Set Intersection Size At Least Two

An integer interval `[a, b]` (for integers `a < b`) is a set of all consecutive integers from `a` to `b`, including `a` and `b`.

Find the minimum size of a set S such that for every integer interval A in `intervals`, the intersection of S with A has size at least 2.

Example 1:

```
Input: intervals = [[1, 3], [1, 4], [2, 5], [3, 5]]
Output: 3
Explanation:
Consider the set S = {2, 3, 4}.  For each interval, there are at least 2 elements fro
m S in the interval.
Also, there isn't a smaller size set that fulfills the above condition.
Thus, we output the size of this set, which is 3.
```

Example 2:

```
Input: intervals = [[1, 2], [2, 3], [2, 4], [4, 5]]
Output: 5
Explanation:
An example of a minimum sized set is {1, 2, 3, 4, 5}.
```

Note:

1. `intervals` will have length in range `[1, 3000]`.
2. `intervals[i]` will have length `2`, representing some integer interval.
3. `intervals[i][j]` will be an integer in `[0, 10^8]`.

Solution 1

First sort the intervals, with their starting points from low to high
Then use a stack to eliminate the intervals which fully overlap another interval.
For example, if we have [2,9] and [1,10], we can get rid of [1,10]. Because as long as we pick up two numbers in [2,9], the requirement for [1,10] can be achieved automatically.

Finally we deal with the sorted intervals one by one.
(1) If there is no number in this interval being chosen before, we pick up 2 biggest number in this interval. (the biggest number have the most possibility to be used by next interval)
(2) If there is one number in this interval being chosen before, we pick up the biggest number in this interval.
(3) If there are already two numbers in this interval being chosen before, we can skip this interval since the requirement has been fulfilled.

```
class Solution {
    public int intersectionSizeTwo(int[][] intervals) {
        Arrays.sort(intervals,(a,b)->((a[0]==b[0])?(-a[1]+b[1]):(a[0]-b[0])));
        Stack<int[]> st=new Stack<>();
        for (int[] in:intervals)
        {
            while (!st.isEmpty() && st.peek()[1]>=in[1]) st.pop();
            st.push(in);
        }
        int n=st.size();
        int[][] a=new int[n][2];
        for (int i=n-1;i>=0;i--)
        {
            a[i][0]=st.peek()[0];
            a[i][1]=st.pop()[1];
        }
        int ans=2;
        int p1=a[0][1]-1,p2=a[0][1];
        for (int i=1;i<n;i++)
        {
            boolean bo1=(p1>=a[i][0] && p1<=a[i][1]),bo2=(p2>=a[i][0] && p2<=a[i][1
]);
            if (bo1 && bo2) continue;
            if (bo2)
            {
                p1=p2;
                p2=a[i][1];
                ans++;
                continue;
            }
            p1=a[i][1]-1;
            p2=a[i][1];
            ans+=2;
        }
        return ans;
    }
}
```

written by KakaHiguain original link here

Solution 2

```
class Solution {
public:
    int intersectionSizeTwo(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end(), [](vector<int>& a, vector<int>& b)
{
            return a[1] < b[1] || (a[1] == b[1] && a[0] > b[0]);
        });
        int n = intervals.size(), ans = 0, p1 = -1, p2 = -1;
        for (int i = 0; i < n; i++) {
            // current p1, p2 works for intervals[i]
            if (intervals[i][0] <= p1) continue;
            // Neither of p1, p2 works for intervals[i]
            // replace p1, p2 by ending numbers
            if (intervals[i][0] > p2) {
                ans += 2;
                p2 = intervals[i][1];
                p1 = p2-1;
            }
            // only p2 works;
            else {
                ans++;
                p1 = p2;
                p2 = intervals[i][1];
            }
        }
        return ans;
    }
};
```

written by zestypanda original link here

Solution 3

The greedy algorithm for constructing the minimum intersection set is not easy to come up with, as it replies on two properties of the minimum intersection set. In this post, I will explain what these properties are and how to find the size of the minimum intersection set using these properties.

`I -- Definitions and notations`

To ease the explanation, we first spell out some definitions and notations that will be used in this post.

Intersection set: we define an intersection set for a given array (containing intervals) as a set `S` such that for every interval `A` in the given array, the intersection of `S` with `A` has size at least `2`.

Minimum intersection set: of all the intersection sets for a given array, let `m` be the minimum value of their sizes, then any one of the intersection sets with size `m` will be referred to as a minimum intersection set.

Notations: let `intervals` be the input array with length `n`; `intervals[0, i]` denote the subarray containing intervals of indices from `0` up to `i`; `S_i` be the minimum intersection set for the subarray `intervals[0, i]`; `m_i` be the size of `S_i`.

`II -- Properties of the minimum intersection set`

From the notations in part `I`, our goal is to figure out `m_(n-1)`, the size of `S_(n-1)`, which is the minimum intersection set for the whole input array. To apply the greedy algorithm, we need to take advantage of the following two properties of `S_i`:

1. `S_i` **does not** depend on the order of the intervals in the subarray `intervals[0, i]`.
2. `m_i` is **non-decreasing** with increasing `i`, that is, we always have `m_(i-1) <= m_i`.

The first property is straightforward from the definition of an intersection set, where the intersection requirement (i.e., `intersection size >= 2`) does not depend on the order of the intervals in the given array.

The second property comes from the fact that if `S_i` is a minimum intersection set for subarray `intervals[0, i]`, then it will also be an intersection set for subarray `intervals[0, i-1]` (but not necessarily a minimum intersection set). Since both `S_(i-1)` and `S_i` are intersection sets for the subarray `intervals[0, i-1]`, but the former is a minimum intersection set, by definition we conclude: `m_(i-1) <= m_i`.

`III -- The greedy algorithm for constructing the minimum intersection set`

The first property above suggests we are free to rearrange the intervals in the input array, which means the intervals can be sorted to our advantage. There are two choices for sorting: we can sort the intervals either with ascending start points or with ascending end points. Both will work but here we will sort them with ascending end points, and if two intervals have the same end points, the one with larger start point will come first (we want the shorter interval to be processed first).

The second property above suggests to minimize `m_i`, we need to minimize `m_(i-1)`, which in turn requires minimization of `m_(i-2)`, and so on. This is actually where the greedy idea comes from. So assume for now we have minimized `m_(i-1)`, how can we minimize `m_i`?

We know that `m_(i-1) <= m_i`, so the minimum value of `m_i` we can achieve is `m_(i-1)`. If this is the case, what does it imply? It means `S_(i-1)` is not only a minimum intersection set for the subarray `intervals[0, i-1]`, but also a minimum intersection set for the subarray `intervals[0, i]`. This is equivalent to saying that we can find two different elements from `S_(i-1)` such that both elements intersect with the interval `intervals[i]`.

But how do we find such two elements? If the intervals come in arbitrary order, we probably have to check each element in `S_(i-1)` one by one and see if it intersects with `intervals[i]`, which is rather inefficient. Fortunately, the intervals can be sorted. In our case, they are sorted in ascending order according to their end points. This means all elements in `S_(i-1)` will be no greater than the end point of the interval `intervals[i]`. This is because every element `e` in `S_(i-1)` will intersect with at least one interval in the subarray `intervals[0, i-1]` (otherwise, we can remove `e` to make `S_(i-1)` smaller without violating the intersection requirement). Without loss of generality, assume the interval intersecting with `e` has index `j`, where `0 <= j <= i - 1 < i`, then we have `e <= intervals[j][1] <= intervals[i][1]`.

Therefore we only need to check the largest two elements (denoted as `largest` and `second`) in `S_(i-1)` to see if they intersect with the interval `intervals[i]`. This is because if they don't, other elements won't either. Note this also implies that of all the minimum intersection sets for the subarray `intervals[0, i-1]`, we will choose `S_(i-1)` to be the one with its largest two elements maximized. That is, for the subarray `intervals[0, i-1]`, we not only minimize the size of the intersection set, but also maximize its largest two elements (after the size is minimized).

Checking if the largest two elements intersect with `interval[i]` is equivalent to comparing them with the start point of `interval[i]`. There are three cases here:

1. **Case 1**: both elements intersect with `intervals[i]`. For this case, we show `m_i = m_(i-1)`, and no updates are needed for the largest two elements of `S_i`.
2. **Case 2**: only the largest element intersects with `intervals[i]`. For this case, we show `m_i = 1 + m_(i-1)`, and the largest two elements of `S_i` need to be

updated.

3. **Case 3**: neither of them intersects with `intervals[i]`. For this case, we show `m_i = 2 + m_(i-1)`, and the largest two elements of `S_i` need to be updated.

Consider **Case 1** first. From our analyses above, we know `S_(i-1)` is also a minimum intersection set for subarray `intervals[0, i]`, therefore we can choose `S_i` to be the same as `S_(i-1)` and get `m_i = m_(i-1)`. Can we make the largest two elements of `S_i` even larger? Negative. It is because if such a set `S_i` exists, it will also be a minimum intersection set for the subarray `intervals[0, i-1]`, so its largest two elements cannot exceed those of `S_(i-1)` (note we've already chosen `S_(i-1)` to be the minimum intersection set for `intervals[0, i-1]` with the largest two elements maximized), contradicting the assumption.

Consider **Case 2** next. Can we still have `m_i = m_(i-1)`? Nope. If this is true, then `S_i` will also be a minimum intersection set for `intervals[0, i-1]`, indicating its two largest elements won't exceed those of `S_(i-1)`. Since the largest two elements of `S_i` intersect with `intervals[i]`, the largest two elements of `S_(i-1)` must also intersect with `intervals[i]`, contradicting the fact that only one of them intersects with `intervals[i]`. Therefore we conclude: `m_i >= 1 + m_(i-1)`. Can we have `m_i = 1 + m_(i-1)`? Yes. `S_i` can be constructed by simply adding the end point of `intervals[i]` to `S_(i-1)`. This not only makes `S_i` a minimum intersection set for the subarray `intervals[0, i]`, but also maximizes its largest two elements.

Consider **Case 3** last. Can we have either `m_i = m_(i-1)` or `m_i = 1 + m_(i-1)`? No, with the same reasoning in **Case 2**. So we conclude: `m_i >= 2 + m_(i-1)`. Can we have `m_i = 2 + m_(i-1)`? Yes. `S_i` can be constructed by simply adding the end point and the point immediately before the end point of `intervals[i]` to `S_(i-1)`. This will turn `S_i` into a minimum intersection set for the subarray `intervals[0, i]`, meanwhile maximize its largest two elements.

For all the three cases, `S_i` is constructed in such a way that its size is minimized first, then its largest two elements are maximized. When all the intervals are processed, `S_(n-1)` will be a minimum intersection set of the whole input array and `m_(n-1)` is the minimum size we are looking for.

`IV -- The solution`

Here is the actual code for the finding the size of the minimum intersection set. We first sort the intervals as described above, then build the minimum intersection set for each of the subarrays one by one (take different measures depending on which case it is). Space complexity is `O(1)` while the time complexity is bound by the sorting part, which is `O(nlogn)`.

```java
public int intersectionSizeTwo(int[][] intervals) {
    Arrays.sort(intervals, new Comparator<int[]>() {
     public int compare(int[] a, int[] b) {
         return a[1] != b[1] ? Integer.compare(a[1], b[1]) : Integer.compare(b[0],
a[0]);
     }
    });

    int m = 0, largest = -1, second = -1;

    for (int[] interval : intervals) {
        int a = interval[0], b = interval[1];

 boolean is_largest_in = (a <= largest);
        boolean is_second_in = (a <= second);

 if (is_largest_in && is_second_in) continue;

        m += (is_largest_in ? 1 : 2);

 second = (is_largest_in ? largest : b - 1);
 largest = b;
    }

    return m;
}
```

written by fun4LeetCode original link here