56. Merge Intervals

Previous Next

Nov. 15, 2017 | 311.2K views

Average Rating: 4.55 (94 votes)

Given a collection of intervals, merge all overlapping intervals.

Example 1:

```
Input: [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
```

Example 2:

```
Input: [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.
```

NOTE: input types have been changed on April 15, 2019. Please reset to default code definition to get new method signature.

Solution

Approach 1: Connected Components

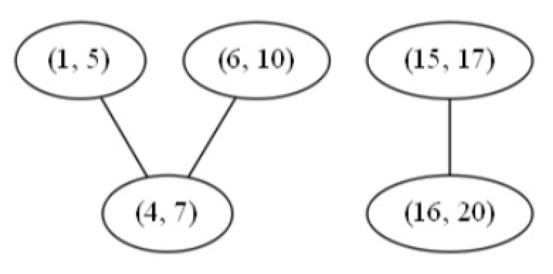
Intuition

If we draw a graph (with intervals as nodes) that contains undirected edges between all pairs of intervals that overlap, then all intervals in each *connected component* of the graph can be merged into a single interval.

Algorithm

With the above intuition in mind, we can represent the graph as an adjacency list, inserting directed edges in both directions to simulate undirected edges. Then, to determine which connected component each node is it, we perform graph traversals from arbitrary unvisited nodes until all nodes have been visited. To do this efficiently, we store visited nodes in a `Set`, allowing for constant time containment checks and insertion. Finally, we consider each connected component, merging all of its intervals by constructing a new `Interval` with `start` equal to the minimum start among them and `end` equal to the maximum end.

This algorithm is correct simply because it is basically the brute force solution. We compare every interval to every other interval, so we know exactly which intervals overlap. The reason for the connected component search is that two intervals may not directly overlap, but might overlap indirectly via a third interval. See the example below to see this more clearly.



Although (1, 5) and (6, 10) do not directly overlap, either would overlap with the other if first merged with (4, 7). There are two connected components, so if we merge their nodes, we expect to get the following two merged intervals:

(1, 10), (15, 20)

Java | Python3 | Copy

```python
class Solution:
    def overlap(self, a, b):
        return a[0] <= b[1] and b[0] <= a[1]

    # generate graph where there is an undirected edge between intervals u
    # and v iff u and v overlap.
    def build_graph(self, intervals):
        graph = collections.defaultdict(list)

        for i, interval_i in enumerate(intervals):
            for j in range(i+1, len(intervals)):
                if self.overlap(interval_i, intervals[j]):
                    graph[tuple(interval_i)].append(intervals[j])
                    graph[tuple(intervals[j])].append(interval_i)

        return graph

    # merges all of the nodes in this connected component into one interval.
    def merge_nodes(self, nodes):
        min_start = min(node[0] for node in nodes)
        max_end = max(node[1] for node in nodes)
        return [min_start, max_end]

    # gets the connected components of the interval overlap graph.
    def get_components(self, graph, intervals):
        visited = set()
        comp_number = 0
```

Complexity Analysis

- Time complexity : $O(n^2)$

  Building the graph costs $O(V+E) = O(V) + O(E) = O(n) + O(n^2) = O(n^2)$ time, as in the worst case all intervals are mutually overlapping. Traversing the graph has the same cost (although it might appear higher at first) because our `visited` set guarantees that each node will be visited exactly once. Finally, because each node is part of exactly one component, the merge step costs $O(V) = O(n)$ time. This all adds up as follows:

  $$O(n^2) + O(n^2) + O(n) = O(n^2)$$

- Space complexity : $O(n^2)$

  As previously mentioned, in the worst case, all intervals are mutually overlapping, so there will be an edge for every pair of intervals. Therefore, the memory footprint is quadratic in the input size.

Approach 2: Sorting

Intuition

If we sort the intervals by their `start` value, then each set of intervals that can be merged will appear as a contiguous "run" in the sorted list.

Algorithm

First, we sort the list as described. Then, we insert the first interval into our `merged` list and continue considering each interval in turn as follows: If the current interval begins *after* the previous interval ends, then they do not overlap and we can append the current interval to `merged`. Otherwise, they do overlap, and we merge them by updating the `end` of the previous interval if it is less than the `end` of the current interval.

A simple proof by contradiction shows that this algorithm always produces the correct answer. First, suppose that the algorithm at some point fails to merge two intervals that should be merged. This would imply that there exists some triple of indices i, j, and k in a list of intervals ints such that $i < j < k$ and (ints[i], ints[k]) can be merged, but neither (ints[i], ints[j]) nor (ints[j], ints[k]) can be merged. From this scenario follow several inequalities:

$$\begin{aligned} \text{ints[i].end} &< \text{ints[j].start} \\ \text{ints[j].end} &< \text{ints[k].start} \\ \text{ints[i].end} &\geq \text{ints[k].start} \end{aligned}$$

We can chain these inequalities (along with the following inequality, implied by the well-formedness of the intervals: $\text{ints[j].start} \leq \text{ints[j].end}$) to demonstrate a contradiction:

$$\begin{aligned} \text{ints[i].end} < \text{ints[j].start} \leq \text{ints[j].end} &< \text{ints[k].start} \\ \text{ints[i].end} &\geq \text{ints[k].start} \end{aligned}$$

Therefore, all mergeable intervals must occur in a contiguous run of the sorted list.

```
[(1, 9), (2, 5), (19, 20), (10, 11), (12, 20), (0, 3), (0, 1), (0, 2)]
                                sort
[(0, 3), (0, 1), (0, 2), (1, 9), (2, 5), (10, 11), (12, 20), (19, 20)]
```

Consider the example above, where the intervals are sorted, and then all mergeable intervals form contiguous blocks.

Java | Python3 | Copy

```python
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:

        intervals.sort(key=lambda x: x[0])

        merged = []
        for interval in intervals:
            # if the list of merged intervals is empty or if the current
            # interval does not overlap with the previous, simply append it.
            if not merged or merged[-1][1] < interval[0]:
                merged.append(interval)
            else:
            # otherwise, there is overlap, so we merge the current and previous
            # intervals.
                merged[-1][1] = max(merged[-1][1], interval[1])

        return merged
```

Complexity Analysis

- Time complexity : $O(n \log n)$

  Other than the `sort` invocation, we do a simple linear scan of the list, so the runtime is dominated by the $O(nlgn)$ complexity of sorting.

- Space complexity : $O(1)$ (or $O(n)$)

  If we can sort `intervals` in place, we do not need more than constant additional space. Otherwise, we must allocate linear space to store a copy of `intervals` and sort that.

Rate this article:

Previous Next

Comments: 72

Sort By

Type comment here... (Markdown is supported)

Preview Post

kennethwty ★ 125 November 11, 2018 1:23 AM

We can use: Collections.sort(intervals, (a, b) -> Integer.compare(a.start, b.start));

90 Share Reply

SHOW 6 REPLIES

yongzx ★ 224 August 10, 2019 7:48 PM

Facebook Follow-Up

Question: How do you add intervals and merge them for a large stream of intervals? (Facebook Follow-up Question)

Read More

77 Share Reply

SHOW 12 REPLIES

sikp ★ 607 March 6, 2018 8:56 AM

The line 5 looks unnecessarily verbose to me : return a.start < b.start ? -1 : a.start == b.start ? 0 : 1;
You can just return a.start - b.start
And you don't have to write a nested Comparator class, an anonymous class will work
Collections.sort(intervals, new Comparator(){
@Override

Read More

23 Share Reply

SHOW 4 REPLIES

fuyaoli ★ 94 July 15, 2018 2:42 PM

The first approach is time limit exceeded.

15 Share Reply

SHOW 1 REPLY

aa11 ★ 19 June 5, 2018 3:47 AM

Shouldn't the space complexity of the second solution be O(n) as we are creating a whole new merged list?

18 Share Reply

SHOW 2 REPLIES

nikpap123 ★ 16 November 4, 2018 12:05 PM

I think there might be a O(N*inverse Ackermann(n)) time, O(N) space solution using the optimised UnionFind (Disjoint Set) Data Structure. The idea is that UnionFind supports quick find and quick update and you can create a set for each interval. Then, as you iterate through the list you update the sets you already have or create new ones. Has anyone come up with a similar solution?

12 Share Reply

SHOW 2 REPLIES

sacerdoti ★ 16 February 19, 2019 6:14 AM

Short C++ sort-style, 16ms 97.8%. Space is O(1) as in-place sort and move semantics.

```
class Solution {
public:
    vector<Interval> merge(vector<Interval>& intervals) {
```

Read More

8 Share Reply

SHOW 2 REPLIES

azimbaghadiya ★ 14 September 24, 2019 7:34 PM

Note that `interval.start` and `interval.end` will no longer work for Python, as the question doesn't use those classes anymore. They are now using Python lists, so `interval[0]` and `interval[1]` must be used instead.

7 Share Reply

robbiefj ★ 39 January 20, 2018 9:17 AM

Your space complexity for the sorting answer is not O(1). You are creating a new list for the output. The space usage is then proportional to the number of intervals in the output, which in the worst case is O(n).

10 Share Reply

SHOW 5 REPLIES

suckcodefuckU ★ 26 September 14, 2018 12:28 PM

We can use `Integer.compare(a.start,b.start)`

6 Share Reply

SHOW 2 REPLIES

‹ 1 2 3 4 5 6 7 8 ›