Applied Algorithms

Programming Assignment-3

Jaya Sandeep Ketha

October 2023

Longest Increasing Subsequence:
The given code implements the Longest Increasing Sub-sequence (LIS) problem using a dynamic programming approach with a time complexity of O(n log n). Let's break down the logic behind the algorithm and analyze its running time.

Algorithm Explanation:
1. Initialization:
The algorithm begins by initializing several variables:

n:	The length of the input array A.
tail_elements:	An array to store the tail elements of active subsequences.
B_length:	The length of the longest increasing subsequence found so far.
B:	An array of lists to store the actual LIS at each step.

2. Loop through the input array A:
The algorithm iterates through the input array A from left to right, considering each element one by one (using the loop variable i).

3. Handling Three Cases:
For each element A[i], the algorithm handles three cases based on its value in relation to the tail_elements array:

a. If A[i] is less than the smallest tail element (tail_elements[0]):
In this case, A[i] starts a new, smaller increasing subsequence. Therefore, we update tail_elements[0] to A[i], and we replace the first element of B with A[i].

b. If A[i] is greater than the largest tail element (tail_elements[B_length - 1]):
Here, A[i] can be added to the end of the longest increasing subsequence found so far. So, we extend the LIS by appending A[i] to the LIS stored at B[B_length - 1]. We also increment B_length by 1 to indicate the new length of the LIS.

c. If A[i] falls between the smallest and largest tail elements:
In this case, we perform a binary search to find the position left to insert A[i] in the tail_elements array. We want to find the place where A[i] can extend an existing subsequence without breaking the increasing order. After finding the position, we update tail_elements[left] to A[i]. We then copy the LIS from B[left - 1] to B[left], indicating that the LIS has been extended by A[i].

4. Conclusion:
After processing all elements in A, the algorithm returns the longest increasing subsequence, which is stored in B[B_length - 1]

Recursive Formula:

- Let $LIS(i)$ represent the length of the LIS ending at index i.

- The recursive formula is defined as follows:

$$LIS(i) = \max(LIS(j) + 1) \text{ for all } j \text{ where } 0 \leq j < i \text{ and } A[j] < A[i]$$
$$LIS(i) = 1 \text{ if no such } j \text{ exists}$$

Running Time Analysis:
1. Initialization:
- Initializing 'n' and other variables takes O(n) time since it requires iterating through the input array A once to set up these initial values. Specifically:
- Setting 'n' to the length of the input array A takes O(1) time.
- Initializing 'tail_elements' and 'B' as empty lists takes O(n) time.
- Setting the initial values for 'tail_elements[0]' and 'B[0]' also takes O(1) time.

2. Main Loop:
The main loop iterates through the input array A from the second element to the last element i from 1 to n-1). In each iteration of the loop, it performs the following operations:
- Checking if A[i] is smaller than the smallest tail element ('if $A[i] < tail\_elements[0]$') takes O(1) time.
- Checking if A[i] is larger than the largest tail element ('if $A[i] > tail\_elements[B\_length - 1]$') also takes O(1) time.

Binary Search:
- The binary search operation to find the position to insert the current number in the 'tail_elements' array takes O(log(n)) time. This is because it divides the search space in half in each iteration.

Array Copying:
- Copying the 'B' array in the case of extending the LIS ('B[left] = B[left - 1][:]') takes O(n) time in the worst case, where n is the length of the current LIS. This is because it involves copying all the elements in the LIS to create a new LIS.

Overall, the main loop runs for (n-1) iterations, and within each iteration, the binary search and array copying operations are the most significant in terms of time complexity.

3. Final Result: - At the end of the algorithm, the code returns the longest increasing subsequence 'B[B_length - 1]', which has been constructed during the process. Constructing this subsequence takes O(n) time since we are copying the elements of the longest subsequence.

4. Overall Time Complexity:
Combining all these steps and considering that the most significant operation inside the loop is the binary search ($O(\log(n))$) with loop running over 'n' times and array copying (O(n)), the overall time complexity of the algorithm is dominated by the binary search operation.

- In this case, while the binary search operation is ($O(\log(n))$), it is nested inside a loop that iterates through the input array, which makes the overall time complexity of the algorithm ($O(n \log(n))$).

- Therefore, the overall time complexity of the given code for finding the Longest Increasing Subsequence is ($O(n \log(n))$), making it an efficient solution for moderately large input arrays.

Longest Increasing and Decreasing Subsequence:

The INC_DEC(A) that takes an input list A and returns a subsequence of A that is both an increasing and decreasing subsequence, such that the combined subsequence is as long as possible. The code uses dynamic programming to find the longest increasing and decreasing subsequences ending at each index and then combines these subsequences to obtain the desired result. Let's break down the approach, algorithm, and analyze its running time:

Approach:
1. Initialize two arrays, inc_subseq_length and dec_subseq_length, to store the length of the longest increasing and decreasing subsequences ending at each index, respectively.
2. Initialize two arrays, prev_inc_subseq and prev_dec_subseq, to store the previous index for the increasing and decreasing subsequences, respectively.
3. Calculate the length of the longest increasing subsequence (INC) for each index using dynamic programming.
4. Calculate the length of the longest decreasing subsequence (DEC) for each index using dynamic programming.
5. Find the index with the maximum combined length of INC and DEC subsequences.
6. Reconstruct the INC subsequence starting from this index by following the prev_inc_subseq array.
7. Reconstruct the DEC subsequence starting from this index by following the prev_dec_subseq array.
8. Combine the INC and DEC subsequences to obtain the final result.

Algorithm Explanation:
1. Initialize several arrays to keep track of information about subsequences:
inc_subseq_length: An array to store the length of the longest increasing subsequence ending at each index.
dec_subseq_length: An array to store the length of the longest decreasing subsequence ending at each index.
prev_inc_subseq: An array to store the previous index for the increasing subsequence.
prev_dec_subseq: An array to store the previous index for the decreasing subsequence.

2. Initialize variables to keep track of the maximum combined length and the index where it occurs:
max_len: Initialized to 0.
max_len_idx: Initialized to -1.

3. Compute the length of the longest increasing subsequence (INC) ending at each index:
For each index i from 1 to arr_length $- 1$, iterate over all indices j from 0 to $i - 1$.
If $A[i] > A[j]$ and inc_subseq_length[i] is less than inc_subseq_length[j] $+ 1$, update inc_subseq_length[i] and prev_inc_subseq[i] accordingly.

4. Compute the length of the longest decreasing subsequence (DEC) ending at each index:
For each index i from arr_length $- 2$ down to 0, iterate over all indices j from arr_length $- 1$ down to $i + 1$.
If $A[i] > A[j]$ and dec_subseq_length[i] is less than dec_subseq_length[j] $+ 1$, update dec_subseq_length[i] and prev_dec_subseq[i] accordingly.

5. Find the index with the maximum combined length of INC and DEC subsequences:
For each index i from 0 to arr_length $- 1$, calculate inc_subseq_length[i] $+$ dec_subseq_length[i] $- 1$ and update max_len and max_len_idx if this value is greater than the current maximum.

6. Reconstruct the INC subsequence starting from max_len_idx by following the prev_inc_subseq array:
Start from max_len_idx and repeatedly add elements from A to the INC subsequence while moving to the previous index indicated by prev_inc_subseq. This creates the increasing subsequence.

7. Reconstruct the DEC subsequence starting from max_len_idx by following the prev_dec_subseq

array:
Start from max_len_idx and repeatedly add elements from A to the DEC subsequence while moving to the previous index indicated by prev_dec_subseq. This creates the decreasing subsequence.

8. Combine the INC and DEC subsequences to obtain the final result:
Concatenate the INC subsequence (in reverse order) and the DEC subsequence (excluding the first element) to create a subsequence that is both increasing and decreasing.

Return the final combined subsequence as the result.

Recursive Formula:

- The recursive formulas for the dynamic programming are as follows:
- For the longest increasing subsequence (INC):

$$\text{inc\_subseq\_length}[i] = \max(\text{inc\_subseq\_length}[i], \text{inc\_subseq\_length}[j] + 1), \\ \text{for all } j \text{ where } 0 \leq j < i \text{ and } A[i] > A[j]$$

- For the longest decreasing subsequence (DEC):

$$\text{dec\_subseq\_length}[i] = \max(\text{dec\_subseq\_length}[i], \text{dec\_subseq\_length}[j] + 1), \\ \text{for all } j \text{ where } \text{arr\_length} - 1 \geq j > i \text{ and } A[i] > A[j]$$

Running Time Analysis:
1. Initializing Arrays:
- The code initializes several arrays of length arr_length. This takes $O(\text{arr\_length}^2) \approx O(n^2)$ time.

2. Longest Increasing Subsequence (INC):
- The code uses two nested loops to compute the length of the longest increasing subsequence:
- The outer loop iterates from 1 to n − 1.
- The inner loop iterates from 0 to $i - 1$ for each iteration of the outer loop.
- In the inner loop, comparisons and updates are performed.
- This part results in a time complexity of $O(\text{n}^2)$ since it has two nested loops iterating over the input.

3. Longest Decreasing Subsequence (DEC):
- Similar to the INC subsequence, the code uses two nested loops to compute the length of the longest decreasing subsequence:
- The outer loop iterates from n − 2 down to 0.
- The inner loop iterates from n − 1 down to $i + 1$ for each iteration of the outer loop.
- In the inner loop, comparisons and updates are performed.
- This part also results in a time complexity of $O(\text{n}^2)$ since it has two nested loops iterating over the input.

4. Finding the Maximum Length:
- After computing INC and DEC subsequences, the code uses a loop that iterates from 0 to n − 1 to find the index with the maximum combined length of INC and DEC subsequences.
- This loop runs in $O(\text{n})$ time as it iterates through the entire array once.

5. Reconstructing the INC and DEC Subsequences:
- Reconstructing both the INC and DEC subsequences involves following the prev_inc_subseq and prev_dec_subseq arrays from the index found in step 4.
- Each reconstruction runs in $O(\text{n})$ time because it follows a sequence of indices from the maximum length index to previous indices.

6. Combining the INC and DEC Subsequences:
- Combining the INC and DEC subsequences involves concatenating them, which takes $O(\mathrm{n})$ time.

7. Overall Time Complexity:
- The dominant time complexity comes from the INC and DEC calculations, which are both $O(\mathrm{n}^2)$.
- The overall time complexity of the code is $O(\mathrm{n}^2)$ due to the nested loops for calculating the subsequences.

In summary, the code has a time complexity of $O(\mathrm{n}^2)$ due to the dynamic programming approach used to calculate the longest increasing and decreasing subsequences.

Note:
Test cases provided have been modified (added other possible answers) in order to consider multiple solutions that are possible for a given array.

References:
https://leetcode.com/problems/longest-increasing-subsequence/
https://interviewbit.com/blog/longest-increasing-subsequence/
https://cp-algorithms.com/sequences/longest_increasing_subsequence.html
https://takeuforward.org/data-structure/longest-bitonic-subsequence-dp-46/
https://www.codingninjas.com/studio/library/longest-bitonic-subsequence