354. Russian Doll Envelopes

Hard Array Binary Search Dynamic Programming Sorting

Leetcode Link

Problem Description

In this problem, we are given a list of `envelopes`, each with its width and height represented as a pair of integers in a 2D array. The primary goal is to figure out how many unique envelopes can be nested inside one another, one at a time, in what's known as a Russian doll fashion. An envelope can only fit into another one if both its width and height are strictly smaller than those of the envelope it is being fit into. Rotating envelopes is not allowed; we have to consider the width and height in the order they are given. The challenge lies in maximizing the sequence of envelopes that can be nested like this.

Intuition

When we face a problem that asks for the 'maximum number of something', a common approach is to think of it as a dynamic programming problem. However, naive dynamic programming will not be sufficient due to the complexity of nested conditions.

A smarter approach involves sorting and then applying a variation of the longest increasing subsequence (LIS) problem. By sorting the envelopes properly, we ensure that once we are considering an envelope for nesting, all potential outer envelopes have already been considered.

Here are the detailed steps of the intuition and approach:

1. Sort the `envelopes` array primarily by width (`a`) and, in the case where widths are equal, by height (`b`) in descending order. This might seem counter-intuitive at first because for LIS problems we generally sort in ascending order.
   - The idea behind the custom sorting is that when widths are the same, sorting heights in descending order ensures that they cannot be placed in the same increasing subsequence, which is essential since we can't nest envelopes with the same width.
2. We then seek to find the LIS based solely on the heights of our sorted envelopes because we know all widths are increasing in the array due to our sort. So if we find an increasing sequence by height, we have automatically found increasing widths as well.
3. To implement this using efficient time complexity, we use a binary search with an auxiliary array `d` (usually termed `tails` in LIS problems) where we store the last element of the currently known increasing subsequences of various lengths.
4. Iterating through the sorted envelopes, for each height, we determine where it would fit in our `d` array:
   - If it's larger than the largest element in `d`, this height can extend the longest increasing subsequence, so we append it to `d`.
   - Otherwise, we use binary search to find the smallest element in `d` that is larger than or equal to the current height, and replace it with the current height, thereby extending the longest increasing subsequence possible at this point (while discarding subsequences that were not the shortest possible).
5. After processing all envelopes, the length of `d` represents the length of the longest subsequence of widths and heights that meet the nesting condition, which is the desired result of the problem.

The key insight is that the sorting step simplifies the problem for us, reducing it to an LIS problem that can be solved in (O(N\log N)) time.

Solution Approach

The implementation of the solution follows a series of steps which involves sorting, a binary search algorithm, and dynamic programming concepts to efficiently solve the problem.

Firstly, let's discuss the data structure used. The array `d` plays a crucial role in the implementation. It stores the heights of the envelopes in a way that they form an increasing sequence. This array represents the end elements of the currently known increasing subsequences of varying lengths.

Now, let's walk through each part of the algorithm:

1. Sort the `envelopes` array by width in ascending order, and in the case of a tie on the width, sort by height in descending order. This is done using the `sort` method with a custom lambda function as the key:

```
envelopes.sort(key=lambda x: (x[0], -x[1]))
```

   Here, `x[0]` represents the width and `x[1]` represents the height of each envelope.

2. Initialize the array `d` with the height of the first envelope because at least one envelope (the smallest) can be Russian dolled thus far:

```
d = [envelopes[0][1]]
```

3. Iterate over the sorted envelopes (ignoring the first one since it's already in `d`), and apply a binary search to find the correct position of each envelope's height in `d`:

```
for _, h in envelopes[1:]:
    if h > d[-1]:
        d.append(h)
    else:
        idx = bisect_left(d, h)
        d[idx] = h
```

   - We use the `bisect_left` method from the `bisect` module in Python, which implements binary search in an efficient manner. The `bisect_left` function finds the index at which the given height `h` can be inserted to maintain the ordered sequence.
   - If the height `h` of the current envelope is greater than the last element in `d`, it can form a new, longer increasing subsequence by being placed on top of the previous sequence, so it is appended to `d`.
   - Otherwise, we replace the found position in `d` with the current height `h`. This ensures the subsequence remains an increasing one, and we maintain the smallest possible heights in `d`, which opens up opportunities for future envelopes to be placed in this increasing subsequence.

4. After processing all envelopes, we return the length of `d`:

```
return len(d)
```

   The length of `d` represents the maximum number of envelopes that can be Russian dolled, because it symbolizes the longest increasing subsequence that we have been able to construct out of the heights while maintaining the sorted order of the widths.

This algorithm is an elegant combination of sorting, dynamic programming, and binary search, resulting in a solution with a time complexity of (O(N\log N)), where (N) is the number of envelopes.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach outlined above. Suppose we are given the following 2D array of `envelopes`:

```
envelopes = [[5,4],[6,4],[6,7],[2,3]]
```

This array represents the following envelopes with width and height:

1. Envelope 1: width=5, height=4
2. Envelope 2: width=6, height=4
3. Envelope 3: width=6, height=7
4. Envelope 4: width=2, height=3

Now, let's apply the steps of the solution approach:

1. **Sort the `envelopes` array** by width in ascending order and by height in descending order within widths:

   After sorting, the `envelopes` array looks like this:

```
sorted_envelopes = [[2,3],[5,4],[6,7],[6,4]]
```

   Envelopes with width=6 have been sorted by their heights in descending order.

2. **Initialize the array `d`** with the height of the first envelope:

```
d = [3]
```

3. **Iterate over the sorted envelopes** and apply a binary search to find the correct position of each envelope's height in `d`:

   - For envelope [5,4]:

     Since the height 4 is greater than the last element in `d` (3), we append it to `d`:

```
d = [3,4]
```

   - For envelope [6,7]:

     Again, the height 7 is greater than the last element in `d` (4), so we append it:

```
d = [3,4,7]
```

   - For envelope [6,4]:

     The height 4 is not greater than the last element in (7). We use binary search to find the correct position of 4 in `d`, which is at index 1, replacing the element at that position:

```
d = [3,4,7]
```

     Note that this step maintains the longest increasing subsequence possible and does not change our `d` in this case because 4 is already present.

4. **After processing all envelopes**, we find that the length of `d` is 3:

```
len(d) = 3
```

This length indicates that the maximum number of envelopes that can be Russian doll is 3, corresponding to the following sequence of envelopes: [2,3], [5,4], [6,7]. Thus, by applying the smart sorting technique followed by finding the longest increasing subsequence on the heights, we have effectively solved the problem in (O(N\log N)) time.

Python Solution

```
from bisect import bisect_left
from typing import List

class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> int:
        # Sort the envelopes by width in increasing order and then by height in decreasing order
        envelopes.sort(key=lambda x: (x[0], -x[1]))

        # Initialize a list to store the increasing heights sequence
        dp = [envelopes[0][1]]

        # Iterate through the sorted envelopes starting from the second one
        for _, height in envelopes[1:]:
            # If the current height is greater than the last in dp
            if height > dp[-1]:
                dp.append(height) # Append it as it forms an increasing sequence
            else:
                # Find the index to replace with the current height to keep sequence increasing
                index = bisect_left(dp, height)
                # Ensure the index is within range, replace the current height
                dp[index] = height

        # The length of dp is the length of the longest increasing subsequence
        return len(dp)
```

Java Solution

```
import java.util.Arrays;

class Solution {
    public int maxEnvelopes(int[][] envelopes) {
        // Sort envelopes by width in ascending order; if widths are equal, sort by height in descending order.
        Arrays.sort(envelopes, (a, b) -> a[0] == b[0] ? b[1] - a[1] : a[0] - b[0]);

        int numEnvelopes = envelopes.length;
        // 'heights' array stores the heights at which envelopes will end at each position in the subsequence.
        int[] heights = new int[numEnvelopes + 1];
        // Initialize the first height in the 'heights' array
        heights[0] = envelopes[0][1];
        // 'maxSize' represents the length of the longest increasing subsequence so far.
        int maxSize = 1;

        for (int i = 1; i < numEnvelopes; ++i) {
            int currentHeight = envelopes[i][1];
            // If the current envelope's height is greater than the last envelope's height in the 'heights' array,
            // it can be appended to the subsequence.
            if (currentHeight > heights[maxSize]) {
                heights[++maxSize] = currentHeight;
            } else {
                // If further binary search is the correct position to place current envelope's height.
                int left = 1, right = maxSize;
                while (left < right) {
                    int mid = (left + right) >> 1; // Equivalent to (left + right) / 2
                    if (heights[mid] >= currentHeight) {
                        right = mid;
                    } else {
                        left = mid + 1;
                    }
                }
                // Update the 'heights' array with the current envelope's height at the correct position.
                int pos = heights[left] >= currentHeight ? left : 1;
                heights[pos] = currentHeight;
            }
        }
        // Return the length of the longest increasing subsequence which corresponds to the maximum number of envelopes.
        return maxSize;
    }
}
```

C++ Solution

```
class Solution {
public:
    int maxEnvelopes(vector<vector<int>>& envelopes) {
        // Sort envelopes by width; if the same, then by height in decreasing order
        sort(envelopes.begin(), envelopes.end(), [](const vector<int>& env1, const vector<int>& env2) {
            return env1[0] < env2[0] || (env1[0] == env2[0] && env1[1] > env2[1]);
        });

        // Variable to store the number of envelopes
        int numEnvelopes = envelopes.size();

        // Dynamic Programming vector to store the increasing heights
        vector<int> heightSequence{ envelopes[0][1] };

        // Process each envelope
        for (int i = 1; i < numEnvelopes; ++i) {
            int currentHeight = envelopes[i][1];
            // If current envelope's height is greater than the last height in the sequence
            if (currentHeight > heightSequence.back()) {
                heightSequence.push_back(currentHeight);
            } else {
                // Find the first element in the sequence which is not less than currentHeight
                *lower_bound(heightSequence.begin(), heightSequence.end(), currentHeight) =
                    currentHeight; // Update the current element with the currentHeight
            }
        }

        // Return the size of the longest increasing height sequence
        return heightSequence.size();
    }
};
```

Typescript Solution

```
// Function to compare two envelopes: sort by width, then by reverse height if widths are equal
function compareEnvelopes(env1: number[], env2: number[]): boolean {
    return env1[0] < env2[0] || (env1[0] === env2[0] && env1[1] > env2[1]);
}

// Perform a binary search and return the index of the first element in a sorted array
// which does not compare less than the target height.
function binarySearch(sequence: number[], targetHeight: number): number {
    let start = 0,
        end = sequence.length - 1;
    while (start <= end) {
        let mid = Math.floor((start + end) / 2);
        if (sequence[mid] < targetHeight) {
            start = mid + 1;
        } else {
            end = mid - 1;
        }
    }
    // Return start, // The insertion point for the targetHeight
}

// Envelope sorting and processing to find the maximum envelopes that can be nested
function maxEnvelopes(envelopes: number[][]): number {
    // Sort the envelopes based on specified comparison logic
    envelopes.sort((a, b) => compareEnvelopes(a, b) ? -1 : 1);

    // Initialize the array to store the maximum increasing sequence of heights
    const heightSequence: number[] = [envelopes[0][1]];

    // Process each envelope starting from the second one
    for (let i = 1; i < envelopes.length; ++i) {
        const currentHeight = envelopes[i][1];

        // If the current envelope's height is greater than the last element in the sequence
        if (currentHeight > heightSequence[heightSequence.length - 1]) {
            // Append the current height to the sequence
            heightSequence.push(currentHeight);
        } else {
            // Find the index to replace with the current envelope's height
            const indexToReplace = binarySearch(heightSequence, currentHeight);
            heightSequence[indexToReplace] = currentHeight;
        }
    }

    // Return the length of the longest increasing subsequence of heights
    return heightSequence.length;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the `maxEnvelopes` function is determined by several factors:

1. Sorting the envelopes requires $O(N * \log(N))$ time, where N is the number of envelopes.
2. The for loop iterates through the sorted envelopes list, which results in $O(N)$ iterations.
3. Inside the for loop, a binary search is performed using `bisect_left`, which takes $O(\log(N))$ time in the worst case for each iteration.

Multiplying the number of iterations (the for loop) by the complexity of each iteration (the binary search) gives us a total for the loop of $O(N * \log(N))$.

Adding the sorting and iteration parts, the final time complexity remains $O(N * \log(N))$, since both have the same asymptotic behavior.

Space Complexity

The space complexity of the function is determined by the storage used for the list `d`.

1. The list `d` is dynamically expanded based on the sequence of heights we can include in the increasing subsequence. In the worst case, it can contain all the heights from the envelopes, which would require $O(N)$ space.
2. The envelopes list itself takes $O(N)$ space.

Therefore, the overall space complexity is $O(N)$ because it's the largest space requirement.