<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: 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: 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;

}

}