Code Jam 2019 - World Finals

Analysis: Sorting Permutation Unit

A rotation-based solution

For simplicity, let's assume that none of the arrays we want to sort contain any repeated entries. If there are repeatred entries, we can arbitrarily choose a correct sorted order for them.

To illustrate the sorting algorithm, let's first ignore the constraint on the number of permutations, and use **N**-1 permutations: the i-th permutation (1 ≤ i ≤ **N**-1) swaps the i-th element with the **N**-th element.

For example, when **N** = 5, we use the following 4 permutations:

- 5 2 3 4 1
- 1 5 3 4 2
- 1 2 5 4 3
- 1 2 3 5 4

With this setup, we can use the **N**-th element as a "buffer" to sort the first **N**-1 elements. The algorithm is as follows:

1. If the element at position **N** is not the largest one, swap it to the correct position.
2. Otherwise, there are 2 cases:
    - The array is sorted, and we are done.
    - The array is not sorted, so we swap the **N**-th element with any element that is at an incorrect position (so that we can continue using it as buffer).
3. Repeat from step 1.

For example, let's consider the array [30, 50, 40, 10, 20]. We can:

- Swap 20 to the correct position: [30, 20, 40, 10, 50].
- Now 50 is at the last position, but the array is not sorted, so we swap it with any element that is at an incorrect position, for instance, 30: [50, 20, 40, 10, 30].
- Swap 30 to the correct position: [50, 20, 30, 10, 40].
- Swap 40 to the correct position: [50, 20, 30, 40, 10].
- Swap 10 to the correct position: [10, 20, 30, 40, 50].
- The array is now sorted.

Note that this solution uses **N**-1 permutations and 1.5**N** operations:

- **N** operations to swap **N** elements to their correct positions,
- At most **N**/2 to swap the **N**-th element in case it is the largest. This is because after we swap the largest element with an element at the wrong position, we won't need to do so again on the next step.

Reducing the number of permutations

To fit within the limit on the number of allowed permutations, we can instead use the following 5 operations:

- One permutation that swaps elements **N**-1 and **N** — the next-to-last and last elements.
- 4 permutations that rotate each of the elements from 1 to **N**-1 by 1, 3, 9 and 27, respectively. (Notice that element **N** remains the same.)

With these 4 permutations, when **N** ≤ 50, we can rotate the first **N**-1 elements by any amount (from 1 to **N**-2) in at most 6 operations. For example, to rotate by 26 = 9 + 9 + 3 + 3 + 1 + 1, we rotate by 9 two times, rotate by 3 two times, and then rotate by 1 two times. To rotate by 47, we use 27 + 9 + 9 + 1 + 1. (Equivalently, we can express any number less than 50 using a base three string with trinary digits summing to 6 or less. The smallest number that would take 7 or more is 53.)

Our new algorithm is similar to the one above.

1. If the element E at position **N** is not the largest one, swap it to the correct position as follows:
    - Apply rotations to the first **N**-1 elements until the (**N**-1)-th position is the slot where E should go.
    - Swap the (**N**-1)-th and **N**th positions, moving E into place (and some other element into the **N**-th position.)
2. Otherwise, E is the largest element. We need to temporarily move it out of the **N**-th position. We can swap in some element that is not already in its correct relative position among the first **N**-1 positions. We can use the same strategy as above to rotate to that element. If there is more than one element in the wrong relative position, we choose the element such that the amount we need to rotate to get it in place is minimized.
3. We repeat the above until all of the elements in the first **N**-1 positions are in the correct relative order. Then, we may need to do some final rotations to put them in the correct absolute positions.

For example, let's consider the array [30, 50, 40, 10, 20]. We will use 3 permutations:

- Swap the last 2 elements: 1 2 3 <u>5</u> <u>4</u>
- Rotate the first **N**-1 elements by 1: 4 1 2 3 5
- Rotate the first **N**-1 elements by 3: 2 3 4 1 5

The algorithm works as follows:

- We first want to swap 20 to the relative position after 10:
    - Rotate the first **N**-1 elements by 3: [50, 40, 10, 30, 20]
    - Swap the last 2 elements: [50, 40, 10, 20, 30]
- Now we want to swap 30 to the relative position after 20:
    - Rotate the first **N**-1 elements by 3: [40, 10, 20, 50, 30]
    - Swap the last 2 elements: [40, 10, 20, 30, 50]
- The largest element, 50, is now at the last position. We need to swap it with some element at the wrong position. In this case, there is exactly one such element: 40.
    - Rotate the first **N**-1 elements by 3: [10, 20, 30, 40, 50]
    - Swap the last 2 elements: [10, 20, 30, 50, 40]
- Finally, we want to move 40 to its correct position:
    - Swap the last 2 elements: [10, 20, 30, 40, 50].

The number of operations we use are:

- 1.5**N** operations for swapping (as shown in our first algorithm)
- 6**N** operations for rotating the first **N**-1 elements, to swap them to correct relative positions.
- For the case where the largest element is at the **N**-th position, we need an addition of **N** operations for rotating. This is because we always rotate the minimum amount, after which the first step will rotate the array until it's back at the same relative position. So in total, the rotations in this step will rotate the array at most one full cycle.

Thus we are using total of 8.5**N** = 425 operations.

An even tighter solution

For each **N** ≤ 50, there also exists a set of 4 rotations that allows us to rotate the first **N**-1 elements by any amount (from 1 to **N**-2) in at most 4 operations, making use of the fact that rotating by **k** is the same as rotating by **k+N-1**. As an example, for **N** = 50, {1, 3, 12, 20} is one such set.

This yields a solution that uses just 325 operations in the worst case.

A randomized variant

In an alternative solution, instead of four rotations of the other= **N**-1 elements, we have four random involutions which swap randomly selected pairs of the **N**-1 elements.

Now we use the same algorithm as in the rotation solution, but instead of rotating elements into place, we use a sequence of the random permutations to move the right element to where it can be swapped with the buffer, and then apply the sequence in reverse to move everything back to where it was.

(We can use a breadth-first search to find the shortest sequence to move any given element into place.)

This process may result in a set of permutations that can't solve the input or can't solve it in few enough permutations, but we can simply retry by choosing a new set of random involutions, as getting a solution with under 450 permutations is highly likely.