WEEK6

Contents

Reorder List

You are given the head of a singly linked-list. The list can be represented as:

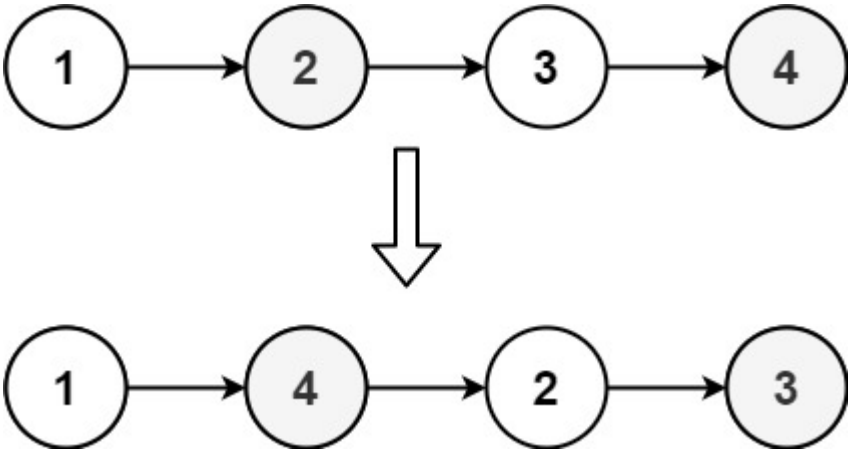
$L_0 \rightarrow L_1 \rightarrow \ldots \rightarrow L_{n-1} \rightarrow L_n$

Reorder the list to be on the following form:

$L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \ldots$

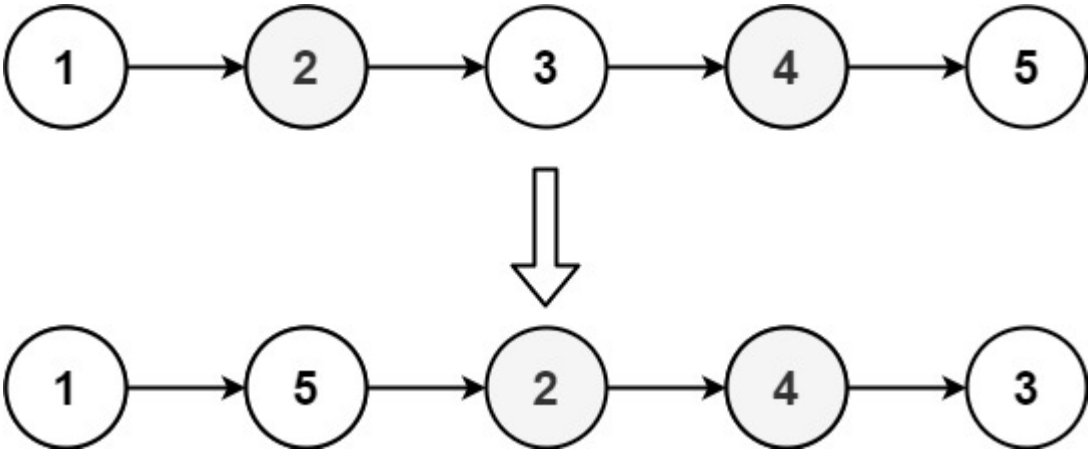
You may not modify the values in the list's nodes. Only nodes themselves may be changed.

Example 1:



Input: head = [1,2,3,4] **Output:** [1,4,2,3]

Example 2:



Input: head = [1,2,3,4,5] **Output:** [1,5,2,4,3]

Constraints:

- The number of nodes in the list is in the range $[1, 5 * 10^4]$.
- $1 <= Node.val <= 1000$

/** * Definition for singly-linked list. * struct ListNode { * int val; * ListNode *next; * ListNode() : val(0), next(nullptr) {} * ListNode(int x) : val(x), next(nullptr) {} * ListNode(int x, ListNode *next) : val(x), next(next) {} * }; */ class Solution { public: void reorderList(ListNode* head) { } };

Clone Graph

Given a reference of a node in a **connected** undirected graph.

Return a **deep copy** (clone) of the graph.

Each node in the graph contains a value (`int`) and a list (`List[Node]`) of its neighbors.

```
class Node { public int val; public List<Node> neighbors; }
```

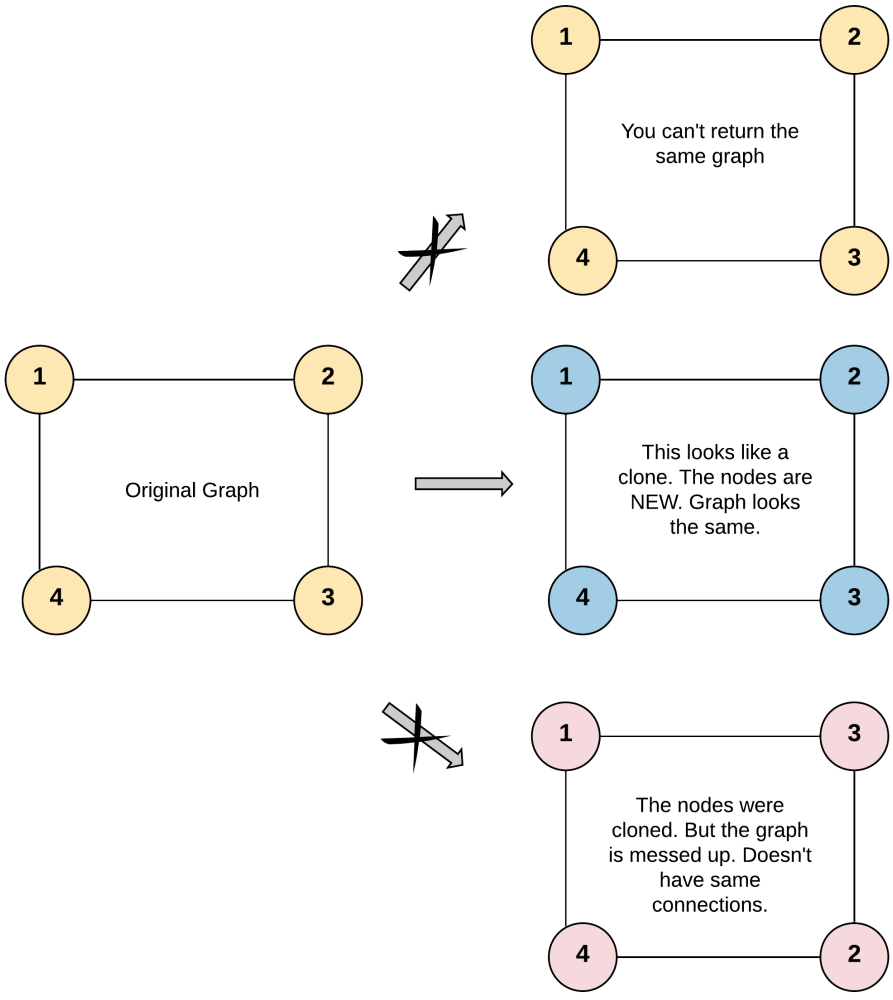
Test case format:

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with `val == 1`, the second node with `val == 2`, and so on. The graph is represented in the test case using an adjacency list.

An adjacency list is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with `val = 1`. You must return the **copy of the given node** as a reference to the cloned graph.

Example 1:



Input: `adjList = [[2,4],[1,3],[2,4],[1,3]]` **Output:** `[[2,4],[1,3],[2,4],[1,3]]` **Explanation:** `There are 4 nodes in the graph. 1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4). 2nd node (val = 2)'s`

neighbors are 1st node (val = 1) and 3rd node (val = 3). 3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4). 4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

Example 2:

Input: adjList = [[]] **Output:** [[]] **Explanation:** Note that the input contains one empty list. The graph consists of only one node with val = 1 and it does not have any neighbors.

Example 3:

Input: adjList = [] **Output:** [] **Explanation:** This an empty graph, it does not have any nodes.

Constraints:

- The number of nodes in the graph is in the range [0, 100].
- 1 <= Node.val <= 100
- Node.val is unique for each node.
- There are no repeated edges and no self-loops in the graph.
- The Graph is connected and all nodes can be visited starting from the given node.

/* // Definition for a Node. class Node { public: int val; vector neighbors; Node() { val = 0; neighbors = vector(); } Node(int _val) { val = _val; neighbors = vector(); } Node(int _val, vector _neighbors) { val = _val; neighbors = _neighbors; } }; */ class Solution { public: Node* cloneGraph(Node* node) { } };

Pacific Atlantic Water Flow

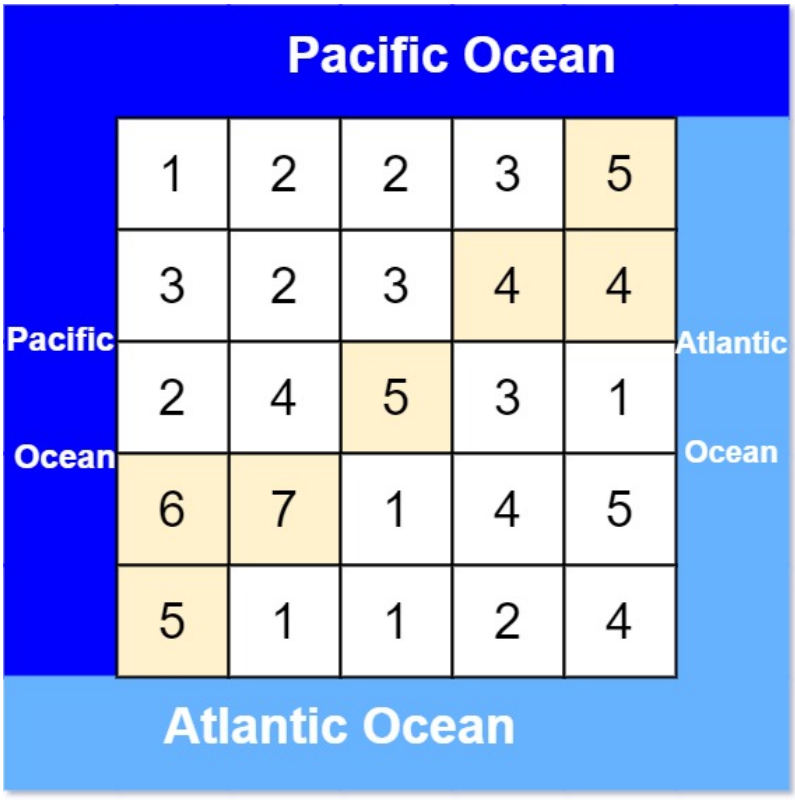
There is an `m x n` rectangular island that borders both the **Pacific Ocean** and **Atlantic Ocean**. The **Pacific Ocean** touches the island's left and top edges, and the **Atlantic Ocean** touches the island's right and bottom edges.

The island is partitioned into a grid of square cells. You are given an `m x n` integer matrix `heights` where `heights[r][c]` represents the **height above sea level** of the cell at coordinate `(r, c)`.

The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is **less than or equal to** the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

Return *a **2D list** of grid coordinates* `result` *where* `result[i] = [r_i, c_i]` *denotes that rain water can flow from cell* `(r_i, c_i)` *to **both** the Pacific and Atlantic oceans*.

Example 1:



Input: heights = [[1,2,2,3,5],[3,2,3,4,4],[2,4,5,3,1],[6,7,1,4,5],[5,1,1,2,4]] **Output:** [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]] **Explanation:** The following cells can flow to the Pacific and Atlantic oceans, as shown below: [0,4]: [0,4] -> Pacific Ocean [0,4] -> Atlantic Ocean [1,3]: [1,3] -> [0,3] -> Pacific Ocean [1,3] -> [1,4] -> Atlantic Ocean [1,4]: [1,4] -> [1,3] -> [0,3] -> Pacific Ocean [1,4] -> Atlantic Ocean [2,2]: [2,2] -> [1,2] -> [0,2] -> Pacific Ocean [2,2] -> [2,3] -> [2,4] -> Atlantic Ocean [3,0]: [3,0] -> Pacific Ocean [3,0] -> [4,0] -> Atlantic Ocean [3,1]: [3,1] -> [3,0] -> Pacific Ocean [3,1] -> [4,1] -> Atlantic Ocean [4,0]: [4,0] -> Pacific Ocean [4,0] -> Atlantic Ocean Note that there are other possible paths for these cells to flow to the Pacific and Atlantic oceans.

Example 2:

Input: heights = [[1]] **Output:** [[0,0]] **Explanation:** The water can flow from the only cell to the Pacific and Atlantic oceans.

Constraints:

- `m == heights.length`
- `n == heights[r].length`
- `1 <= m, n <= 200`
- `0 <= heights[r][c] <= 10^5`

class Solution { public: vector> pacificAtlantic(vector>& heights) { } };

Number of Islands

Given an `m x n` 2D binary grid `grid` which represents a map of `'1'`s (land) and `'0'`s (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
Input: grid = [ ["1","1","1","1","0"], ["1","1","0","1","0"], ["1","1","0","0","0"], ["0","0","0","0","0"] ]
Output: 1
```

Example 2:

```
Input: grid = [ ["1","1","0","0","0"], ["1","1","0","0","0"], ["0","0","1","0","0"], ["0","0","0","1","1"] ]
Output: 3
```

Constraints:

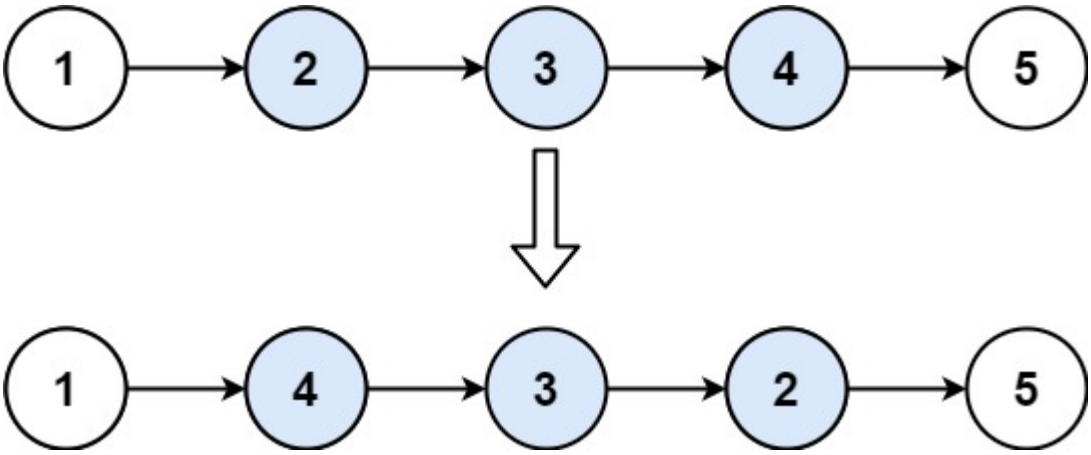
- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 300`
- `grid[i][j]` is `'0'` or `'1'`.

class Solution { public: int numIslands(vector>& grid) { } };

Reverse Linked List II

Given the `head` of a singly linked list and two integers `left` and `right` where `left <= right`, reverse the nodes of the list from position `left` to position `right`, and return *the reversed list*.

Example 1:



Input: `head = [1,2,3,4,5], left = 2, right = 4` **Output:** `[1,4,3,2,5]`

Example 2:

Input: `head = [5], left = 1, right = 1` **Output:** `[5]`

Constraints:

- The number of nodes in the list is `n`.
- `1 <= n <= 500`
- `-500 <= Node.val <= 500`
- `1 <= left <= right <= n`

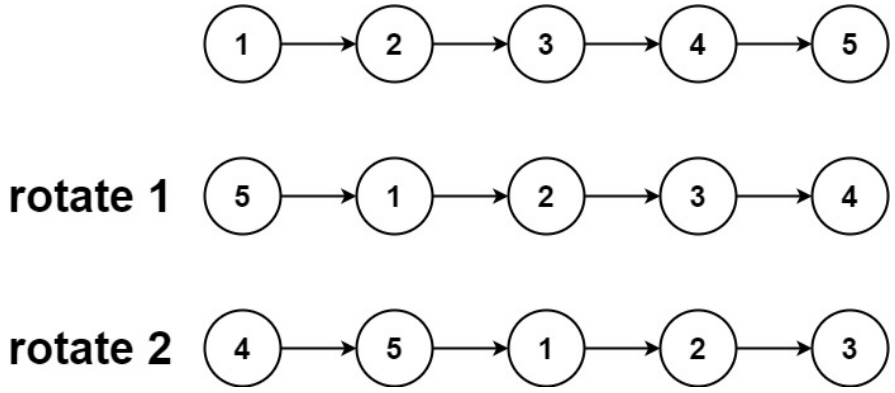
Follow up: Could you do it in one pass?

/** * Definition for singly-linked list. * struct ListNode { * int val; * ListNode *next; * ListNode() : val(0), next(nullptr) {} * ListNode(int x) : val(x), next(nullptr) {} * ListNode(int x, ListNode *next) : val(x), next(next) {} * }; */ class Solution { public: ListNode* reverseBetween(ListNode* head, int left, int right) { } };

Rotate List

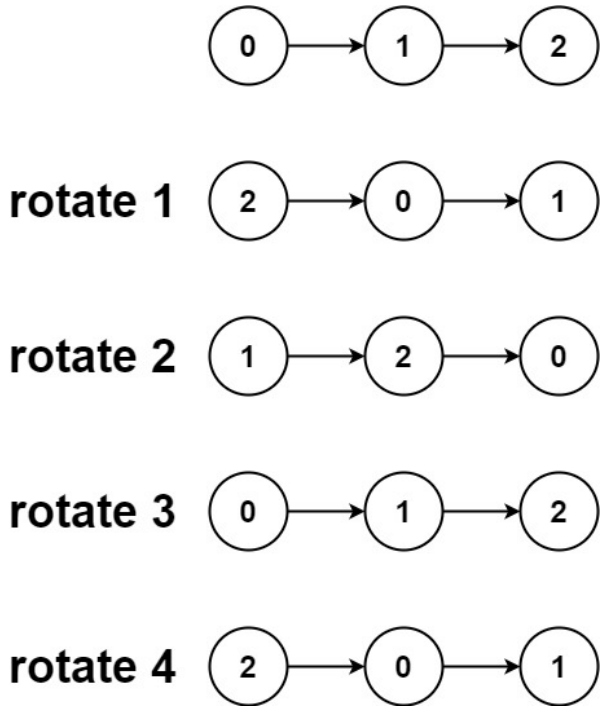
Given the head of a linked list, rotate the list to the right by k places.

Example 1:



Input: head = [1,2,3,4,5], k = 2 **Output:** [4,5,1,2,3]

Example 2:



Input: head = [0,1,2], k = 4 **Output:** [2,0,1]

Constraints:

- The number of nodes in the list is in the range [0, 500].
- -100 <= Node.val <= 100
- 0 <= k <= $2 * 10^9$

```
/** * Definition for singly-linked list. * struct ListNode { * int val; * ListNode *next; * ListNode() : val(0), next(nullptr) {} * ListNode(int x) : val(x), next(nullptr) {} * ListNode(int x, ListNode *next) : val(x), next(next) {} * }; */ class Solution { public: ListNode* rotateRight(ListNode* head, int k) { } };
```

Swap Nodes in Pairs

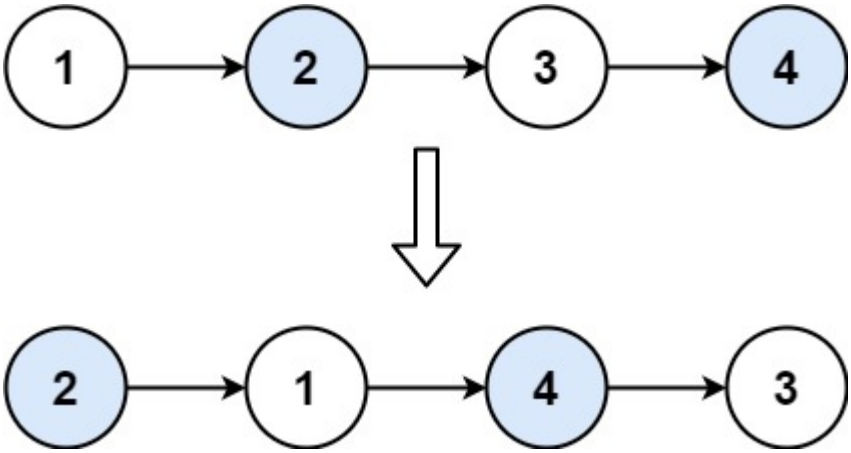
Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

Example 1:

Input: head = [1,2,3,4]

Output: [2,1,4,3]

Explanation:



Example 2:

Input: head = []

Output: []

Example 3:

Input: head = [1]

Output: [1]

Example 4:

Input: head = [1,2,3]

Output: [2,1,3]

Constraints:

- The number of nodes in the list is in the range `[0, 100]`.
- `0 <= Node.val <= 100`

/** * Definition for singly-linked list. * struct ListNode { * int val; * ListNode *next; * ListNode() : val(0), next(nullptr) {} * ListNode(int x) : val(x), next(nullptr) {} * ListNode(int x, ListNode *next) : val(x), next(next) {} * }; */ class Solution { public: ListNode* swapPairs(ListNode* head) { } };

Odd Even Linked List

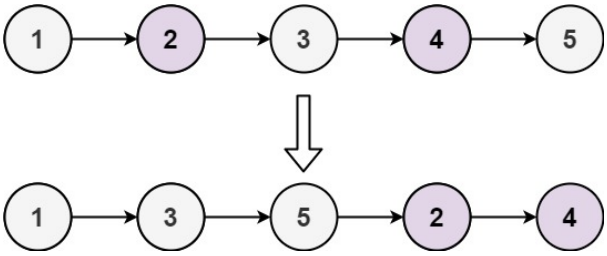
Given the `head` of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

The **first** node is considered **odd**, and the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

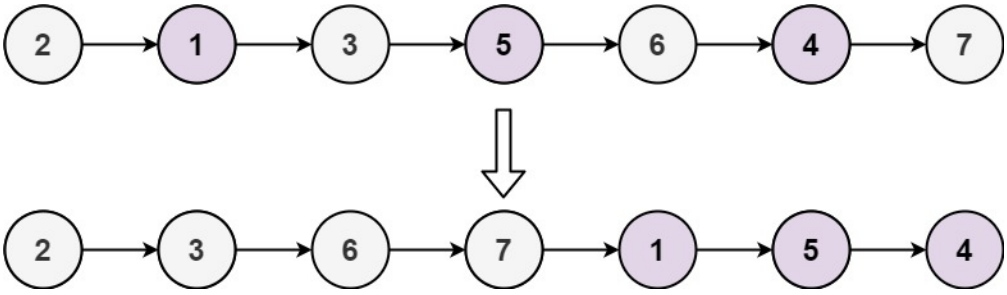
You must solve the problem in `O(1)` extra space complexity and `O(n)` time complexity.

Example 1:



Input: `head = [1,2,3,4,5]` **Output:** `[1,3,5,2,4]`

Example 2:



Input: `head = [2,1,3,5,6,4,7]` **Output:** `[2,3,6,7,1,5,4]`

Constraints:

- The number of nodes in the linked list is in the range `[0, 10^4]`.
- `-10^6 <= Node.val <= 10^6`

/** * Definition for singly-linked list. * struct ListNode { * int val; * ListNode *next; * ListNode() : val(0), next(nullptr) {} * ListNode(int x) : val(x), next(nullptr) {} * ListNode(int x, ListNode *next) : val(x), next(next) {} * }; */ class Solution { public: ListNode* oddEvenList(ListNode* head) { } };

Kth Smallest Element in a Sorted Matrix

Given an `n x n matrix` where each of the rows and columns is sorted in ascending order, return *the* k^{th} *smallest element in the matrix*.

Note that it is the k^{th} smallest element **in the sorted order**, not the k^{th} **distinct** element.

You must find a solution with a memory complexity better than $O(n^2)$.

Example 1:

Input: matrix = [[1,5,9],[10,11,13],[12,13,15]], k = 8 **Output:** 13 **Explanation:** The elements in the matrix are [1,5,9,10,11,12,13,**13**,15], and the 8^{th} smallest number is 13

Example 2:

Input: matrix = [[-5]], k = 1 **Output:** -5

Constraints:

- `n == matrix.length == matrix[i].length`
- `1 <= n <= 300`
- -10^9 `<= matrix[i][j] <=` 10^9
- All the rows and columns of `matrix` are **guaranteed** to be sorted in **non-decreasing order**.
- `1 <= k <=` n^2

Follow up:

- Could you solve the problem with a constant memory (i.e., $O(1)$ memory complexity)?
- Could you solve the problem in $O(n)$ time complexity? The solution may be too advanced for an interview but you may find reading this paper fun.

class Solution { public: int kthSmallest(vector>& matrix, int k) { } };

Find K Pairs with Smallest Sums

You are given two integer arrays `nums1` and `nums2` sorted in **non-decreasing order** and an integer `k`.

Define a pair `(u, v)` which consists of one element from the first array and one element from the second array.

Return *the* k *pairs* $(u_1, v_1), (u_2, v_2), ..., (u_k, v_k)$ *with the smallest sums*.

Example 1:

Input: `nums1 = [1,7,11], nums2 = [2,4,6], k = 3` **Output:** `[[1,2],[1,4],[1,6]]` **Explanation:** `The first 3 pairs are returned from the sequence: [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]`

Example 2:

Input: `nums1 = [1,1,2], nums2 = [1,2,3], k = 2` **Output:** `[[1,1],[1,1]]` **Explanation:** `The first 2 pairs are returned from the sequence: [1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]`

Constraints:

- `1 <= nums1.length, nums2.length <=` 10^5
- `-`10^9 `<= nums1[i], nums2[i] <=` 10^9
- `nums1` and `nums2` both are sorted in **non-decreasing order**.
- `1 <= k <=` 10^4
- `k <= nums1.length * nums2.length`

class Solution { public: vector> kSmallestPairs(vector& nums1, vector& nums2, int k) { } };

Merge Intervals

Given an array of `intervals` where `intervals[i] = [start`$_i$`, end`$_i$`]`, merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

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] overlap, merge them into [1,6].`

Example 2:

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

Constraints:

- `1 <= intervals.length <= 10`^{4}
- `intervals[i].length == 2`
- `0 <= start`$_i$ `<= end`$_i$ `<= 10`^{4}

class Solution { public: vector> merge(vector>& intervals) { } };

Interval List Intersections

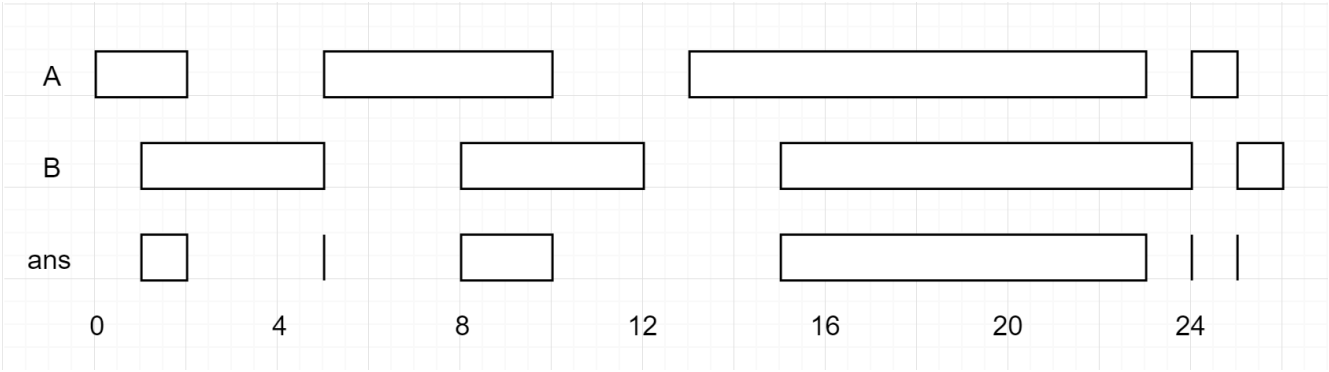
You are given two lists of closed intervals, `firstList` and `secondList`, where `firstList[i] = [start`$_i$`, end`$_i$`]` and `secondList[j] = [start`$_j$`, end`$_j$`]`. Each list of intervals is pairwise **disjoint** and in **sorted order**.

Return *the intersection of these two interval lists*.

A **closed interval** `[a, b]` (with a `<=` b) denotes the set of real numbers x with a `<=` x `<=` b.

The **intersection** of two closed intervals is a set of real numbers that are either empty or represented as a closed interval. For example, the intersection of `[1, 3]` and `[2, 4]` is `[2, 3]`.

Example 1:



Input: `firstList = [[0,2],[5,10],[13,23],[24,25]], secondList = [[1,5],[8,12],[15,24],[25,26]]` **Output:** `[[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]`

Example 2:

Input: `firstList = [[1,3],[5,9]], secondList = []` **Output:** `[]`

Constraints:

- `0 <= firstList.length, secondList.length <= 1000`
- `firstList.length + secondList.length >= 1`
- `0 <= start`$_i$ `< end`$_i$ `<=` 10^9
- `end`$_i$ `< start`$_{i+1}$
- `0 <= start`$_j$ `< end`$_j$ `<=` 10^9
- `end`$_j$ `< start`$_{j+1}$

class Solution { public: vector> intervalIntersection(vector>& firstList, vector>& secondList) { } };

Non-overlapping Intervals

Given an array of intervals `intervals` where `intervals[i] = [start`$_i$`, end`$_i$`]`, return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.

Example 1:

Input: `intervals = [[1,2],[2,3],[3,4],[1,3]]` **Output:** `1` **Explanation:** `[1,3] can be removed and the rest of the intervals are non-overlapping.`

Example 2:

Input: `intervals = [[1,2],[1,2],[1,2]]` **Output:** `2` **Explanation:** `You need to remove two [1,2] to make the rest of the intervals non-overlapping.`

Example 3:

Input: `intervals = [[1,2],[2,3]]` **Output:** `0` **Explanation:** `You don't need to remove any of the intervals since they're already non-overlapping.`

Constraints:

- `1 <= intervals.length <= 10`^{5}
- `intervals[i].length == 2`
- `-5 * 10`^{4} `<= start`$_i$ `< end`$_i$ `<= 5 * 10`^{4}

class Solution { public: int eraseOverlapIntervals(vector>& intervals) { } };