<https://leetcode.com/problems/next-permutation/>

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

* For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

* For example, the next permutation of arr = [1,2,3] is [1,3,2].
* Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
* While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, *find the next permutation of* nums.

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

**Attempt 1: 2023-08-01**

**Solution 1: Two Pointers (30min)**

class Solution {

public void nextPermutation(int[] nums) {

// e.g {0,1,2,5,3,3,0}

// i = 6, nums[6] < nums[5], i--

// i = 5, nums[5] == nums[4], i--

// i = 4, nums[4] < nums[3], i--

// i = 3, nums[3] > nums[2], break

// e.g {3,2,1} monotonically decreasing example

// i = 2, nums[2] < nums[1], i--

// i = 1, nums[1] < nums[0], i--

// i = 0, break out while loop

// e.g {1,2,3} monotonically increasing example

// i = 2, nums[2] > nums[1], break

// Find longest increasing suffix (scan from right to left)

int i = nums.length - 1;

while(i >= 1) {

if(nums[i] <= nums[i - 1]) {

i--;

} else {

break;

}

}

// Identify pivot

int pivot = i - 1;

// Monotonically decreasing has next permutation

// just by reversing it

// e.g While the next permutation of arr = [3,2,1] is

// [1,2,3] because [3,2,1] does not have a lexicographical

// larger rearrangement.

if(pivot == -1) {

reverse(nums, 0, nums.length - 1);

} else {

// Find least significant (right most) number larger

// than number at pivot position on its right side

int j = nums.length - 1;

while(j > pivot) {

if(nums[j] > nums[pivot]) {

break;

}

j--;

}

swap(nums, pivot, j);

reverse(nums, pivot + 1, nums.length - 1);

}

}

private void swap(int[] nums, int i, int j) {

int tmp = nums[i];

nums[i] = nums[j];

nums[j] = tmp;

}

private void reverse(int[] nums, int i, int j) {

while(i < j) {

swap(nums, i, j);

i++;

j--;

}

}

}

**Refer to**

<https://leetcode.com/problems/next-permutation/solutions/13994/readable-code-without-confusing-i-j-and-with-explanation/>

public class Solution {

/\*0\*/ public void nextPermutation(int[] nums) {

// pivot is the element just before the non-increasing (weakly decreasing) suffix

/\*2\*/ int pivot = indexOfLastPeak(nums) - 1;

// paritions nums into [prefix pivot suffix]

if (pivot != -1) {

int nextPrefix = lastIndexOfGreater(nums, nums[pivot]); // in the worst case it's suffix[0]

// next prefix must exist because pivot < suffix[0], otherwise pivot would be part of suffix

/\*4\*/ swap(nums, pivot, nextPrefix); // this minimizes the change in prefix

}

/\*5\*/ reverseSuffix(nums, pivot + 1); // reverses the whole list if there was no pivot

/\*6\*/ }

/\*\*

\* Find the last element which is a peak.

\* In case there are multiple equal peaks, return the first of those.

\* @return first index of last peak

\*/

/\*1\*/ int indexOfLastPeak(int[] nums) {

for (int i = nums.length - 1; 0 < i; --i) {

if (nums[i - 1] < nums[i]) return i;

}

return 0;

}

/\*\* @return last index where the {@code num > threshold} or -1 if not found \*/

/\*3\*/ int lastIndexOfGreater(int[] nums, int threshold) {

for (int i = nums.length - 1; 0 <= i; --i) {

if (threshold < nums[i]) return i;

}

return -1;

}

/\*\* Reverse numbers starting from an index till the end. \*/

void reverseSuffix(int[] nums, int start) {

int end = nums.length - 1;

while (start < end) {

swap(nums, start++, end--);

}

}

void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

**Refer to**

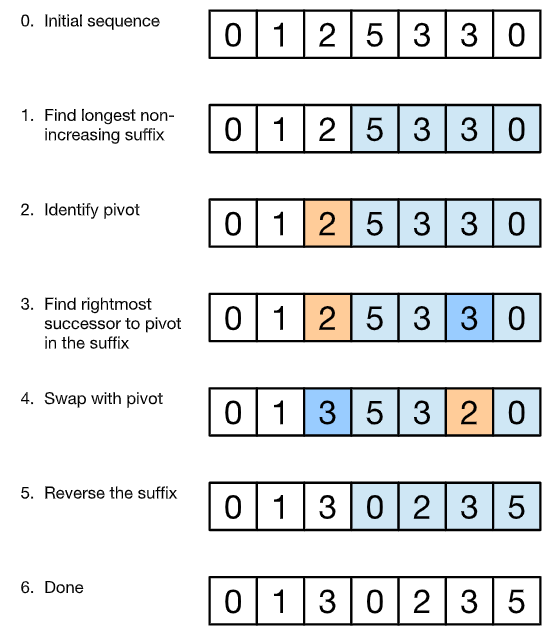
<https://leetcode.wang/leetCode-31-Next-Permutation.html>

这道题的的难度我觉得理解题意就占了一半。题目的意思是给定一个数，然后将这些数字的位置重新排列，得到一个刚好比原数字大的一种排列。如果没有比原数字大的，就升序输出。

关键就是刚好是什么意思？比如说原数字是 A，然后将原数字的每位重新排列产生了 B C D E，然后把这 5 个数字从小到大排列，比如是 D **A** B E C ,那么，我们要找的就是 B，就是那个刚好比 A 大的数字。

再比如 123，其他排列有 132，213，231，312，321，从小到大排列就是 **123** 132 213 231 312 321，那么我们要找的就是 132。

题目还要求空间复杂度必须是 O（1）。



# **解法一**

我们想几个问题。

要想使得数字变大，只要任意一位变大就可以。

要想得到刚好大于原来的数字，要变个位。

这里变大数字，只能利用交换。

如果从个位开始，从右往左进行，找一个比个位大的，交换过来，个位的数字交换到了更高位，由于个位的数字较小，所以交换过去虽然个位变大了，但数字整体变小了。例如 1 3 2，把 2 和 3 交换，变成 1 2 3，个位变大了，但整体数字变小了。

个位不行，我们再看十位，如果从十位左边找一个更大的数字交换过来，和个位的情况是一样的，数字会变小。例如 4 1 2 3，把 2 和 4 交换，2 1 4 3，数字会变小。如果从右边找一个更大的数字交换过来，由于是从低位交换过来的，所以数字满足了会变大。如 4 1 2 3，把 2 和 3 交换，变成 4 1 3 2 数字变大了。

如果十位右边没有比十位数字大的，我们就左移看下一位，再看当前位右边，有没有更大的数字，没有就一直左移就可以。

还有一个问题，如果右边有不止一个大于当前位的数字选哪个？选那个刚好大于当前位的，这样会保证数字整体尽可能的小。

交换完结束了吗？并没有。因为交换完数字变大了，但并不一定是刚好大于原数字的。例如 158476531，我们从十位开始，十位右边没有大于 3 的。再看百位，百位右边没有大于 5 的。直到 4 ，右边出现了很多大于 4 的，选那个刚好大于 4 的，也就是 5 。然后交换，变成 158**5**76**4**31，数字变大了，但并不是刚好大于 158476531，我们还需要将 5 右边的数字从小到大排列。变成158**5**13467，就可以结束了。

而最后的排序，我们其实并不需要用排序函数，因为交换的位置也就是 5 的右边的数字一定是降序的，我们只需要倒序即可了。看一下 LeetCode 提供的动图更好理解一些。

**This is a GIF**



**再看这个过程，我们其实是从右向左找到第一个数字不再递增的位置，然后从右边找到一个刚好大于当前位的数字即可。**

再看下代码吧。

public void nextPermutation(int[] nums) {

int i = nums.length - 2;

//找到第一个不再递增的位置

while (i >= 0 && nums[i + 1] <= nums[i]) {

i--;

}

//如果到了最左边，就直接倒置输出

if (i < 0) {

reverse(nums, 0);

return;

}

//找到刚好大于 nums[i]的位置

int j = nums.length - 1;

while (j >= 0 && nums[j] <= nums[i]) {

j--;

}

//交换

swap(nums, i, j);

//利用倒置进行排序

reverse(nums, i + 1);

}

private void swap(int[] nums, int i, int j) {

int temp = nums[j];

nums[j] = nums[i];

nums[i] = temp;

}

private void reverse(int[] nums, int start) {

int i = start, j = nums.length - 1;

while (i < j) {

swap(nums, i, j);

i++;

j--;

}

}

时间复杂度：最坏的情况就是遍历完所有位，O（n），倒置也是 O（n），所以总体依旧是 O（n）。

空间复杂度：O（1）。