

We can further refine the Insertion Sort implementation to use a ternary search (in this case, this means a search similar to binary search that splits into three parts instead of two) to extract the full potential out of the median query. At each step, we query the two elements that are $1/3$ and $2/3$ of the way into our current range, together with the element to be inserted. The result narrows down the search to one of three ranges (roughly first, middle or last third). Notice that any query that we do to handle border cases (like inserting at the very beginning or very end) does not get the optimal 1.58 bits of information, so we want to be careful not to use those, or to use them infrequently.

A different option is to do a two-pivot randomized quicksort. By using two pivots, we make full use of each median query involving them and each other element, as there are three buckets where the other elements can fall, each corresponding to a possible response to the query. The only issue with this approach (and any other approach based on a divide and conquer sorting algorithm) is that there are two ways to orient each recursive result if they have more than one element. If we use a query to decide which way is consistent with our decision on how to order the pivots, that query gives us only 1 bit of information. Luckily, this is rather infrequent as it only happens proportional to the number of branches in the recursion tree that contain more than one element, which is a small number.