Longest Increasing Subsequence

Two classic approaches - pick based on constraints:

1) $O(n^2)$ DP:

Let dp[i] = length of the LIS ending at i.

Transition:

- For each i, check all $j < i$:
  - If nums[j] < nums[i], we can extend: dp[i] = max(dp[i], dp[j] + 1).
- Initialize all dp[i] = 1.
- Answer = max_i dp[i].

Why it works: you extend the best increasing subsequence that ends before i and is smaller than nums[i].

2) $O(n \log n)$ "patience sorting" trick (preferred for large n):

Maintain an array tails, where:

- tails[k] = the smallest possible tail value of an increasing subsequence of length k+1.

Process each x in nums:

- Find the first index i in tails with tails[i] >= x (lower_bound).
  - If found, set tails[i] = x (we improved the tail for length i+1).
  - If not found (i.e. x is larger than all tails), append x to tails (we extended the LIS by 1).
- The length of LIS is tails.size() at the end.

Key intuition:

- tails is kept sorted.
- We never claim tails itself is a valid subsequence; it's a bookkeeping structure ensuring minimal tails for each length.
- Using >= gives strictly increasing subsequences (duplicates don't extend).

Complexity:

- Time: $O(n \log n)$ for the binary searches.
- Space: $O(n)$ for tails.