757. Set Intersection Size At Least Two

Hard Overview Array Greedy Leetcode Link

Problem Description

The problem requires us to find the smallest number of elements in a set that covers every given interval in a 2D array `intervals`, with the condition that each interval must have at least two elements from the set. An interval is represented by `[start, end)`, which includes all integers between `start` and `end` inclusive.

To clarify, let's consider an example `intervals = [[1,3], [2,5], [6,9]]`: a valid containing set could be `[1, 2, 4, 6, 7]` as it has at least two numbers covered in every interval (for `[1,3]` it covers `1` and `2`, for `[2,5]` it covers `2` and `4`, and for `[6,9]` it covers `6` and `7`). The goal is to find such a containing set that has the smallest possible number of elements.

Intuition

To arrive at the solution for this problem, we need to come up with a strategy that helps us minimize the number of elements in the containing set while ensuring that each interval has at least two elements from the set.

The intuition is to always pick the last two elements of an interval whenever possible, so as to maximize the chances of those elements being within the subsequent intervals. Sorting the intervals by their ending element in ascending order and, in case of a tie, by their starting element in descending order assists with this strategy. This ordering ensures that we process intervals in a manner that's optimal for choosing common elements.

The algorithm iterates through the sorted intervals and, for each interval, determines whether new elements need to be added to the containing set (`nums`). If an interval's start is greater than the last element (`b`) added to `nums`, then the interval is disjoint from the current containing set, and two new elements (`b-1` and `b`) from this interval must be added to `nums`. If the interval's start is greater than the second-last element (`a`) but less than or equal to `a`, then the interval partially overlaps with the current containing set, and only one new element needs to be added (`b`). If an interval's start is less than or equal to `a`, it is fully covered by the current containing set, and no new elements are added.

The `ans` variable keeps track of the size of the containing set, incrementing by 2 when two elements are added and by 1 when one element is added. At the end, `ans` gives us the minimum size of a containing set that satisfies the problem's conditions.

Solution Approach

The implementation of the solution in Python makes use of a greedy algorithm to minimize the size of the containing set.

The approach starts by sorting the `intervals` list. The sorting is done based on the end value of each interval in ascending order, using the `lambda` function `key=lambda x: (x[1], -x[0])`. This means that if we have two intervals with the same ending value, the one with the larger starting value will be considered first. Sorting the intervals this way prioritizes intervals that finish earlier, and for intervals with the same end, it prioritizes the wider range, which may encompass more of the following intervals.

After sorting, the algorithm initializes two pointers `s` and `e` with a value of `-1` and a counter `ans` set to `0`. These pointers track the last two elements that have been added to the containing set.

The algorithm iterates through each interval `(a, b)` in the sorted list of `intervals`. During each iteration, there are three scenarios to consider:

1. **The interval is already covered**: If `a` (start of the current interval) is less than or equal to `s` (second-to-last element added to the containing set), the current interval is already covered by the set, and we do not need to add any more elements.
2. **Only one element of the interval is covered**: If `a` (start of the current interval) is greater than `s` but less than or equal to `e` (last element added to the containing set), this means that the current interval overlaps with the last element, but we still need one more. So we increase `ans` by `1`, and update `s` to `e` and `e` to `b` (end of the current interval) to include the last two elements of the interval.
3. **The interval is disjoint**: If `a` (start of the current interval) is greater than `e` (last element added to the containing set), then the current interval does not overlap with the elements in the set, and we must include its last two elements. Thus, we increase `ans` by `2`, and update `s` to `b-1` and `e` to `b`, adding these two elements as the new last elements of the set.

After processing all intervals, the value of `ans` represents the size of the minimum containing set that fulfills the criteria outlined in the problem, which is returned at the end of the method.

This greedy solution is efficient because at each step it makes a locally optimal choice (adding as few elements as possible to cover the current interval) that leads to a globally optimal solution (minimal size of the containing set).

Example Walkthrough

Let's illustrate the solution approach with a small example where the given `intervals` is `[[1,3], [2,4], [5,7]]`.

First, we apply the sorting strategy to the `intervals` list. The sorted intervals will be `[[1,3], [2,4], [5,7]]`. In this example, sorting doesn't change the order since the intervals are already in ascending order based on their end values.

Now, we initialize two pointers `s` and `e` to `-1` and set the `ans` counter to `0`. These variables help us track elements in the containing set and its size.

We start iterating through each interval `(a, b)` in the sorted list:

1. The first interval is `[1,3]`. Since `a` is greater than `e`, this interval isn't covered by the containing set. So we add its last two elements `(2, 3)` to the set, increment `ans` by `2`, and update `s` to `2` and `e` to `3`.
2. The set now covers `{2, 3}`, and `ans` is `2`.

Continuing with the second interval `[2,4]`:

1. Here, `a (2)` is equal to `s (2)`, so one element of this interval is already covered. We need one more element from this interval to make sure we have two. Therefore, we add `4`, the last element, increasing `ans` by `1`, and update `s` to `e (3)` and `e` to `4`.
2. The set now covers `{2, 3, 4}`, and `ans` is `3`.

Next, we move on to the third interval `[5,7]`:

1. For this interval, `a (5)` is greater than `e (4)`, signifying that the interval is disjoint from the set. We add the last two elements `(6, 7)` from this interval into the set, increment `ans` by `2`, and update `s` to `6` and `e` to `7`.
2. The set now covers `{2, 3, 4, 6, 7}`, and `ans` is `5`.

Having processed all intervals, we can see that the smallest containing set that covers every given interval with at least two elements is `{2, 3, 4, 6, 7}` with a size of `5`.

This walkthrough demonstrates how the greedy algorithm works step by step, adding as few elements as possible while ensuring each interval is covered by at least two elements from the set. The size of the final set, represented by `ans`, gives us the minimum number of elements required to satisfy the problem criteria.

Python Solution

```python
from typing import List

class Solution:
    def intersectionSizeTwo(self, intervals: List[List[int]]) -> int:
        # Sort the intervals based on their ending values (primary) and
        # starting values (secondary, in descending order).
        intervals.sort(key=lambda x: (x[1], -x[0]))

        # Initialize 'start' & 'end' to keep track of the last two elements
        # in the current intersection set.
        start = end = -1
        # Initialize 'ans' to store the total number of elements in the intersection set.
        ans = 0

        # Iterate through each interval.
        for current_start, current_end in intervals:
            # If the current interval's start is less than or equal to 'start',
            # it is already covered by the intersection set and we can skip it.
            if current_start <= start:
                continue

            # If the current interval's start is greater than 'end',
            # we need to add two new elements to cover this interval.
            if current_start > end:
                # Update 'start' and 'end' to the last two elements of this interval.
                start, end = current_end - 1, current_end
            else:
                # If the current interval's start is within '(start, end)',
                # we need to add only one new element to cover this interval.
                ans += 1
                # Update 'start' to 'end' and 'end' to the current interval's end.
                start, end = end, current_end

        # Return the total number of elements in the intersection set.
        return ans

# The solution class can now be used to instantiate an object and call the 'intersectionSizeTwo' method.
# Here is how you can use it:

sol = Solution()
results = sol.intersectionSizeTwo([[1, 3], [4, 9], [8, 10]])
print(results) # Example usage; should print out the result of the method call
```

Java Solution

```java
class Solution {
    public int intersectionSizeTwo(int[][] intervals) {
        // Sort the intervals based on their end value.
        // If two intervals have the same end value, sort based on start value in descending order.
        Arrays.sort(intervals, (a, b) -> {
            if (a[1] == b[1]) {
                return b[0] - a[0];
            }
            return a[1] - b[1];
        });

        // Initialize answer as 0, and start/end as -1.
        int answer = 0;
        int start = -1, end = -1;

        // Iterate over each interval.
        for (int[] interval : intervals) {
            int intervalStart = interval[0];
            int intervalEnd = interval[1];

            // If the current interval's start is less than or equal to the current `start`,
            // it is already covered by the set of points we have.
            if (intervalStart <= start) {
                continue;
            }

            // If the current interval's start is greater than the current `end`,
            // we need to add two points to cover this interval.
            if (intervalStart > end) {
                answer += 2;

                // Update `start` and `end` to the last two elements of the interval.
                start = intervalEnd - 1;
                end = intervalEnd;
            } else {
                // Only if the current interval's start is within the range (start, end],
                // we only need to add one more point to cover this interval.
                answer += 1;

                // Shift `start` to `end` and update `end` to the end of the current interval.
                start = end;
                end = intervalEnd;
            }
        }

        // Return the minimum size of the set of points to cover all intervals.
        return answer;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to calculate the minimum size of a set so that for every
    // interval in `intervals` there are at least two distinct set elements which
    // are in the interval.
    int intersectionSizeTwo(std::vector<std::vector<int>>& intervals) {
        // Sort the intervals based on their end points. If end points are the same,
        // sort based on the start points in decreasing order. This way, we prefer
        // intervals with larger start points for same end points.
        sort(intervals.begin(), intervals.end(), [](const std::vector<int>& a, const std::vector<int>& b) {
            return a[1] == b[1] ? a[0] > b[0] : a[1] < b[1];
        });

        int ans = 0; // Initialize the answer to 0.
        int smallest = -1, secondSmallest = -1; // Initialize the two smallest elements seen so far to -1.

        // Iterate through the sorted intervals
        for (const auto& interval : intervals) {
            int start = interval[0], end = interval[1];

            // If the current start is less than or equal to smallest, this interval is already covered
            // by the elements chosen so far.
            if (start <= smallest) continue;

            // If the current start is greater than secondSmallest, we need to add two more elements to the set.
            if (start > secondSmallest) {
                ans += 2;
                // The secondSmallest is now the end of the interval minus one, and the smallest
                // is the end of the interval.
                secondSmallest = end - 1;
                smallest = end;
            } else {
                // If start is between smallest and secondSmallest, we need to add one more element.
                ans += 1;
                // The new secondSmallest becomes the smallest we had, and the new smallest becomes the end of this interval.
                secondSmallest = smallest;
                smallest = end;
            }
        }

        // Return the total number of elements added to the set.
        return ans;
    }
};
```

Typescript Solution

```typescript
// Import statements are not required in TypeScript as they were in the C++ code.
// Import statements for standard collections and algorithms are unnecessary in TypeScript.

// Function to calculate the minimum size of a set so that for every
// interval in `intervals`, there are at least two distinct set elements
// that are within the interval.
function intersectionSizeTwo(intervals: number[][]): number {
    // Sort the intervals based on their end points. If end points are the same,
    // sort by the start points in decreasing order. This ensures
    // preference is given to intervals with larger start points for the same end points.
    intervals.sort((a, b) => {
        return a[1] === b[1] ? b[0] - a[0] : a[1] - b[1];
    });

    let answer = 0; // Initialize the answer to 0.
    let smallest = -1, secondSmallest = -1; // Initialize the two smallest elements seen so far to -1.

    // Iterate through the sorted intervals
    for (const interval of intervals) {
        const [start, end] = interval;

        // If the current start is less than or equal to smallest, this interval is already covered
        // by the elements chosen so far.
        if (start <= smallest) continue;

        // If the current start is greater than secondSmallest, we need to add two more elements to the set.
        if (start > secondSmallest) {
            answer += 2;
            // The secondSmallest is now the end of the interval minus one, and the smallest
            // is the end of the interval.
            secondSmallest = end - 1;
            smallest = end;
        } else {
            // If start is between smallest and secondSmallest, we need to add one more element.
            answer += 1;
            // New secondSmallest becomes the smallest we had before, and the new smallest becomes the end of this interval.
            secondSmallest = smallest;
            smallest = end;
        }
    }

    // Return the total number of elements added to the set.
    return answer;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is dominated by the sort operation applied to the list of intervals. Sorting a list of `n` intervals has a time complexity of `O(n log n)`. After sorting, the code iterates over the sorted intervals once, which has a time complexity of `O(n)`.

Thus, the total time complexity of the code is `O(n log n) + O(n)` which simplifies to `O(n log n)`.

Space Complexity

The space complexity of the code is determined by the additional space used by the algorithm besides the input. The solution does not create any additional data structures that are dependent on the size of the input. The extra variables `s`, `e`, and `ans` use constant space.

Therefore, the space complexity of the code is `O(1)`, which means it uses a constant amount of extra space.