next_permutation : find next lexicographically greater permutation

Problem Statement: Given an array Arr[] of integers, rearrange the numbers of the given array into the lexicographically next greater permutation of numbers.

If such an arrangement is not possible, it must rearrange to the lowest possible order (i.e., sorted in ascending order).

▼ **Examples**
Example 1 :

```
Input format: Arr[] = {1,3,2}
Output: Arr[] = {2,1,3}
Explanation: All permutations of {1,2,3} are {{1,2,3} , {1,3,2}, {2,13} , {2,3,1}
, {3,1,2} , {3,2,1}}. So, the next permutation just after {1,3,2} is {2,1,3}.
```

Example 2:

```
Input format: Arr[] = {3,2,1}
Output: Arr[] = {1,2,3}
Explanation: As we see all permutations of {1,2,3}, we find {3,2,1} at the last
position. So, we have to return the topmost permutation.
```

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

▼ Solution 1
▼ Algorithm / Intuition ›
Brute Force: Finding all possible permutations.

Approach :

Step 1: Find all possible permutations of elements present and store them.

Step 2: Search input from all possible permutations.

Step 3: Print the next permutation present right after it.

For reference of how to find all possible permutations, follow up https://www.youtube.com/watch?v=f2ic2Rsc9pU&t=32s. This video shows for distinct elements but code works for duplicates too.

▼ Complexity Analysis ›
For finding, all possible permutations, it is taking N!xN. N represents the number of elements present in the input array. Also for searching input arrays from all possible permutations will take N!. Therefore, it has a Time complexity of O(N!xN).

Space Complexity :

Since we are not using any extra spaces except stack spaces for recursion calls. So, it has a space complexity of O(1).

▼ Solution 2
Using in-built function

C++ provides an in-built function called next_permutation() which directly returns the lexicographically next greater permutation of the input.

▼ Code ›
▼ Solution 3
▼ Algorithm / Intuition ›
The steps are the following:

1. **Find the break-point, i:** Break-point means the ***first index i from the back of the given array*** where arr[i] becomes smaller than arr[i+1].
   For example, if the given array is {2,1,5,4,3,0,0}, the break-point will be index 1(*0-based indexing*). Here from the back of the array, index 1 is the first index where arr[1] i.e. 1 is smaller than arr[i+1] i.e. 5.
   To find the break-point, using a loop we will traverse the array backward and store the index i where arr[i] is less than the value at index (i+1) i.e. arr[i+1].
2. **If such a break-point does not exist i.e. if the array is sorted in decreasing order,** the given permutation is the last one in the sorted order of all possible permutations. So, the next permutation must be the first i.e. the permutation in increasing order.
   So, ***in this case, we will reverse the whole array and will return it as our answer.***
3. **If a break-point exists:**
   1. Find the smallest number i.e. > arr[i] and in the right half of index i(i.e. from index i+1 to n-1) and swap it with arr[i].
   2. Reverse the entire right half(i.e. from index i+1 to n-1) of index i. And finally, return the array.

Note: *For a better understanding of intuition, please watch the video at the bottom of the page.*

Intuition:

We build up the intuition of the algorithm through the following observations.

▼ Observations ›
▼ Observation 1 ›
Let’s try to observe some dictionary-ordered strings like “raj”, “rax”, and “rbx”. If we carefully observe, we can notice that these strings contain a common prefix, and the rankings are done based on the differentiating characters.

For example, “raj” and “rax” has a common prefix i.e. “ra” and the differentiating characters are ‘j’ and ‘x’. Now, as ‘j’ appears before ‘x’ in the alphabet, “raj” appears before “rax” in the given order. The same logic is applicable for “rax” and “rbx”(*Common prefix: “r”, differentiating characters: ‘a’ and ‘b’)*.

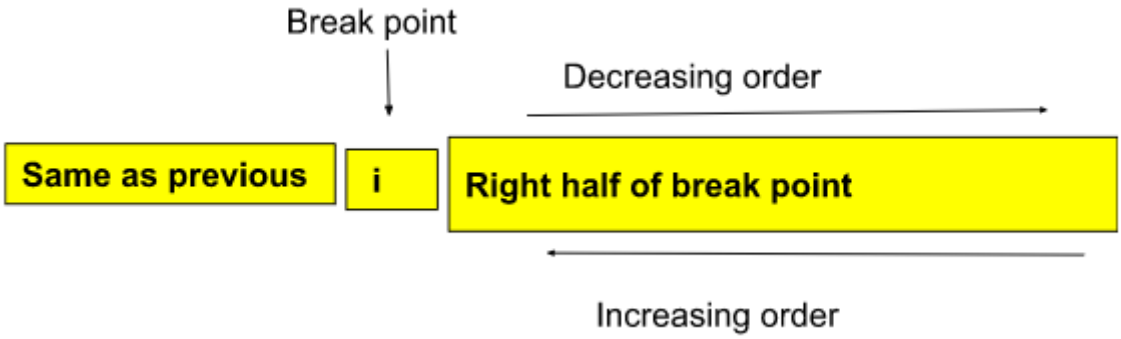
The same observation can be done on the permutations of numbers. For example, if the array is [1, 2, 3], all possible permutations in sorted order will look like the following:

- [1, 2, 3]
- [1, 3, 2]
- [2, 1, 3]
- [2, 3, 1]
- [3, 1, 2]
- [3, 2, 1]

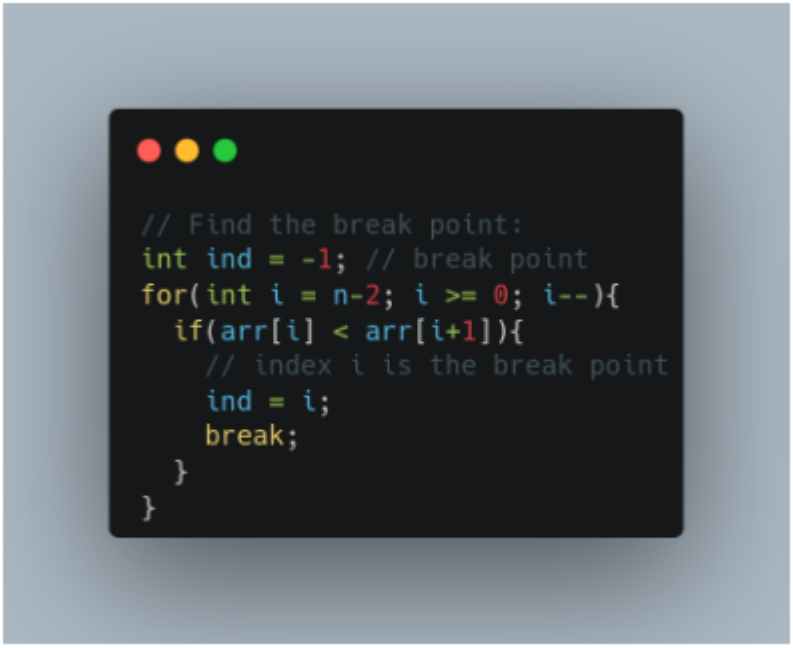
In the above cases, we can also notice that all the permutations contain an index i(*between the first and second last index*) such that its right part is sorted in decreasing order. Now, **if we look at the array in the backward direction, it is sorted in increasing order up to index i (*from n-1 to index i+1*).**

We can call this index i as the break-point of the array. The left half of index i (*the length of the left half might be 0*) in the current permutation is the same as in the previous permutation. And the right half of the break-point is always in decreasing order.

The structure of every possible permutation is the following:



After all, we can conclude that the difference between the next and current permutation always starts at the index i i.e. the break-point. How to find the break-point in an array:
We can clearly observe that the right half of the break-point will always be in decreasing order. So, ***from the backside, the array will be in increasing order up to the break-point index***. Keeping this in mind, we will traverse the array from the backside and we will break from the first index where arr[i] becomes smaller than arr[i+1]. The code will look like the following:

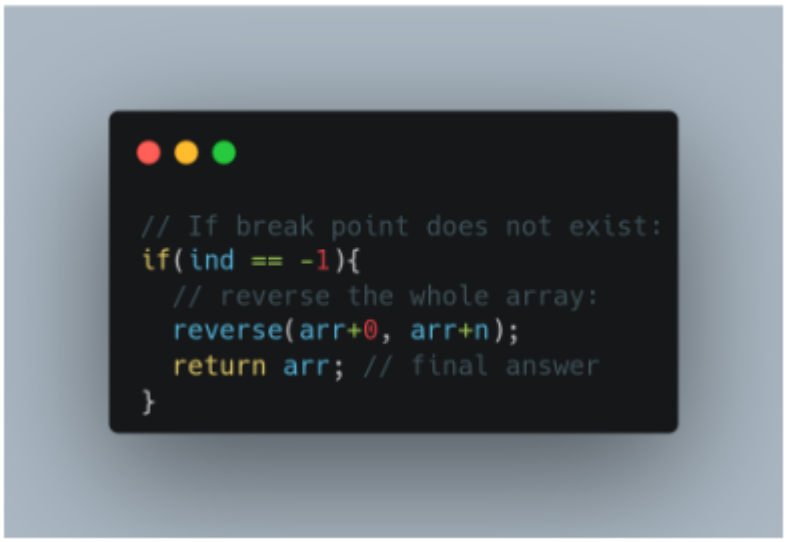


```
// Find the break point:
int ind = -1; // break point
for(int i = n-2; i >= 0; i--){
  if(arr[i] < arr[i+1]){
    // index i is the break point
    ind = i;
    break;
  }
}
```

▼ Observation 2 ›

If the break-point does not exist i.e. ind remains -1 in the code: For an array that is sorted in decreasing order break-point does not exist. Here, we can assure that the given array is the last one in the sorted order of all permutations. In this case, the break-point index will be -1, the right half of the break-point will be the whole array and the left half will be of length 0.

In this case, the next permutation should be as minimum as possible. So to achieve it, we just need to reverse the whole array. And this will be our answer in this case. The code will look like the following:



```
// If break point does not exist:
if(ind == -1){
  // reverse the whole array:
  reverse(arr+0, arr+n);
  return arr; // final answer
}
```

▼ Observation 3 ›

Until now, we have found the break-point where the difference starts in the permutations. ***As the left half of the break-point should remain the same, we will not perform any operation on the left half.***

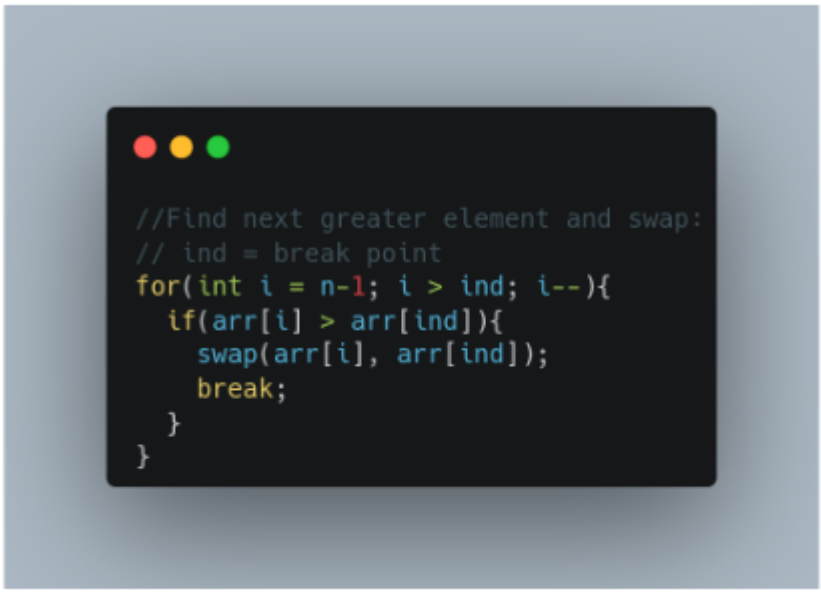
Now we need to modify the break-point and the right half to get the next permutation. By the convention of ordering, we can say that the element at the break-point in the next permutation should be the next greater element of arr[break-point] in the current permutation.

Now to find the next greater element, we cannot use the whole array as the left half of the break-point should remain the same. But we can use the right half as it can be modified.

So, we will find the next greater element of arr[break-point] from the right half and swap it with arr[break-point] itself. Next greater element of arr[break-point] means the smallest element in the right half, greater than arr[break-point].

How to find and swap the next greater element of arr[break-point]:
The right half is sorted in decreasing order(or increasing order from the backside). So, we will again ***traverse the array in the backward direction, and at the first index where arr[index] > arr[break-point], we will swap arr[index] and arr[break-point]***.

The code will be the following:

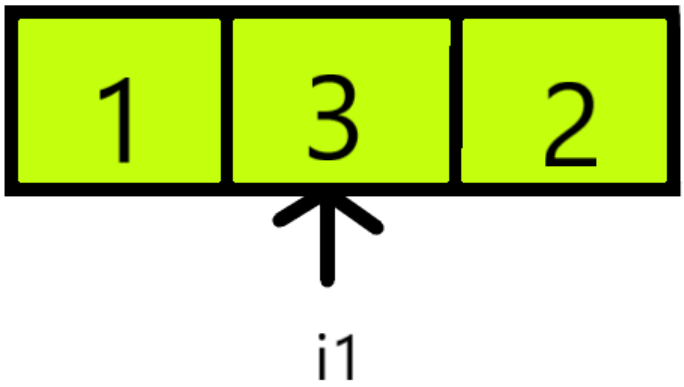


```
//Find next greater element and swap:
// ind = break point
for(int i = n-1; i > ind; i--){
  if(arr[i] > arr[ind]){
    swap(arr[i], arr[ind]);
    break;
  }
}
```

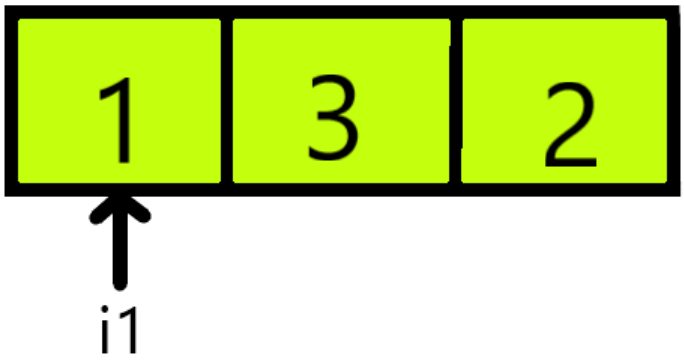
For example, if the given array is {2, 1, 5, 4, 3, 0, 0}, the break-point will be index 1, and the next greater element of arr[1] i.e. 1 is 3 from the right half. After swapping the array will be like: {2, 3, 5, 4, 1, 0, 0}.

▶ Observation 4 ›
▼ Dry Run ›
We will take the input array {1,3,2}.

Step 1: First find an increasing sequence. We take i1 = 1. Starting traversing backward.

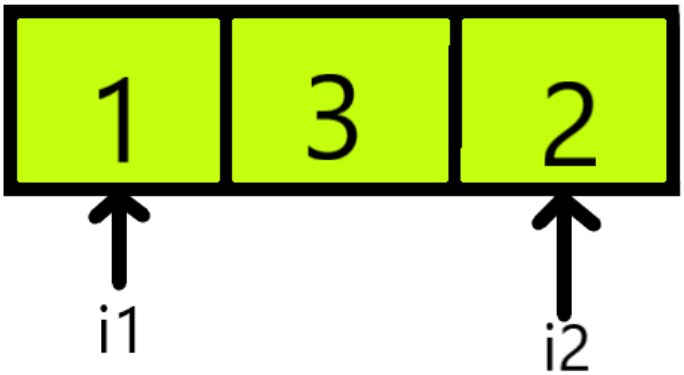


Step 2: Since 3 is not less than 2, we decrease i1 by 1.



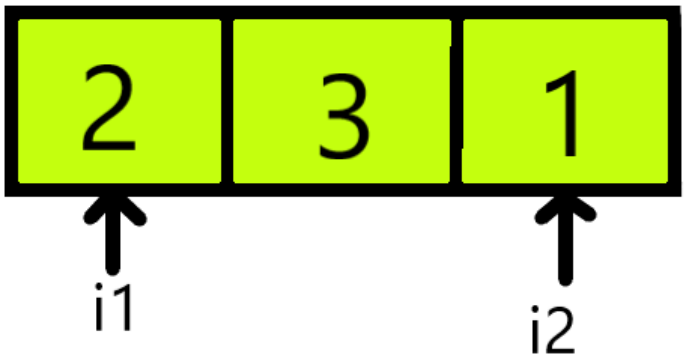
Step 3: Since 1 is less than 2, we achieved our start of the increasing sequence. Now, i1 = 0.

Step 4: i2 will be another index to find just greater than i1 indexed elements in the array. Point i2 to the last element.



Step 5: i2 indexed element is greater than i1 indexed element. So, i2 has a value of 2.

Step 6: Swapping values present in i1 and i2 indices.



Step 7: Reversing from i1+1 index to last of the array.

Thus, we achieved our final answer.

▼ Code ›

```
from typing import List

def nextGreaterPermutation(A: List[int]) -> List[int]:
    n = len(A) # size of the array.

    # Step 1: Find the break point:
    ind = -1 # break point
    for i in range(n-2, -1, -1):
        if A[i] < A[i + 1]:
            # index i is the break point
            ind = i
            break

    # If break point does not exist:
    if ind == -1:
        # reverse the whole array:
        A.reverse()
        return A

    # Step 2: Find the next greater element
    #         and swap it with arr[ind]:
    for i in range(n - 1, ind, -1):
        if A[i] > A[ind]:
            A[i], A[ind] = A[ind], A[i]
            break

    # Step 3: reverse the right half:
    A[ind+1:] = reversed(A[ind+1:])

    return A

if __name__ == "__main__":
    A = [2, 1, 5, 4, 3, 0, 0]
    ans = nextGreaterPermutation(A)

    print("The next permutation is: [", end="")
    for it in ans:
        print(it, end=" ")
    print("]")
```

Output: The next permutation is: [2 3 0 0 1 4 5]

▼ Complexity Analysis ›

Time Complexity: O(3N), where N = size of the given array
Finding the break-point, finding the next greater element, and reversal at the end takes O(N) for each, where N is the number of elements in the input array. This sums up to 3*O(N) which is approximately O(3N).

Space Complexity: Since no extra storage is required. Thus, its space complexity is O(1).