TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)
Personal Member ID#: 117667

Session 2: Review

Session Overview

Congratulations on making it to the final session! 🎉 In this session, students will review everything that has been covered in the course so far, practicing matching the algorithmic strategies and data structures covered to different easy and medium level Leetcode problems.

> You can find all resources from today including session slide decks, session recordings, and more on the resources tab

📊 Part 1 : Instructor Led Session

We'll spend the first portion of the synchronous class time in large groups, where the instructor will lead class instruction for 30-45 minutes.

👨‍💻 Part 2: Breakout Session

In breakout sessions, we will explore and collaboratively solve problem sets in small groups. Here, the **collaboration, conversation, and approach** are just as important as “solving the problem” - please engage warmly, clearly, and plentifully in the process!

In breakout rooms you will:

- Screen-share the problem/s, and verbally review them together
- Screen-share an interactive coding environment, and talk through the steps of a solution approach
    - ProTip: - An Integrated Development Environment (IDE) is a fancy name for a tool you could use for shared writing of code - like Replit.com, Collabed.it, CodePen.io, or other - your staff team will specify which tool to use for this class!
- Screen-share an implementation of your proposed solution
- Independently follow-along, or create an implementation, in your own IDE.

Your program leader/s will indicate which code sharing tool/s to use as a group, and will help break down or provide specific scaffolding with the main concepts above.

▶ **Note on Expectations**

🔎 Problem Solving Approach

To build a long-term organized approach to problem solving, we'll start with three main steps. We'll refer to them as **UPI: Understand, Plan, and Implement**.

We'll apply these three steps to most of the problems we'll see in the first half of the course.

We will learn to:

- **Understand** the problem,
- **Plan** a solution step-by-step, and
- **Implement** the solution

▶ Comment on UPI

▶ UPI Example

ℹ️ Note: Testing your Linked Lists (Printing)

The following function takes in the `head` of a linked list and prints out the values of each node int he list with an `->` between values of linked nodes. You may copy the function below to use it as needed while you complete the problem sets.

```
def print_linked_list(head):
    current = head
    if not head:
        print("Empty List")
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next
```

ℹ️ Note: Testing your Binary Tree (Printing)

To keep the amount of starter code manageable, we have chosen not to include a function to print a binary tree as part of each relevant problem statement. You may instead copy the function in the drop-down below `print_tree()` and use it as needed while you complete the problem sets.

▼ Print Binary Tree Function

Accepts the root of a binary tree and prints out the values of each node level by level from left to right. Values of `None` are used to indicate a null child node between non-null children on the same level. Prints `"Empty"` for an empty tree.

```
from collections import deque

# Tree Node class
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.val = value
        self.left = left
        self.right = right

def print_tree(root):
    if not root:
        return "Empty"
    result = []
    queue = deque([root])
    while queue:
        node = queue.popleft()
        if node:
            result.append(node.val)
            queue.append(node.left)
            queue.append(node.right)
        else:
            result.append(None)
    while result and result[-1] is None:
        result.pop()
    print(result)
```

Example Usage:

```
"""
          1
        /   \
       2     3
      /     / \
     4     5   6
"""

root = Node(1, Node(2, Node(4)), Node(3, Node(5), Node(6)))

print_tree(root)
print_tree(None)
```

Example Output:

```
[1, 2, 3, 4, None, 5, 6]
'Empty'
```

ℹ️ Note: Testing your Binary Tree (Generating a Tree)

Now that you have practice manually building trees for testing in previous sessions, we are providing a function that builds binary trees based off of a list of values to speed up the testing process. We have chosen not to include this function in the starter code for each problem to keep the length of problems manageable. You may instead copy the function in the drop-down below `build_tree()` and use it as needed while you complete the problem sets.

▼ Build Binary Tree Function

Takes in a list `values` where each element in the list corresponds to a node in the binary tree you would like to build. The values should be in level order (from top to bottom, left to right). Use `None` to indicate a null child between non-null children on the same level.

Some problems may ask you to build a tree where nodes have both keys and values. This function may be used to build trees with just values *and* trees with both keys and values:

- If building a tree with only values, `values` should be given in the form: `[value1, value2, value3, ...]`.
- If building a tree with both keys and values `values` should be given in the form `[(key1, value1), (key2, value2), (key3, value3), ...]`.

Returns the `root` of the binary tree made from `values`.

```
from collections import deque

# Tree Node class
class TreeNode:
  def __init__(self, value, key=None, left=None, right=None):
      self.key = key
      self.val = value
      self.left = left
      self.right = right

def build_tree(values):
  if not values:
      return None

  def get_key_value(item):
      if isinstance(item, tuple):
          return item[0], item[1]
      else:
          return None, item

  key, value = get_key_value(values[0])
  root = TreeNode(value, key)
  queue = deque([root])
  index = 1

  while queue:
      node = queue.popleft()
      if index < len(values) and values[index] is not None:
          left_key, left_value = get_key_value(values[index])
          node.left = TreeNode(left_value, left_key)
          queue.append(node.left)
      index += 1
      if index < len(values) and values[index] is not None:
          right_key, right_value = get_key_value(values[index])
          node.right = TreeNode(right_value, right_key)
          queue.append(node.right)
      index += 1

  return root
```

Example Usage:

```
"""
          1
        /   \
       2     3
      /     / \
     4     5   6
"""

tree_with_just_values = [1, 2, 3, 4, None, 5, 6]
val_tree = build_tree(tree_with_just_values)

"""
          (1, 'A')
          /       \
       (2, 'B')   (3, 'C')
      /           /      \
  (4, 'D')    (5, 'E')   (6, 'F')
"""

tree_with_keys_and_values = [(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'), None, (5, 'E
key_val_tree = build_tree(tree_with_keys_and_values)

# Using print_tree() function included above
print_tree(val_tree)
print_tree(key_val_tree) # Only values will be printed
```

Example Output:

```
[1, 2, 3, 4, None, 5, 6]
['A', 'B', 'C', 'D', None, 'E', 'F']
```

Breakout Problems Session 2

▸ Standard Problem Set Version 1

▸ Standard Problem Set Version 2

▾ Advanced Problem Set Version 1

Problem 1: Copy List with Random Pointer

A linked list of length `n` is given such that each node contains an additional random pointer, which could point to any node in the list, or `None`.

Construct a deep copy of the list. The deep copy should consist of exactly `n` brand new nodes, where each new node has its value set to the value of its corresponding original node. Both the `next` and `random` pointer of the new nodes should point to new nodes in the copied list such

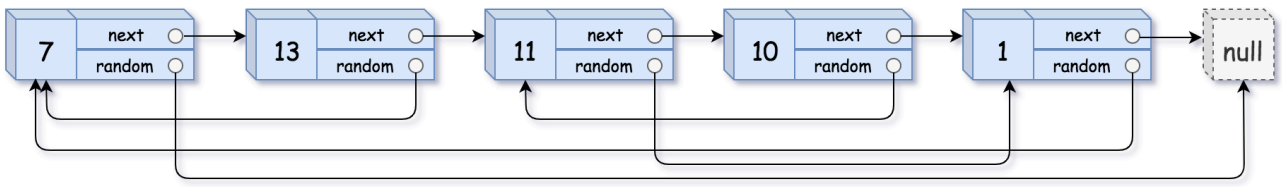
that the pointers in the original list and copied list represent the same list state. None of the pointers in the new list should point to nodes in the original list.

For example, if there are two nodes `X` and `Y` in the original list, where `X.random --> Y`, then for the corresponding two nodes `x` and `y` in the copied list, `x.random --> y`.

Given the `head` of the original linked list, return the head of the copied linked list.

```
class Node:
    def __init__(self, val, next=None, random=None):
        self.value = val
        self.next = next
        self.random = random

# For testing, prints (value, random pointer value) for each node
def print_linked_list(head):
    current = head
    if not head:
        print("Empty List")
    while current:
        print((current.value, current.random.value), end=" -> " if current.next else
        current = current.next

# For testing, prints object ids for each node
def print_list_ids(head):
    current = head
    if not head:
        print("Empty List")
    while current:
        print(id(current), end=" -> " if current.next else "\n")
        current = current.next

def copy_random_list(head):
    pass
```

Example Usage 1:

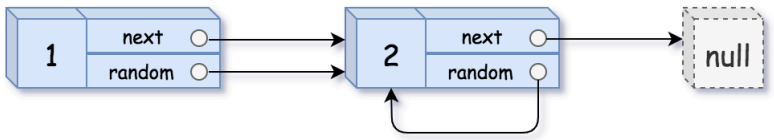


```
seven = Node(7)
thirteen = Node(13)
eleven = Node(11)
ten = Node(10)
one = Node(1)

seven.next = thirteen
thirteen.next = eleven
eleven.next = ten
ten.next = one

thirteen.random = seven
eleven.random = one
ten.random = eleven
one.random = seven

copied_list = copy_random_list(seven)
print_linked_list(copied_list)
print_list_ids(seven)
print_list_ids(copied_list)
```

Example Output 1:

```
(7, None) -> (13, 7) -> (11, 1) -> (10, 11) -> (1, 7)
4341867088 -> 4341866992 -> 4341866848 -> 4341866704 -> 4341866608
4341858208 -> 4341858064 -> 4341857968 -> 4341857872 -> 4341857776
Example 1 Explanation:
The node and random pointer values match that of the input list. The second two prin
which represent the object ids of each node are different from. Note that object ids
each time you run your code and will likely differ from those shown above. The impor
that the ids of the original list do not match the ids of the copied list.
```

Example Usage 2:

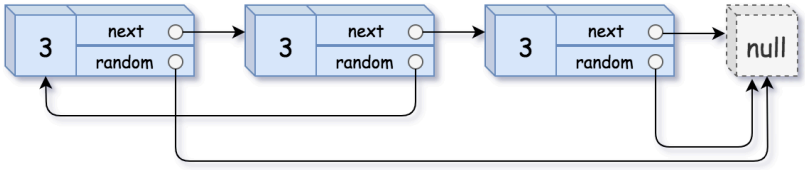



```
one = Node(1)
two = Node(2)

one.next = two

one.random = two
two.random = two

copied_list = copy_random_list(one)
print_linked_list(copied_list)
print_list_ids(one)
print_list_ids(copied_list)
```

Example Output 2:

```
(1, 2) -> (2, 2)
4368363472 -> 4368363376
4368363232 -> 4368363088
```

Example Usage 3:




```
first = Node(3)
second = Node(3)
third = Node(3)

first.next = second
second.next = third

second.random = first

copied_list = copy_random_list(first)
print_linked_list(copied_list)
print_list_ids(first)
print_list_ids(copied_list)
```

Example Output 3:

```
(3, None) -> (3, 3) -> (3, None)
4375769040 -> 4375768944 -> 4375768800
4375768656 -> 4375768560 -> 4375768416
```

▶ 💡 **Hint: Python Object IDs**

Problem 2: Longest Palindromic Substring

Given a string `s`, return the longest palindromic substring in `s`.

A string is palindromic if it reads the same forwards and backwards.

```
def longest_palindrome(s):
    pass
```

Example Usage:

```
print(longest_palindrome("babad"))
print(longest_palindrome("cbbd"))
```

Example Output:

```
bab
Example 1 Explanation: 'aba' is also a valid answer.

bb
```

Problem 3: Graph Valid Tree

You have a graph of `n` nodes labeled from `0` to `n - 1`. You are given an integer `n` and a list of edges where `edges[i] = [$a_i$, $b_i$]` indicates that there is an undirected edge between nodes `$a_i$` and `$b_i$` in the graph.

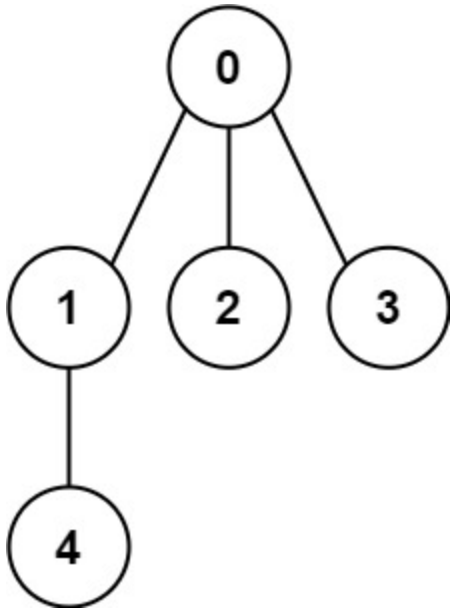
Return `True` if the edges of the given graph make up a valid tree, and `False` otherwise.

A graph is a valid tree if it meets two conditions:

- Connected: All nodes must be reachable from any other node, meaning there is exactly one connected component.
- Acyclic: The graph must not contain any cycles.

```
def valid_tree(n, edges):
    pass
```

Example Usage 1:

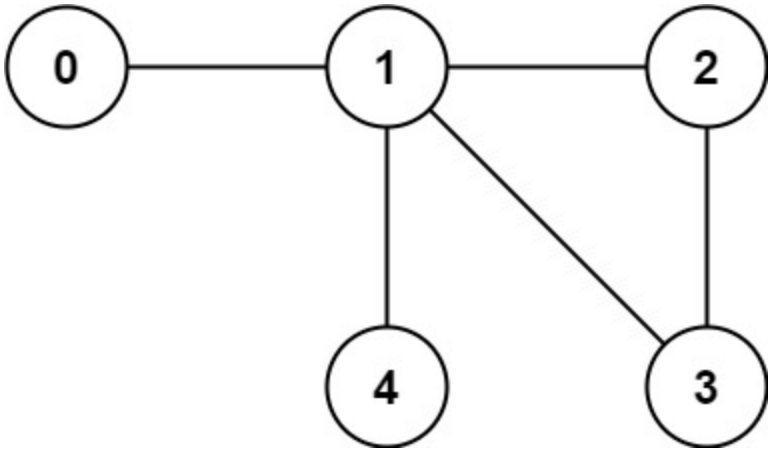



```
edges_1 = [[0,1],[0,2],[0,3],[1,4]]

print(valid_tree(5, edges_1))
```

Example Output 1:

```
True
```

Example Usage 2:




```
edges_2 = [[0,1],[1,2],[2,3],[1,3],[1,4]]

print(valid_tree(5, edges_2))
```

Example Output 2:

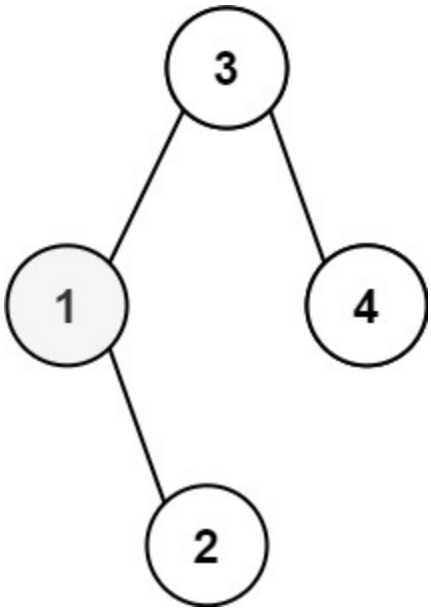
```
False
```

Problem 4: Kth Smallest Element in a BST

Given the `root` of a binary search tree, and an integer `k`, return the `kth` smallest value (1-indexed) of all the values of the nodes in the tree.

```
class TreeNode():
     def __init__(self, value, left=None, right=None):
        self.val = value
        self.left = left
        self.right = right

def kth_smallest(root, k):
    pass
```

Example Usage 1:

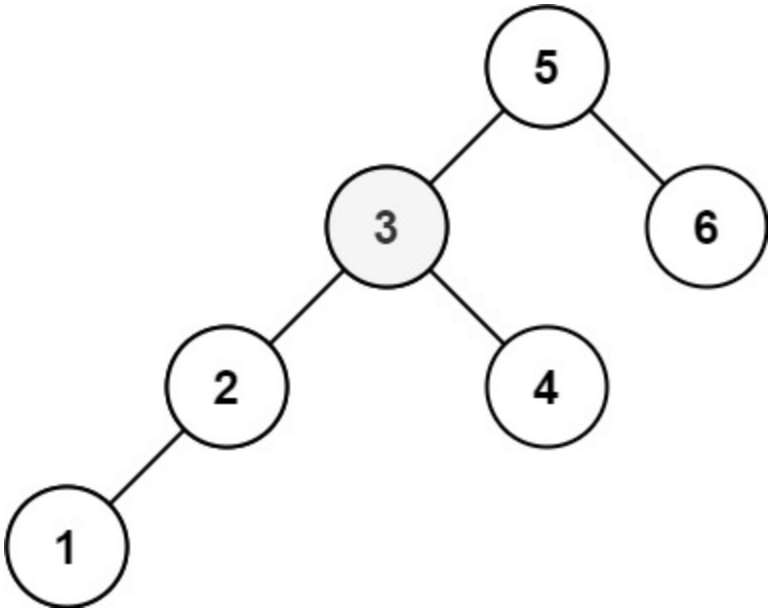



```
values_1 = [3, 1, 4, None, 2]

# Using build_tree() function included at top of page
tree_1 = build_tree(values_1)
print(kth_smallest(tree_1, 1))
```

Example Output 1:

```
1
```

Example Usage 2:




```
values_2 = [5, 3, 6, 2, 4, None, None, 1]

# Using build_tree() function included at top of page
tree_2 = build_tree(values_2)
print(kth_smallest(tree_2, 3))
```

Example Output 2:

```
3
```

Problem 5: Longest Consecutive Sequence

Given an unsorted array of integers `nums`, return the length of the longest consecutive elements sequence.

You must write an algorithm that runs in `O(n)` time.

```
def longest_consecutive(nums):
    pass
```

Example Usage:

```
nums_1 = [100,4,200,1,3,2]
nums_2 = [0,3,7,2,5,8,4,6,0,1]
```

Example Output:

```
4
Example 1 Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Th

9
```

Problem 6: Reverse Nodes in K-Group

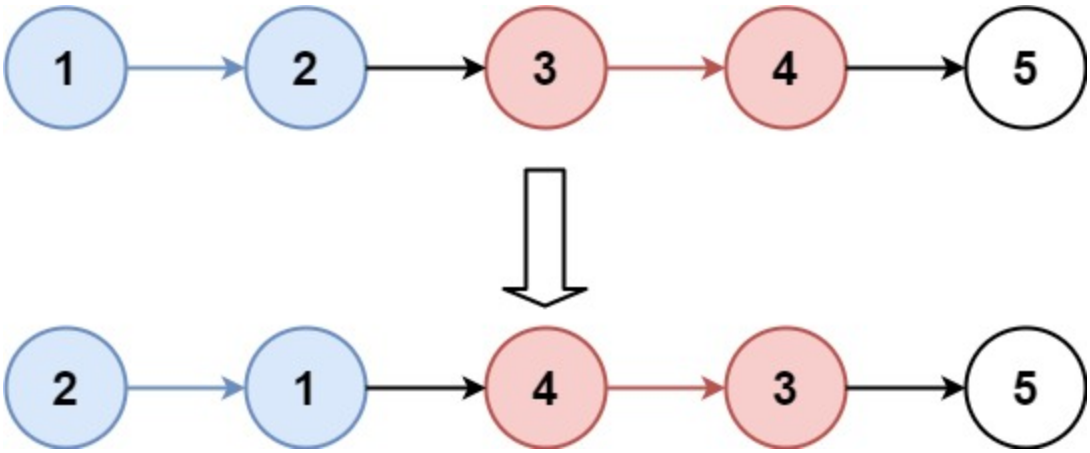
Given the head of a linked list, reverse the nodes of the list `k` at a time, and return the head of the modified list.

`k` is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of `k` then left-out nodes, in the end, should remain as it is.

You may not alter the values in the list's nodes, only nodes themselves may be changed.

```
class Node():
    def __init__(self, val=0, next=None):
        self.value = val
        self.next = next

def reverse_k_group(head, k):
    pass
```

Example Usage 1:

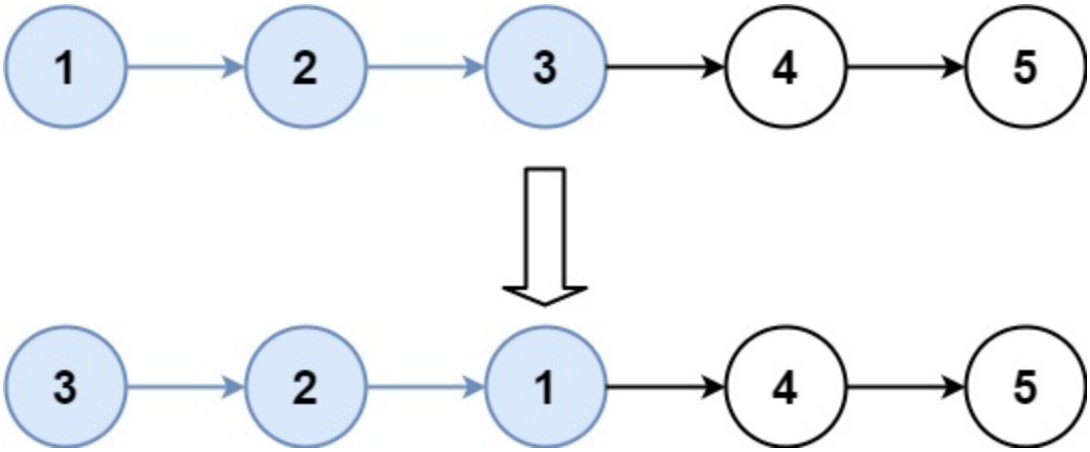



```
list_1 = Node(1, Node(2, Node(3, Node(4, Node(5)))))

# Using print_linked_list() function included at top of page
print_linked_list(reverse_k_group(list_1, 2))
```

Example Output 1:

```
2 -> 1 -> 4 -> 3 -> 5
```

Example Usage 2:




```
list_2 = Node(1, Node(2, Node(3, Node(4, Node(5)))))

# Using print_linked_list() function included at top of page
print_linked_list(reverse_k_group(list_2, 3))
```

Example Output 2:

```
3 -> 2 -> 1 -> 4 -> 5
```

▶ 💡 **Hint: Leetcode Hard**

Close Section

Advanced Problem Set Version 2

Problem 1: Populating Next Right Pointers in Each Node

You are given the `root` of 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:

```
class TreeNode():
    def __init__(self, val=0, left=None, right=None, next=None):
        self.val = val
        self.left = left
        self.right = right
        self.next = 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 `None`.

Initially, all next pointers are set to `None`. Return the root of the modified tree.

Note that the `build_tree()` function included at the top of this page will work with this problem so long as you use the updated `TreeNode()` class provided below. The `print_tree()` function needs to be modified. A modified version that prints a list of tuples where each tuple has the form `(node.val, node.next.val)` is provided below.

```
class TreeNode():
    def __init__(self, val=0, left=None, right=None, next=None):
        self.val = val
        self.left = left
        self.right = right
        self.next = next

# For testing
def print_tree(root):
    if not root:
        return "Empty"
    result = []
    queue = deque([root])
    while queue:
        node = queue.popleft()
        if node:
            if node.next is not None:
                result.append((node.val, node.next.val))
            else:
                result.append((node.val, None))
            queue.append(node.left)
            queue.append(node.right)
        else:
            result.append(None)
    while result and result[-1] is None:
        result.pop()
    print(result)

def connect(root):
    pass
```

Example Usage:

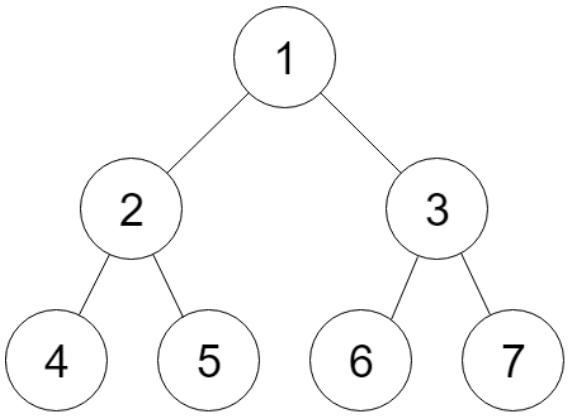


Figure A

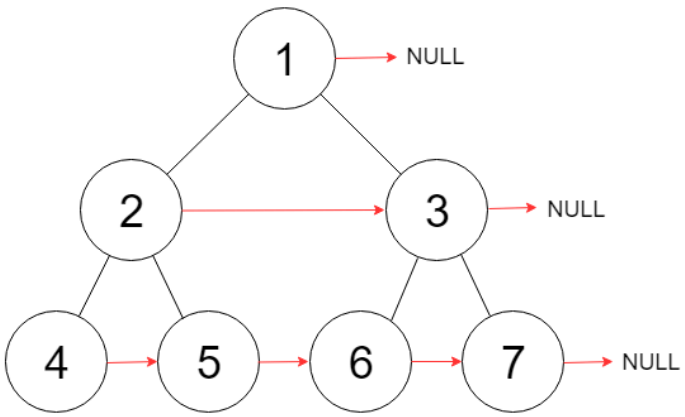


Figure B

```
values = [1, 2, 3, 4, 5, 6, 7]

root = build_tree(values)
print_tree(connect(root))
```

Example Output:

```
[(1, None), (2, 3), (3, None), (4, 5), (5, 6), (6, 7), (7, None)]
```

Problem 2: 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.

```
def merge(intervals):
    pass
```

Example Usage:

```
intervals_1 = [[1,3],[2,6],[8,10],[15,18]]
intervals_2 = [[1,4],[4,5]]
```

Example Output:

```
[[1,6],[8,10],[15,18]]
Example 1 Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6

[[1,5]]
Example 2 Explanation: Intervals [1,4] and [4,5] are considered overlapping.
```

Problem 3: Minimum Path Sum

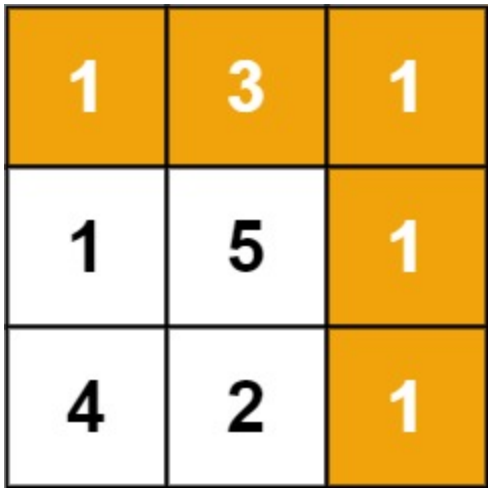
Given a `m x n` `grid` filled with non-negative numbers, return the sum of the path from the top left cell to bottom right cell which minimizes the sum of all numbers along its path.

You can only move either down or right at any point in time.

```
def min_path_sum(grid):
    pass
```

Example Usage:

Example 1:



```
grid_1 = [[1,3,1],[1,5,1],[4,2,1]]
grid_2 = [[1,2,3],[4,5,6]]

print(min_path_sum(grid_1))
print(min_path_sum(grid_2))
```

Example Output:

```
7
Example 1 Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.

12
```

Problem 4: 4Sum

Given an array `nums` of `n` integers and an integer `target`, return an array of all the unique quadruplets `[nums[a], nums[b], nums[c], nums[d]]` such that:

- `0 <= a, b, c, d < n`
- `a`, `b`, `c`, and `d` are distinct.
- `nums[a] + nums[b] + nums[c] + nums[d] == target`

You may return the answer in any order.

```
def four_sum(nums, target):
    pass
```

Example Usage:

```
nums_1 = [1,0,-1,0,-2,2]
nums_2 = [2,2,2,2,2]

print(four_sum(nums_1, 0))
print(four_sum(nums_2, 8))
```

Example Output:

```
[[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]
[[2,2,2,2]]
```

Problem 5: Find Closest Node to Two Other Nodes

You are given a directed graph of `n` nodes numbered from `0` to `n - 1`, where each node has at most one outgoing edge.

The graph is represented with a given 0-indexed array `edges` of size `n`, indicating that there is a directed edge from node `i` to node `edges[i]`. If there is no outgoing edge from `i`, then `edges[i] == -1`.

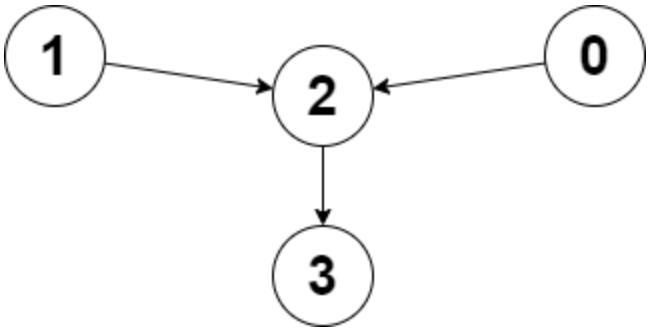
You are also given two integers `node1` and `node2`.

Return the index of the node that can be reached from both `node1` and `node2`, such that the maximum between the distance from `node1` to that node, and from `node2` to that node is minimized. If there are multiple answers, return the node with the smallest index, and if no possible answer exists, return `-1`.

Note that `edges` may contain cycles.

```
def closest_meeting_node(edges, node1, node2):
    pass
```

Example Usage 1:

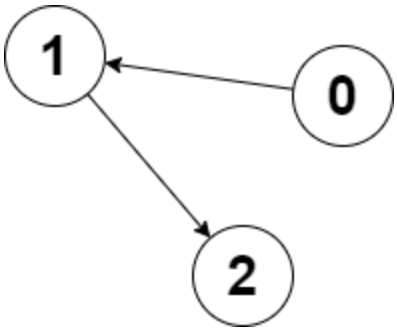



```
edges_1 = [2,2,3,-1]

print(closest_meeting_node(edges_1, 0, 1))
```

Example Output 1:

```
2
Example 1 Explanation: The distance from node 0 to node 2 is 1, and the distance fro
to node 2 is 1.
The maximum of those two distances is 1. It can be proven that we cannot get a node
smaller maximum distance than 1, so we return node 2.
```

Example Usage 2:




```
edges_2 = [1,2,-1]

print(closest_meeting_node(edges_1, 0, 2))
```

Example Output 2:

```
2
Example 2 Explanation: The distance from node 0 to node 2 is 2, and the distance fro
itself is 0.
The maximum of those two distances is 2. It can be proven that we cannot get a node
maximum distance than 2, so we return node 2.
```

Problem 6: Word Ladder

A transformation sequence from word `begin_word` to word `end_word` using an array `word_list` is a sequence of words `begin_word -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every `si` for `1 <= i <= k` is in `word_list`. Note that `begin_word` does not need to be in `word_list`.
- `sk == end_word`

Given two words, `begin_word` and `end_word`, and an array `word_list`, return the number of words in the shortest transformation sequence from `begin_word` to `end_word`, or `0` if no such sequence exists.

```
def ladder_length(begin_word, end_word, word_list):
    pass
```

Example Usage:

```
word_list_1 = ["hot","dot","dog","lot","log","cog"]
word_list_2 = ["hot","dot","dog","lot","log"]

print(ladder_length("hit", "cog", word_list_1))
print(ladder_length("hit", "cog", word_list_2))
```

```
5
Example 1 Explanation: One shortest transformation sequence is
"hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.

0
Example 2 Explanation: The end_word "cog" is not in word_list, therefore there is no
transformation sequence.
```

▶ 💡 **Hint: Leetcode Hard**

Close Section