* Problem statement :-

→ We are given a input array & we are supposed to find its next permutation.

→ Permutation is arrangement of elements in a sequence.

→ The next permutation of an array of integers is the next lexicographically greater permutation of its integers.

* Example :-

nums = [1, 2, 3]

All possible permutations are :- [1, 2, 3] ← [1, 3, 2] [2, 1, 3] [2, 3, 1] [3, 1, 2] [3, 2, 1]

Here for every arrangement, its next permutation is the next lexicographical greater permutation.

If an array has duplicate elements then [1 1 5] ← [1 5 1] [5 1 1]

Note / Constraints :-

→ Replacement should be in-place.

→ Only use constant extra memory.

* Solution :-

arr = [1, 5, 8, 4, 7, 6, 5, 3, 1]

indices: 0 1 2 3 4 5 6 7 8

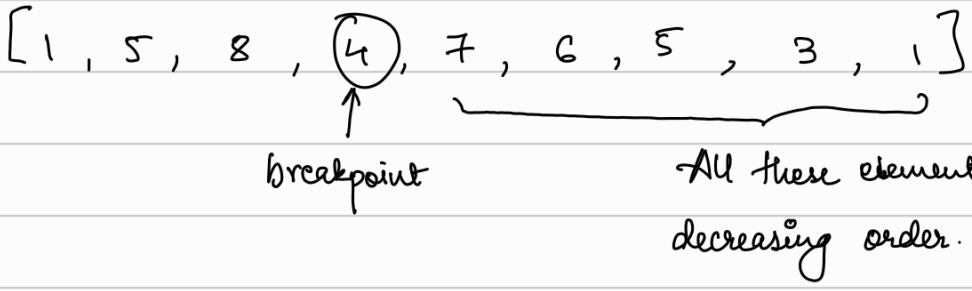
..... 10K 1000 100 10 1

→ So far we know if all the elements are arranged in descending order, the next permutation will be the entire array sorted in ascending order.

→ If that's the case we can just reverse the array because the TC will be $O(n)$.

→ But if that's not the case, then we need to find a breakpoint in the array from the end of the array to start such that arr[i] > arr[i-1].

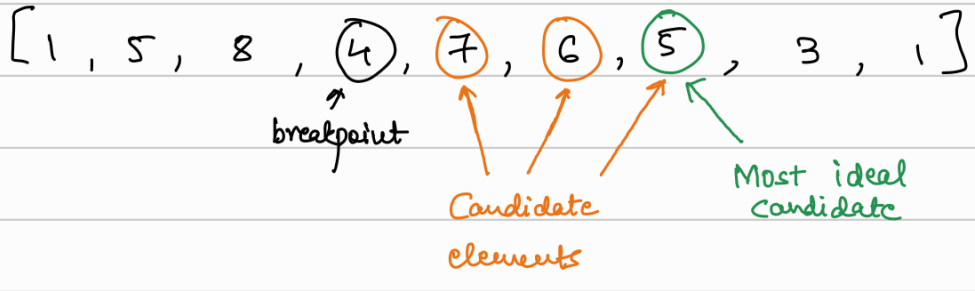
→ We are doing it from the end because if there is a decreasing sequence then its next permutation will be just the array in ascending order.



→ Once the breakpoint is found, we store that element & its respective index to easy manipulation.

→ If no breakpoint is found, the array is in decreasing order & we just need to reverse it.

→ Since, we have found a breakpoint, now we need to swap it with the element that is just greater than the breakpoint.



→ A good way to find the most ideal candidate is to start looking for it from the end of the array because all the elements to the right of the breakpoint are in decreasing order.

→ Once we find the ideal candidate, we swap it with the breakpoint.

After swapping,

[1, 5, 8, 5, 7, 6, 4, 3, 1]

→ In this way, we see that everything to the right of the breakpoint index is in decreasing order.

→ Now, we just reverse from (breakpoint index + 1) to the end of the array.

[1, 5, 8, 5, 1, 3, 4, 6, 7]

→ And this is the next possible permutation.

```
function nextPermutation (arr) {
    const n = nums.length;
    let breakpoint = -1, idxBreakpoint = -1;

    for (let i = n-1; i > 0; i--) {
        if (arr[i] > arr[i-1]) { ← Breakpoint found
            breakpoint = arr[i-1];
            idxBreakpoint = i-1;
            break; ← Because we don't want to keep of searching for
        }                    new breakpoint.
    }
    if (idxBreakpoint != -1) {
        for (let j = n-1; j > idxBreakpoint; j--) {
            if (arr[j] > breakpoint) {
                [arr[j], arr[idxBreakpoint]] = [arr[idxBreakpoint], arr[j]];
                break; → No more swapping required
            }
        }
    }
    reverse (arr, idxBreakpoint + 1, n-1);
}
```

TC :- O(n)

SC :- O(1)