Bad Professor Tutorial

Pre-requisites:

This problem will require concepts of greedy algorithm to solve. Make sure your basics of algorithm design paradigms and greedy algorithms are clear before attempting this problem. The following links can help you learn the prerequisites:

- https://www.topcoder.com/community/data-science/data-science-tutorials/greedy-is-good/
- http://www.geeksforgeeks.org/greedy-algorithms/
- https://www.hackerrank.com/domains/algorithms/greedy
- https://www.codechef.com/tags/problems/greedy
- https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/

Problem Description:

In this problem, initially n variables have been assigned a random value each and the task is to change the values of the least number of variables such that all variables have a value from 1 to n and no two variables have the same value.

Difficulty Level:

Easy-Medium

Editorial:

Let us begin by understanding how we got the given sample output from the given sample input. In the example, there are two pairs of equal numbers (2 and 3). In each pair if we replace one number and if the two new numbers after replacement are not the same, the task is done. Also, we had to choose numbers from 1 to n and n is 4 in this case so the final output becomes 1 2 3 4. From this example, it is clear that we also have to keep track of the new values of the variables after replacement and have to ensure that we do not replace those as well.

Let's look at the problem from another side: how many numbers can we leave unchanged such that all numbers are from 1 to n and no two numbers are the same? It is obvious: these numbers must be from 1 to *n* and they must be pairwise distinct. This condition is necessary and sufficient.

This problem can be solved with greedy algorithm. If me meet the number we have never met before and this number is between 1 and *n*, we will leave this number unchanged. To implement this we can use array where we will mark used numbers.

After that we will look over the array again and allocate numbers that weren't used. Traversing the array twice is sufficient to get the required sequence.

Complexity of solution:

This algorithm works in $O(n)$ because we are traversing the array twice and the number of elements in the array is n.