<https://leetcode.com/problems/russian-doll-envelopes/>

You are given a 2D array of integers envelopes where envelopes[i] = [wi, hi] represents the width and the height of an envelope.

One envelope can fit into another if and only if both the width and height of one envelope are **greater than** the other envelope's width and height.

Return *the maximum number of envelopes you can Russian doll (i.e., put one inside the other)*.

**Note:** You cannot rotate an envelope.

**Example 1:**

Input: envelopes = [[5,4],[6,4],[6,7],[2,3]]

Output: 3

Explanation: The maximum number of envelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7]).

**Example 2:**

Input: envelopes = [[1,1],[1,1],[1,1]]

Output: 1

**Constraints:**

1 <= envelopes.length <= 10^5

envelopes[i].length == 2

1 <= wi, hi <= 10^5

**Attempt 1: 2023-04-08**

**Solution 1: DP (30 min, global 'max' update more strict than L300.Longest Increasing Subsequence, TLE)**

class Solution {

    public int maxEnvelopes(int[][] envelopes) {

        Arrays.sort(envelopes, (a, b) -> a[0] != b[0] ? a[0] - b[0] : a[1] - b[1]);

        int len = envelopes.length;

        int[] dp = new int[len];

        Arrays.fill(dp, 1);

        int max = Integer.MIN\_VALUE;

        for(int i = 0; i < len; i++) {

            for(int j = 0; j < i; j++) {

                if(envelopes[i][0] > envelopes[j][0] && envelopes[i][1] > envelopes[j][1]) {

                    dp[i] = Math.max(dp[i], dp[j] + 1);

                    // Not same as L300.Longest Increasing Subsequence,

                    // we cannot update global 'max' in inside for loop,

                    // have to update in ourside for loop, because have

                    // to handle test case like:

                    // int[][] envelopes = {{1,1},{1,1},{1,1}} when

                    // no envelopes[i] strictly surround envelopes[j] exist

                    //max = Math.max(max, dp[i]);

                }

            }

            max = Math.max(max, dp[i]);

        }

        return max;

    }

}

Time Complexity : O(N^2)

Space Complexity : O(N)

**Refer to**

<https://leetcode.com/problems/russian-doll-envelopes/solutions/82778/two-solutions-in-c-well-explained/>

**DP**

It's quite intuitive to adopt DP to solve this problem:

sorting the envelopes first via its first value (width)

allocating an array to record the maximal amount for each envelope (the maximal amount we can get ending with the current envelope)

Directly the time cost here will be o(nlogn+n^2) which is o(n^2) and meantime taking up o(n) extra space.

int maxenvelopes(vector<pair<int, int>>& envelopes)

{

int size = envelopes.size();

if(!size) return 0;

sort(envelopes.begin(), envelopes.end());

int maxrolls[size]{0}, maxroll = 1;

maxrolls[0] = 1;

for(int i = 1; i < size; ++i)

{

maxrolls[i] = 1;

for(int j = i-1; j >= 0; --j)

if(envelopes[i].first>envelopes[j].first && envelopes[i].second>envelopes[j].second)

maxrolls[i] = max(maxrolls[i], maxrolls[j]+1);

maxroll = max(maxroll, maxrolls[i]);

}

return maxroll;

}

**Solution 2: Binary Search (60 min, width sort ascending, height sort descending, then problem convert into L300.Longest Increasing Subsequence by binary search on second dimension to find longest increasing subsequence)**

class Solution {

    public int maxEnvelopes(int[][] envelopes) {

        Arrays.sort(envelopes, (a, b) -> a[0] == b[0] ? b[1] - a[1] : a[0] - b[0]);

        List<Integer> list = new ArrayList<Integer>();

        for(int[] cur : envelopes) {

            if(list.size() == 0 || list.get(list.size() - 1) < cur[1]) {

                list.add(cur[1]);

            } else {

                int index = binarySearch(list, cur[1]);

                list.set(index, cur[1]);

            }

        }

        return list.size();

    }

    private int binarySearch(List<Integer> list, int target) {

        int start = 0;

        int end = list.size() - 1;

        while(start <= end) {

            int mid = start + (end - start) / 2;

            if(list.get(mid) >= target) {

                end = mid - 1;

            } else {

                start = mid + 1;

            }

        }

        return start;

    }

}

Time Complexity : O(NlogN)

Space Complexity : O(N)

**Why sort width as ascending and height as descending ?**

**Refer to**

<https://leetcode.com/problems/russian-doll-envelopes/solutions/2071477/c-java-python-best-explanation-with-pictures/>

**Why we need to sort?**

**In these types of problem when we are dealing with two dimensions, we need to reduce the problem from two-dimensional array into a one-dimensional array in order to improve time complexity.**

**"Sort first when things are undecided", sorting can make the data orderly, reduce the degree of confusion, and often help us to sort out our thinking. the same is true with this question. Now, after doing the correct sorting, we just need to find Longest Increasing Subsequence of that one dimensional array.**

**Now, you may be wondered what correct sorting actually is?**

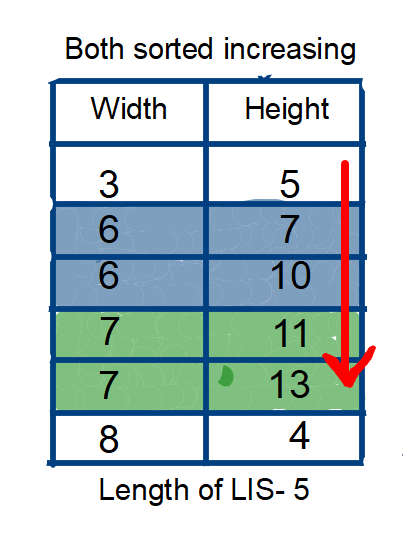
It is the sorting which we do to order to achieve the answer. Like, increasing, non-increasing sorting. Without any further discussion, let's dig into Intuition followed by algorithm.

**Algorithm**

**We sort the array in increasing order of width. And if two widths are same, we need to sort height in decreasing order.**

**Now why we need to sort in decreasing order if two widths are same. By this practice, we're assuring that no width will get counted more than one time. Let's take an example**

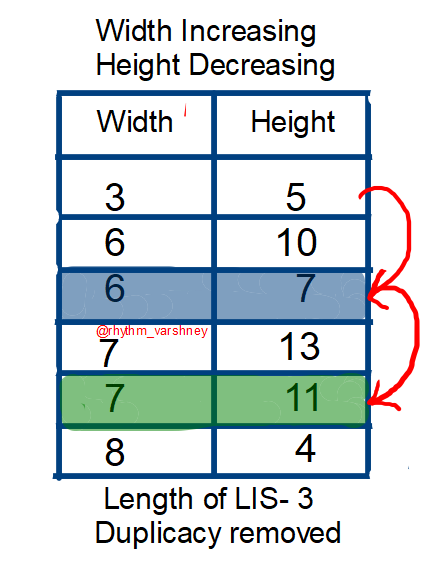
envelopes=[[3, 5], [6, 7], [7, 13], [6, 10], [8, 4], [7, 11]]



Now, if you see for a while,**6 and 7** is **counted twice** while we're calculating the length of LIS, which will give the wrong ans. **As question is asking, if any width/height are less than or equal, then, it is not possible to russian doll these envelopes.**

Now, we know the problem. So, how can we tackle these conditions when two width are same, so that it won't affect our answer. We can simple **reverse sort the height if two width are equal, to remove duplicacy.**

Now, you may question, how reverse sorting the height would remove duplicity? As the name itself says, Longest Increasing Subsequence, the next coming height would be less than the previous one. Hence, forbidding it to increase length count.



If you don't understand how LIS is calculated here, I strongly refer you to follow the prerequisite.

Now, we have **successfully reduced the problem to LIS!** All you need to apply classical LIS on heights, to calculate the ans. This would be the maximum number of envelopes can be Russians doll.

class Solution {

    public int binarySearch(int[] dp, int val){

        int lo=0,hi=dp.length-1,res=0;

        while(lo<=hi){

            int mid=(lo+hi)/2;

            if(dp[mid]<val){

                res=mid;

                lo=mid+1;

            }else{

                hi=mid-1;

            }

        }

        return res+1;

    }

    public int maxEnvelopes(int[][] envelopes) {

        Arrays.sort(envelopes,(a,b)->a[0]==b[0]?b[1]-a[1]:a[0]-b[0]);

        int[] LIS=new int[envelopes.length+1];

        Arrays.fill(LIS,Integer.MAX\_VALUE);

        LIS[0]=Integer.MIN\_VALUE;

        int ans=0;

        for(int i=0;i<envelopes.length;i++){

            int val=envelopes[i][1];

            int insertIndex=binarySearch(LIS,val);

            ans=Math.max(ans,insertIndex);

            if(LIS[insertIndex]>=val){

                LIS[insertIndex]=val;

            }

        }

        return ans;

    }

}

Now, if you compare the code of this problem with the classical LIS, it is very similar.

In fact, we have added only one line to get the maximum Russian Doll.

Arrays.sort(envelopes,(a,b)->a[0]==b[0]?b[1]-a[1]:a[0]-b[0]);

Time Complexity- **O(nlogn)**

Space Complexity- **O(n)**

**Find longest increasing subsequence on the second dimension exactly same way as L300.Longest Increasing Subsequence**

**Refer to**

**L300. Longest Increasing Subsequence Binary Search Solution**

class Solution {

    public int lengthOfLIS(int[] nums) {

        List<Integer> list = new ArrayList<Integer>();

        for(int cur : nums) {

            if(list.size() == 0 || list.get(list.size() - 1) < cur) {

                list.add(cur);

            } else {

                int index = binarySearch(list, cur);

                list.set(index, cur);

            }

        }

        return list.size();

    }

    // Find the index of first element larger than 'target'

    private int binarySearch(List<Integer> list, int target) {

        int start = 0;

        int end = list.size() - 1;

        while(start <= end) {

            int mid = start + (end - start) / 2;

            if(list.get(mid) >= target) {

                end = mid - 1;

            } else {

                start = mid + 1;

            }

        }

        return start;

    }

}

**Refer to**

<https://leetcode.com/problems/russian-doll-envelopes/solutions/82763/java-nlogn-solution-with-explanation/comments/87032>

// binary search solution: O(nlogn)

// width is increasing, but if two widths are the same, the height is decreasing

// after sorting, all envolopes are valid 'based on width', so we just binary search based on 'heights'

// to choose 'some of them' to meet the requirement

// Ex. after sorting: (1,3), (3,5), (6,8), (6,7), (8,4), (9,5)

// transform to question find LIS: [3,5,8,7,4,5] => like '300. Longest Increasing Subsequence'

public class Solution {

    public int maxEnvelopes(int[][] envelopes) {

        if(envelopes.length < 2) return envelopes.length;

        Arrays.sort(envelopes, new EnvelopeComparator());

        int[] dp = new int[envelopes.length];

        int size = 0;

        for(int[] envelope: envelopes) {

            // binary search

            int left = 0, right = size, middle = 0;    // right = size

            while(left < right) {

                middle = left + (right - left) / 2;

                if(dp[middle] < envelope[1]) left = middle + 1;

                else right = middle;

            }

            // left is the right position to 'replace' in dp array

            dp[left] = envelope[1];

            if(left == size) size++;

        }

        return size;

    }

    class EnvelopeComparator implements Comparator<int[]> {

        public int compare(int[] e1, int[] e2) {

            return e1[0] == e2[0] ? e2[1] - e1[1] : e1[0] - e2[0];

        }

    }

}

**Refer to**

[L300.Longest Increasing Subsequence](note://20C954956F6E4AA6A3575EA3146A1FCA)