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

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.

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.

Here are the detailed steps of the intuition and approach: 1. Sort the envelopes array primarily by width (w) and, in the case where widths are equal, by height (h) in descending order. This might seem counter-intuitive at first because for LIS problems we generally sort in ascending order.

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 idea behind the custom sorting is that when the widths are the same, sorting heights in descending order ensures that

- 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))

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

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.

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

1 for _, h in envelopes[1:]:

d.append(h)

d[idx] = h

idx = bisect_left(d, h)

complexity of (O(N\log N)), where (N) is the number of envelopes.

This array represents the following envelopes with width and height:

if h > d[-1]:

else:

Example Walkthrough

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

3. Envelope 3: width=6, height=7

4. Envelope 4: width=2, height=3

envelopes:

1 d = [3]

1 d = [3,4]

For envelope [6,7]:

1 d = [3,4,7]

1 d = [3,4,7]

Python Solution

class Solution:

10

11

12

13

14

16

17

18

19

20

21

22

23

24

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

40

42

41 }

from bisect import bisect_left

dp = [envelopes[0][1]]

from typing import List

4 is already present.

For envelope [6,4]:

far:

subsequences of varying lengths.

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

time.

Here, \times [0] represents the width and \times [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

This is done using the sort method with a custom lambda function as the key:

```
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:
```

 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

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

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

```
this increasing subsequence.
4. After processing all envelopes, we return the length of d:
  1 return len(d)
  The length of d represents the maximum number of envelopes that can be Russian dolled, because it symbolizes the longest
```

by being placed on top of the previous sequence, so it is appended to d.

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

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

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:

```
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]:
```

at index 1, replacing the element at that position:

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

After sorting, the envelopes array looks like this:

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

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

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

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

The height 4 is not greater than the last element in d (7). We use binary search to find the correct position of 4 in d, which is

Note that this step maintains the longest increasing subsequence possible and does not change our d in this case because

This length indicates that the maximum number of envelopes that can be Russian dolled is 3, corresponding to the following

Sort the envelopes by width in increasing order and then by height in decreasing order

sequence of envelopes: [[2,3], [5,4], [6,7]]. Thus, by applying the smart sorting technique followed by finding the longest increasing

```
4. After processing all envelopes, we find that the length of d is 3:
   1 \operatorname{len}(d) = 3
```

subsequence on the heights, we have effectively solved the problem in (O(N\log N)) time.

def maxEnvelopes(self, envelopes: List[List[int]]) -> int:

Initialize a list to store the increasing heights sequence

Iterate through the sorted envelopes starting from the second one

Ensure the index is within range, replace the current height

// 'maxSize' represents the length of the longest increasing subsequence so far.

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

int numEnvelopes = envelopes.length;

heights[1] = envelopes[0][1];

int maxSize = 1;

} else {

return maxSize;

int[] heights = new int[numEnvelopes + 1];

for (int i = 1; i < numEnvelopes; ++i) {</pre>

int currentHeight = envelopes[i][1];

// it can be appended to the subsequence.

heights[++maxSize] = currentHeight;

if (heights[mid] >= currentHeight) {

int pos = (heights[left] >= currentHeight) ? left : 1;

// Sort envelopes by width; if the same, then by height in decreasing order

return env1[0] < env2[0] || (env1[0] == env2[0] && env1[1] > env2[1]);

sort(envelopes.begin(), envelopes.end(), [](const vector<int>& env1, const vector<int>& env2) {

if (currentHeight > heights[maxSize]) {

int left = 1, right = maxSize;

right = mid;

left = mid + 1;

heights[pos] = currentHeight;

int maxEnvelopes(vector<vector<int>>& envelopes) {

return start; // The insertion point for the targetHeight

// Sort the envelopes based on specified comparison logic

envelopes.sort((a, b) => compareEnvelopes(a, b) ? -1 : 1);

// Append the current height to the sequence

function maxEnvelopes(envelopes: number[][]): number {

for (let i = 1; i < envelopes.length; ++i) {</pre>

const currentHeight = envelopes[i][1];

const heightSequence: number[] = [envelopes[0][1]];

heightSequence.push(currentHeight);

// Process each envelope starting from the second one

// Envelope sorting and processing to find the maximum envelopes that can be nested

// Initialize the array to store the maximum increasing sequence of heights

if (currentHeight > heightSequence[heightSequence.length - 1]) {

// Return the length of the longest increasing subsequence of heights

// If the current envelope's height is greater than the last element in the sequence

while (left < right) {</pre>

} else {

// Initialize the first height in the 'heights' array

```
for _, height in envelopes[1:]:
   if height > dp[-1]: # If the current height is greater than the last in dp
       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)
```

```
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) \rightarrow a[0] == b[0] ? b[1] - a[1] : a[0] - b[0]);
```

// 'heights' array stores the heights at which envelopes will end at each position in the subsequence.

// If the current envelope's height is greater than the last envelope's height in the 'heights' array,

// Perform binary search to find the correct position to place current envelope's height.

// Update the 'heights' array with the current envelope's height at the correct position.

// Return the length of the longest increasing subsequence which corresponds to the maximum number of envelopes.

int mid = (left + right) >> 1; // Equivalent to (left + right) / 2

C++ Solution class Solution {

});

public:

```
// Variable to store the number of envelopes
           int numEnvelopes = envelopes.size();
10
11
12
           // Dynamic Programming vector to store the increasing heights
           vector<int> heightSequence{ envelopes[0][1] };
13
14
           // Process each envelope
15
           for (int i = 1; i < numEnvelopes; ++i) {</pre>
16
               int currentHeight = envelopes[i][1];
               // If current envelope's height is greater than the last height in the sequence
               if (currentHeight > heightSequence.back()) {
19
20
                   // Add the height to the sequence
21
                   heightSequence.push_back(currentHeight);
22
               } else {
                   // Find the first element in the sequence which is not less than currentHeight
                   auto it = lower_bound(heightSequence.begin(), heightSequence.end(), currentHeight);
24
25
                   // Update the sequence element with the currentHeight
26
                   *it = currentHeight;
27
28
           // Return the size of the longest increasing height sequence
30
           return heightSequence.size();
31
32 };
33
Typescript Solution
  1 // Function to compare two envelopes; sort by width, then by reverse height if widths are equal
  2 function compareEnvelopes(env1: number[], env2: number[]): boolean {
         return env1[0] < env2[0] || (env1[0] === env2[0] && env1[1] > env2[1]);
  4 }
  5
  6 // Perform a binary search and return the index of the first element in a sorted array
  7 // which does not compare less than the target height.
     function binarySearch(sequence: number[], targetHeight: number): number {
         let start = 0;
         let end = sequence.length - 1;
 10
         while (start <= end) {
 11
 12
             let mid = Math.floor((start + end) / 2);
 13
             if (sequence[mid] < targetHeight) {</pre>
 14
                 start = mid + 1;
 15
             } else {
 16
                 end = mid - 1;
 17
```

39 // Find the index to replace with the current envelope's height 40 const indexToReplace = binarySearch(heightSequence, currentHeight); heightSequence[indexToReplace] = currentHeight; 41 42 43 44

return heightSequence.length;

Time and Space Complexity

18

19

21

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

45

46

48

47 }

20 }

Time Complexity

- 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.
- 1. The list d is dynamically expanded based on the sequence of heights we can include in the increasing subsequence. In the worst case, d can contain all the heights from the envelopes, which would require O(N) space.
- 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 0(log(N)) time in the worst case for each
 - Space Complexity The space complexity of the function is determined by the storage used for the list d.
 - 2. The envelopes list itself takes O(N) space.
 - Therefore, the overall space complexity is O(N) because it's the largest space requirement.