Range Equals Size

Problem category:	Dynamic Programming
Expected difficulty:	1500

Solution

First, note that the order of elements in a group is unimportant. Hence, we can sort the input array to make it easier to work with, and we shall consider it sorted in what follows.

Now let's consider an easier version of the problem: one in which the elements are pairwise *distinct*. In this case, there is strict monotonicity of the difference between the maximum and minimum elements in any chosen subarray (i.e., extending it would increase the difference as well as the length, whereas shrinking it would do the opposite). Therefore, the problem would lend itself to a solution using two pointers.

However, in the original problem, the strictness of the monotonicity property does *not* hold: when we extend a subarray, we might remain with the same difference between its maximum and minimum elements (even though its length is increased). Thus, even if the difference is greater than the length, we could still find a valid group with that difference (or a larger one) by extending the subarray. So it wouldn't always be clear which pointer we should move in each iteration.

Thereby, we will examine a different approach. First, let's make a few observations:

1. We can count the occurrences of each number and work as if there were no duplicates.
2. We are interested in differences that are no greater than their group's size. This is because we can always form a group using a subset of the elements (as long as we keep the maximum and minimum); the only exception being a difference of one, in which case it is not possible.

To rephrase: we want to maximize the difference between any two numbers such that it is no greater than the size of the group containing all elements within their range. To this effect, let's make some definitions:

- a_i — the i-th element in the sorted and deduplicated array
- b_i — the number of occurrences of a_i in the original array
- c_i — the total number of elements up to, and including, a_i (i.e., $c_i = \sum_{j=1}^{i} b_j$)

In other words, we want to maximize the value of $a_j - a_i$ for any $i < j$ such that

$$a_j - a_i \leq c_j - c_i + b_i$$

or, equivalently,

$$a_j - c_j \leq a_i - c_i + b_i$$

Note that fixing a value of i also fixes the right side of this equation. Let's call d_j the left side. Then,

$$d_j \leq d_i + b_i$$

Thus, it suffices to precompute all d_i and, for each i, find the maximum j among those that satisfy the equation above. This can be done efficiently using a Fenwick Tree.

Alternatively, one may compute c_i as the total number of elements up to, but *excluding*, a_i. Then, by fixing a value of j, one can find the minimum i among those that satisfy $d_i \geq d_j - b_j$. This can be done incrementally while updating the Fenwick Tree.

Complexity

Since we need to sort the input and use a Fenwick Tree, the overall time complexity is $O(n \log n)$.