435. Non Overlapping Intervals

Jan. 27, 2017 | 21K views

Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Example 1:

```
Input: [[1,2],[2,3],[3,4],[1,3]]
Output: 1
Explanation: [1,3] can be removed and the rest of intervals are non-overlapping.
```

Example 2:

```
Input: [[1,2],[1,2],[1,2]]
Output: 2
Explanation: You need to remove two [1,2] to make the rest of intervals non-overlappi
```

Example 3:

```
Input: [[1,2],[2,3]]
Output: 0
Explanation: You don't need to remove any of the intervals since they're already non-
```

Note:

1. You may assume the interval's end point is always bigger than its start point.
2. Intervals like [1,2] and [2,3] have borders "touching" but they don't overlap each other.

Summary

Given a collection of intervals, we need to find the minimum number of intervals to be removed to make the rest of the intervals non-overlapping.

Solution

Approach #1 Brute Force [Time Limit Exceeded]

In the brute force approach, we try to remove the overlapping intervals in different combinations and then check which combination needs the minimum number of removals. To do this, firstly we sort the given intervals based on the starting point. Then, we make use of a recursive function `eraseOverlapIntervals` which takes the index of the previous interval $prev$ and the index of the current interval $curr$ (which we try to add to the list of intervals not removed), and returns the count of intervals that need to be removed from the current index onwards.

We start by using $prev = -1$ and $curr = 0$. In each recursive call, we check if the current interval overlaps with the previous interval. If not, we need not remove the current interval from the final list and we can call the function `eraseOverlapIntervals` with $prev = curr$ and $curr = curr + 1$. The result of this function call in which we have included the current element is stored in $taken$ variable.

We also make another function call by removing the current interval because this could be overlapping with the upcoming intervals in the next function call and thus, its removal could eventually require lesser total number of removals. Thus, the recursive call takes the arguments $prev = prev$ and $curr = curr + 1$. Since, we have removed one interval, the result if the current interval isn't included is the sum of the value returned by the function call incremented by 1, which is stored in $notTaken$ variable. While returning the count of removals following a particular index, we return the minimum of $taken$ and $notTaken$.

Java

```java
class Solution {
  class myComparator implements Comparator<int[]> {
    public int compare(int[] a, int[] b) {
      return a[1] - b[1];
    }
  }
  public int eraseOverlapIntervals(int[][] intervals) {
    Arrays.sort(intervals, new myComparator());
    return erase_Overlap_Intervals(-1, 0, intervals);
  }
  public int erase_Overlap_Intervals(int prev, int curr, int[][] intervals) {
    if (curr == intervals.length) {
      return 0;
    }
    int taken = Integer.MAX_VALUE, nottaken;
    if (prev == -1 || intervals[prev][1] <= intervals[curr][0]) {
      taken = erase_Overlap_Intervals(curr, curr + 1, intervals);
    }
    nottaken = erase_Overlap_Intervals(prev, curr + 1, intervals) + 1;
    return Math.min(taken, nottaken);
  }
}
```

Complexity Analysis

- Time complexity : $O(2^n)$. Total possible number of Combinations of subsets are 2^n.
- Space complexity : $O(n)$. Depth of recursion is n.

Approach #2 Using DP based on starting point [Accepted]

Algorithm

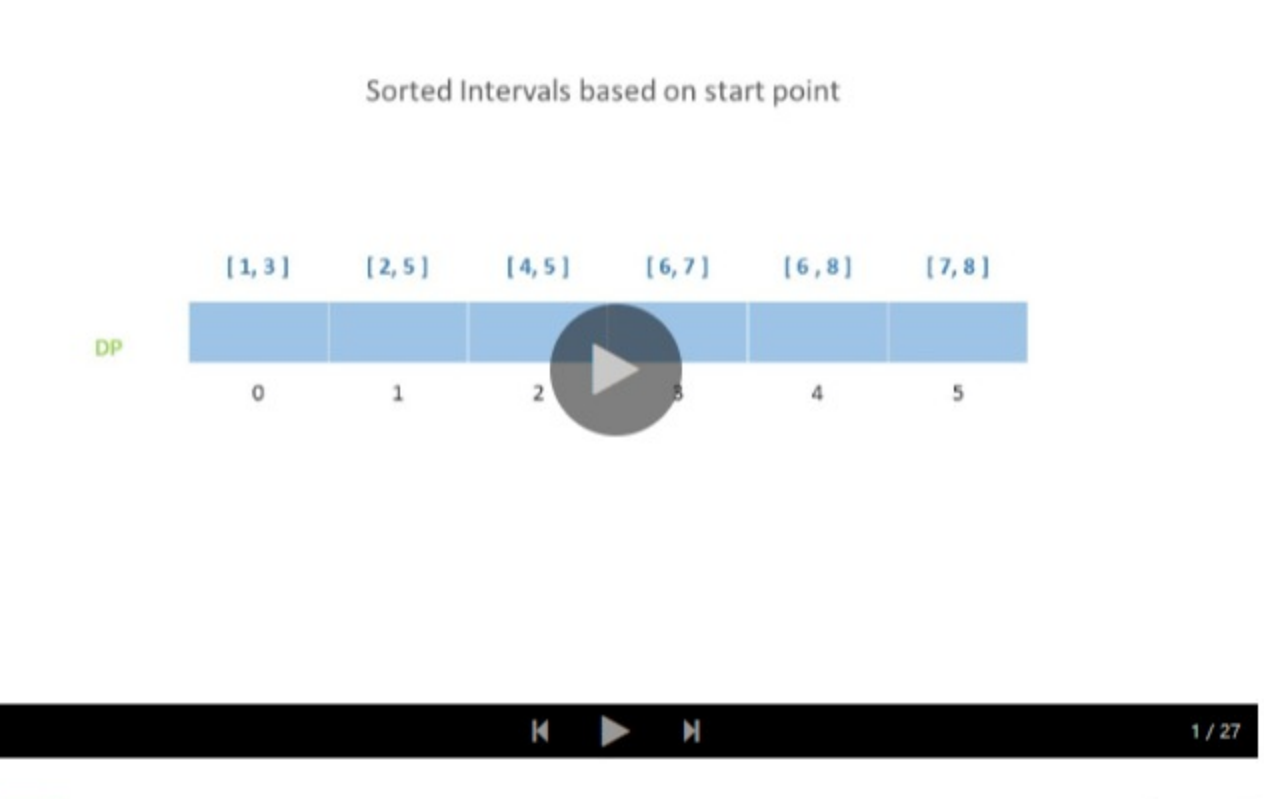
The given problem can be simplified to a great extent if we sort the given interval list based on the starting points. Once it's done, we can make use of a dp array, scuh that $dp[i]$ stores the maximum number of valid intervals that can be included in the final list if the intervals upto the i^{th} interval only are considered, including itself. Now, while finding $dp[i+1]$, we can't consider the value of $dp[i]$ only, because it could be possible that the i^{th} or any previous interval could be overlapping with the $(i+1)^{th}$ interval. Thus, we need to consider the maximum of all $dp[j]$'s such that $j \leq i$ and j^{th} interval and i^{th} don't overlap, to evaluate $dp[i+1]$. Therefore, the equation for $dp[i+1]$ becomes:

$$dp[i+1] = \max(dp[j]) + 1,$$

such that j^{th} interval and i^{th} don't overlap, for all $j \leq i$.

In the end, to obtain the maximum number of intervals that can be included in the final list(ans) we need to find the maximum value in the dp array. The final result will be the total number of intervals given less the result just obtained($intervals.length - ans$).

The animation below illustrates the approach more clearly:



Java

```java
  class myComparator implements Comparator<int[]> {
    public int compare(int[] a, int[] b) {
      return a[1] - b[1];
    }
  }
  public boolean isOverlapping(int[] i, int[] j) {
    return i[1] > j[0];
  }
  public int eraseOverlapIntervals(int[][] intervals) {
    if (intervals.length == 0) {
      return 0;
    }
    Arrays.sort(intervals, new myComparator());
    int dp[] = new int[intervals.length];
    dp[0] = 1;
    int ans = 1;
    for (int i = 1; i < dp.length; i++) {
      int max = 0;
      for (int j = i - 1; j >= 0; j--) {
        if (!isOverlapping(intervals[j], intervals[i])) {
          max = Math.max(dp[j], max);
        }
      }
      dp[i] = max + 1;
      ans = Math.max(ans, dp[i]);
    }
    return intervals.length - ans;
```

Complexity Analysis

- Time complexity : $O(n^2)$. Two nested loops are required to fill dp array.
- Space complexity : $O(n)$. dp array of size n is used.

Approach #3 Using DP based on the end points [Accepted]

Algorithm

In the DP approach just discussed above, for calculating the value of every $dp[i]$, we need to traverse the dp array back till the starting index. This overhead can be removed, if we use an interval list sorted on the basis of the end points. Thus, now, again we use the same kind of dp array, where $dp[i]$ is used to store the maximum number of intervals that can be included in the final list if the intervals only upto the i^{th} index in the sorted list are considered. Thus, in order to find $dp[i+1]$ now, we've to consider two cases:

Case 1:

> The interval corresponding to $(i+1)^{th}$ interval needs to be included in the final list to obtain the minimum number of removals:

In this case, we need to traverse back in the sorted interval array form the $(i+1)^{th}$ index upto the starting index to find the first interval which is non-overlapping. This is because, if we are including the current interval, we need to remove all the intervals which are overlapping with the current interval. But, we need not go back till the starting index everytime. Instead we can stop the traversal as soon as we hit the first non-overlapping interval and use its $dp[j] + 1$ to fill in $dp[i+1]$, since $dp[j]$ will be the element storing the maximum number of intervals that can be included considering elements upto the j^{th} index.

Case 2:

> The interval corresponding to $(i+1)^{th}$ interval needs to be removed from the final list to obtain the minimum number of removals:

In this case, the current element won't be included in the final list. So, the count of intervals to be included upto $(i+1)^{th}$ index is the same as the count of intervals upto the i^{th} index. Thus, we can use $dp[i]$'s value to fill in $dp[i+1]$.

The value finally entered in $dp[i+1]$ will be the maximum of the above two values.

The final result will again be the total count of intervals less the result obtained at the end from the dp array.

The animation below illustrates the approach more clearly:

Java

```java
class Solution {
  class myComparator implements Comparator<int[]> {
    public int compare(int[] a, int[] b) {
      return a[1] - b[1];
    }
  }
  public boolean isOverlapping(int[] i, int[] j) {
    return i[1] > j[0];
  }
  public int eraseOverlapIntervals(int[][] intervals) {
    if (intervals.length == 0) {
      return 0;
    }
    Arrays.sort(intervals, new myComparator());
    int dp[] = new int[intervals.length];
    dp[0] = 1;
    int ans = 1;
    for (int i = 1; i < dp.length; i++) {
      int max = 0;
      for (int j = i - 1; j >= 0; j--) {
        if (!isOverlapping(intervals[j], intervals[i])) {
          max = Math.max(dp[j], max);
          break;
        }
      }
      dp[i] = Math.max(max + 1, dp[i - 1]);
      ans = Math.max(ans, dp[i]);
```

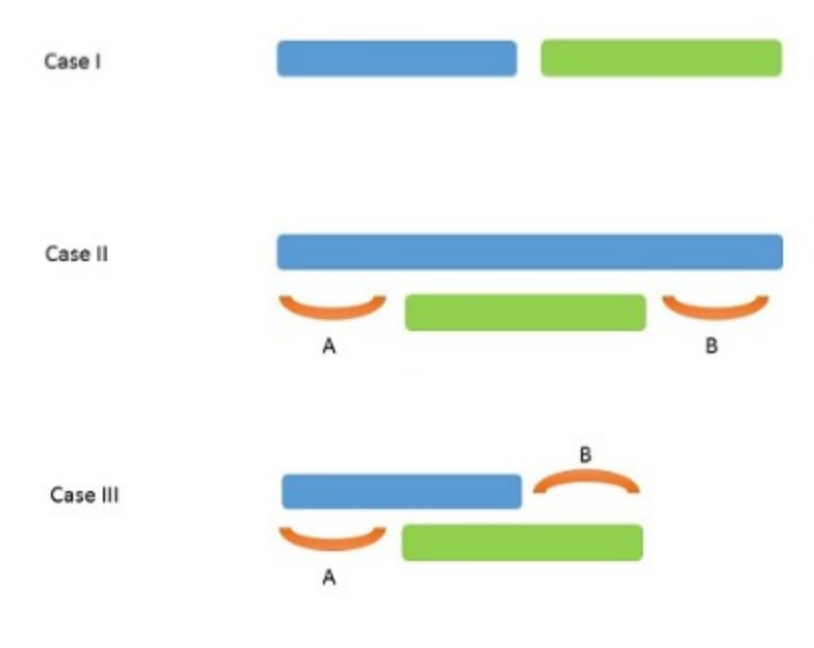
Complexity Analysis

- Time complexity : $O(n^2)$. Two nested loops are required to fill dp array.
- Space complexity : $O(n)$. dp array of size n is used.

Approach #4 Using Greedy Approach based on starting points [Accepted]

Algorithm

If we sort the given intervals based on starting points, the greedy approach works very well. While considering the intervals in the ascending order of starting points, we make use of a pointer $prev$ pointer to keep track of the interval just included in the final list. While traversing, we can encounter 3 possibilities as shown in the figure:



Case 1:

> The two intervals currently considered are non-overlapping:

In this case, we need not remove any interval and we can continue by simply assigning the $prev$ pointer to the later interval and the count of intervals removed remains unchanged.

Case 2:

> The two intervals currently considered are overlapping and the end point of the later interval falls before the end point of the previous interval:

In this case, we can simply take the later interval. The choice is obvious since choosing an interval of smaller width will lead to more available space labelled as A and B, in which more intervals can be accommodated. Hence, the $prev$ pointer is updated to current interval and the count of intervals removed is incremented by 1.

Case 3:

> The two intervals currently considered are overlapping and the end point of the later interval falls after the end point of the previous interval:

In this case, we can work in a greedy manner and directly remove the later interval. To understand why this greedy approach works, we need to see the figure below, which includes all the subcases possible. It is clear from the figures that we choosing interval 1 always leads to a better solution in the future. Thus, the $prev$ pointer remains unchanged and the count of intervals removed is incremented by 1.

Java

```java
class Solution {
  class myComparator implements Comparator<int[]> {
    public int compare(int[] a, int[] b) {
      return a[1] - b[1];
    }
  }

  public int eraseOverlapIntervals(int[][] intervals) {
    if (intervals.length == 0) {
      return 0;
    }
    Arrays.sort(intervals, new myComparator());
    int end = intervals[0][1], prev = 0, count = 0;
    for (int i = 1; i < intervals.length; i++) {
      if (intervals[prev][1] > intervals[i][0]) {
        if (intervals[prev][1] > intervals[i][1]) {
          prev = i;
        }
        count++;
      } else {
        prev = i;
      }
    }
    return count;
  }
}
```

Complexity Analysis

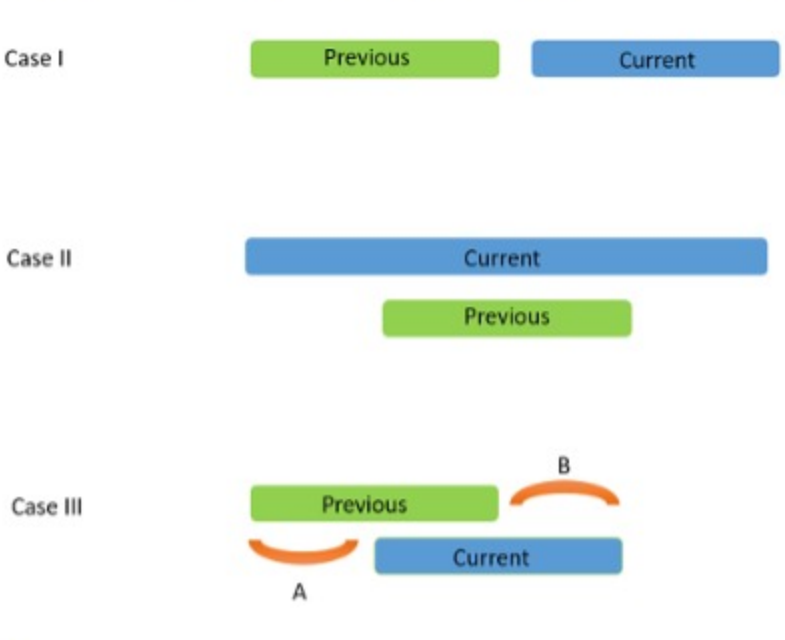
- Time complexity : $O(n\log(n))$. Sorting takes $O(n\log(n))$ time.
- Space complexity : $O(1)$. No extra space is used.

Approach #5 Using Greedy Approach based on end points [Accepted]

Algorithm

The Greedy approach just discussed was based on choosing intervals greedily based on the starting points. But in this approach, we go for choosing points greedily based on the end points. For this, firstly we sort the given intervals based on the end points. Then, we traverse over the sorted intervals. While traversing, if there is no overlapping between the previous interval and the current interval, we need not remove any interval. But, if an overlap exists between the previous interval and the current interval, we always drop the current interval.

To explain how it works, again we consider every possible arrangement of the intervals.



Case 1:

> The two intervals currently considered are non-overlapping:

In this case, we need not remove any interval and for the next iteration the current interval becomes the previous interval.

Case 2:

> The two intervals currently considered are overlapping and the starting point of the later interval falls before the starting point of the previous interval:

In this case, as shown in the figure below, it is obvious that the later interval completely subsumes the previous interval. Hence, it is advantageous to remove the later interval so that we can get more range available to accommodate future intervals. Thus, previous interval remains unchanged and the current interval is updated.

Case 3:

> The two intervals currently considered are overlapping and the starting point of the later interval falls before the starting point of the previous interval:

In this case, the only opposition to remove the current interval arises because it seems that more intervals could be accommodated by removing the previous interval in the range marked by A. But that won't be possible as can be visualized with a case similar to Case 3a and 3b shown above. But, if we remove the current interval, we can save the range B to accommodate further intervals. Thus, previous interval remains unchanged and the current interval is updated.

Java

```java
class Solution {
  class myComparator implements Comparator<int[]> {
    public int compare(int[] a, int[] b) {
      return a[1] - b[1];
    }
  }

  public int eraseOverlapIntervals(int[][] intervals) {
    if (intervals.length == 0) {
      return 0;
    }
    Arrays.sort(intervals, new myComparator());
    int end = intervals[0][1];
    int count = 1;
    for (int i = 1; i < intervals.length; i++) {
      if (intervals[i][0] >= end) {
        end = intervals[i][1];
        count++;
      }
    }
    return intervals.length - count;
  }
}
```

Complexity Analysis

- Time complexity : $O(n\log(n))$. Sorting takes $O(n\log(n))$ time.
- Space complexity : $O(1)$. No extra space is used.