December 11, 2021 ▪ Arrays / Data Structure

Merge Overlapping Sub-intervals

Problem Statement: Given an array of intervals, merge all the overlapping intervals and return an array of non-overlapping intervals.

Examples

```
Example 1:

Input: intervals=[[1,3],[2,6],[8,10],[15,18]]

Output: [[1,6],[8,10],[15,18]]

Explanation: Since intervals [1,3] and [2,6] are overlapping we can merge them to form [1,6]
 intervals.

Example 2:

Input: [[1,4],[4,5]]

Output: [[1,5]]

Explanation: Since intervals [1,4] and [4,5] are overlapping we can merge them to form [1,5].
```

Solution

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Solution 1: Brute force

Approach: First check whether the array is sorted or not.If not sort the array. Now linearly iterate over the array and then check for all of its next intervals whether they are overlapping with the interval at the current index. Take a new data structure and insert the overlapped interval. If while iterating if the interval lies in the interval present in the data structure simply continue and move to the next interval.

Code:

C++ Code

```
#include<bits/stdc++.h>
using namespace std;

vector < pair < int, int >> merge(vector < pair < int, int >> & arr) {

    int n = arr.size();
    sort(arr.begin(), arr.end());
    vector < pair < int, int >> ans;

    for (int i = 0; i < n; i++) {
        int start = arr[i].first, end = arr[i].second;

        //since the intervals already lies
        //in the data structure present we continue
        if (!ans.empty()) {
            if (start <= ans.back().second) {
                continue;
            }
        }

        for (int j = i + 1; j < n; j++) {
            if (arr[j].first <= end) {
                end = max(end, arr[j].second);
            }
        }

        end = max(end, arr[i].second);

        ans.push_back({start, end});
    }

    return ans;
}
```

```
int main() {
    vector < pair < int, int >> arr;
    arr = {{1,3},{2,4},{2,6},{8,9},{8,10},{9,11},{15,18},{16,17}};
    vector < pair < int, int >> ans = merge(arr);

    cout << "Merged Overlapping Intervals are " << endl;

    for (auto it: ans) {
        cout << it.first << " " << it.second << "\n";
    }
}
```

Output:

```
Merged Overlapping Intervals are
1 6
8 11
15 18
```

Time Complexity: O(NlogN)+O(N*N). O(NlogN) for sorting the array, and O(N*N) because we are checking to the right for each index which is a nested loop.

Space Complexity: O(N), as we are using a separate data structure.

Java Code

```
import java.util.*;

// Pair class.
class Pair {
 int start;
 int end;

 public Pair(int start, int end) {
 this.start = start;
 this.end = end;

    }
}

public class Tuf {

    public static List<Pair> merge(List<Pair> intervals) {

          int n = intervals.size();
          Collections.sort(intervals, (a, b) -> Integer.compare(a.start, b.start));
          List<Pair> ans = new ArrayList<>();

          for (int i = 0; i < n; i++) {

             Pair temp = intervals.get(i);
             int start = temp.start;
             int end = temp.end;


             if (!ans.isEmpty()) {
             if (start <= ans.get(ans.size()-1).end) {
                continue;
             }
        }

        for (int j = i + 1; j < n; j++) {
            if (intervals.get(j).start <= end) {
                end = Math.max(end, intervals.get(j).end);
            }
        }

        end = Math.max(end, intervals.get(i).end);

        ans.add(new Pair(start, end));
    }

       return ans;

    }

    public static void main(String[] args) {

        List<Pair> input = new ArrayList<>();
        input.add(new Pair(1, 3));
        input.add(new Pair(2, 4));
        input.add(new Pair(2, 6));
        input.add(new Pair(8, 9));
        input.add(new Pair(8, 10));
        input.add(new Pair(9, 11));
        input.add(new Pair(15, 18));
        input.add(new Pair(16, 17));

         List<Pair> output = merge(input);

         for(Pair result : output) {
                System.out.println(result.start + " " + result.end);
            }
        }
    }
```

Output:

Merged Overlapping Intervals are

1 6

8 11

15 18

Time Complexity: O(NlogN)+O(N*N). O(NlogN) for sorting the array, and O(N*N) because we are checking to the right for each index which is a nested loop.

Space Complexity: O(N), as we are using a separate data structure.

Python Code

```python
from typing import List
def merge(arr: List[List[int]]) -> List[List[int]]:
    n = len(arr)
    arr.sort()
    ans = []


    for i in range(n):
        start, end = arr[i][0], arr[i][1]


        # since the intervals already lies
        # in the data structure present we continue
        if ans:
            if start <= ans[-1][1]:
                continue


        for j in range(i + 1, n):
            if arr[j][0] <= end:
                end = max(end, arr[j][1])


        end = max(end, arr[i][1])


        ans.append([start, end])


    return ans




if __name__ == "__main__":
    arr = [[1, 3], [2, 4], [2, 6], [8, 9], [
        8, 10], [9, 11], [15, 18], [16, 17]]
    ans = merge(arr)


    print("Merged Overlapping Intervals are ")


    for start, end in ans:
        print(start, end)
```

Output:

Merged Overlapping Intervals are

1 6

8 11

15 18

Time Complexity: O(NlogN)+O(N*N). O(NlogN) for sorting the array, and O(N*N) because we are checking to the right for each index which is a nested loop.

Space Complexity: O(N), as we are using a separate data structure.

Solution 2: Optimal approach

Approach: Linearly iterate over the array if the data structure is empty insert the interval in the data structure. If the last element in the data structure overlaps with the current interval we merge the intervals by updating the last element in the data structure, and if the current interval does not overlap with the last element in the data structure simply insert it into the data structure.

Intuition: Since we have sorted the intervals, the intervals which will be merging are bound to be adjacent. We kept on merging simultaneously as we were traversing through the array and when the element was non-overlapping we simply inserted the element in our data structure.

Code:

C++ Code

```cpp
#include<bits/stdc++.h>

using namespace std;
vector < vector < int >> merge(vector < vector < int >> & intervals) {

  sort(intervals.begin(), intervals.end());
  vector < vector < int >> merged;

  for (int i = 0; i < intervals.size(); i++) {
    if (merged.empty() || merged.back()[1] < intervals[i][0]) {
      vector < int > v = {
        intervals[i][0],
        intervals[i][1]
      };

      merged.push_back(v);
    } else {
      merged.back()[1] = max(merged.back()[1], intervals[i][1]);
    }
  }

  return merged;
}

int main() {
  vector < vector < int >> arr;
  arr = {{1, 3}, {2, 4}, {2, 6}, {8, 9}, {8, 10}, {9, 11}, {15, 18}, {16, 17}};
  vector < vector < int >> ans = merge(arr);

  cout << "Merged Overlapping Intervals are " << endl;

  for (auto it: ans) {
    cout << it[0] << " " << it[1] << "\n";
  }
}
```

Output:

```
Merged Overlapping Intervals are
1 6
8 11
15 18
```

Time Complexity: O(NlogN) + O(N). O(NlogN) for sorting and O(N) for traversing through the array.

Space Complexity: O(N) to return the answer of the merged intervals.

Java Code

```java
import java.util.*;

public class TUF {
    static ArrayList < List < Integer >> merge(ArrayList < List < Integer >> intervals) {

        Collections.sort(intervals, (a,b)->a.get(0)-b.get(0));
        ArrayList < List < Integer >> merged = new ArrayList < > ();

        for (int i = 0; i < intervals.size(); i++) {
            if (merged.isEmpty() || merged.get(merged.size() - 1).get(1) < intervals.get(i).get(0)) {
                ArrayList < Integer > v = new ArrayList < > ();
                v.add(intervals.get(i).get(0));
                v.add(intervals.get(i).get(1));

                merged.add(v);
            } else {

                merged.get(merged.size() - 1).set(1, Math.max(merged.get(merged.size() - 1).get(1), intervals.g
            }
        }

        return merged;
    }

    public static void main(String args[]) {
        ArrayList < List < Integer >> arr = new ArrayList < > ();
        arr.add(Arrays.asList(new Integer[]{1,3}));
        arr.add(Arrays.asList(new Integer[]{2,4}));
        arr.add(Arrays.asList(new Integer[]{2,6}));
        arr.add(Arrays.asList(new Integer[]{8,9}));
        arr.add(Arrays.asList(new Integer[]{8,10}));
        arr.add(Arrays.asList(new Integer[]{9,11}));
        arr.add(Arrays.asList(new Integer[]{15,18}));
        arr.add(Arrays.asList(new Integer[]{16,17}));
        ArrayList < List < Integer >> ans = merge(arr);
        System.out.println("Merged Overlapping Intervals are ");
        for (List < Integer > it: ans) {
            System.out.println(it.get(0) + " " + it.get(1));
        }
    }
```

Output:

```
Merged Overlapping Intervals are
1 6
8 11
15 18
```

Time Complexity: O(NlogN) + O(N). O(NlogN) for sorting and O(N) for traversing through the array.

Space Complexity: O(N) to return the answer of the merged intervals.

Python Code

```python
from typing import List
def merge(intervals: List[List[int]]) -> List[List[int]]:


    intervals.sort()
    merged = []


    for i in range(len(intervals)):
        if not merged or merged[-1][1] < intervals[i][0]:
            merged.append(intervals[i])
        else:
            merged[-1][1] = max(merged[-1][1], intervals[i][1])


    return merged




if __name__ == "__main__":
    arr = [[1, 3], [2, 4], [2, 6], [8, 9], [
        8, 10], [9, 11], [15, 18], [16, 17]]
    ans = merge(arr)


    print("Merged Overlapping Intervals are ")


    for start, end in ans:
        print(start, end)
```

Output:

```
Merged Overlapping Intervals are
1 6
8 11
15 18
```

Time Complexity: O(NlogN) + O(N). O(NlogN) for sorting and O(N) for traversing through the array.

Space Complexity: O(N) to return the answer of the merged intervals.

> Special thanks to Pranav Padawe and **Sudip Ghosh** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article

Merge Intervals | Leetcode | Problem-6 | Brute-Optimal | C++/Java

« Previous Post
Implement LRU Cache

Next Post »
Sprinklr Interview Experience | Set – 1

Load Comments