数组

1. 两数之和

> 给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出 和为目标值 target 的那 两个 整数，并返回它们的数组下标。
>
> 你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。
>
> 你可以按任意顺序返回答案。
>
> 示例 1：
>
> 输入：nums = [2,7,11,15], target = 9
> 输出：[0,1]
> 解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1] 。
> 示例 2：
>
> 输入：nums = [3,2,4], target = 6
> 输出：[1,2]
> 示例 3：
>
> 输入：nums = [3,3], target = 6
> 输出：[0,1]

- 哈希表：创建哈希表，对每个`x`，首先查询哈希表中是否存在`target - x`，若存在则返回，否则将`x`插入到哈希表中，即可保证不会让`x`和自己匹配
- 代码

```
// https://leetcode-cn.com/problems/two-sum/solution/liang-shu-zhi-he-by-leetcode-solution/
// 哈希表
class Solution {
```

```
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        std::unordered_map<int, int> lookup;
        for (int i = 0; i < nums.size(); ++i) {
            if (lookup.find(target - nums[i]) != lookup.end()) {
                return {lookup[target - nums[i]], i};
            }
            lookup[nums[i]] = i;
        }
        return {};
    }
};
```

- 复杂度
  - 时间复杂度：O(n)
  - 空间复杂度：O(n)

15. 三数之和

> 给你一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c，使得 a + b + c = 0？请你找出所有和为 0 且不重复的三元组。
>
> 注意：答案中不可以包含重复的三元组。
>
> 示例 1：
>
> 输入：nums = [-1,0,1,2,-1,-4]
> 输出：[[-1,-1,2],[-1,0,1]]
> 示例 2：
>
> 输入：nums = []
> 输出：[]
> 示例 3：
>
> 输入：nums = [0]
> 输出：[]

- 思路
  1. 先将 nums 排序，时间复杂度为 O(NlogN)。
  2. 固定 3 个指针中最左（最小）元素的指针 k，双指针 i，j 分设在数组索引 (k,len(nums)) 两端。
  3. 双指针 i , j 交替向中间移动，记录对于每个固定指针 k 的所有满足 nums[k] + nums[i] + nums[j] == 0 的 i,j 组合：
     1. 当 nums[k] > 0 时直接break跳出：因为 nums[j] >= nums[i] >= nums[k] > 0，即 3 个元素都大于 0 ，在此固定指针 k 之后不可能再找到结果了。
     2. 当 k > 0且nums[k] == nums[k - 1]时即跳过此元素nums[k]：因为已经将 nums[k - 1] 的所有组合加入到结果中，本次双指针搜索只会得到重复组合。
     3. i，j 分设在数组索引 (k,len(nums)) 两端，当i < j时循环计算s = nums[k] + nums[i] + nums[j]，并按照以下规则执行双指针移动：
        1. 当s < 0时，i += 1并跳过所有重复的nums[i]；
        2. 当s > 0时，j -= 1并跳过所有重复的nums[j]；
        3. 当s == 0时，记录组合[k, i, j]至res，执行i += 1和j -= 1并跳过所有重复的nums[i]和nums[j]，防止记录到重复组合。

- 代码

```
// https://leetcode.cn/problems/3sum/solutions/11525/3sumpai-xu-shuang-zhi-zhen-yi-dong-by-jyd/
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
      vector<vector<int>> res;
      if (nums.size() < 3) { return res; }

      sort(nums.begin(), nums.end());

      for (int i = 0; i < nums.size() - 2; ++i) {
        if (nums[i] > 0) { continue; }
        if (i > 0 && nums[i] == nums[i - 1]) { continue; }
        int j = i + 1;
        int k = nums.size() - 1;
        while (j < k) {
          int sum = nums[i] + nums[j] + nums[k];
          if (sum < 0) {
            ++j;
          } else if (sum > 0) {
            --k;
          } else {
            res.push_back(vector<int>{nums[i], nums[j], nums[k]});
            while (j < k && nums[j] == nums[j + 1]) { ++j; }
            while (j < k && nums[k] == nums[k - 1]) { --k; }
            ++j;
            --k;
          }
        }
      }
      return res;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n^2)$
  - 空间复杂度：$O(1)$

88. 合并两个有序数组

> 给你两个按 **非递减顺序** 排列的整数数组 `nums1` 和 `nums2`，另有两个整数 `m` 和 `n`，分别表示 `nums1` 和 `nums2` 中的元素数目。
>
> 请你 **合并** `nums2` 到 `nums1` 中，使合并后的数组同样按 **非递减顺序** 排列。
>
> **注意：**最终，合并后数组不应由函数返回，而是存储在数组 `nums1` 中。为了应对这种情况，`nums1` 的初始长度为 `m + n`，其中前 `m` 个元素表示应合并的元素，后 `n` 个元素为 `0` ，应忽略。`nums2` 的长度为 `n` 。
>
> **示例 1：**

```
输入：nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
输出：[1,2,2,3,5,6]
解释：需要合并 [1,2,3] 和 [2,5,6] 。
合并结果是 [1,2,2,3,5,6] ，其中斜体加粗标注的为 nums1 中的元素。
```

示例 2：

```
输入：nums1 = [1], m = 1, nums2 = [], n = 0
输出：[1]
解释：需要合并 [1] 和 [] 。
合并结果是 [1] 。
```

示例 3：

```
输入：nums1 = [0], m = 0, nums2 = [1], n = 1
输出：[1]
解释：需要合并的数组是 [] 和 [1] 。
合并结果是 [1] 。
注意，因为 m = 0 ，所以 nums1 中没有元素。nums1 中仅存的 0 仅仅是为了确保合并结果可以顺利存放到 nums1 中。
```

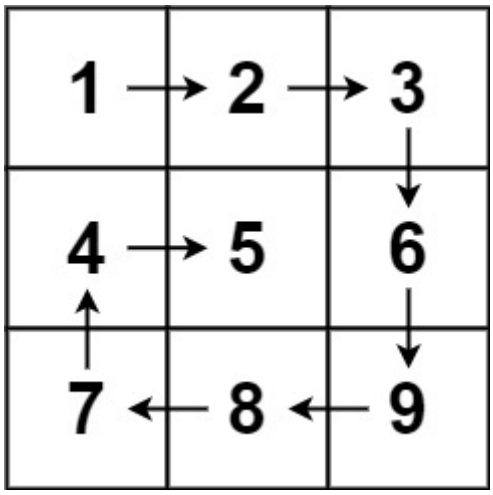
- 思路：逆序双指针合并
- 代码

```cpp
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        int new_idx = m + n - 1;
        --m;
        --n;
        while (m >= 0 && n >= 0) {
            if (nums1[m] >= nums2[n]) {
                nums1[new_idx--] = nums1[m--];
            } else {
                nums1[new_idx--] = nums2[n--];
            }
        }

        while (m >= 0) {
            nums1[new_idx--] = nums1[m--];
        }
        while (n >= 0) {
            nums1[new_idx--] = nums2[n--];
        }
    }
};
```

54. 螺旋矩阵

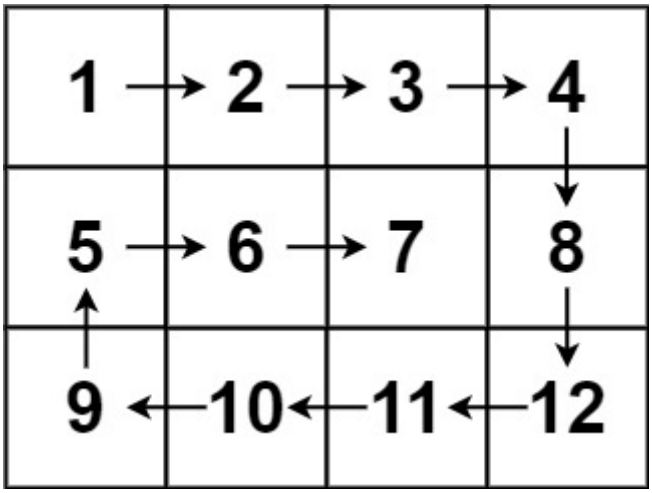
给你一个 `m` 行 `n` 列的矩阵 `matrix` ，请按照 **顺时针螺旋顺序** ，返回矩阵中的所有元素。

示例 1：




```
输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
输出：[1,2,3,6,9,8,7,4,5]
```

示例 2：




```
输入：matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
输出：[1,2,3,4,8,12,11,10,9,5,6,7]
```

思路：

1. 初始设定上下左右边界
2. 按照从左到右、从上到下、从右到左、从下到上 的顺序循环遍历，每次遍历完一次，修改并判断左右边界、上下边界、右左边界、下上边界，确定是否结束

```
// 参考 https://leetcode.cn/problems/spiral-matrix/solutions/7155/cxiang-xi-ti-jie-by-youlookdeliciousc-3/
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        vector<int> res;
        if (matrix.empty() || matrix[0].empty()) { return res;}
        int top = 0, bottom = matrix.size() - 1, left = 0, right =
matrix[0].size() - 1;

        while (true) {
            // 从左到右
```

```
            for (int i = left; i <= right; ++i) {
                res.push_back(matrix[top][i]);
            }
            if (++top > bottom) { break; };

            // 从上到下
            for (int i = top; i <= bottom; ++i) {
                res.push_back(matrix[i][right]);
            }
            if (--right < left) { break; };

            // 从右到左
            for (int i = right; i >= left; --i) {
                res.push_back(matrix[bottom][i]);
            }
            if (--bottom < top) { break; };

            // 从下到上
            for (int i = bottom; i >= top; --i) {
                res.push_back(matrix[i][left]);
            }
            if (++left > right) { break; };
        }
        return res;
    }
};
```

200. 岛屿数量

给你一个由 `'1'`（陆地）和 `'0'`（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

示例 1：

```
输入：grid = [
["1","1","1","1","0"],
["1","1","0","1","0"],
["1","1","0","0","0"],
["0","0","0","0","0"]
]
输出：1
```

示例 2：

```
输入：grid = [
["1","1","0","0","0"],
["1","1","0","0","0"],
["0","0","1","0","0"],
["0","0","0","1","1"]
]
输出：3
```

深搜：https://leetcode.cn/problems/number-of-islands/solutions/211211/dao-yu-lei-wen-ti-de-tong-yong-jie-fa-dfs-bian-li-/

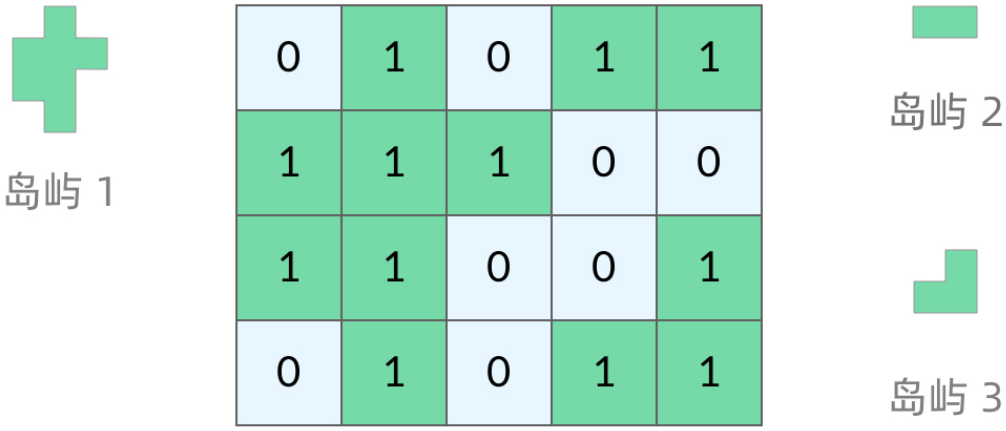
网格类问题的 DFS 遍历方法

网格问题的基本概念

我们首先明确一下岛屿问题中的网格结构是如何定义的，以方便我们后面的讨论。

网格问题是由 m×n 个小方格组成一个网格，每个小方格与其上下左右四个方格认为是相邻的，要在这样的网格上进行某种搜索。

岛屿问题是一类典型的网格问题。每个格子中的数字可能是 0 或者 1。我们把数字为 0 的格子看成海洋格子，数字为 1 的格子看成陆地格子，这样相邻的陆地格子就连接成一个岛屿。



在这样一个设定下，就出现了各种岛屿问题的变种，包括岛屿的数量、面积、周长等。不过这些问题，基本都可以用 DFS 遍历来解决。

DFS 的基本结构

网格结构要比二叉树结构稍微复杂一些，它其实是一种简化版的图结构。要写好网格上的 DFS 遍历，我们首先要理解二叉树上的 DFS 遍历方法，再类比写出网格结构上的 DFS 遍历。我们写的二叉树 DFS 遍历一般是这样的：

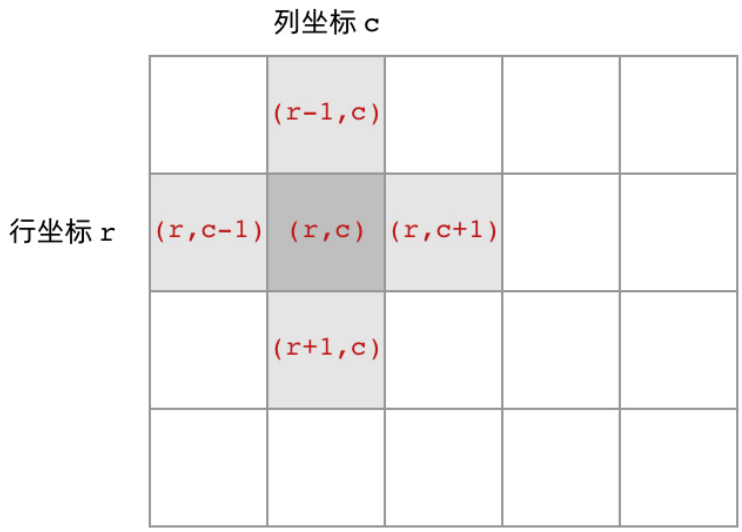
```
void traverse(TreeNode root) {
    // 判断 base case
    if (root == null) {
        return;
    }
    // 访问两个相邻结点：左子结点、右子结点
    traverse(root.left);
    traverse(root.right);
}
```

可以看到，二叉树的 DFS 有两个要素：**「访问相邻结点」**和**「判断 base case」**。

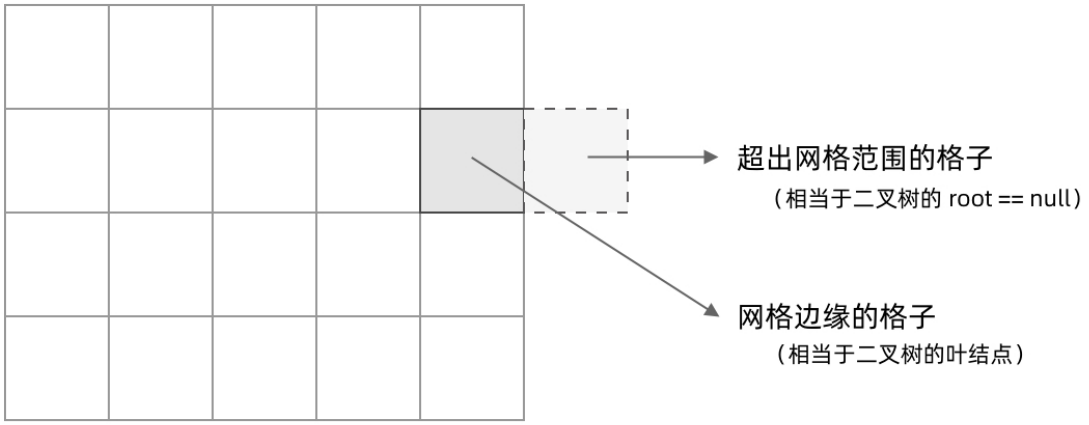
1. 第一个要素是**访问相邻结点**。二叉树的相邻结点非常简单，只有左子结点和右子结点两个。二叉树本身就是一个递归定义的结构：一棵二叉树，它的左子树和右子树也是一棵二叉树。那么我们的 DFS 遍历只需要递归调用左子树和右子树即可。
2. 第二个要素是 **判断 base case**。一般来说，二叉树遍历的 base case 是 root == null。这样一个条件判断其实有两个含义：一方面，这表示 root 指向的子树为空，不需要再往下遍历了。另一方面，在 root == null 的时候及时返回，可以让后面的 root.left 和 root.right 操作不会出现空指针异常。

对于网格上的 DFS，我们完全可以参考二叉树的 DFS，写出网格 DFS 的两个要素：

1. 首先，网格结构中的格子有多少相邻结点？答案是上下左右四个。对于格子 (r, c) 来说（r 和 c 分别代表行坐标和列坐标），四个相邻的格子分别是 (r-1, c)、(r+1, c)、(r, c-1)、(r, c+1)。换句话说，网格结构是「四叉」的。



2. 其次，网格 DFS 中的 base case 是什么？从二叉树的 base case 对应过来，应该是网格中不需要继续遍历、grid[r][c] 会出现数组下标越界异常的格子，也就是那些超出网格范围的格子。



这一点稍微有些反直觉，坐标竟然可以临时超出网格的范围？这种方法我称为「先污染后治理」—— 甭管当前是在哪个格子，先往四个方向走一步再说，如果发现走出了网格范围再赶紧返回。这跟二叉树的遍历方法是一样的，先递归调用，发现 root == null 再返回。

这样，我们得到了**网格 DFS 遍历的框架代码**：

```
void dfs(int[][] grid, int r, int c) {
    // 判断 base case
    // 如果坐标 (r, c) 超出了网格范围，直接返回
```

```
    if (!inArea(grid, r, c)) {
        return;
    }
    // 访问上、下、左、右四个相邻结点
    dfs(grid, r - 1, c);
    dfs(grid, r + 1, c);
    dfs(grid, r, c - 1);
    dfs(grid, r, c + 1);
}

// 判断坐标 (r, c) 是否在网格中
boolean inArea(int[][] grid, int r, int c) {
    return 0 <= r && r < grid.length
            && 0 <= c && c < grid[0].length;
}
```

如何避免重复遍历

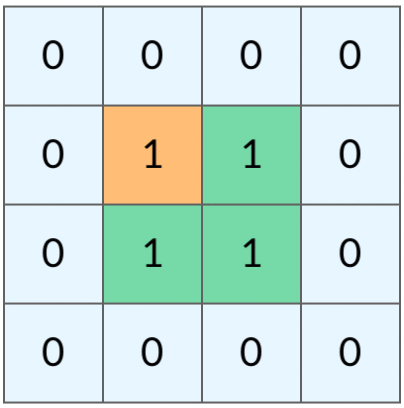
网格结构的 DFS 与二叉树的 DFS 最大的不同之处在于，遍历中可能遇到遍历过的结点。这是因为，网格结构本质上是一个「图」，我们可以把每个格子看成图中的结点，每个结点有向上下左右的四条边。在图中遍历时，自然可能遇到重复遍历结点。

这时候，DFS 可能会不停地「兜圈子」，永远停不下来，如下图所示：

如何避免这样的重复遍历呢？答案是标记已经遍历过的格子。以岛屿问题为例，我们需要在所有值为 1 的陆地格子上做 DFS 遍历。每走过一个陆地格子，就把格子的值改为 2，这样当我们遇到 2 的时候，就知道这是遍历过的格子了。也就是说，每个格子可能取三个值：

0 —— 海洋格子
1 —— 陆地格子（未遍历过）
2 —— 陆地格子（已遍历过）
我们在框架代码中加入避免重复遍历的语句：

```
void dfs(int[][] grid, int r, int c) {
    // 判断 base case
    if (!inArea(grid, r, c)) {
        return;
    }
    // 如果这个格子不是岛屿，直接返回
    if (grid[r][c] != 1) {
        return;
    }
    grid[r][c] = 2; // 将格子标记为「已遍历过」
    // 访问上、下、左、右四个相邻结点
    dfs(grid, r - 1, c);
    dfs(grid, r + 1, c);
    dfs(grid, r, c - 1);
    dfs(grid, r, c + 1);
}

// 判断坐标 (r, c) 是否在网格中
boolean inArea(int[][] grid, int r, int c) {
    return 0 <= r && r < grid.length
            && 0 <= c && c < grid[0].length;
}
```



这样，我们就得到了一个岛屿问题、乃至各种网格问题的通用 DFS 遍历方法。以下所讲的几个例题，其实都只需要在 DFS 遍历框架上稍加修改而已。

小贴士

> 在一些题解中，可能会把「已遍历过的陆地格子」标记为和海洋格子一样的 0，美其名曰「陆地沉没方法」，即遍历完一个陆地格子就让陆地「沉没」为海洋。这种方法看似很巧妙，但实际上有很大隐患，因为这样我们就无法区分「海洋格子」和「已遍历过的陆地格子」了。如果题目更复杂一点，这很容易出 bug。

代码

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int res = 0;
        int rows = grid.size();
        int cols = grid[0].size();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; ++j) {
                if (grid[i][j] == '1') {
                    ++res;
                    dfs(grid, i, j);
                }
            }
        }
        return res;
    }

    void dfs(vector<vector<char>> &grid, int row, int col) {
        int rows = grid.size();
        int cols = grid[0].size();
        if (!inArea(grid, row, col)) {
            return;
        }

        if (grid[row][col] != '1') {
            return;
        }

        grid[row][col] = '2';

        dfs(grid, row - 1, col);
```

```
        dfs(grid, row + 1, col);
        dfs(grid, row, col - 1);
        dfs(grid, row, col + 1);
    }

    bool inArea(vector<vector<char>> &grid, int row, int col) {
        int rows = grid.size();
        int cols = grid[0].size();
        return 0 <= row && row < rows && 0 <= col && col < cols;
    }
};
```

56. 合并区间

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [starti, endi]` 。请你合并所有重叠的区间，并返回 一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间 。

示例 1：

```
输入：intervals = [[1,3],[2,6],[8,10],[15,18]]
输出：[[1,6],[8,10],[15,18]]
解释：区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6].
```

示例 2：

```
输入：intervals = [[1,4],[4,5]]
输出：[[1,5]]
解释：区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

- 排序

```
class Solution {
public:
    // 结果集中最后一个区间表示正在处理的合并区间：
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        vector<vector<int>> res;
        // 先按照区间起始位置排序
        sort(intervals.begin(), intervals.end(), [](vector<int> &left,
vector<int> &right) -> bool {
            return left[0] < right[0];
        });

        // 如果结果数组是空的，或者当前区间的起始位置 > 结果数组中最后区间的终止位置，表示没有交集，则不合并，直接将当前区间加入结果数组。
        for (int i = 0; i < intervals.size(); ++i) {
            if (res.empty() || intervals[i][0] > res.back()[1]) {
                res.push_back(intervals[i]);
            } else {  // 反之将当前区间合并至结果数组的最后区间
                res.back()[1] = max(res.back()[1], intervals[i][1]);
```

```
            }
        }

        return res;
    }
};
```

- 时间复杂度：O(nlogn)，其中 n 为区间的数量。除去排序的开销，我们只需要一次线性扫描，所以主要的时间开销是排序的 O(nlogn)。
- 空间复杂度：O(logn)，其中 n 为区间的数量。这里计算的是存储答案之外，使用的额外空间。O(logn) 即为排序所需要的空间复杂度

252. （会议室）重叠区间

> 给定一个会议时间安排的数组 intervals ，每个会议时间都会包括开始和结束的时间 intervals[i] = [starti, endi] ，请你判断一个人是否能够参加这里面的全部会议。
>
> **示例 1::**
> **输入:** intervals = [[0,30],[5,10],[15,20]]
> **输出:** false
> **解释:** 存在重叠区间，一个人在同一时刻只能参加一个会议。
>
> **示例 2::**
> **输入:** intervals = [[7,10],[2,4]]
> **输出:** true
> **解释:** 不存在重叠区间。

- 排序：因为一个人在同一时刻只能参加一个会议，因此题目实质是判断是否存在重叠区间，这个简单，将区间按照会议开始时间进行排序，然后遍历一遍判断即可。

```
class Solution {
public:
    bool canAttendMeetings(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end(), [](vector<int> &left,
vector<int> &right) -> bool {
            return left[0] < right[0];
        });
        for (int i = 1; i < intervals.size(); ++i) {
            if (intervals[i][0] < intervals[i - 1][1]) {
                return false;
            }
        }
        return true;
    }
};
```

57. 插入区间

> 给你一个 **无重叠的** ，按照区间起始端点排序的区间列表 `intervals`，其中 `intervals[i] = [starti, endi]` 表示第 `i` 个区间的开始和结束，并且 `intervals` 按照 `starti` 升序排列。同样给定一个区间 `newInterval = [start, end]` 表示另一个区间的开始和结束。
>
> 在 `intervals` 中插入区间 `newInterval`，使得 `intervals` 依然按照 `starti` 升序排列，且区间之间不重叠（如果有必要的话，可以合并区间）。
>
> 返回插入之后的 `intervals`。

> **注意** 你不需要原地修改 `intervals`。你可以创建一个新数组然后返回它。

示例 1：

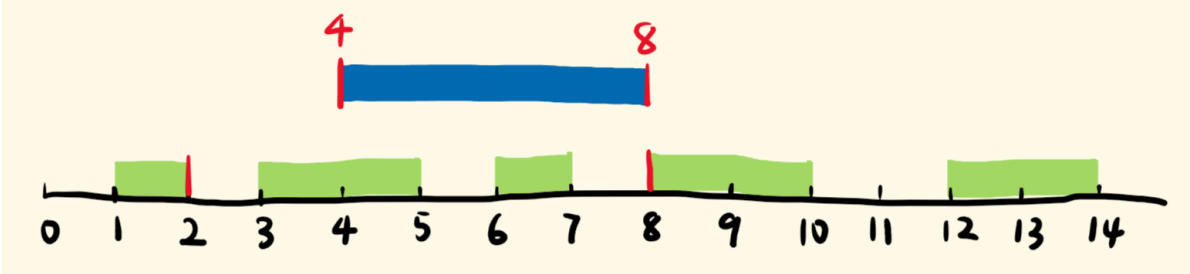
```
输入：intervals = [[1,3],[6,9]], newInterval = [2,5]
输出：[[1,5],[6,9]]
```

示例 2：

```
输入：intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]
输出：[[1,2],[3,10],[12,16]]
解释：这是因为新的区间 [4,8] 与 [3,5],[6,7],[8,10] 重叠。
```

用指针去扫 `intervals`，最多可能有三个阶段：

1. 不重叠的绿区间，在蓝区间的左边
2. 有重叠的绿区间
3. 不重叠的绿区间，在蓝区间的右边



逐个分析

1. 不重叠，需满足：绿区间的右端，位于蓝区间的左端的左边，如 [1,2]。
    - 则当前绿区间，推入 res 数组，指针 +1，考察下一个绿区间。
    - 循环结束时，当前绿区间的屁股，就没落在蓝区间之前，有重叠了，如 [3,5]。
2. 现在看重叠的。我们反过来想，没重叠，就要满足：绿区间的左端，落在蓝区间的屁股的后面，反之就有重叠：绿区间的左端 <= 蓝区间的右端，极端的例子就是 [8,10]。
    1. 和蓝有重叠的区间，会合并成一个区间：左端取蓝绿左端的较小者，右端取蓝绿右端的较大者，不断更新给蓝区间。
    2. 循环结束时，将蓝区间（它是合并后的新区间）推入 res 数组。
3. 剩下的，都在蓝区间右边，不重叠。不用额外判断，依次推入 res 数组。

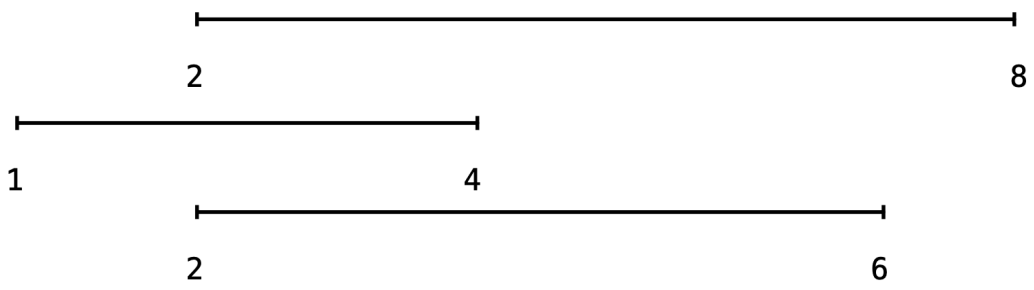
```
class Solution {
public:
    vector<vector<int>> insert(vector<vector<int>>& intervals, vector<int>&
newInterval) {
        vector<vector<int>> res;
        int i = 0;
        // newInterval区间左边不想交的绿色区间
        while (i < intervals.size() && intervals[i][1] < newInterval[0]) {
            res.push_back(intervals[i++]);
        }

        // newInterval区间 重叠的 蓝色区间
        while (i < intervals.size() && intervals[i][0] <= newInterval[1]) {
            newInterval[0] = min(intervals[i][0], newInterval[0]);
            newInterval[1] = max(intervals[i][1], newInterval[1]);
```

```
            i++;
        }
        res.push_back(newInterval);

        // newInterval区间右边不想交的绿色区间
        while (i < intervals.size()) {
            res.push_back(intervals[i++]);
        }

        return res;
    }
};
```

1288. 删除被覆盖区间

> 给你一个区间列表，请你删除列表中被其他区间所覆盖的区间。
>
> 只有当 `c <= a` 且 `b <= d` 时，我们才认为区间 `[a,b)` 被区间 `[c,d)` 覆盖。
>
> 在完成所有删除操作后，请你返回列表中剩余区间的数目。
>
> **示例：**
>
> ```
> 输入：intervals = [[1,4],[3,6],[2,8]]
> 输出：2
> 解释：区间 [3,6] 被区间 [2,8] 覆盖，所以它被删除了。
> ```

当区间左端点相同的时候，右端点靠后的应该放在前面。



在区间左端点相等的时候，应该让右端点大的先出现。
如图，让区间 [2, 8] 先出现，这样接下来扫描到 [2, 6] 的时候，
就能检测到 [2, 6] 是被覆盖的区间了。

```
class Solution {
public:
    int removeCoveredIntervals(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end(), [](vector<int> &left,
vector<int> &right) -> bool {
            if (left[0] != right[0]) {
```

```
                return left[0] < right[0];
            } else {
                return left[1] > right[1];
            }
        });

        int removedCnt = intervals.size();
        int rMax = intervals[0][1];
        for (int i = 1; i < intervals.size(); ++i) {
            if (intervals[i][1] <= rMax) {
                --removedCnt;
            } else {
                rMax = intervals[i][1];
            }
        }

        return removedCnt;
    }
};
```