

1A

1. Given an array of integers $X = \{x_0, \dots, x_{n-1}\}$, we want to partition the array into k groups P_1, \dots, P_k so that the numbers in every group P_j are distinct, minimizing k , and preserving order.

An algorithm is as follows. Start with array X . Create sorted list $P = \{\}$. Create array $A = []$. Let $k = 0$.

While X is not empty, remove the first x from X . If $x \notin P$, then insert x into P . If x already in P , then append the length of P to A , let $k = k + 1$, and reset $P = \{\}$.

The solution to the problem is k when X is empty.

To construct the actual partitions of X (the P we used above needed to be sorted to get $O(n \log n)$) we can use A , which contains the lengths each partition should be, to create the partitions from X . Note that while the partitions P are not unique, k is unique.

2. Now we prove that the algorithm is correct.

Proof. Let k_A be the solution given by the algorithm, and let k_O be the optimal solution. We know that $k_A \not< k_O$, since that would contradict the optimality of k_O .

Seeking a contradiction, suppose that $k_O < k_A$, i.e. that there are strictly fewer partitions in the optimal solution than the number in the algorithm solution. This means that all elements in P_{k_A} (i.e. all elements in the last partition of the algorithm solution) belong to some partition P'_i in the optimal solution, where $i < k_A$.

This is a contradiction, because our algorithm only creates a new partition when adding another element to a partition would mean that that partition had duplicate elements. In particular, the first element in P_{k_A} (the first element in the last partition of the algorithm solution), cannot belong to P_{k_A-1} , since by the algorithm construction, that element's number is already in P_{k_A-1} , and it certainly cannot belong to any other partition, since then the elements would be out of order.

Thus, it cannot be that $k_O < k_A$, and we also know that $k_A \not< k_O$. We therefore conclude that $k_O = k_A$, i.e. the algorithm solution is optimal and our algorithm is correct. \square

3. Finally we analyze the running time. At the i th step of the algorithm, we need to decide if the i th element of X is contained in a partition P which has at most $i - 1$ elements. Finding an element in a list of size n can be implemented in $O(\log n)$ time. At each step we also do some $O(1)$ operations. We have a step for each element in X , which has length n . At the end we use A to construct the final partitions, which takes $O(n)$ time. Thus the total running time of the algorithm is $O(n \log n)$.

1B

1. Given an array of integers $X = \{x_0, \dots, x_{n-1}\}$, we want to partition the array into k groups P_1, \dots, P_k so that the numbers in every group P_j are distinct, minimizing k , but we *don't* care about preserving order.

An algorithm is as follows:

Create a map M , of elements (x, count) where $x \in X$, and count is the number of times x appears in X . Let $k = 1$, create partition P_1 .

While X not empty, remove the first element of X . If x is not a key in M , then let $c = 1$ and add (x, c) to M . Assign x to partition P_1 .

If x is a key in M , that is, $(x, c) \in M$ for some c , then let $c = c + 1$ (i.e. increment c). If $c > k$, then let $k = c$ and create a new partition P_k . Assign x to partition c .

When X is empty, k is the minimum number of partitions needed.

Note that the partitions themselves are not unique, but k is unique. However, the partitions given by the algorithm are a valid partitioning of X , because as we go through X , we assign the element x to the partition corresponding to the number of times x has appeared so far. Thus no partition has repetitions.

2. Now we prove that the algorithm is correct. (We only prove that the number of partitions k returned by the algorithm is correct. The partitioning of X is not unique.)

Proof. Let k_A be the number of partitions returned by the algorithm, and let k_O be the optimal (minimum) number of partitions. Clearly, $k_A \not< k_O$, since that would contradict the optimality of k_O . Seeking a contradiction, suppose $k_O < k_A$, i.e. that there are strictly fewer partitions in the optimal solution than the number in the algorithm solution. However, by the construction of the algorithm, we know that there must be some element x which appears in X k_A times. But $k_O < k_A$, so if x appears k_A times then there must be some partition in the optimal solution which contains a repetition of an element, which is not allowed. Therefore it cannot be that $k_O < k_A$. Thus if $k_O \not< k_A$ and $k_A \not< k_O$, then we must have $k_A = k_O$. Our algorithm is therefore correct. \square

3. Finally we analyze the running time. We have a step for each element in X , which has length n . With appropriate implementation, we can find/update a particular element in the map M in $O(\log n)$ time (for `c++`, we can use the `map` object). Finding the element with the highest number of repetitions takes $O(n)$. Together the running time is $n \times O(\log n) + O(n) = O(n \log n)$.