<https://leetcode.com/problems/maximum-length-of-pair-chain/description/>

You are given an array of n pairs pairs where pairs[i] = [lefti, righti] and lefti < righti.

A pair p2 = [c, d] **follows** a pair p1 = [a, b] if b < c. A **chain** of pairs can be formed in this fashion.

Return *the length longest chain which can be formed*.

You do not need to use up all the given intervals. You can select pairs in any order.

**Example 1:**

Input: pairs = [[1,2],[2,3],[3,4]]

Output: 2

Explanation: The longest chain is [1,2] -> [3,4].

**Example 2:**

Input: pairs = [[1,2],[7,8],[4,5]]

Output: 3

Explanation: The longest chain is [1,2] -> [4,5] -> [7,8].

**Constraints:**

n == pairs.length

1 <= n <= 1000

-1000 <= lefti < righti <= 1000

**Attempt 1: 2023-04-09**

**Solution 1: Sorting + DP (10 min)**

class Solution {

    public int findLongestChain(int[][] pairs) {

        int result = 0;

        Arrays.sort(pairs, (a, b) -> a[0] - b[0]);

        int len = pairs.length;

        // dp[i] be the length of the longest chain ending at pairs[i]

        int[] dp = new int[len];

        Arrays.fill(dp, 1);

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

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

                if(pairs[i][0] > pairs[j][1]) {

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

                }

            }

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

        }

        return result;

    }

}

Time Complexity: O(N^2)) where N is the length of pairs. There are two for loops, and N^2 dominates the sorting step.

Space Complexity: O(N) for sorting and to store dp.

**Refer to**

<https://leetcode.com/problems/maximum-length-of-pair-chain/editorial/>

<https://leetcode.com/problems/maximum-length-of-pair-chain/solutions/105602/easy-dp/>

**Approach #1: Dynamic Programming [Accepted]**

**Intuition**

If a chain of length k ends at some pairs[i], and pairs[i][1] < pairs[j][0], we can extend this chain to a chain of length k+1.

**Algorithm**

Sort the pairs by first coordinate, and let dp[i] be the length of the longest chain ending at pairs[i]. When i < j and pairs[i][1] < pairs[j][0], we can extend the chain, and so we have the candidate answer dp[j] = max(dp[j], dp[i] + 1).

class Solution {

    public int findLongestChain(int[][] pairs) {

        Arrays.sort(pairs, (a, b) -> a[0] - b[0]);

        int N = pairs.length;

        // dp[i] be the length of the longest chain ending at pairs[i]

        int[] dp = new int[N];

        Arrays.fill(dp, 1);

        for (int j = 1; j < N; ++j) {

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

                if (pairs[i][1] < pairs[j][0])

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

            }

        }

        int ans = 0;

        for (int x: dp) if (x > ans) ans = x;

        return ans;

    }

}

**Complexity Analysis**

Time Complexity: O(N^2)) where N is the length of pairs. There are two for loops, and N^2 dominates the sorting step.

Space Complexity: O(N) for sorting and to store dp.

**Solution 2: Sorting + Greedy + One Pass (30 min)**

class Solution {

    public int findLongestChain(int[][] pairs) {

        int count = 0;

        Arrays.sort(pairs, (a, b) -> a[1] - b[1]);

        int cur = Integer.MIN\_VALUE;

        for(int[] pair : pairs) {

            if(cur < pair[0]) {

                cur = pair[1];

                count++;

            }

        }

        return count;

    }

}

Time Complexity: O(Nlog⁡N) where N is the length of S.

The complexity comes from the sorting step, but the rest of the solution does linear work.

Space Complexity: O(N). The additional space complexity of storing cur and ans,

but sorting uses O(N) space. Depending on the implementation of the language used,

sorting can sometimes use less space.

**Refer to**

<https://leetcode.com/problems/maximum-length-of-pair-chain/editorial/>

**Approach #2: Greedy [Accepted]**

**Intuition**

We can greedily add to our chain. Choosing the next addition to be the one with the lowest second coordinate is at least better than a choice with a larger second coordinate.

**Algorithm**

Consider the pairs in increasing order of their ***second*** coordinate. We'll try to add them to our chain. If we can, by the above argument we know that it is correct to do so.

class Solution {

    public int findLongestChain(int[][] pairs) {

        Arrays.sort(pairs, (a, b) -> a[1] - b[1]);

        int cur = Integer.MIN\_VALUE, ans = 0;

        for (int[] pair: pairs) if (cur < pair[0]) {

            cur = pair[1];

            ans++;

        }

        return ans;

    }

}

**Complexity Analysis**

Time Complexity: O(Nlog⁡N) where N is the length of S. The complexity comes from the sorting step, but the rest of the solution does linear work.

Space Complexity: O(N). The additional space complexity of storing cur and ans, but sorting uses O(N) space. Depending on the implementation of the language used, sorting can sometimes use less space.

**Why no pure Binary Search like L300.Longest Increasing Subsequence or L354.Russian Doll Envelops ?**

**Note:**

**This problem could not be resolved by the similar Binary Search way implement on L300.Longest Increasing Subsequence or L354.Russian Doll Envelops, because in both L300 or L354 we just have to compare second dimension as a[1] with b[1], but in L646 we have to get relationship between two dimensions as a[1] with b[0], which Binary Search brings no more benefits than DP**

**Below is an example use DP + Binary Search**

**Refer to**

<https://leetcode.com/problems/maximum-length-of-pair-chain/editorial/comments/1220344>

In fact, the first DP approach can be speed up to O(NlogN) by using binary search.

class Solution {

    public int findLongestChain(int[][] pairs) {

        Arrays.sort(pairs, (o1, o2) -> (o1[0] - o2[0]));

        int n = pairs.length;

        int[] dp = new int[n + 1];

        dp[n - 1] = 1;  // base case

        for (int i = n - 2; i >= 0; i--) {

            // find the first pair with pairs[j][0] > pairs[i][1] using bs

            int j = find(pairs, i + 1, n, pairs[i][1]);

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

        }

        return dp[0];

    }

    private int find(int[][] pairs, int from, int to, int val) {

        int left = from, right = to - 1;

        while (left <= right) {

            int mid = (right + left) / 2;

            if (pairs[mid][0] > val) {

                if (mid == from || pairs[mid - 1][0] <= val)

                    return mid;

                else

                    right = mid - 1;

            } else {

                left = mid + 1;

            }

        }

        return left;

    }

}

**Why in Greedy solution sorting using the first value(a[0]-b[0]) does not work ?**

**Refer to**

<https://leetcode.com/problems/maximum-length-of-pair-chain/solutions/105602/easy-dp/comments/2030660>

Take this example: [[1,5], [1,2] , [3,4]]. Sorting with a[0] will give ans = 1 which is wrong. In range based questions, most of the time sorting by a[1] is used.

**Refer to**

<https://leetcode.com/problems/maximum-length-of-pair-chain/solutions/105602/easy-dp/comments/286102>

At the first step, we sorted the array based on ENDING value - Arrays.sort(pairs, (p1, p2) -> p1[1] - p2[1]);

the problem can be translated as how many meetings can be scheduled in ONE single meeting room

how can we decide the next interval can be added? only if there is no overlap with previous interval. so the start value, should be larger than end value in previous inerval

**Refer to**

<https://leetcode.com/problems/maximum-length-of-pair-chain/solutions/105649/earliest-deadline-first-algorithm-greedy-same-as-maximum-jobs-we-can-accomplish/>

Earliest Deadline first algorithm (greedy). Same as Maximum jobs we can accomplish.

Consider pairs as jobs, with [start time, end time],

Then the problem is converted to ask the maximum jobs we can accomplish.

**Refer to**

<https://leetcode.com/problems/maximum-length-of-pair-chain/solutions/105613/java-solution-10-lines-dp/comments/108182>

Just some thoughts on this problem.

Sorting on either the first element or the second, this problem essentially reduces to the Longest Increasing Subsequence problem and we can use DP to solve it.

However, if we want to use Greedy to solve this problem, we need to sort on the second element. Sorting on the first element will not work. For example, try (8,9) (10,11) (1,100).

My initial trial uses DFS+memorization. It is not efficient in both time and space though.

DFS + memorization:

class Solution {

public:

int findLongestChain(vector<vector<int>>& pairs) {

vector<bool> visited(pairs.size(), false);

vector<int> record(pairs.size(), 0);

int max\_depth = 1;

for(int i=0; i<pairs.size(); i++) {

visited[i]=true;

int depth = dfs(pairs,visited,i,record);

max\_depth = max(max\_depth,depth);

visited[i]=false;

}

return max\_depth;

}

private:

int max\_depth = 0;

for(int i=0; i<pairs.size(); i++) {

if(!visited[i] && pairs[i][0]>pairs[index][1]) {

visited[i] = true;

int depth = getOrUpdate(pairs,visited,i,record);

max\_depth = max(max\_depth,depth);

visited[i] = false;

}

}

return max\_depth+1;

}

if(record[index]!=0) return record[index];

return record[index] = dfs(pairs,visited,index,record);

}

};

**Refer to**

<https://algo.monster/liteproblems/646>

**Problem Description**

You are provided with an array of n pairs, with each pair formatted as [left\_i, right\_i], and it's guaranteed that left\_i < right\_i for each pair. The concept of one pair "following" another is introduced as such: a pair p2 = [c, d] is said to follow another pair p1 = [a, b] if b < c. Chains can be formed by linking pairs that follow one another. The aim is to find the length of the longest possible chain of pairs that can be built from the given array. It's important to note that there is no requirement to use all the pairs provided; you are free to choose any sequence of pairs that forms the longest possible chain.

**Intuition**

The intuition behind the problem solution is to apply a greedy algorithm. The key idea of the greedy approach is to always choose the next pair in the chain with the smallest second element that is not yet connected to the chain. This is because selecting the pair with the smallest right\_i minimizes the chance of precluding the selection of future pairs while maximizing the potential chain length.

How do we implement this strategy? First, we sort the pairs in ascending order based on their right\_i components. Sorting by the second element ensures that as we iterate through the pairs, we always have the pair with the smallest possible right\_i that could extend our current chain.

Once the pairs are sorted, we loop over them and keep track of the current end of the chain we have built (initially, we use negative infinity to ensure we can start our chain with the first pair). For each pair, we compare the current end of the chain (cur) with the start of the next pair (a). If cur < a, then the current pair can be appended to the chain, extending it by one. Then we set cur to the end of the current pair (b) and increase our chain length count (ans).

This approach guarantees we'll end up with the longest chain by always choosing pairs that extend the chain while blocking the fewest possible pairs that come after.

**Solution Approach**

The implementation of the provided solution involves the use of the greedy algorithm and is grounded in Python's list sorting capabilities. Here is a step-by-step explanation of the solution approach:

We start by sorting the pairs list. The sorting is done based on the second element of each pair (right\_i) using a lambda function as the key: sorted(pairs, key=lambda x: x[1]). This arranges the pairs in ascending order of their right\_i values, allowing us to consider the pairs that preclude the fewest future pairs first when forming the chain.

An ans variable is initialized to 0, which will eventually represent the length of the longest chain of pairs. The cur variable is set to negative infinity (-inf) to ensure that the first pair in the sorted list can be taken as the starting pair of the chain.

A for loop is used to iterate over each pair in the sorted pairs list. At each step, the loop checks whether the current pair [a, b] can follow the last pair added to the chain. Specifically, it checks if cur < a:

If cur < a, this means that the current chain (cur representing the end of the last pair in the chain) does not overlap with the starting point of the current pair (a), hence the current pair can be appended to the chain.

The cur variable is updated to the value of b of the current pair, which becomes the new end of the chain.

The ans (answer) variable is incremented by 1 since we just added a pair to our chain.

After the loop finishes, the variable ans holds the length of the longest chain that can be formed and is returned as the final result.

By utilizing this approach, each pair is added to the chain optimally, ensuring that we can move to the next possible pair without excluding too many other pairs. This implementation is efficient and demonstrates the power of the greedy algorithm in solving such problems.

**Example Walkthrough**

Consider the list of pairs: pairs = [[1, 2], [2, 3], [3, 4]].

We first sort the pairs by their second element, resulting in no change in this case since the array is already sorted by right\_i: [[1, 2], [2, 3], [3, 4]].

Initialize ans to 0. This variable keeps track of the length of the longest chain.

Initialize cur to negative infinity (-inf) to ensure that we can compare it with the first element a of the first pair.

Now we start looping through each sorted pair:

First pair is [1, 2]. We compare cur (-inf) < 1. Since this is true, we can start a chain with this pair. Therefore, we update cur to 2 (the second element of the pair) and increment ans to 1.

Next pair is [2, 3]. We compare cur (2) < 2. This is false, so we cannot include this pair in our chain — it conflicts with the previous pair [1, 2].

The last pair is [3, 4]. We compare cur (2) < 3. This is true, so we can append this pair to our chain. We update cur to 4 and increment ans to 2.

By the end of the loop, ans is 2, representing the length of the longest chain, which includes the pairs [1, 2] and [3, 4].

This example demonstrates the application of the greedy algorithm where we prioritize adding pairs to our chain based on the smallest right\_i that doesn't overlap with the current chain. The longest chain possible in this case is of length 2, which is our final result.

**Solution Implementation**

class Solution {

public int findLongestChain(int[][] pairs) {

// Sort the pairs array by the second element of each pair (i.e., end time of the interval)

Arrays.sort(pairs, Comparator.comparingInt(pair -> pair[1]));

// Initialize the count of the longest chain as 0

int longestChainLength = 0;

// Initialize 'currentEnd' to the minimum integer value

int currentEnd = Integer.MIN\_VALUE;

// Iterate through the sorted pairs

for (int[] pair : pairs) {

// If the current pair's start time is greater than 'currentEnd'

if (currentEnd < pair[0]) {

// Update 'currentEnd' to the end time of the current pair

currentEnd = pair[1];

// Increment the count of the chain as we've found a non-overlapping pair

++longestChainLength;

}

}

// Return the length of the longest chain found

return longestChainLength;

}

}

**Time and Space Complexity**

The time complexity of the given code is primarily dictated by the sorting operation. Sorting an array of pairs with a total of n pairs has a time complexity of O(n log n). The for-loop that follows has a time complexity of O(n), as it iterates through the list of pairs only once. Therefore, the overall time complexity of the algorithm is O(n log n) due to the sorting step, since O(n log n) + O(n) simplifies to O(n log n).

Additionally, the space complexity of the code is O(1) or constant space complexity, since no additional space that scales with the input size is used, and the sorting is done in-place, assuming the sorting algorithm used is space-optimized, like Timsort, which is the default sorting algorithm in Python. The only extra variables used are for the current end of the chain (cur) and the answer counter (ans), which both require a constant amount of space.

**Elaborate why in below Greedy solution we are sorting pairs based on the second element ? not the first element ?**

**Refer to chatGPT**

import java.util.Arrays;

class Solution {

public int findLongestChain(int[][] pairs) {

// Step 1: Sort pairs based on the second element

Arrays.sort(pairs, (a, b) -> Integer.compare(a[1], b[1]));

// Initialize the chain length and the end of the last pair in the chain

int chainLength = 0;

int currentEnd = Integer.MIN\_VALUE;

// Step 2: Iterate through the sorted pairs

for (int[] pair : pairs) {

// If the current pair can be added to the chain

if (pair[0] > currentEnd) {

chainLength++; // Increase the chain length

currentEnd = pair[1]; // Update the end of the last pair in the chain

}

}

// Return the maximum length of the chain

return chainLength;

}

}

In the given solution for finding the maximum length of a pair chain, we sort the pairs based on their second element (the b value in each pair (a, b)) rather than the first element. This sorting strategy is crucial for the greedy algorithm to work correctly. Let's break down why this is the case.

**Why Sort by the Second Element (b)?**

The problem requires finding the maximum number of pairs where each pair (a, b) can be followed by another pair (c, d) if and only if b < c.

**Sorting by the second element (b)** helps in ensuring that each selected pair leaves the maximum possible space for the next pair to be added to the chain. Here's why:

Maximizing the Chain:

By sorting pairs by their second element (b), the greedy approach picks the pair that ends the earliest first. This strategy ensures that the "gap" after the end of the current pair is as large as possible, thereby maximizing the chances of being able to include more pairs in the chain.

If we sorted by the first element (a), there would be no guarantee that the pair with the earliest start would also be the best choice to allow the longest chain, because a pair with a later start but an earlier end might be a better candidate for forming a longer chain.

Avoiding Overlaps:

When you pick a pair with the smallest possible end value (b), you're minimizing the chance of overlap with the next pairs you might want to add to the chain. This is because any pair that starts after b can still be included in the chain.

Sorting by the first element wouldn't help avoid overlaps as effectively, because it wouldn't directly address the issue of where one pair ends relative to the start of the next.

Greedy Choice Property:

Greedy algorithms work well when you can make a "locally optimal choice" that leads to a globally optimal solution. In this problem, the locally optimal choice is to pick the pair that ends earliest (after ensuring it can follow the previous pair in the chain). Sorting by b and picking the first valid pair ensures that each step is making the optimal choice for building the longest chain.

**An Example to Illustrate:**

Consider the following pairs:

[[1, 2], [2, 3], [3, 4]]

If we sort by the second element:

After sorting: [[1, 2], [2, 3], [3, 4]]

Now, using the greedy approach:

Start with currentEnd = -∞.

Pick [1, 2] since 1 > currentEnd, so now currentEnd = 2.

Next, pick [3, 4] since 3 > currentEnd, so now currentEnd = 4.

The maximum chain length is 2 (pairs [1, 2] -> [3, 4]).

If we were to sort by the first element:

Sorting would give us the same list, but consider different pairs:

[[5, 24], [15, 25], [27, 40], [50, 60]]

Sorting by the first element: [[5, 24], [15, 25], [27, 40], [50, 60]]

Sorting by the second element: [[5, 24], [15, 25], [27, 40], [50, 60]] (which happens to be the same, but the choice becomes critical if overlaps exist)

If we focus on the end of the intervals, sorting by b ensures that when we pick 24, we can easily continue with 27 and 50, forming a valid chain.

**Why Sorting by the First Element is Suboptimal**

Sorting by the first element would help if you wanted to start with the smallest a, but it doesn't guarantee that the subsequent pairs can be connected. It doesn't help directly with managing overlaps because a pair that starts earlier but ends later might block other pairs from being added to the chain.

**Summary**

Sorting by the second element (**b)** allows the algorithm to greedily select the next pair that leaves the maximum room for additional pairs to be added to the chain, ensuring the longest possible chain.

Sorting by the first element (**a)** does not provide the same guarantee because it doesn't effectively manage the "end" of one interval relative to the "start" of the next, leading to suboptimal chain lengths.

This strategy leverages the properties of greedy algorithms to solve the problem efficiently, ensuring that we always make the best possible local choice to achieve the global maximum chain length.

**But why in below DP solution it not require sort on second element, and sort on first element also gives correct answer ?**

class Solution {

public int findLongestChain(int[][] pairs) {

int result = 0;

Arrays.sort(pairs, (a, b) -> a[0] - b[0]);

int len = pairs.length;

// dp[i] be the length of the longest chain ending at pairs[i]

int[] dp = new int[len];

Arrays.fill(dp, 1);

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

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

if(pairs[i][0] > pairs[j][1]) {

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

}

}

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

}

return result;

}

}

The dynamic programming (DP) solution you provided works correctly even when sorting the pairs based on the first element because the DP approach inherently checks all possible valid chains, regardless of the sorting order. Here's why this happens:

**Why Sorting by the First Element Works for the DP Approach**

DP Nature:

The DP approach in this problem computes the longest chain ending at each pair i. For each pair i, it looks at all previous pairs j (where j < i) and checks if the pair j can be connected to pair i (i.e., pairs[j][1] < pairs[i][0]).

If such a connection is possible, the DP table (dp[i]) is updated with the maximum chain length that can end at pair i.

Comprehensive Checks:

In the DP solution, every pair i is checked against every previous pair j, which means that all possible combinations are explored. This ensures that even if the pairs are sorted by the first element, the algorithm still correctly finds the longest chain by examining all valid previous pairs.

Sorting by the first element ensures that when the DP loop is considering pair i, it has already considered all possible earlier pairs j that could potentially form a chain with i.

No Need for Early Termination:

Unlike the greedy approach, which tries to pick pairs optimally in a single pass and thus benefits from sorting by the second element to ensure minimal blocking, the DP approach doesn't require early termination of pair selection. It systematically evaluates all potential chains and chooses the best one.

As a result, sorting by the first element (a) is sufficient because the DP approach does not rely on the pairs ending as early as possible; it merely needs to ensure that all possible pairs are considered in the correct order to build the longest chain.

**Why Sorting by the Second Element Isn't Necessary in DP**

Sorting by the First Element (**a)**:

When sorted by the first element, the pairs are already in an order where the DP algorithm can progressively build up longer chains by considering all valid previous pairs. Since DP checks all possibilities, it doesn't miss any valid combinations that could contribute to the maximum chain length.

Efficiency:

Sorting by the first element makes sure that when you are processing pair i, any potential pairs j that could connect to i have already been processed and stored in the DP array. This ensures that when you reach the end of the array, you have considered all possible chains.

**Contrast with Greedy Approach**

Greedy Approach:

The greedy algorithm requires sorting by the second element because it makes local decisions (i.e., picking the pair that ends the earliest), which needs to be globally optimal to build the longest chain. Sorting by the first element in the greedy approach might lead to suboptimal choices because it doesn’t necessarily minimize overlap with the next pair.

DP Approach:

The DP approach, on the other hand, doesn't need to make these local decisions, so it can afford to sort by the first element. It systematically considers all possibilities, and therefore doesn't require sorting by the second element to ensure optimality.

**Summary**

Greedy Approach: Requires sorting by the second element to ensure each local choice is optimal, thereby achieving a globally optimal chain length.

DP Approach: Works with sorting by the first element because it checks all possible chains and thus doesn't need the pairs to be sorted by the end value.

The DP solution's strength lies in its ability to exhaustively explore all valid chains, making it robust to the order in which pairs are sorted, as long as they are sorted in a way that respects the constraints of the problem (i.e., checking pairs[j][1] < pairs[i][0]).

**Refer to**

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

[L354.Russian Doll Envelopes (Ref.L300)](note://0D383CA392844387B1B3AB6A7C206D42)

[L491.Increasing Subsequences (Ref.L90)](note://6519624114A34866BC5B26A6BE4E32F5)