Documentation

Problem Overview

The **Longest Increasing Subsequence (LIS)** problem asks us to find the length of the longest subsequence of a given sequence of numbers that is strictly increasing. Unlike a subarray, subsequences don't require the elements to be contiguous, but the order of elements in the subsequence must be preserved. The problem provides an integer array of nums, and we are tasked with returning the longest-long, strictly increasing subsequence. The problem asks for an efficient algorithm that can solve this problem in **O(n log n)** time complexity, especially when the array length is large (up to 2500).

Approach 1: *Dynamic Programming ($O(n^2)$ Time Complexity)*

The simplest approach to solving the LIS problem is using a **dynamic programming** (DP) approach. This method involves constructing an auxiliary array dp, where dp[i] represents the length of the longest increasing subsequence that ends at index i. We initialize each entry of up to 1, as each element in the array can be considered a LIS of length 1 by itself. Then, for each element nums[i], we check all previous elements nums[j] (where j < i). If nums[j] < nums[i], it means we can append nums[i] to the subsequence ending at nums[j], and we update dp[i] as dp[i] = max(dp[i], dp[j] + 1). Finally, the length of the longest increasing subsequence is the maximum value in the dp array.

While this approach works correctly, its time complexity is **$O(n^2)$** because for each element i, we potentially examine all previous elements j (which results in an inner loop for each outer loop iteration). This method is manageable for smaller inputs but can become inefficient when the input size is large (e.g., when n = 2500).

Approach 2: *Binary Search and Dynamic Programming (O(n log n) Time Complexity)*

For larger input sizes, we can improve the time complexity to **O(n log n)** by using a more advanced method that combines **binary search** with dynamic programming. In this approach, we maintain a list sub that will store the smallest possible tail values of increasing subsequences of different lengths. The key observation is that we perform a binary search for each number in nums to find its position in the sub-list, *where we either extend or replace elements to maintain the smallest possible tails. Specifically:*

1. If the number is greater than the largest element in sub, we append it to sub.
2. If the number can replace an element in sub, we find the appropriate position using binary search (bisect_left) and replace the element at that position.

At the end of the iteration, the length of the sub list represents the length of the longest increasing subsequence. This method is efficient because the binary search allows us to insert or replace elements in sub in **O(log n)** time, and we perform this operation for each element in the nums array, resulting in an overall time complexity of

O(n log n).

Time and Space Complexity

The **Dynamic Programming ($O(n^2)$)** approach has a time complexity of **$O(n^2)$** because we use two nested loops to compare each element with all previous ones. The space complexity is **O(n)** due to the dp array storing the length of LIS for each index.

On the other hand, the **Binary Search with Dynamic Programming (O(n log n))** approach has a time complexity of **O(n log n)** because we use binary search to maintain the sub array. The space complexity remains **O(n)** because the sub-array holds at most n elements, which is the worst-case scenario when the entire array forms an increasing subsequence.

Practical Use and Efficiency

The **O(n log n)** solution is highly efficient and practical, especially when dealing with larger input sizes. The **$O(n^2)$** solution, while simpler and easier to understand, is more suitable for smaller datasets. However, the

O(n log n) approach should be preferred when the input size can be as large as 2500 elements. This solution efficiently solves the problem within the constraints and ensures that large inputs are handled in a reasonable time frame. Furthermore, the binary search approach makes the solution more adaptable and can be extended to related problems, such as finding the longest subsequence that satisfies other conditions.