Subtask 1 (25 points):

The main idea is to simulate bubble sort, where the array is sorted by swapping adjacent members of an array. We can perform a swap of two adjacent members of the array A_i and A_j by three-step operation (Where $\oplus$ is XOR):

1. $A_j \oplus A_i$
2. $A_i \oplus A_j$
3. $A_j \oplus A_i$

Array will transform like this: $[A_i,\ A_j] \rightarrow [A_i,\ A_j \oplus A_i] \rightarrow [A_j,\ A_j \oplus A_i] \rightarrow [A_j,\ A_i]$. Since the bubble sort can take at most $\frac{n*(n-1)}{2}$ swaps to sort, our method will finish in at most $\frac{n*(n-1)}{2} * 3 \leq 40000$ operations.

Subtask 2 (35 points):

Here, since swapping is not a viable option, it's likely that we should rearrange the array with transformed values and then recover original values. Recovering the original values is important, because the resulting array should not have equal members and preserving original array elements is one way to ensure that. Our approach can be divided into n stages. In each stage, we will swap the largest element to the end of the unsorted part of the array. Consider the array $A = [a,\ b,\ c,\ d,\ e,\ f]$, where c is the maximal element. Firstly, we transform our array to the following one in $n-1$ operations: $[a \oplus b,\ b \oplus c,\ c \oplus d,\ d \oplus e,\ e \oplus f,\ f]$, after that, we will make following operations: $[A_{c+1}, A_c]$, $[A_{c+2}, A_{c+1}]$, $\dots$, $[A_n, A_{n-1}]$, so the array will change to $[a \oplus b,\ b \oplus c,\ c \oplus d,\ c \oplus e,\ c \oplus f,\ c]$. After that, we can change left part of A same way: $[a \oplus c,\ b \oplus c,\ c \oplus d,\ c \oplus e,\ c \oplus f,\ c]$. Last two steps took maximum of $n-1$ (when the maximum element is the first one) operations in total, i.e. each stage can take $2*n-2$ operations. As we can see, all the elements, apart from the last, are the original elements XOR-ed with c. Now we can move on to the next stage, but we have to pick the maximum from the **original** array, not the transformed one. For example, if the second maximum is e, we will move fourth element of A to fifth position (as c already holds the last) even if some element became bigger after the previous transformations.

Suppose that A would be increasingly sorted as follows: $[b, f, a, d, e, c]$. Then, after completing all the stages, which takes $n*(n-1)$ operations, A will be like this: $[b⊕f, f⊕a, a⊕d, d⊕e, e⊕c, c]$ (for example, like in the first stage, at the end of the second stage, last (fifth) element would be $e⊕c$, while all the elements before fifth would be XOR-ed with $e⊕c$ and $a⊕c⊕e⊕c = a⊕e$, so we would have a similar situation as at the end of the first stage). In the final sweep, we can recover original elements of A doing following operations: $[A_{n-1}, A_n]$, $[A_{n-2}, A_{n-1}]$, $\dots$, $[A_1, A_2]$. Finally, array is strictly increasingly sorted and we took total of $n^2 - 1$ operations, which just below the limit when $n = 200$.

Subtask 3 (40 points):

This subtask does not require distinct numbers, which gives us more freedom. We can sort any array A in $O(n \, log \, n)$ operations in $log \, n$ sweeps. In each sweep we find the leftmost element in A with the largest bit in its binary representation and then drag it to the end of the array. Suppose that such element is A_i. We can perform the following operations:

1. If A_{i+1} does not contain A_i-s maximum bit, then $A_{i+1} \oplus A_i$, after which A_{i+1} will contain that bit
2. $A_i \oplus A_{i+1}$ which removes maximal bit from A_i. In this way, we will drag this bit to the end in at most $2*n$ operations.

It's easy to see that after each sweep the maximal bit we can find in unsorted part of A will decrease and since $A_i < 2^{20}$, after 20 sweeps only zeros will be left besides the elements to which we already dragged some maximal bits, which is non-decreasing, thus the array is sorted. In total, this method takes $2*n*20$ operations, which is exactly 40000 when $n = 1000$.