Solution Review: Problem Challenge 1

We'll cover the following

Highlight

Next Interval (hard)#

Given an array of intervals, find the next interval of each interval. In a list of intervals, for an interval `i` its next interval `j` will have the smallest 'start' greater than or equal to the 'end' of `i`.

Write a function to return an array containing indices of the next interval of each input interval. If there is no next interval of a given interval, return -1. It is given that none of the intervals have the same start point.

Example 1:

> **Input:** Intervals [[2,3], [3,4], [5,6]]
> **Output:** [1, 2, -1]
> **Explanation:** The next interval of [2,3] is [3,4] having index ‘1’. Similarly, the next interval of [3,4] is [5,6] having index ‘2’. There is no next interval for [5,6] hence we have ‘-1’.

Example 2:

> **Input:** Intervals [[3,4], [1,5], [4,6]]
> **Output:** [2, -1, -1]
> **Explanation:** The next interval of [3,4] is [4,6] which has index ‘2’. There is no next interval for [1,5] and [4,6].

Solution#

A brute force solution could be to take one interval at a time and go through all the other intervals to find the next interval. This algorithm will take $O(N^2)$ where N is the total number of intervals. Can we do better than that?

We can utilize the **Two Heaps** approach. We can push all intervals into two heaps: one heap to sort the intervals on maximum start time (let’s call it `maxStartHeap`) and the other on maximum end time (let’s call it `maxEndHeap`). We can then iterate through all intervals of the `maxEndHeap` to find their next interval. Our algorithm will have the following steps:

1. Take out the top (having highest end) interval from the `maxEndHeap` to find its next interval. Let's call this interval `topEnd`.
2. Find an interval in the `maxStartHeap` with the closest start greater than or equal to the start of `topEnd`. Since `maxStartHeap` is sorted by 'start' of intervals, it is easy to find the interval with the highest 'start'. Let's call this interval `topStart`.
3. Add the index of `topStart` in the result array as the next interval of `topEnd`. If we can't find the next interval, add '-1' in the result array.
4. Put the `topStart` back in the `maxStartHeap`, as it could be the next interval of other intervals.
5. Repeat steps 1-4 until we have no intervals left in `maxEndHeap`.

Code#

Here is what our algorithm will look like:

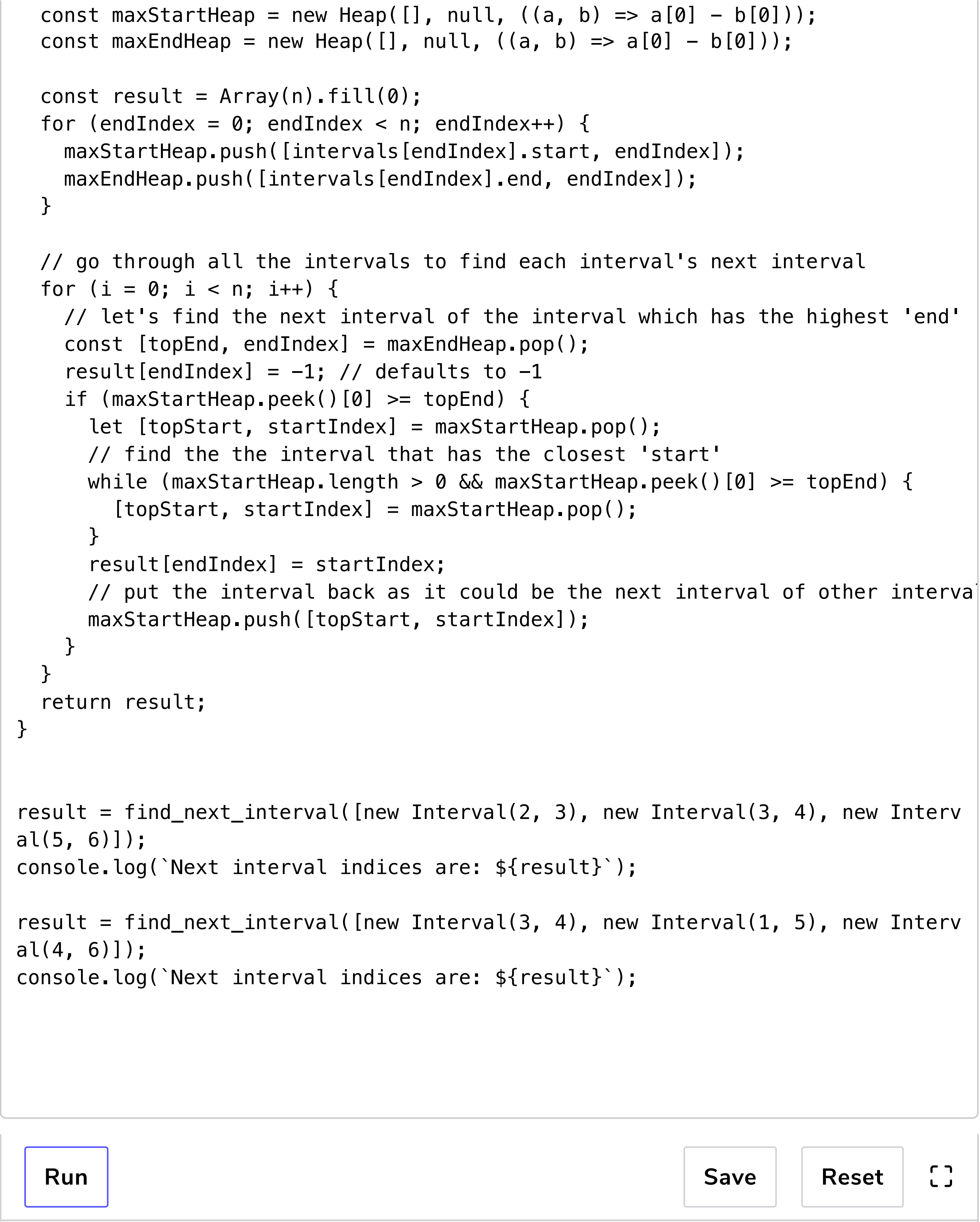
Java | Python3 | C++ | JS

```
const Heap = require('./collections/heap'); //http://www.collectionsjs.com


class Interval {
  constructor(start, end) {
    this.start = start;
    this.end = end;
  }
}


function find_next_interval(intervals) {
  const n = intervals.length;

  // heaps for finding the maximum start and end
```



```
  const maxStartHeap = new Heap([], null, ((a, b) => a[0] - b[0]));
  const maxEndHeap = new Heap([], null, ((a, b) => a[0] - b[0]));

  const result = Array(n).fill(0);
  for (endIndex = 0; endIndex < n; endIndex++) {
    maxStartHeap.push([intervals[endIndex].start, endIndex]);
    maxEndHeap.push([intervals[endIndex].end, endIndex]);
  }

  // go through all the intervals to find each interval's next interval
  for (i = 0; i < n; i++) {
    // let's find the next interval of the interval which has the highest 'end'
    const [topEnd, endIndex] = maxEndHeap.pop();
    result[endIndex] = -1; // defaults to -1
    if (maxStartHeap.peek()[0] >= topEnd) {
      let [topStart, startIndex] = maxStartHeap.pop();
      // find the the interval that has the closest 'start'
      while (maxStartHeap.length > 0 && maxStartHeap.peek()[0] >= topEnd) {
        [topStart, startIndex] = maxStartHeap.pop();
      }
      result[endIndex] = startIndex;
      // put the interval back as it could be the next interval of other interva
      maxStartHeap.push([topStart, startIndex]);
    }
  }
  return result;
}


result = find_next_interval([new Interval(2, 3), new Interval(3, 4), new Interv
al(5, 6)]);
console.log(`Next interval indices are: ${result}`);

result = find_next_interval([new Interval(3, 4), new Interval(1, 5), new Interv
al(4, 6)]);
console.log(`Next interval indices are: ${result}`);
```

Time complexity#

The time complexity of our algorithm will be $O(NlogN)$, where N is the total number of intervals.

Space complexity#

The space complexity will be $O(N)$ because we will be storing all the intervals in the heaps.

← Back

Problem Challenge 1

Next →

Introduction

☑ Mark as Completed

Report an Issue