300. Longest Increasing Subsequence

Given an integer array `nums`, return the length of the longest strictly increasing subsequence.

A **subsequence** is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements. For example, `[3,6,2,7]` is a subsequence of the array `[0,3,1,6,2,2,7]`.

Example 1:

```
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore
the length is 4.
```

Example 2:

```
Input: nums = [0,1,0,3,2,3]
Output: 4
```

Example 3:

```
Input: nums = [7,7,7,7,7,7,7]
Output: 1
```

```
import bisect
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        dp = []
        dp.append(nums[0])
        for i in range(1,len(nums)):
            if nums[i]>dp[-1]:
                dp.append(nums[i])
            else:
                # idx = self.bisect_left(dp,nums[i],0,len(dp)-1)
                idx = bisect.bisect_left(dp,nums[i])
                dp[idx] = nums[i]
        return len(dp)

    def bisect_left(self,arr,val,start,end):
        if start==end:
```

```
            return start
        while start<=end:
            mid = (start+end)//2
            if arr[mid]>val:
                end = mid-1
            elif arr[mid]<val:
                start = mid+1
            else:
                return mid
        return start
```

```
dp = [0]*(len(nums))
        dp[0]=1
        for i in range(1,len(nums)):
            temp = 0
            for j in range(i):
                if nums[j]<nums[i]:
                    temp = max(temp,dp[j])
```

Solution 2: Greedy with Binary Search

- Let's construct the idea from following example.
- Consider the example `nums = [2, 6, 8, 3, 4, 5, 1]`, let's try to build the increasing subsequences starting with an empty one: `sub1 = []`.
  1. Let pick the first element, `sub1 = [2]`.
  2. `6` is greater than previous number, `sub1 = [2, 6]`
  3. `8` is greater than previous number, `sub1 = [2, 6, 8]`
  4. `3` is less than previous number, we can't extend the subsequence `sub1`, but we must keep `3` because in the future there may have the longest subsequence start with `[2, 3]`, `sub1 = [2, 6, 8], sub2 = [2, 3]`.
  5. With `4`, we can't extend `sub1`, but we can extend `sub2`, so `sub1 = [2, 6, 8], sub2 = [2, 3, 4]`.
  6. With `5`, we can't extend `sub1`, but we can extend `sub2`, so `sub1 = [2, 6, 8], sub2 = [2, 3, 4, 5]`.
  7. With `1`, we can't extend neighter `sub1` nor `sub2`, but we need to keep `1`, so `sub1 = [2, 6, 8], sub2 = [2, 3, 4, 5], sub3 = [1]`.
  8. Finally, length of longest increase subsequence = `len(sub2)` = 4.
- In the above steps, we need to keep different `sub` arrays (`sub1`, `sub2`..., `subk`) which causes poor performance. But we notice that we can just keep one `sub` array, when new number `x` is not

greater than the last element of the subsequence `sub`, we do binary search to find the smallest element >= `x` in `sub`, and replace with number `x`.

- Let's run that example `nums = [2, 6, 8, 3, 4, 5, 1]` again:
  1. Let pick the first element, `sub = [2]`.
  2. `6` is greater than previous number, `sub = [2, 6]`
  3. `8` is greater than previous number, `sub = [2, 6, 8]`
  4. `3` is less than previous number, so we can't extend the subsequence `sub`. We need to find the smallest number >= `3` in `sub`, it's `6`. Then we overwrite it, now `sub = [2, 3, 8]`.
  5. `4` is less than previous number, so we can't extend the subsequence `sub`. We overwrite `8` by `4`, so `sub = [2, 3, 4]`.
  6. `5` is greater than previous number, `sub = [2, 3, 4, 5]`.
  7. `1` is less than previous number, so we can't extend the subsequence `sub`. We overwrite `2` by `1`, so `sub = [1, 3, 4, 5]`.
  8. Finally, length of longest increase subsequence = `len(sub)` = 4.

Now the nlogn approach uses binary search.