INTERVAL

Some Common Patterns

Merge Intervals

- Sort by **start time**
- Merge if `current.start <= previous.end`
- Classic use: *merging calendar events, range compression*

Interval Scheduling (Max Non-overlapping Intervals)

- Sort by **end time**
- Greedy: select interval only if `start >= last_selected.end`
- Optimal because ending earlier leaves more time for future

Minimum Number of Arrows to Burst Balloons

- Same as interval scheduling
- Sort by **end time**
- Count how many non-overlapping intervals = minimum arrows needed

Some Common Patterns II

Minimum Meeting Rooms

- Sort by start and end times separately
- Use a min-heap to track active meetings
- Greedy, but uses more advanced data structure

Can Attend All Meetings?

- Sort by start time
- Check if any overlap with previous: if `start < previous.end`, return false

Problem – 57. Insert Interval

Medium

LeetCode leetcode.com/problems/insert-interval

Problem Statement

- You are given an array of **intervals,** where **intervals[i] = [$start_i$, end_i]** and **newInterval = [start, end]**
- **newInterval** must be inserted into **intervals**
- Overlapping intervals must be merged
- Example

  intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]] **newInterval** = [4,8]

  Output: [[1,2],[3,10],[12,16]]

Solution – 57. Insert Interval

Medium

LeetCode leetcode.com/problems/insert-interval

Solution

- Sort intervals by the first element (start)
- Initialize **result**
- Solve in three loops:
  1. While there is no overlap with **newInterval**, add to **intervals[i]** to **result**
  2. While it overlaps, merge **newInterval**
  3. While until the end intervals and add the remaining **intervals[i]**

Code – 57. Insert Interval

Medium

LeetCode leetcode.com/problems/insert-interval

Code

Time: **O(n)** Space: **O(n)** where n is the size of intervals

```
vector<vector<int>> insert(vector<vector<int>>& intervals, vector<int>& newInterval) {
    vector<vector<int>> result;
    int tupleIndex = 0;
    int totalTuples = intervals.size();
    // 1. check if it overlaps
    // 1 ------ 2
    //              4 ----- 8
    while (tupleIndex < totalTuples && intervals[tupleIndex][1] < newInterval[0]) {
        result.push_back(intervals[tupleIndex]);
        ++tupleIndex;
    }


    // 2. merge overlap. We already know there is an overlap here,
    // otherwise it should be sorted out in the previous step
    // 3 ---- 5
    //     4 ----- 8
    while (tupleIndex < totalTuples && intervals[tupleIndex][0] <= newInterval[1]) {
        newInterval[0] = min(newInterval[0], intervals[tupleIndex][0]);
        newInterval[1] = max(newInterval[1], intervals[tupleIndex][1]);
        ++tupleIndex;
    }
    result.push_back(newInterval);


    // 3. add remaining parts
    while (tupleIndex < totalTuples) {
        result.push_back(intervals[tupleIndex]);
        ++tupleIndex;
    }
    return result;
}
```

Problem – 56. Merge Intervals

Medium

LeetCode leetcode.com/problems/merge-intervals

Problem Statement

- You are given an array of intervals, example:

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

- Merge all overlapping intervals. So the output should be:

  `[[1,6],[8,10],[15,18]]`

- Interval [1,3] was merged with [2,6]

Solution – 56. Merge Intervals

Medium

LeetCode leetcode.com/problems/merge-intervals

Solution

- Sort the array based on the beginning of the interval
- In C++, when applying `sort(intervals.begin(), intervals.end())`

  the default comparator compares `vector<vector<int>>` lexicographically:

  - it first compares the first element [0] of each sub-vector

  - if those are equal, it compares the second element [1] and so on
- Go over each interval and compare
- `interval[i][begin] <= interval[i – 1][end]` ? then merge
- To merge, set the current `interval[i][begin] to interval[i -1][begin]` and set the `interval[i][end]` to the maximum value between `interval[i][end]` and `interval[i -1][end]`
- If no merge is necessary, push the previous interval to the result array
- Once the loop finishes, add the last element and return the result

Code – 56. Merge Intervals

Medium

LeetCode leetcode.com/problems/merge-intervals

Code

Time: **O(n log n)** Space: **O(n)**

```
vector<vector<int>> merge(vector<vector<int>>& intervals) {
    if (intervals.empty()) return {};
    sort(intervals.begin(), intervals.end());
    vector<vector<int>> result;

    result.push_back(intervals[0]);

    for (int i = 1; i < intervals.size(); ++i) {
        vector<int>& current = intervals[i];
        vector<int>& previous = result.back();

        // check if they overlap, if so merge...
        // they're sorted, we know that:
        // previous[0] >= current[0]
        // 1 --- 3 (previous)
        //   2 ----- 6 (current)
        if (current[0] <= previous[1]) {
            // merge
            previous[1] = max(previous[1], current[1]);
        } else {
            result.push_back(current);
        }
    }
    return result;
}
```

Problem – 435. Non-overlapping Intervals

Medium

LeetCode leetcode.com/problems/non-overlapping-intervals

Problem Statement

- You are given an array of intervals vector<vector<int>> with start and end, Example:

  `intervals = [[1,2],[2,3],[3,4],[1,3]]`
- The intervals **must not** overlap each over
- You have to remove the minimum number of pairs to make it non-overlapping

Solution – 435. Non-overlapping Intervals

Medium

LeetCode leetcode.com/problems/non-overlapping-intervals

Solution

- Sort the array by the ending time:

  [[1,2],[2,3],[3,4],[1,3]] → [[1,2],[2,3],[1,3],[3,4]]
- In C++ a lambda function can be used with sort:

```
sort(intervals.begin(), intervals.end(), [](const vector<int>& a, const vector<int>&b) {
   return a[1] < b[1];
});
```

- Note that **std::sort** is not stable (opposite of **std::stable_sort**), so there is no guarantees that [2,3] comes before [1,3]. But for this algorithm, it doesn't matter
- Iterate over the array and check overlaps by comparing the end[i] with begin[i - 1]
- If they overlap, logically remove the current pair and count + 1
- Logically removing means just setting the end to compare to the previous element, so "skip" the current interval

Code – 435. Non-overlapping Intervals

Medium

LeetCode leetcode.com/problems/non-overlapping-intervals

Code

Time: **O(n log n)** Space: **O(1)**

```
int eraseOverlapIntervals(vector<vector<int>>& intervals) {
    // sort by the ending time O(log n)
    sort(intervals.begin(), intervals.end(), [](const auto& a, const auto& b) {
            return a[1] < b[1];
    });
    int result= 0;
    int end = intervals[0][1];

    // O(n)
    for (int i = 1; i < intervals.size(); ++i) {
        // does it overlaps?
        if (intervals[i][0] < end) {
            ++result;
        } else {
            // it doesn't overlap, just 'skip'
            // the current interval
            end = intervals[i][1];
        }
    }
    return result;
}
```

Problem – 110. Balanced Binary Tree

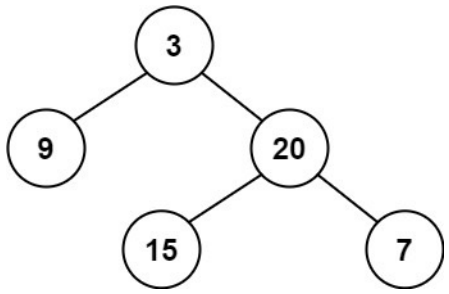
Easy

LeetCode leetcode.com/problems/balanced-binary-tree

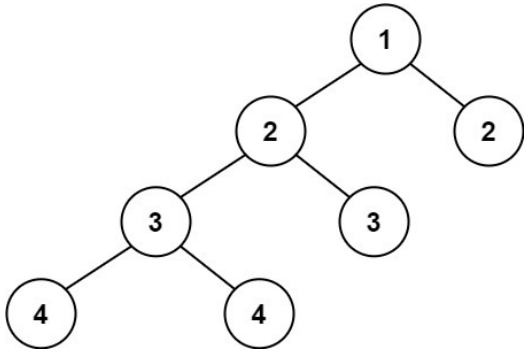
Problem

- You are given the root of a binary tree
- Return true if it is height-balanced
- A tree is height-balanced when the height of two subtrees does not differ by two

Height balanced



Not Height balanced



Problem – 110. Balanced Binary Tree

Easy

LeetCode leetcode.com/problems/balanced-binary-tree

Solution

- Recursive approach: go all the way down
- Calculate the height of the left subtree
- Calculate the height of the right subtree
- Compare both to check if they differ by more than one
- Continue going up the tree to check all the nodes

Problem – 110. Balanced Binary Tree

Easy

LeetCode leetcode.com/problems/balanced-binary-tree

Code

Time: **O(n)** Space: **O(h)** where n is the number of the nodes and h is the height of the tree

```
int checkHeight(TreeNode* node) {
    if (!node) return 0;

    int left = checkHeight(node->left);
    // left tree is unbalanced
    if (left == -1) return -1;

    int right = checkHeight(node->right);
    // right tree is unbalanced
    if (right == -1) return -1;

    // check the different, -1 is unbalanced
    if (abs(left - right) > 1) return -1;

    return max(left, right) + 1;
}

bool isBalanced(TreeNode* root) {
    return checkHeight(root) != -1;
}
```