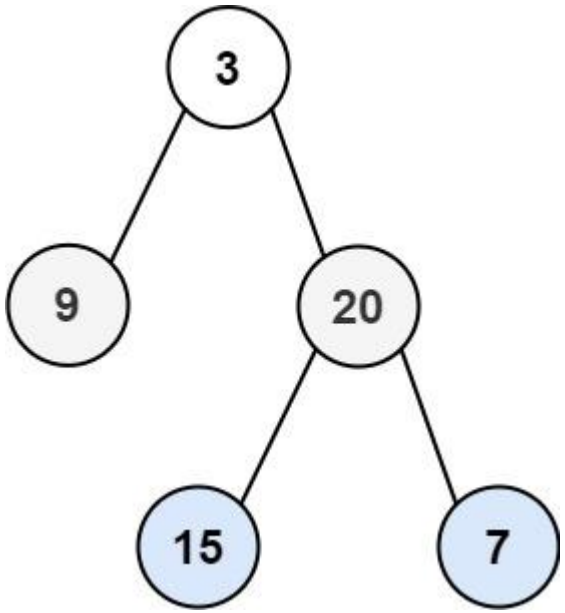
Binary Tree Zigzag Level Order Traversal

Given the `root` of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

Example 1:




```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[20,9],[15,7]]
```

Example 2:

```
Input: root = [1]
Output: [[1]]
```

Example 3:

```
Input: root = []
Output: []
```

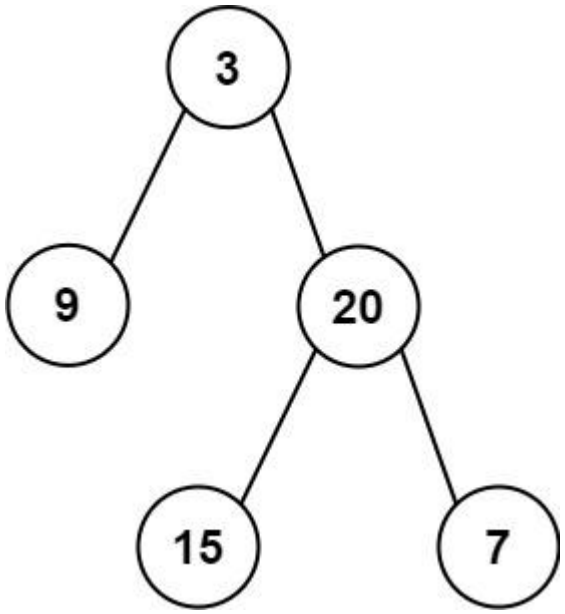
Constraints:

- The number of nodes in the tree is in the range `[0, 2000]`.
- `-100 <= Node.val <= 100`

Construct Binary Tree from Inorder and Postorder Traversal

Given two integer arrays `inorder` and `postorder` where `inorder` is the inorder traversal of a binary tree and `postorder` is the postorder traversal of the same tree, construct and return *the binary tree*.

Example 1:




```
Input: inorder = [9,3,15,20,7], postorder = [9,15,7,20,3]
Output: [3,9,20,null,null,15,7]
```

Example 2:

```
Input: inorder = [-1], postorder = [-1]
Output: [-1]
```

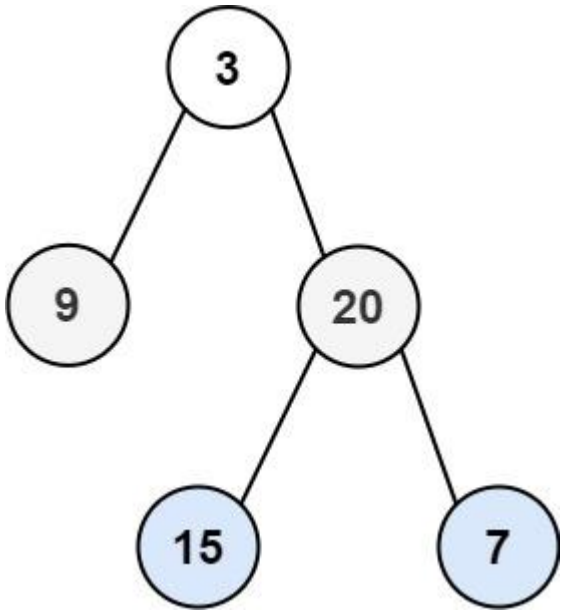
Constraints:

- `1 <= inorder.length <= 3000`
- `postorder.length == inorder.length`
- `-3000 <= inorder[i], postorder[i] <= 3000`
- `inorder` and `postorder` consist of **unique** values.
- Each value of `postorder` also appears in `inorder`.
- `inorder` is **guaranteed** to be the inorder traversal of the tree.
- `postorder` is **guaranteed** to be the postorder traversal of the tree.

Binary Tree Level Order Traversal II

Given the `root` of a binary tree, return *the bottom-up level order traversal of its nodes' values*. (i.e., from left to right, level by level from leaf to root).

Example 1:




```
Input: root = [3,9,20,null,null,15,7]
Output: [[15,7],[9,20],[3]]
```

Example 2:

```
Input: root = [1]
Output: [[1]]
```

Example 3:

```
Input: root = []
Output: []
```

Constraints:

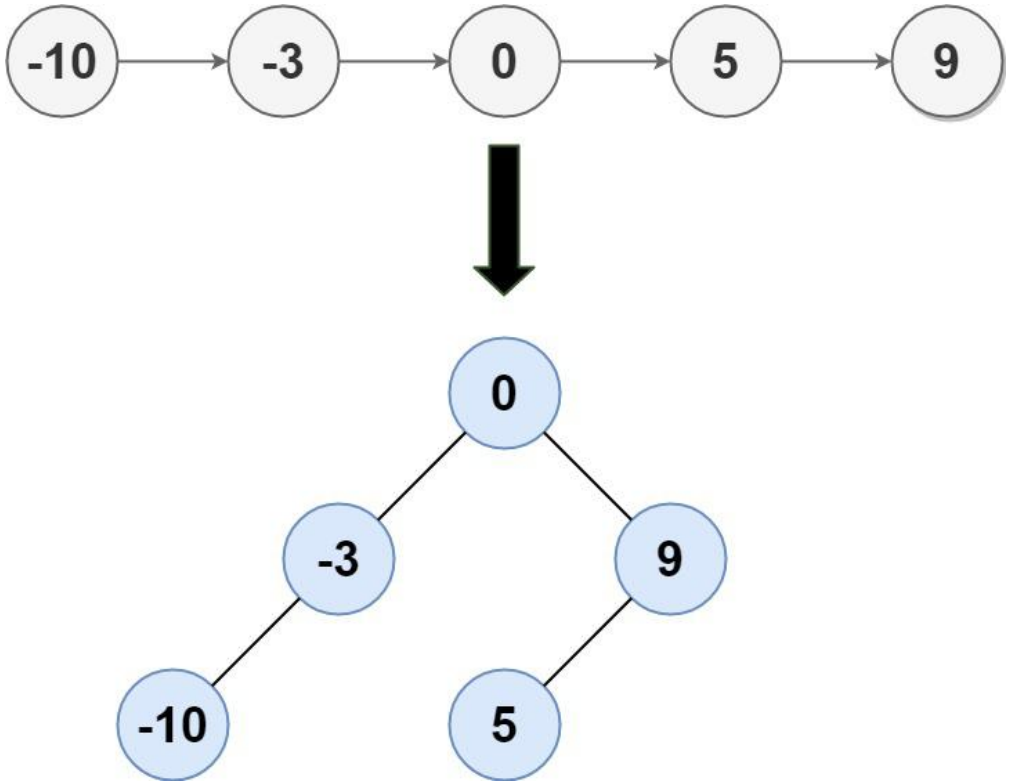
- The number of nodes in the tree is in the range `[0, 2000]`.
- `-1000 <= Node.val <= 1000`

Convert Sorted List to Binary Search Tree

Given the `head` of a singly linked list where elements are **sorted in ascending order**, convert it to a height balanced BST.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

Example 1:




```
Input: head = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
```

```
Explanation: One possible answer is [0,-3,9,-10,null,5], which represents the shown height
balanced BST.
```

Example 2:

```
Input: head = []
Output: []
```

Constraints:

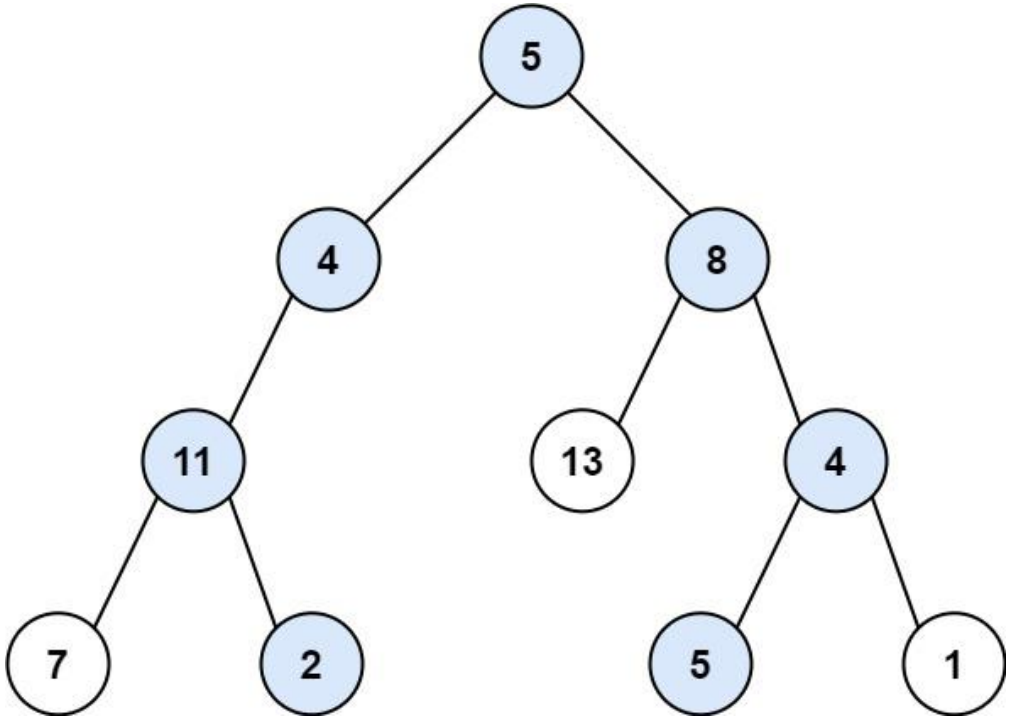
- The number of nodes in `head` is in the range `[0, 2 * 10^4]`.
- `-10^5 <= Node.val <= 10^5`

Path Sum II

Given the `root` of a binary tree and an integer `targetSum`, return *all **root-to-leaf** paths where the sum of the node values in the path equals* `targetSum`*. Each path should be returned as a list of the node **values**, not node references*.

A **root-to-leaf** path is a path starting from the root and ending at any leaf node. A **leaf** is a node with no children.

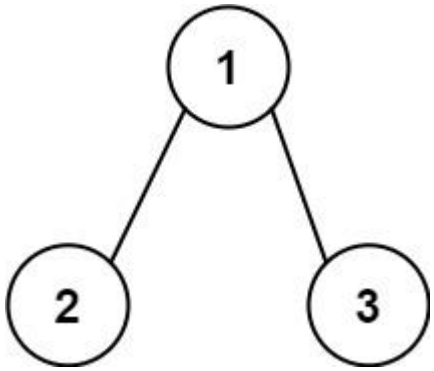
Example 1:




```
Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
Output: [[5,4,11,2],[5,8,4,5]]
Explanation: There are two paths whose sum equals targetSum:
```

```
5 + 4 + 11 + 2 = 22
5 + 8 + 4 + 5 = 22
```

Example 2:




```
Input: root = [1,2,3], targetSum = 5
Output: []
```

Example 3:

```
Input: root = [1,2], targetSum = 0
Output: []
```

Constraints:

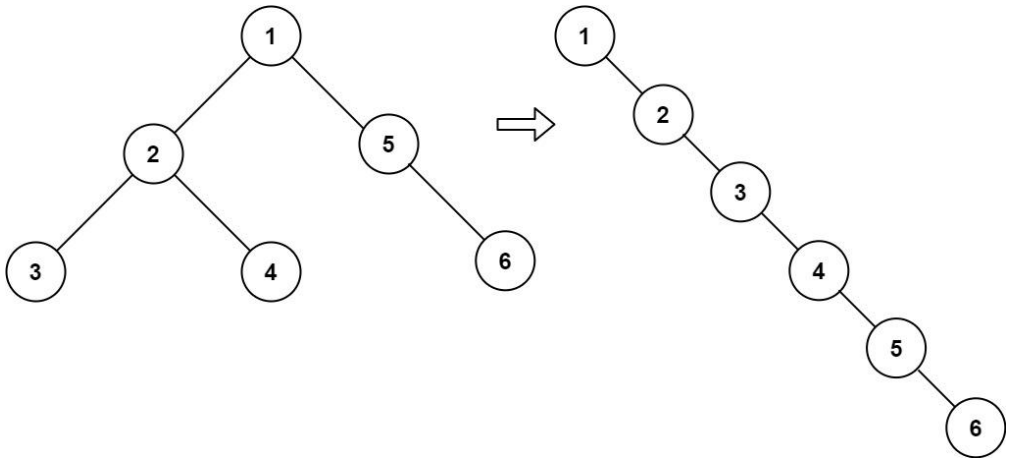
- The number of nodes in the tree is in the range `[0, 5000]`.
- `-1000 <= Node.val <= 1000`
- `-1000 <= targetSum <= 1000`

Flatten Binary Tree to Linked List

Given the `root` of a binary tree, flatten the tree into a "linked list":

- The "linked list" should use the same `TreeNode` class where the `right` child pointer points to the next node in the list and the `left` child pointer is always `null`.
- The "linked list" should be in the same order as a **pre-order traversal** of the binary tree.

Example 1:




```
Input: root = [1,2,5,3,4,null,6]
Output: [1,null,2,null,3,null,4,null,5,null,6]
```

Example 2:

```
Input: root = []
Output: []
```

Example 3:

```
Input: root = [0]
Output: [0]
```

Constraints:

- The number of nodes in the tree is in the range `[0, 2000]`.
- `-100 <= Node.val <= 100`

Follow up: Can you flatten the tree in-place (with `O(1)` extra space)?

Populating Next Right Pointers in Each Node

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Example 1:

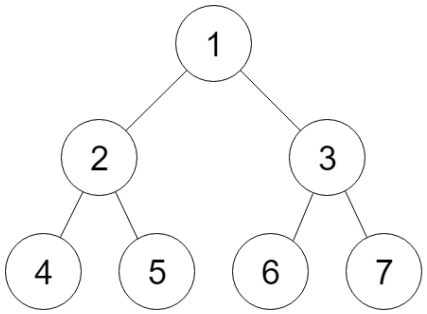


Figure A

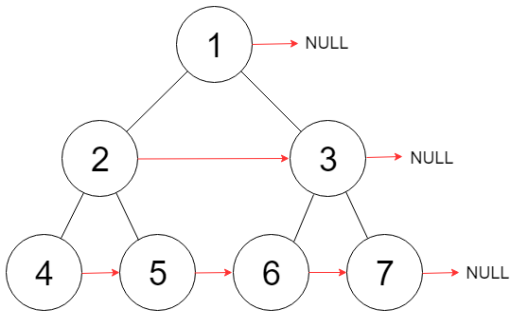


Figure B

```
Input: root = [1,2,3,4,5,6,7]
Output: [1,#,2,3,#,4,5,6,7,#]
Explanation: Given the above perfect binary tree (Figure A), your function should populate
each next pointer to point to its next right node, just like in Figure B. The serialized
```

output is in level order as connected by the next pointers, with '#' signifying the end of each level.

Example 2:

```
Input: root = []
Output: []
```

Constraints:

- The number of nodes in the tree is in the range $[0, 2^{12} - 1]$.
- $-1000 <= Node.val <= 1000$

Follow-up:

- You may only use constant extra space.
- The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem.

Triangle

Given a `triangle` array, return *the minimum path sum from top to bottom*.

For each step, you may move to an adjacent number of the row below. More formally, if you are on index `i` on the current row, you may move to either index `i` or index `i + 1` on the next row.

Example 1:

```
Input: triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]
Output: 11
Explanation: The triangle looks like:
   2
  3 4
 6 5 7
4 1 8 3
The minimum path sum from top to bottom is 2 + 3 + 5 + 1 = 11 (underlined above).
```

Example 2:

```
Input: triangle = [[-10]]
Output: -10
```

Constraints:

- `1 <= triangle.length <= 200`
- `triangle[0].length == 1`
- `triangle[i].length == triangle[i - 1].length + 1`
- -10^4 `<= triangle[i][j] <=` 10^4

Follow up: Could you do this using only `O(n)` extra space, where `n` is the total number of rows in the triangle?

Sum Root to Leaf Numbers

You are given the `root` of a binary tree containing digits from `0` to `9` only.

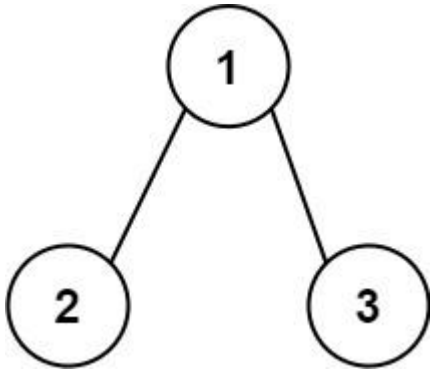
Each root-to-leaf path in the tree represents a number.

- For example, the root-to-leaf path `1 -> 2 -> 3` represents the number `123`.

Return *the total sum of all root-to-leaf numbers*. Test cases are generated so that the answer will fit in a **32-bit** integer.

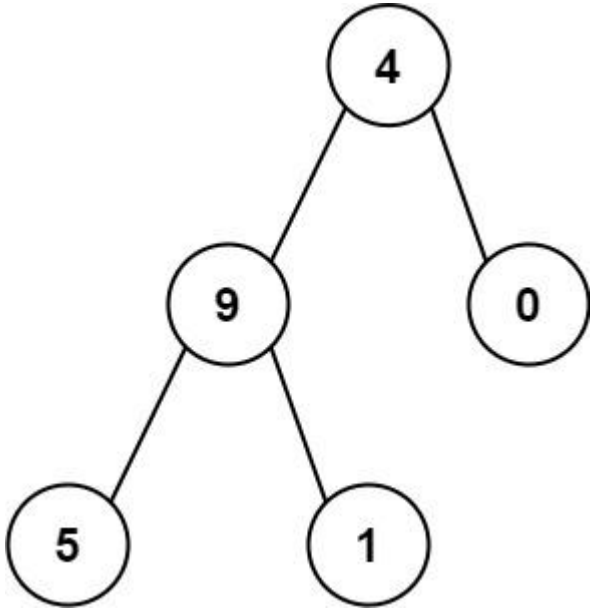
A **leaf** node is a node with no children.

Example 1:




```
Input: root = [1,2,3]
Output: 25
Explanation:
The root-to-leaf path 1->2 represents the number 12.
The root-to-leaf path 1->3 represents the number 13.
Therefore, sum = 12 + 13 = 25.
```

Example 2:




```
Input: root = [4,9,0,5,1]
Output: 1026
Explanation:
The root-to-leaf path 4->9->5 represents the number 495.
The root-to-leaf path 4->9->1 represents the number 491.
The root-to-leaf path 4->0 represents the number 40.
Therefore, sum = 495 + 491 + 40 = 1026.
```

Constraints:

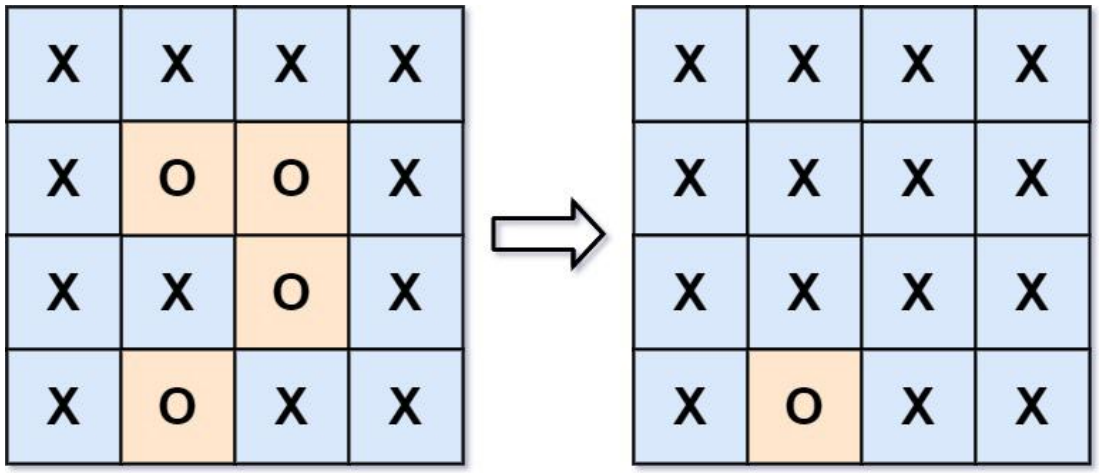
- The number of nodes in the tree is in the range `[1, 1000]`.
- `0 <= Node.val <= 9`
- The depth of the tree will not exceed `10`.

Surrounded Regions

Given an `m x n` matrix `board` containing `'X'` and `'O'`, *capture all regions that are 4-directionally surrounded by* `'X'`.

A region is **captured** by flipping all `'O'`s into `'X'`s in that surrounded region.

Example 1:




```
Input: board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]
Output: [["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]
Explanation: Notice that an 'O' should not be flipped if:
- It is on the border, or
- It is adjacent to an 'O' that should not be flipped.
The bottom 'O' is on the border, so it is not flipped.
The other three 'O' form a surrounded region, so they are flipped.
```

Example 2:

```
Input: board = [["X"]]
Output: [["X"]]
```

Constraints:

- `m == board.length`
- `n == board[i].length`
- `1 <= m, n <= 200`
- `board[i][j]` is `'X'` or `'O'`.

My Calendar I

You are implementing a program to use as your calendar. We can add a new event if adding the event will not cause a **double booking**.

A **double booking** happens when two events have some non-empty intersection (i.e., some moment is common to both events.).

The event can be represented as a pair of integers `start` and `end` that represents a booking on the half-open interval `[start, end)`, the range of real numbers `x` such that `start <= x < end`.

Implement the `MyCalendar` class:

- `MyCalendar()` Initializes the calendar object.
- `boolean book(int start, int end)` Returns `true` if the event can be added to the calendar successfully without causing a **double booking**. Otherwise, return `false` and do not add the event to the calendar.

Example 1:

```
Input
["MyCalendar", "book", "book", "book"]
[[], [10, 20], [15, 25], [20, 30]]
Output
[null, true, false, true]

Explanation
MyCalendar myCalendar = new MyCalendar();
myCalendar.book(10, 20); // return True
myCalendar.book(15, 25); // return False, It can not be booked because time 15 is already
booked by another event.
myCalendar.book(20, 30); // return True, The event can be booked, as the first event takes
every time less than 20, but not including 20.
```

Constraints:

- $0 <= start < end <= 10^9$
- At most 1000 calls will be made to `book`.

The Earliest Moment When Everyone Become Friends

Range Module

A Range Module is a module that tracks ranges of numbers. Design a data structure to track the ranges represented as **half-open intervals** and query about them.

A **half-open interval** `[left, right)` denotes all the real numbers `x` where `left <= x < right`.

Implement the `RangeModule` class:

- `RangeModule()` Initializes the object of the data structure.
- `void addRange(int left, int right)` Adds the **half-open interval** `[left, right)`, tracking every real number in that interval. Adding an interval that partially overlaps with currently tracked numbers should add any numbers in the interval `[left, right)` that are not already tracked.
- `boolean queryRange(int left, int right)` Returns `true` if every real number in the interval `[left, right)` is currently being tracked, and `false` otherwise.
- `void removeRange(int left, int right)` Stops tracking every real number currently being tracked in the **half-open interval** `[left, right)`.

Example 1:

```
Input
["RangeModule", "addRange", "removeRange", "queryRange", "queryRange", "queryRange"]
[[], [10, 20], [14, 16], [10, 14], [13, 15], [16, 17]]
Output
[null, null, null, true, false, true]

Explanation
RangeModule rangeModule = new RangeModule();
rangeModule.addRange(10, 20);
rangeModule.removeRange(14, 16);
rangeModule.queryRange(10, 14); // return True,(Every number in [10, 14) is being tracked)
```

```
rangeModule.queryRange(13, 15); // return False,(Numbers like 14, 14.03, 14.17 in [13, 15)
are not being tracked)
rangeModule.queryRange(16, 17); // return True, (The number 16 in [16, 17) is still being
tracked, despite the remove operation)
```

Constraints:

- `1 <= left < right <= 10^9`
- At most `10^4` calls will be made to `addRange`, `queryRange`, and `removeRange`.

Shortest Way to Form String

Random Pick with Weight

You are given a **0-indexed** array of positive integers `w` where `w[i]` describes the **weight** of the i^{th} index.

You need to implement the function `pickIndex()`, which **randomly** picks an index in the range `[0, w.length - 1]` (**inclusive**) and returns it. The **probability** of picking an index `i` is `w[i] / sum(w)`.

- For example, if `w = [1, 3]`, the probability of picking index `0` is `1 / (1 + 3) = 0.25` (i.e., `25%`), and the probability of picking index `1` is `3 / (1 + 3) = 0.75` (i.e., `75%`).

Example 1:

```
Input
["Solution","pickIndex"]
[[[1]],[]]
Output
[null,0]

Explanation
Solution solution = new Solution([1]);
solution.pickIndex(); // return 0. The only option is to return 0 since there is only one
element in w.
```

Example 2:

```
Input
["Solution","pickIndex","pickIndex","pickIndex","pickIndex","pickIndex"]
[[[1,3]],[],[],[],[],[]]
Output
[null,1,1,1,1,0]
```

```
Explanation
Solution solution = new Solution([1, 3]);
solution.pickIndex(); // return 1. It is returning the second element (index = 1) that has a
probability of 3/4.
solution.pickIndex(); // return 1
solution.pickIndex(); // return 1
solution.pickIndex(); // return 1
solution.pickIndex(); // return 0. It is returning the first element (index = 0) that has a
probability of 1/4.

Since this is a randomization problem, multiple answers are allowed.
All of the following outputs can be considered correct:
[null,1,1,1,1,0]
[null,1,1,1,1,1]
[null,1,1,1,0,0]
[null,1,1,1,0,1]
[null,1,0,1,0,0]
......
and so on.
```

Constraints:

- `1 <= w.length <= 10^4`
- `1 <= w[i] <= 10^5`
- `pickIndex` will be called at most 10^4 times.

Maximum AND Sum of Array

You are given an integer array `nums` of length `n` and an integer `numSlots` such that `2 * numSlots >= n`. There are `numSlots` slots numbered from `1` to `numSlots`.

You have to place all `n` integers into the slots such that each slot contains at **most** two numbers. The **AND sum** of a given placement is the sum of the **bitwise** `AND` of every number with its respective slot number.

- For example, the **AND sum** of placing the numbers `[1, 3]` into slot `1` and `[4, 6]` into slot `2` is equal to `(1 AND 1) + (3 AND 1) + (4 AND 2) + (6 AND 2) = 1 + 1 + 0 + 2 = 4`.

Return *the maximum possible **AND sum** of* `nums` *given* `numSlots` *slots.*

Example 1:

```
Input: nums = [1,2,3,4,5,6], numSlots = 3
Output: 9
Explanation: One possible placement is [1, 4] into slot 1, [2, 6] into slot 2, and [3, 5]
into slot 3.
This gives the maximum AND sum of (1 AND 1) + (4 AND 1) + (2 AND 2) + (6 AND 2) + (3 AND 3)
+ (5 AND 3) = 1 + 0 + 2 + 2 + 3 + 1 = 9.
```

Example 2:

```
Input: nums = [1,3,10,4,7,1], numSlots = 9
Output: 24
Explanation: One possible placement is [1, 1] into slot 1, [3] into slot 3, [4] into slot 4,
[7] into slot 7, and [10] into slot 9.
This gives the maximum AND sum of (1 AND 1) + (1 AND 1) + (3 AND 3) + (4 AND 4) + (7 AND 7)
+ (10 AND 9) = 1 + 1 + 3 + 4 + 7 + 8 = 24.
Note that slots 2, 5, 6, and 8 are empty which is permitted.
```

Constraints:

- `n == nums.length`
- `1 <= numSlots <= 9`
- `1 <= n <= 2 * numSlots`
- `1 <= nums[i] <= 15`

RLE Iterator

We can use run-length encoding (i.e., **RLE**) to encode a sequence of integers. In a run-length encoded array of even length `encoding` (**0-indexed**), for all even `i`, `encoding[i]` tells us the number of times that the non-negative integer value `encoding[i + 1]` is repeated in the sequence.

- For example, the sequence `arr = [8,8,8,5,5]` can be encoded to be `encoding = [3,8,2,5]`. `encoding = [3,8,0,9,2,5]` and `encoding = [2,8,1,8,2,5]` are also valid **RLE** of `arr`.

Given a run-length encoded array, design an iterator that iterates through it.

Implement the `RLEIterator` class:

- `RLEIterator(int[] encoded)` Initializes the object with the encoded array `encoded`.
- `int next(int n)` Exhausts the next `n` elements and returns the last element exhausted in this way. If there is no element left to exhaust, return `-1` instead.

Example 1:

```
Input
["RLEIterator", "next", "next", "next", "next"]
[[[3, 8, 0, 9, 2, 5]], [2], [1], [1], [2]]
Output
[null, 8, 8, 5, -1]

Explanation
RLEIterator rLEIterator = new RLEIterator([3, 8, 0, 9, 2, 5]); // This maps to the sequence
[8,8,8,5,5].
rLEIterator.next(2); // exhausts 2 terms of the sequence, returning 8. The remaining
sequence is now [8, 5, 5].
rLEIterator.next(1); // exhausts 1 term of the sequence, returning 8. The remaining sequence
is now [5, 5].
rLEIterator.next(1); // exhausts 1 term of the sequence, returning 5. The remaining sequence
is now [5].
```

```
rLEIterator.next(2); // exhausts 2 terms, returning -1. This is because the first term
exhausted was 5,
but the second term did not exist. Since the last term exhausted does not exist, we return
-1.
```

Constraints:

- `2 <= encoding.length <= 1000`
- `encoding.length` is even.
- `0 <= encoding[i] <= 10^9`
- `1 <= n <= 10^9`
- At most `1000` calls will be made to `next`.

Maximum Score of a Node Sequence

There is an **undirected** graph with `n` nodes, numbered from `0` to `n - 1`.

You are given a **0-indexed** integer array `scores` of length `n` where `scores[i]` denotes the score of node `i`. You are also given a 2D integer array `edges` where `edges[i] = [a_i, b_i]` denotes that there exists an **undirected** edge connecting nodes `a_i` and `b_i`.

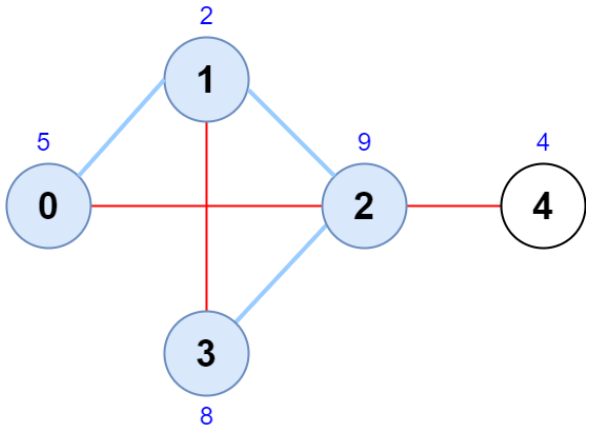
A node sequence is **valid** if it meets the following conditions:

- There is an edge connecting every pair of **adjacent** nodes in the sequence.
- No node appears more than once in the sequence.

The score of a node sequence is defined as the **sum** of the scores of the nodes in the sequence.

Return *the **maximum score** of a valid node sequence with a length of* `4`. If no such sequence exists, return `-1`.

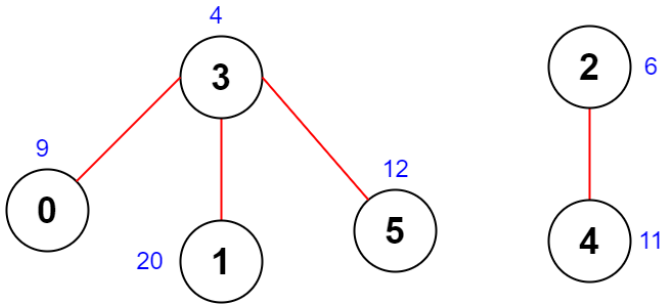
Example 1:




```
Input: scores = [5,2,9,8,4], edges = [[0,1],[1,2],[2,3],[0,2],[1,3],[2,4]]
Output: 24
```

Explanation: The figure above shows the graph and the chosen node sequence [0,1,2,3].
The score of the node sequence is 5 + 2 + 9 + 8 = 24.
It can be shown that no other node sequence has a score of more than 24.
Note that the sequences [3,1,2,0] and [1,0,2,3] are also valid and have a score of 24.
The sequence [0,3,2,4] is not valid since no edge connects nodes 0 and 3.

Example 2:



Input: scores = [9,20,6,4,11,12], edges = [[0,3],[5,3],[2,4],[1,3]]
Output: -1
Explanation: The figure above shows the graph.
There are no valid node sequences of length 4, so we return -1.

Constraints:

- n == scores.length
- 4 <= n <= $5 * 10^4$
- 1 <= scores[i] <= 10^8
- 0 <= edges.length <= $5 * 10^4$
- edges[i].length == 2
- 0 <= a_i, b_i <= n - 1
- a_i != b_i
- There are no duplicate edges.

Longest Line of Consecutive One in Matrix

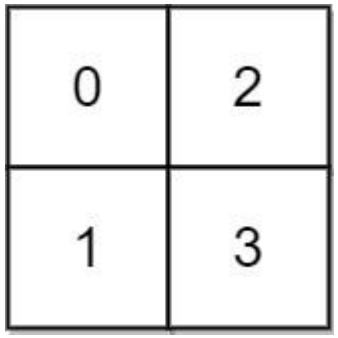
Swim in Rising Water

You are given an `n x n` integer matrix `grid` where each value `grid[i][j]` represents the elevation at that point `(i, j)`.

The rain starts to fall. At time `t`, the depth of the water everywhere is `t`. You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most `t`. You can swim infinite distances in zero time. Of course, you must stay within the boundaries of the grid during your swim.

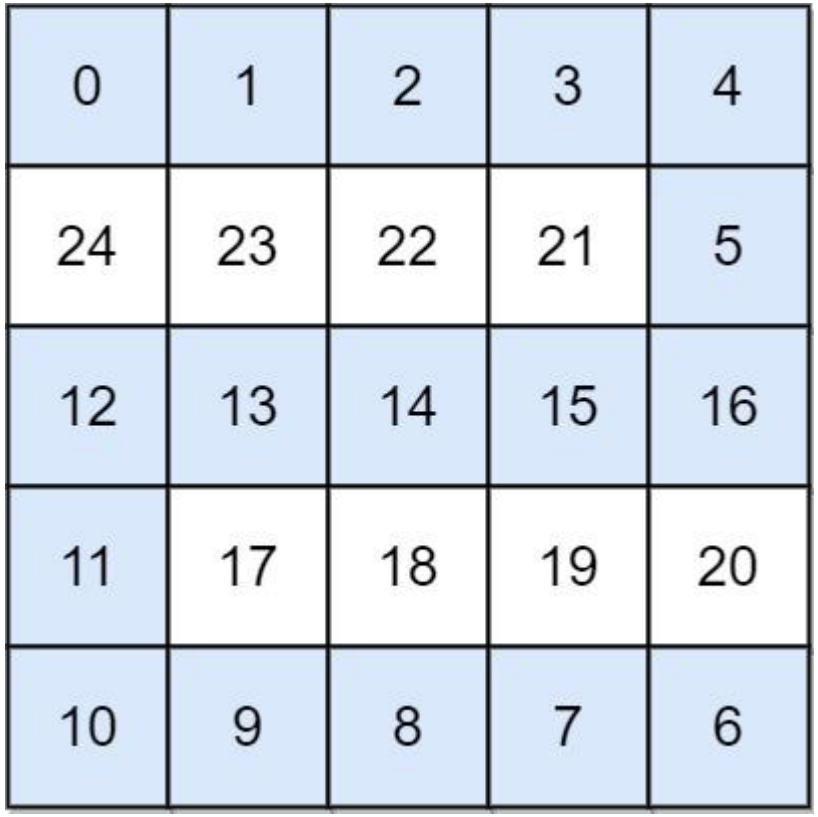
Return *the least time until you can reach the bottom right square* `(n - 1, n - 1)` *if you start at the top left square* `(0, 0)`.

Example 1:




```
Input: grid = [[0,2],[1,3]]
Output: 3
Explanation:
At time 0, you are in grid location (0, 0).
You cannot go anywhere else because 4-directionally adjacent neighbors have a higher
elevation than t = 0.
You cannot reach point (1, 1) until time 3.
When the depth of water is 3, we can swim anywhere inside the grid.
```

Example 2:




```
Input: grid = [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,6]]
Output: 16
Explanation: The final route is shown.
We need to wait until time 16 so that (0, 0) and (4, 4) are connected.
```

Constraints:

- `n == grid.length`
- `n == grid[i].length`

- `1 <= n <= 50`
- `0 <= grid[i][j] < n^2`
- Each value `grid[i][j]` is **unique**.

Word Ladder II

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every s_i for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- s_k `== endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *all the* ***shortest transformation sequences*** *from* `beginWord` *to* `endWord`*, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words* `[beginWord, s1, s2, ..., sk]`.

Example 1:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]
Explanation: There are 2 shortest transformation sequences:
"hit" -> "hot" -> "dot" -> "dog" -> "cog"
"hit" -> "hot" -> "lot" -> "log" -> "cog"
```

Example 2:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: []
Explanation: The endWord "cog" is not in wordList, therefore there is no valid
transformation sequence.
```

Constraints:

- `1 <= beginWord.length <= 5`
- `endWord.length == beginWord.length`
- `1 <= wordList.length <= 500`
- `wordList[i].length == beginWord.length`
- `beginWord`, `endWord`, and `wordList[i]` consist of lowercase English letters.
- `beginWord != endWord`
- All the words in `wordList` are **unique**.
- The **sum** of all shortest transformation sequences does not exceed 10^5.

Word Ladder

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every s_i for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- s_k `== endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *the* ***number of words*** *in the* ***shortest transformation sequence*** *from* `beginWord` *to* `endWord`*, or* `0` *if no such sequence exists.*

Example 1:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: 5
Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" ->
cog", which is 5 words long.
```

Example 2:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: 0
Explanation: The endWord "cog" is not in wordList, therefore there is no valid
transformation sequence.
```

Constraints:

- `1 <= beginWord.length <= 10`
- `endWord.length == beginWord.length`
- `1 <= wordList.length <= 5000`

- `wordList[i].length == beginWord.length`
- `beginWord`, `endWord`, and `wordList[i]` consist of lowercase English letters.
- `beginWord != endWord`
- All the words in `wordList` are **unique**.

Palindrome Partitioning II

Given a string `s`, partition `s` such that every substring of the partition is a palindrome.

Return *the minimum cuts needed* for a palindrome partitioning of `s`.

Example 1:

```
Input: s = "aab"
Output: 1
Explanation: The palindrome partitioning ["aa","b"] could be produced using 1 cut.
```

Example 2:

```
Input: s = "a"
Output: 0
```

Example 3:

```
Input: s = "ab"
Output: 1
```

Constraints:

- `1 <= s.length <= 2000`
- `s` consists of lowercase English letters only.

Candy

There are `n` children standing in a line. Each child is assigned a rating value given in the integer array `ratings`.

You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

Return *the minimum number of candies you need to have to distribute the candies to the children*.

Example 1:

```
Input: ratings = [1,0,2]
Output: 5
Explanation: You can allocate to the first, second and third child with 2, 1, 2 candies
respectively.
```

Example 2:

```
Input: ratings = [1,2,2]
Output: 4
Explanation: You can allocate to the first, second and third child with 1, 2, 1 candies
respectively.
The third child gets 1 candy because it satisfies the above two conditions.
```

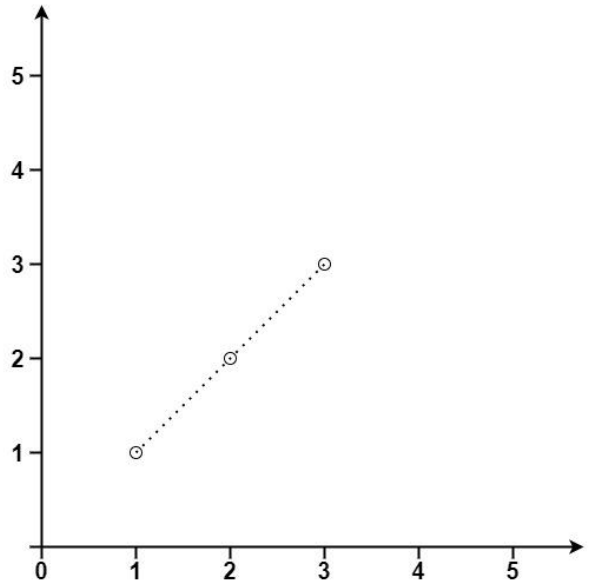
Constraints:

- `n == ratings.length`
- `1 <= n <= 2 * 10^4`
- `0 <= ratings[i] <= 2 * 10^4`

Max Points on a Line

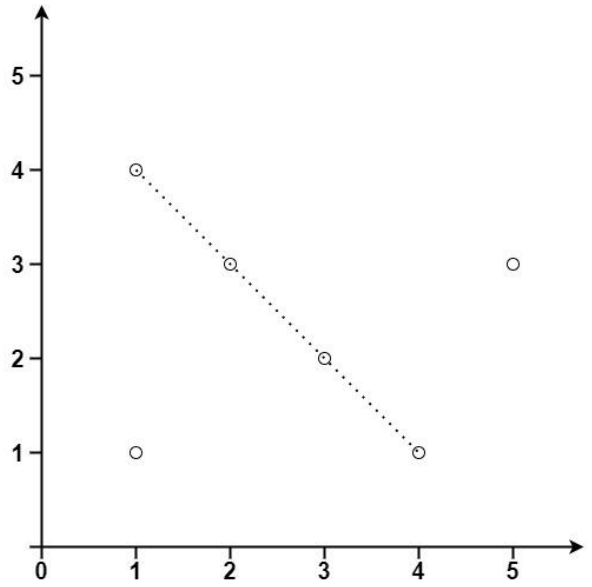
Given an array of `points` where `points[i] = [x_i, y_i]` represents a point on the **X-Y** plane, return *the maximum number of points that lie on the same straight line*.

Example 1:




```
Input: points = [[1,1],[2,2],[3,3]]
Output: 3
```

Example 2:




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

Constraints:

- `1 <= points.length <= 300`
- `points[i].length == 2`
- $-10^4 \le x_i, y_i \le 10^4$
- All the `points` are **unique**.