**Approach 1: Dynamic Programming**

**Realizing a Dynamic Programming Problem**

This problem has two important attributes that let us know it should be solved by dynamic programming. First, the question is asking for the maximum or minimum of something. Second, we have to make decisions that may depend on previously made decisions, which is very typical of a problem involving subsequences.

As we go through the input, each "decision" we must make is simple: is it worth it to consider this number? If we use a number, it may contribute towards an increasing subsequence, but it may also eliminate larger elements that came before it. For example, let's say we have nums = [5, 6, 7, 8, 1, 2, 3]. It isn't worth using the 1, 2, or 3, since using any of them would eliminate 5, 6, 7, and 8, which form the longest increasing subsequence. We can use dynamic programming to determine whether an element is worth using or not.

**A Framework to Solve Dynamic Programming Problems**

Typically, dynamic programming problems can be solved with three main components. If you're new to dynamic programming, this might be hard to understand but is extremely valuable to learn since **most dynamic programming problems can be solved this way**.

First, we need some function or array that represents the answer to the problem from a given state. For many solutions on LeetCode, you will see this function/array named "dp". For this problem, let's say that we have an array dp. As just stated, this array needs to represent **the answer to the problem for a given state**, so let's say that dp[i] represents the length of the **longest increasing subsequence** that **ends with the**ithi^{th}*ith***element**. The "state" is one-dimensional since it can be represented with only one variable - the index i.

Second, we need a way to transition between states, such as dp[5] and dp[7]. This is called a **recurrence relation** and can sometimes be tricky to figure out. Let's say we know dp[0], dp[1], and dp[2]. How can we find dp[3] given this information? Well, since dp[2] represents the length of the longest increasing subsequence that ends with nums[2], if nums[3] > nums[2], then we can simply take the subsequence ending at i = 2 and append nums[3] to it, increasing the length by 1. The same can be said for nums[0] and nums[1] if nums[3] is larger. Of course, we should try to maximize dp[3], so we need to check all 3. Formally, the recurrence relation is: dp[i] = max(dp[j] + 1) for all j where nums[j] < nums[i] and j < i.

The third component is the simplest: we need a base case. For this problem, we can initialize every element of dp to 1, since every element on its own is technically an increasing subsequence.

1 / 18

**Algorithm**

1. Initialize an array dp with length nums.length and all elements equal to 1. dp[i] represents the length of the longest increasing subsequence that ends with the element at index i.
2. Iterate from i = 1 to i = nums.length - 1. At each iteration, use a second for loop to iterate from j = 0 to j = i - 1 (all the elements before i). For each element before i, check if that element is smaller than nums[i]. If so, set dp[i] = max(dp[i], dp[j] + 1).
3. Return the max value from dp.

**Implementation**

**Complexity Analysis**

Given NN*N* as the length of nums,

* Time complexity: O(N2)O(N^2)*O*(*N*2)

We use two nested for loops resulting in 1+2+3+4+...+N=N∗(N+1)21 + 2 + 3 + 4 + ... + N = \dfrac {N \* (N + 1)}{2}1+2+3+4+...+*N*=2*N*∗(*N*+1)​ operations, resulting in a time complexity of O(N2)O(N^2)*O*(*N*2).

* Space complexity: O(N)O(N)*O*(*N*)

The only extra space we use relative to input size is the dp array, which is the same length as nums.

**Approach 2: Intelligently Build a Subsequence**

**Intuition**

As stated in the previous approach, the difficult part of this problem is deciding if an element is worth using or not. Consider the example nums = [8, 1, 6, 2, 3, 10]. Let's try to build an increasing subsequence starting with an empty one: sub = [].

* At the first element 8, we might as well take it since it's better than nothing, so sub = [8].
* At the second element 1, we can't increase the length of the subsequence since 8 >= 1, so we have to choose only one element to keep. Well, this is an easy decision, let's take the 1 since there may be elements later on that are greater than 1 but less than 8, now we have sub = [1].
* At the third element 6, we can build on our subsequence since 6 > 1, now sub = [1, 6].
* At the fourth element 2, we can't build on our subsequence since 6 >= 2, but can we improve on it for the future? Well, similar to the decision we made at the second element, if we replace the 6 with 2, we will open the door to using elements that are greater than 2 but less than 6 in the future, so sub = [1, 2].
* At the fifth element 3, we can build on our subsequence since 3 > 2. Notice that this was only possible because of the swap we made in the previous step, so sub = [1, 2, 3].
* At the last element 10, we can build on our subsequence since 10 > 3, giving a final subsequence sub = [1, 2, 3, 10]. The length of sub is our answer.

It appears the best way to build an increasing subsequence is: for each element num, if num is greater than the largest element in our subsequence, then add it to the subsequence. Otherwise, perform a linear scan through the subsequence starting from the smallest element and replace the first element that is greater than or equal to num with num. This opens the door for elements that are greater than num but less than the element replaced to be included in the sequence.

One thing to add: this algorithm does not always generate a valid subsequence of the input, but the length of the subsequence will always equal the length of the longest increasing subsequence. For example, with the input [3, 4, 5, 1], at the end we will have sub = [1, 4, 5], which isn't a subsequence, but the length is still correct. The length remains correct because the length only changes when a new element is larger than any element in the subsequence. In that case, the element is appended to the subsequence instead of replacing an existing element.

**Algorithm**

1. Initialize an array sub which contains the first element of nums.
2. Iterate through the input, starting from the second element. For each element num:
   * If num is greater than any element in sub, then add num to sub.
   * Otherwise, iterate through sub and find the first element that is greater than or equal to num. Replace that element with num.
3. Return the length of sub.

**Implementation**

**Complexity Analysis**

Given NN*N* as the length of nums,

* Time complexity: O(N2)O(N^2)*O*(*N*2)

This algorithm will have a runtime of O(N2)O(N^2)*O*(*N*2) only in the worst case. Consider an input where the first half is [1, 2, 3, 4, ..., 99998, 99999], then the second half is [99998, 99998, 99998, ..., 99998, 99998]. We would need to iterate (N/2)2(N / 2)^2(*N*/2)2 times for the second half because there are N/2N / 2*N*/2 elements equal to 99998, and a linear scan for each one takes N/2N / 2*N*/2 iterations. This gives a time complexity of O(N2)O(N^2)*O*(*N*2).

Despite having the same time complexity as the previous approach, in the best and average cases, it is much more efficient.

* Space complexity: O(N)O(N)*O*(*N*)

When the input is strictly increasing, the sub array will be the same size as the input.

**Approach 3: Improve With Binary Search**

**Intuition**

In the previous approach, when we have an element num that is not greater than all the elements in sub, we perform a linear scan to find the first element in sub that is greater than or equal to num. Since sub is in sorted order, we can use binary search instead to greatly improve the efficiency of our algorithm.

**Algorithm**

1. Initialize an array sub which contains the first element of nums.
2. Iterate through the input, starting from the second element. For each element num:
   * If num is greater than any element in sub, then add num to sub.
   * Otherwise, perform a binary search in sub to find the smallest element that is greater than or equal to num. Replace that element with num.
3. Return the length of sub.

**Implementation**

In Python, the [bisect](https://docs.python.org/3/library/bisect.html) module provides super handy functions that does binary search for us.

**Complexity Analysis**

Given NN*N* as the length of nums,

* Time complexity: O(N⋅log⁡(N))O(N \cdot \log(N))*O*(*N*⋅log(*N*))

Binary search uses log⁡(N)\log(N)log(*N*) time as opposed to the O(N)O(N)*O*(*N*) time of a linear scan, which improves our time complexity from O(N2)O(N^2)*O*(*N*2) to O(N⋅log⁡(N))O(N \cdot \log(N))*O*(*N*⋅log(*N*)).

* Space complexity: O(N)O(N)*O*(*N*)

When the input is strictly increasing, the sub array will be the same size as the input.