646. Maximum Length of Pair Chain

Medium Greedy Array Dynamic Programming Sorting

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

Python

```python
from typing import List

class Solution:
    def findLongestChain(self, pairs: List[List[int]]) -> int:
        # Initialize the variable to store the length of the longest chain and
        # the current end value of the last pair in the chain
        longest_chain_length, current_end = 0, float('-inf')

        # Sort the pairs based on their second element since we want to ensure
        # we pick the next pair with the smallest possible end value.
        for start, end in sorted(pairs, key=lambda x: x[1]):
            # If the current start is greater than the last stored end,
            # it means we can append the current pair to the chain.
            if current_end < start:
                current_end = end  # Update the end to the current pair's end
                longest_chain_length += 1  # Increase the length of the chain

        # Return the length of the longest chain found
        return longest_chain_length
```

Java

```java
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;
    }
}
```

C++

```cpp
#include <vector> // Include vector header for using vectors
#include <climits> // Include limits header for using INT_MIN

using namespace std; // Use standard namespace

class Solution {
public:
    int findLongestChain(vector<vector<int>>& pairs) {
        // Sort the vector of pairs by the second element of each pair
        sort(pairs.begin(), pairs.end(), [](const vector<int>& a, const vector<int>& b) {
            return a[1] < b[1];
        });

        int chainLength = 0; // Initialize the length of the longest chain to 0
        int currentEnd = INT_MIN; // Initialize the end of current pair to the minimum integer value

        // Iterate over all the pairs
        for (const auto& pair : pairs) {
            // If the current pair can be chained to the previous one
            if (currentEnd < pair[0]) {
                currentEnd = pair[1]; // Update the end of current pair
                ++chainLength; // Increment the length of the chain
            }
        }

        // Return the length of the longest chain found
        return chainLength;
    }
};
```

TypeScript

```typescript
function findLongestChain(pairs: number[][]): number {
    // Sort the array of pairs based on the second element of each pair
    pairs.sort((firstPair, secondPair) => firstPair[1] - secondPair[1]);

    // Initialize the count of the longest chain
    let longestChainCount = 0;

    // Initialize the previous end element of the last pair in the chain to negative infinity
    let previousEnd = -Infinity;

    // Iterate through each pair in the sorted array
    for (const [start, end] of pairs) {
        // If the current pair's start is greater than the end of the last pair added to the chain
        if (previousEnd < start) {
            // Update the previous end to the current pair's end
            previousEnd = end;

            // Increment the count of the longest chain
            longestChainCount++;
        }
    }

    // Return the count of the longest possible chain
    return longestChainCount;
}
```

```python
from typing import List

class Solution:
    def findLongestChain(self, pairs: List[List[int]]) -> int:
        # Initialize the variable to store the length of the longest chain and
        # the current end value of the last pair in the chain
        longest_chain_length, current_end = 0, float('-inf')

        # Sort the pairs based on their second element since we want to ensure
        # we pick the next pair with the smallest possible end value.
        for start, end in sorted(pairs, key=lambda x: x[1]):
            # If the current start is greater than the last stored end,
            # it means we can append the current pair to the chain.
            if current_end < start:
                current_end = end  # Update the end to the current pair's end
                longest_chain_length += 1  # Increase the length of the chain

        # Return the length of the longest chain found
        return longest_chain_length
```

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.