**Next Permutation**

**Medium**

Implement **next permutation**, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such an arrangement is impossible, it must rearrange it to the lowest possible order (i.e., sorted in ascending order).

The replacement must be [**in place**](http://en.wikipedia.org/wiki/In-place_algorithm) and use only constant extra memory.

**Example 1:**

**Input:** nums = [1,2,3]

**Output:** [1,3,2]

**Example 2:**

**Input:** nums = [3,2,1]

**Output:** [1,2,3]

**Example 3:**

**Input:** nums = [1,1,5]

**Output:** [1,5,1]

**Constraints:**

* 1 <= nums.length <= 100
* 0 <= nums[i] <= 100

class Solution {c

public:

    void nextPermutation(vector<int>& nums) {

        int n=nums.size();

        /\*int index=-1, loc=-1;

        int diff=INT\_MAX;

        for (int i=n-1; i>=0; i--) {

            for (int j=i+1; j<n; j++) {

                if (nums[j]>nums[i] and nums[j]-nums[i]<diff) {

                    diff=nums[j]-nums[i];

                    loc=i;

                    index=j;

                }

            }

            if (index!=-1) {

                swap(nums[loc], nums[index]);

                sort(nums.begin()+loc+1, nums.end());

                return;

            }

        }

        if (index==-1) reverse(nums.begin(), nums.end());\*/

        int i=n-1, index=-1;

        while (i>0) {

            if (nums[i]>nums[i-1]) {

                index=i-1;

                break;

            }

            i--;

        }

        if (index==-1) {

            reverse(nums.begin(), nums.end());

            return;

        }

        sort(nums.begin()+index+1, nums.end());

        int loc=upper\_bound(nums.begin()+index+1, nums.end(), nums[index])-nums.begin();

        swap(nums[index], nums[loc]);

    }

};