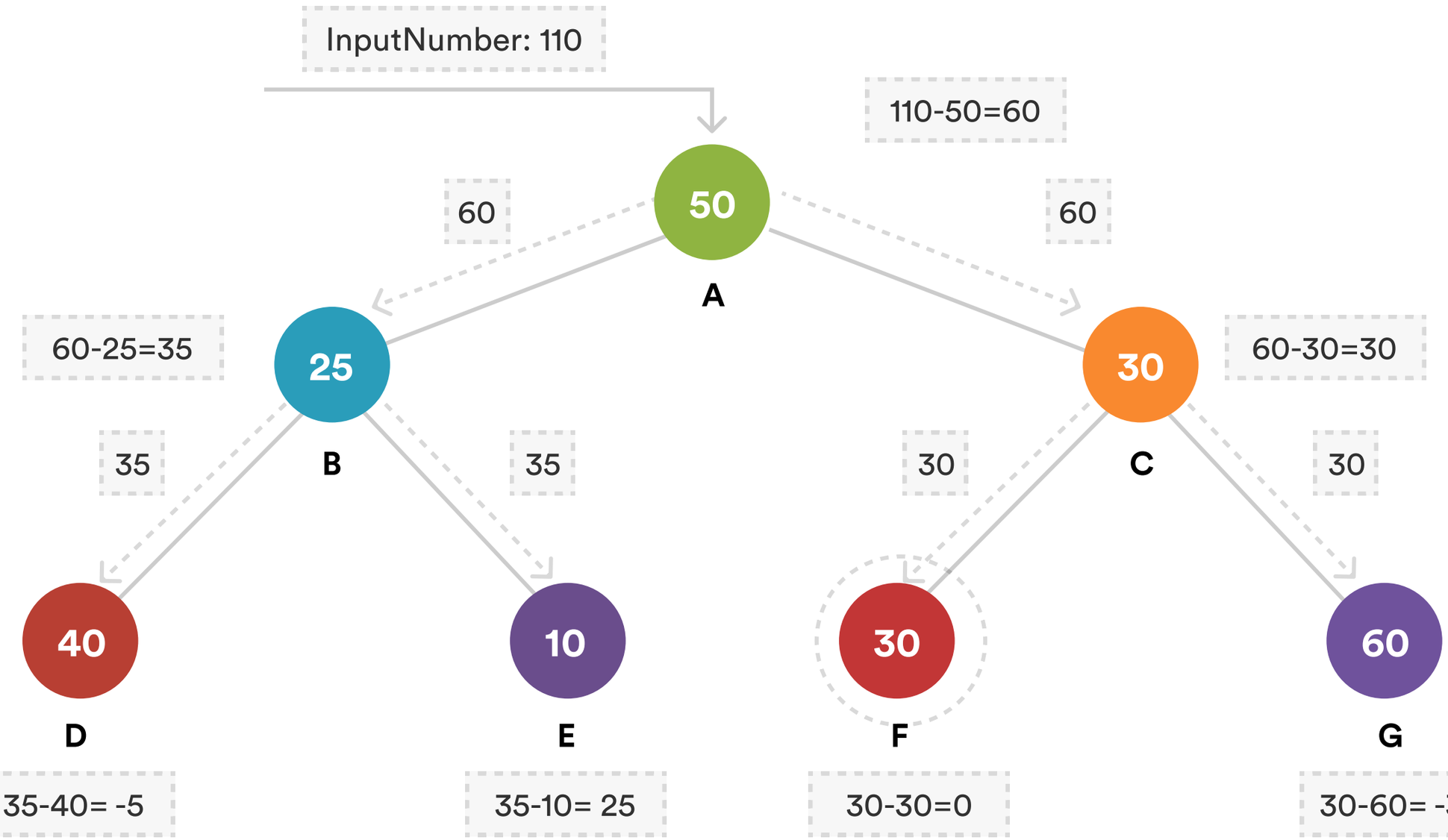
Master Most Used

PROBLEM SOLVING PATTERNS



For Product Based Companies Interviews

How to effectively prepare for Product Based Companies Interviews?

Solving thousands of questions

Recognising frequently occurring patterns and mapping new questions to them

Two Pointers

1

- Two pointers traverse the data structure together until a specific condition is met.
- Handy for finding pairs in sorted arrays or linked lists, such as comparing elements to each other in an array.
- Can be of different types - both pointers starting from same end, or one pointer at each end.

Example Problem

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

The brute force approach would take two loops for forming pairs and comparing their sum to target.

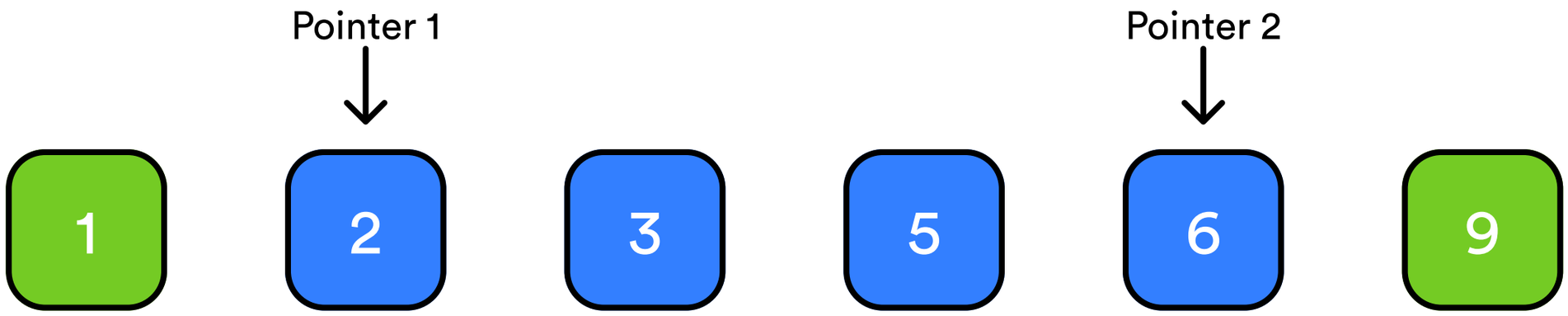
This would take a time complexity of O(n^2) where n is the length of the array.

We can use two pointers approach for an optimised solution

1. Sort the given array.
2. Start two pointers. First pointer at the beginning of the array and second one at end.
3. Get current sum of pointer A and B, if current sum is greater than target move pointer B back, else move pointer A forward

1 + 9 > target sum, therefore let's decrement **Pointer 2**

1 + 6 < target sum, therefore let's increment **Pointer 1**



2 + 6 == target sum, we have found our pair!

The optimised solution using Two pointers would follow the pseudocode

```
function PairSum(nums, target):
 // Sort the array
 sort(nums)
 // Initialize the first pointer at the beginning of the
array
 i = 0
 // Initialize the second pointer at the end of the array
 j = length of nums - 1
 // Loop until the first pointer is less than the second
pointer
 while i < j:
 // If we find a pair adding to the target
     if nums[i] + nums[j] == target:
     // Return the indices i and j as a pair
         return i, j
     // If the current sum is less than the target, move
towards higher values by incrementing i
     else if nums[i] + nums[j] < target:
         i++
     // If the current sum is more than the target, move
towards lower values by decrementing j
     else:
         j--
 // If no pair is found, return 0 or an appropriate value
to indicate no pair was found
 return 0
```

This version would only take O(nlogn) time complexity for sorting the array.

Problems based on Two Pointers:

1. Remove Duplicates from Sorted Array - LeetCode
2. Squares of a Sorted Array - LeetCode
3. 3Sum - LeetCode
4. 3Sum Closest - LeetCode
5. Sort Colors - LeetCode
6. Backspace String Compare - LeetCode

Fast and Slow Pointers

2

- Also known as the Hare & Tortoise algorithm uses two pointers which move through the data structure at different speeds

Example Problem

Given a non-empty, singly linked list with head node head we have to return the middle node. If there are two middle nodes, we have to return the second middle node.

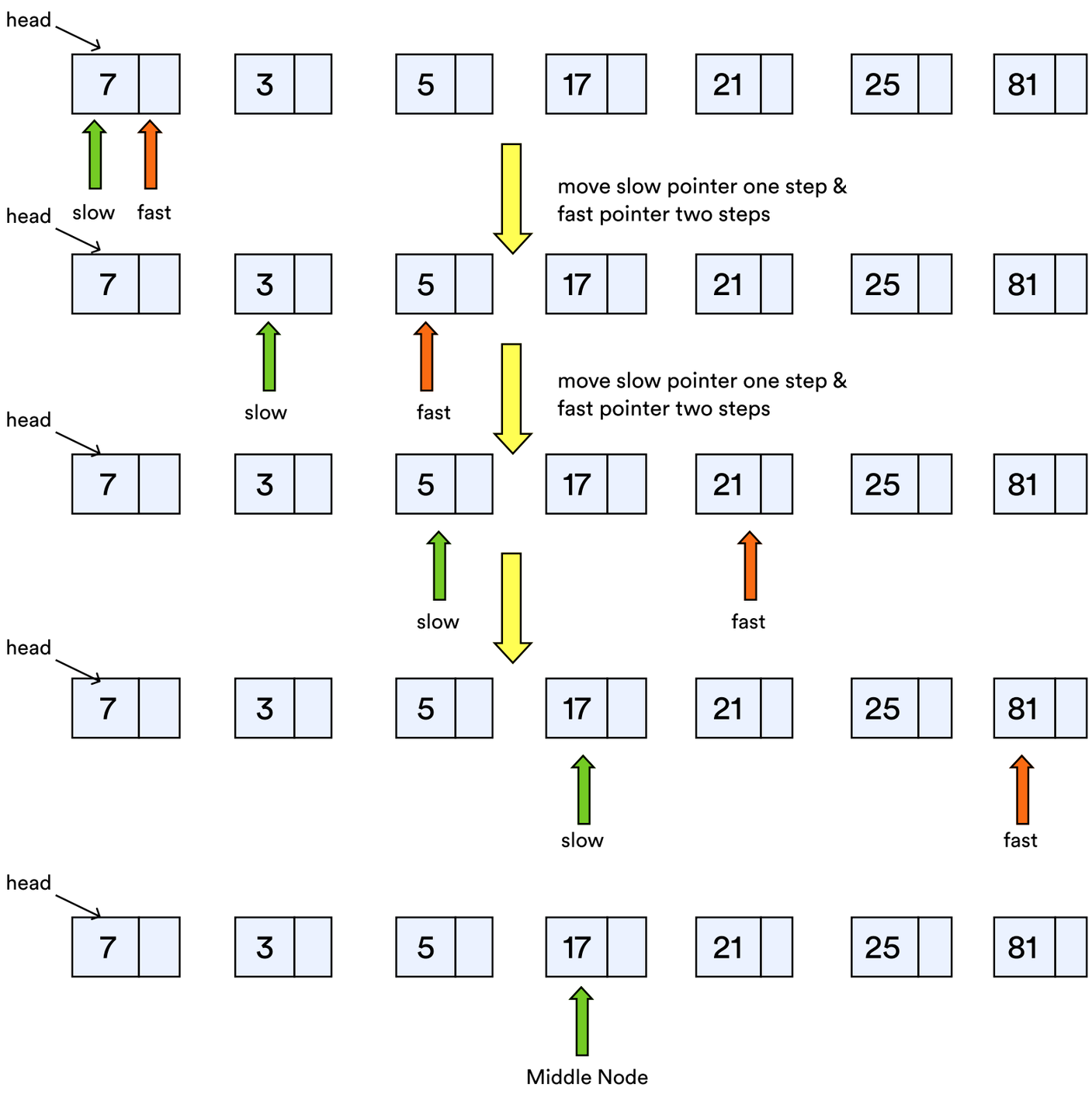
A naive method would include two traversals of the linked list, one to find the total length ‘n’ and one to traverse till the ‘n/2th’ node and return it.

To solve this problem in a single traversal we can use the Hare and Tortoise Approach

There are two pointers used in this method, slow and fast.

1. First with the fast pointer we move forward by two steps in each iteration
2. With the slow pointer we move forward by only one step in each iteration

When the fast pointer reaches the end of the linked list, the slow pointer will have reached the middle of the linked list.



Pseudocode for Fast and Slow Pointer Approach

```
slowPointer = head
fastPointer = head

while fastPointer is not null and fastPointer.next is not
null:
 // Move slow pointer one step
 slowPointer = slowPointer.next

 // Move fast pointer two steps
 fastPointer = fastPointer.next.next

// At this point, slowPointer is at the middle element
middleElement = slowPointer.data
```

Problems based on Slow and Fast Pointer:

1. Linked List Cycle - LeetCode
2. Happy Number - LeetCode
3. Palindrome Linked List - LeetCode
4. Reorder List - LeetCode
5. Circular Array Loop - LeetCode

Sliding Window

3

- Used to perform a required operation on a specific window size of a given array or linked list
- Useful for problems dealing with subarrays or sublists

Example Problem

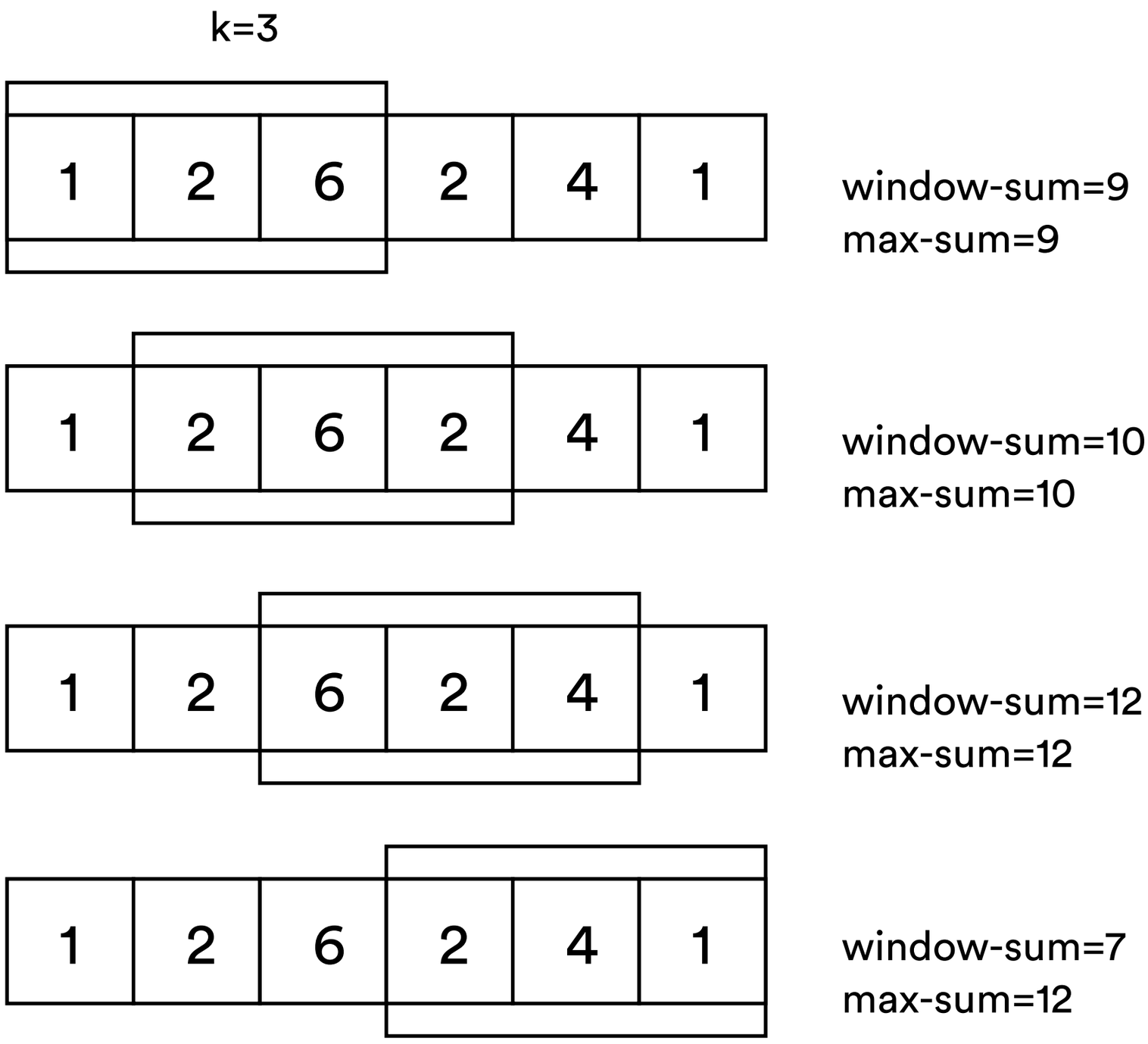
Given an array of positive numbers and a positive number K, find the maximum sum of any contiguous subarray of size K.

The naive solution will be to calculate the sum of all K sized subarrays of the given array to find the subarray with the highest sum.

This would result in a time complexity of O(N*K), where N is the total number of elements in the given array

We can improve this approach by using the sliding window technique to move the window over the array from index K to the end of the array.

1. In each iteration, the window-sum is updated by removing the element at the left side of the window (index i-K) and adding the element at the right side of the window (index i).
2. Check if the window-sum is greater than the current max-sum, and if so, update max-sum to keep track of the maximum sum found so far.



Pseudocode for the Sliding Window approach

```
function maxSumSubarrayOfSizeK(arr, K):
 // Check for invalid input
 if K > length of arr or K <= 0:
     return "Invalid input"

 // Initialize variables
 maxSum = 0
 currentSum = 0

 // Calculate the initial sum of the first window of size K
 for i from 0 to K - 1:
     currentSum = currentSum + arr[i]

 // Set the initial maxSum to the sum of the first window
 maxSum = currentSum

 // Slide the window and update maxSum until the end of the
array is reached
 for i from K to length of arr - 1:
     // Remove the element at the left of the window and
 add the element at the right
     currentSum = currentSum - arr[i - K] + arr[i]
     // Update maxSum if the currentSum is greater
     if currentSum > maxSum:
         maxSum = currentSum
 // Return the maximum sum found
 return maxSum
```

Problems based on Sliding Window Approach:

1. Maximum Average Subarray I - LeetCode
2. Minimum Size Subarray Sum - LeetCode
3. Longest Substring with At Most K Distinct Characters - LeetCode
4. Fruit Into Baskets - LeetCode
5. Longest Substring with At Most Two Distinct Characters - LeetCode
6. Longest Substring Without Repeating Characters - LeetCode
7. Max Consecutive Ones III - LeetCode
8. Minimum Window Substring - LeetCode
9. Find All Anagrams in a String - LeetCode

Merged Intervals

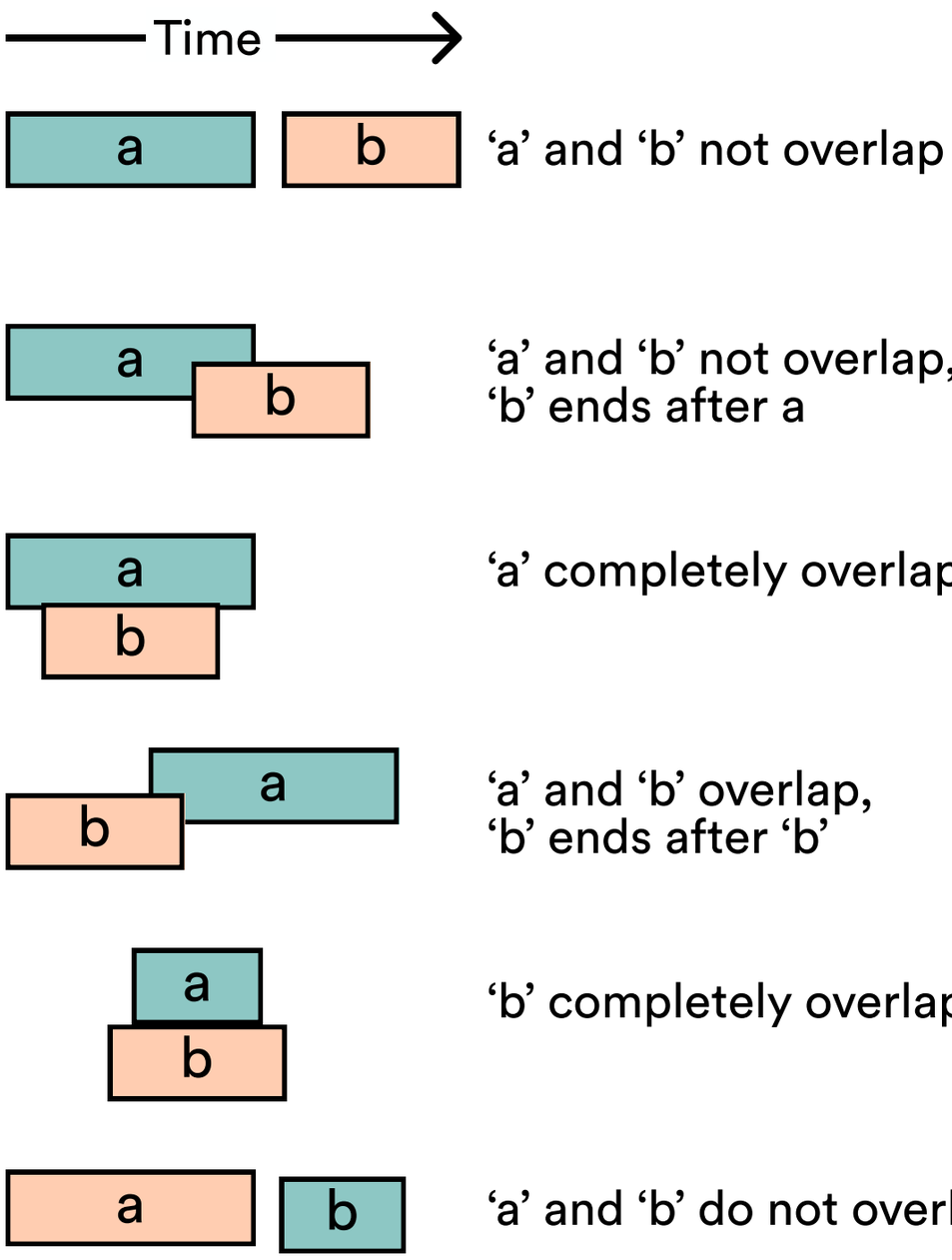
4

- This technique is used to deal with problems that require you to find overlapping intervals.

Example Problem

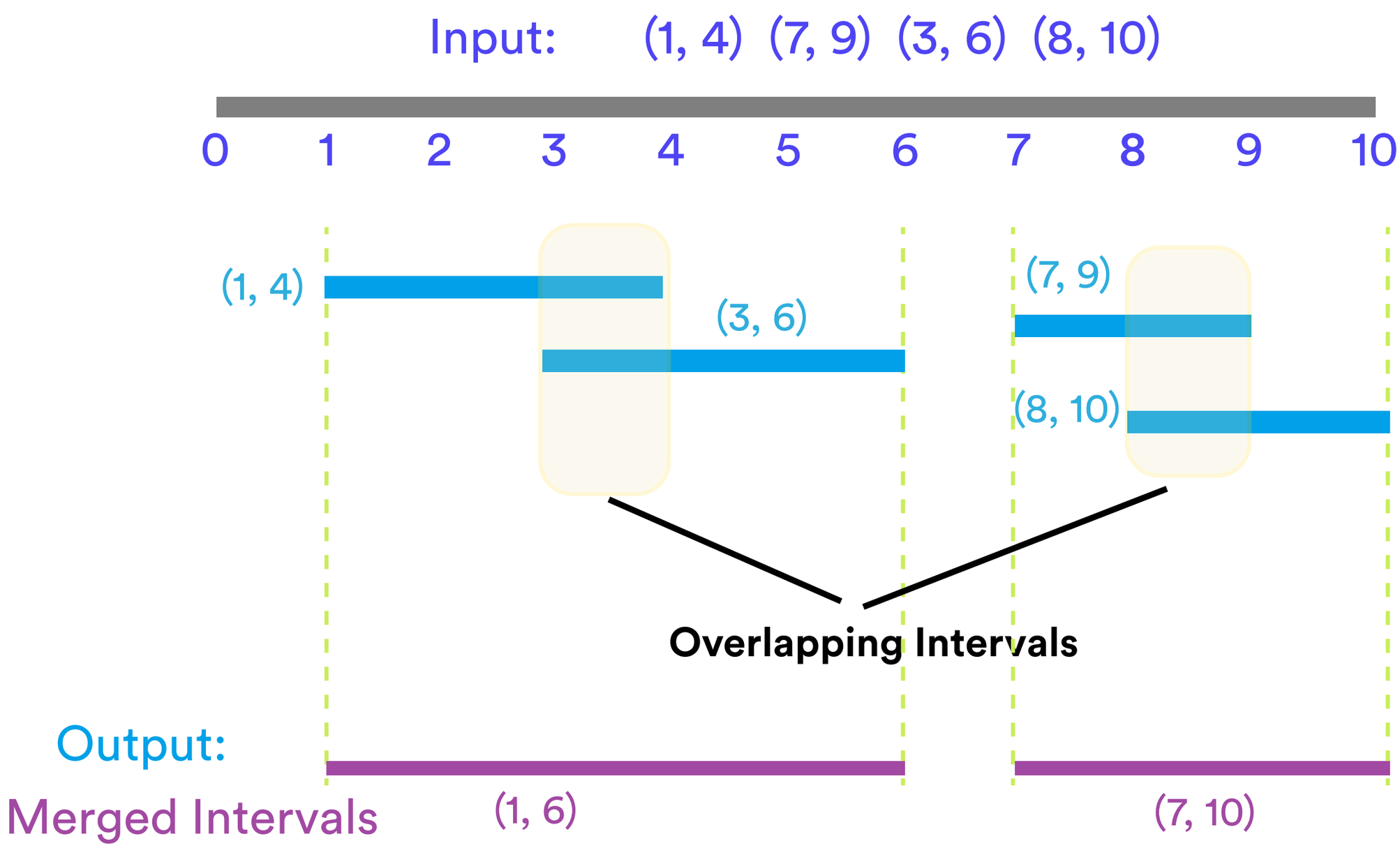
Given a list of intervals, merge all the overlapping intervals to produce a list that has only mutually exclusive intervals.

In general there are 6 possible combinations for a pair of intervals



- The brute force approach would involve starting from the first interval and comparing it with all other intervals
- If current overlaps with any other interval, then remove the other interval from the list and merge the other into the first interval
- This leads to a time complexity of O(n^2) where n is the number of intervals

Merge Overlapping Intervals



The optimised version to solve the problem

1. Sort the intervals according to the starting point of each interval.
2. Push the first interval into the stack.
3. Iterate over each interval:
    a. If the current interval does not overlap with the top of the stack then, push the current interval into the stack.
    b. If the current interval does overlap with the top of the stack then, update the stack top with the ending time of the current interval.
4. The stack now has the output merged intervals.

This version would have a time complexity of O(n*log(n)).

Pseudocode for the optimised approach

```
function mergeIntervals(arr, n):
    // Test if the given set has at least one interval
    if n <= 0:
        return

    // Create an empty stack of intervals
    stack s

    // Sort the intervals in increasing order of start time
    sort arr

    // Push the first interval to the stack
    s.push(arr[0])

    // Start from the next interval and merge if necessary
    for i from 1 to n-1:
        // Get the interval from the top of the stack
        top = s.top()

        // If the current interval is not overlapping with
the stack top, push it to the stack
        if top.end < arr[i].start:
            s.push(arr[i])

        // Otherwise, update the ending time of top if the
ending of the current interval is more
        else if top.end < arr[i].end:
            top.end = arr[i].end
            s.pop()
            s.push(top)
    return s
```

Problems based on Merge Intervals Approach:

1. Insert Interval - LeetCode

2. Interval List Intersections - LeetCode

3. Meeting Rooms - LeetCode

4. Meeting Rooms II - LeetCode

5. Car Pooling - LeetCode

6. Employee Free Time - LeetCode

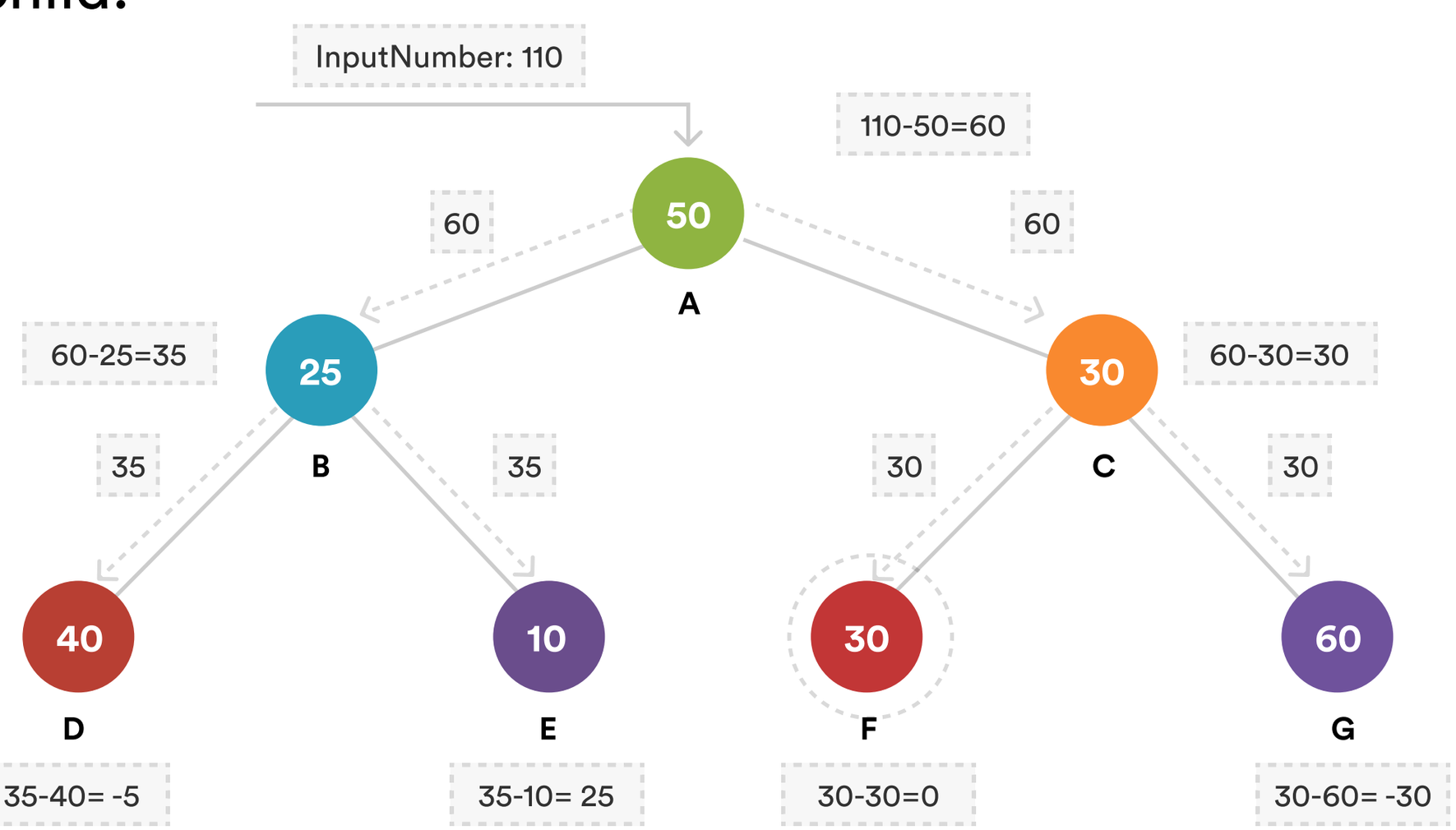
Depth First Search (DFS)

- Recursive Algorithm used to search all the vertices of all a graph or a tree
- It starts from a chosen source node and explores as far as possible along each branch before backtracking

Example Problem

Given a binary tree and a number S, find if the tree has a path from root-to-leaf such that the sum of all the node values of that path equals S.

This fits the DFS pattern as we are searching for a root-to-leaf path
To solve this problem we can start from the root and at every step, make two recursive calls one for the left and one for the right child.



DFS Approach

1. Start from the root of the tree and traverse the tree in a depth-first manner.
2. If the current node is NOT a leaf node, do:
    - Subtract the value of the current node from the given target sum to get updated sum value → S = S - node.value
    - Make two recursive calls for left and right children of the current node with updated sum value
3. At every step, check if the current node is a leaf node and if its value is equal to target sum S. If both conditions are met, we have found the required root-to-leaf path, therefore return true.

- If the current node is a leaf but its value is not equal to the given number S, return false.

Pseudocode for DFS Approach

```
function hasPath(root, sum):
    // If the current node is null (empty tree), return
false
    if root is null:
        return false

    // If the current node is a leaf node and its value
matches the sum we've found a path, so return true
    if root.value is equal to sum and root.left is null and
root.right is null:
        return true

    // Recursively call the function on the left and right
subtrees
    // Return true if any of the two recursive calls return
true,
    // indicating that a path with the desired sum was
found
    return hasPath(root.left, sum - root.value) OR
hasPath(root.right, sum - root.value)
```

Problems based on DFS Approach:

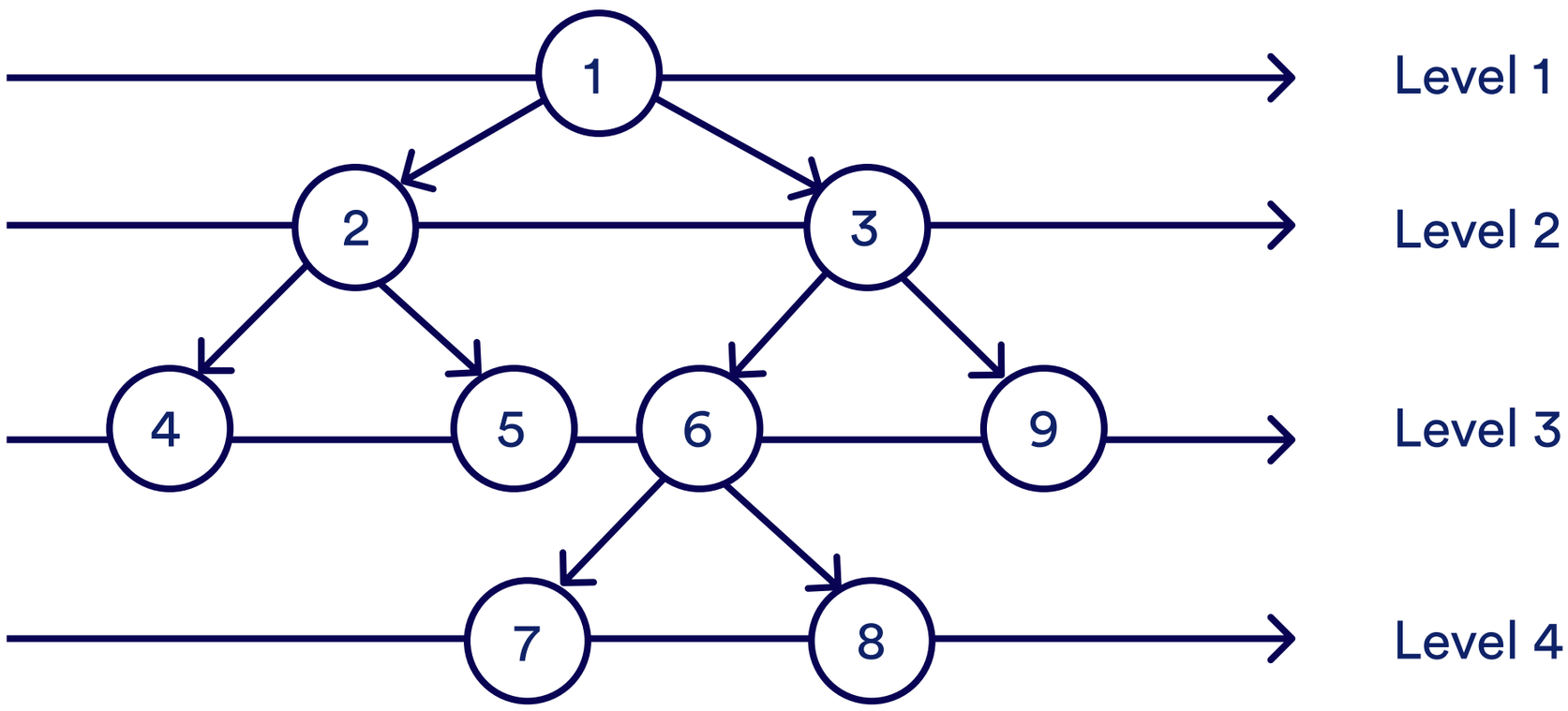
1. Path Sum II - LeetCode
2. Binary Tree Paths - LeetCode
3. Sum Root to Leaf Numbers - LeetCode
4. Path Sum III - LeetCode
5. Diameter of Binary Tree - LeetCode
6. Binary Tree Maximum Path Sum - LeetCode

Breadth First Search (BFS)

- BFS is a graph traversal algorithm that systematically explores all the nodes in a graph by visiting nodes in layers or levels.
- Starts from a chosen source node, visits all its neighbors, then moves on to their neighbors, and so on, until all nodes are visited or a specific condition is met.
- Uses a queue data structure

Example Problem

Given a binary tree, populate an array to represent its level-by-level traversal. You should populate the values of all nodes of each level from left to right in separate sub-arrays.



∴ LEVEL ORDER: 1, 2, 3, 4, 5, 6, 9, 7, 8

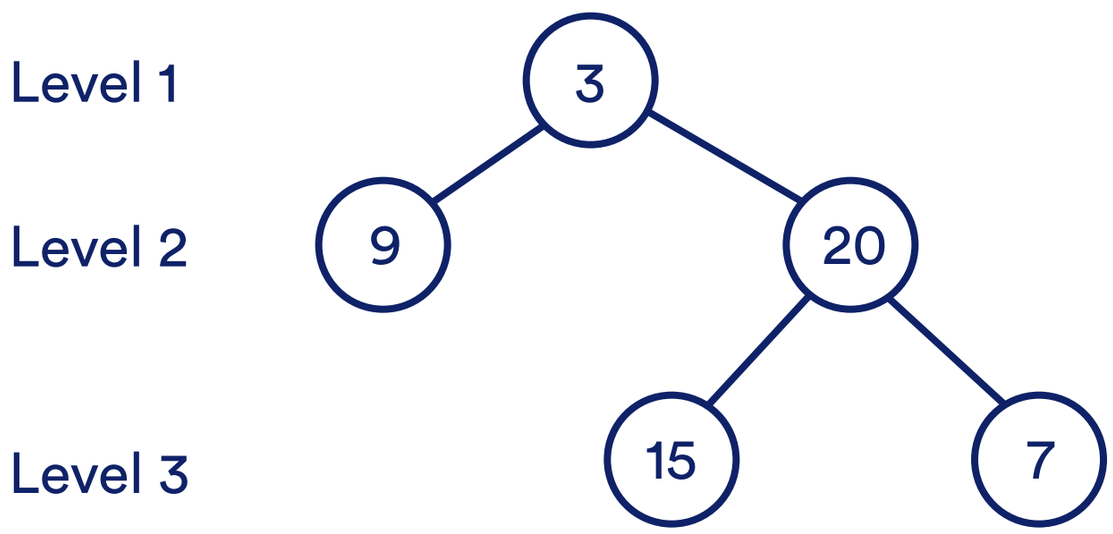
BFS Based Approach:

We basically have to visit the nodes of a higher level before visiting nodes of a lower level
For this we use a queue data structure.

1. Start by pushing the root node to the queue.
2. Keep iterating until the queue is empty.
3. In each iteration, count the elements in the queue (let's call it levelSize). This is the number of nodes in the current level.
4. Remove levelSize nodes from the queue and push their value in an array to represent the current level.
5. After removing each node from the queue, insert both of its children into the queue.

If the queue is not empty, repeat from step 3 for the next level.

BFS Based Approach:



We start by pushing root 3 into the queue.

QUEUE

3					

Since the queue is not empty 3 is popped out and added to the final traversal order.
Its children 9 and 20 are pushed into the queue

QUEUE

9	20				

Queue is not empty so 9 is popped out and added to final traversal order
It has no children to push into the queue.

QUEUE

20					

Now 20 is popped out and added to the final traversal order. Its children 15 and 7 are pushed into queue

QUEUE

15	7				

15 and 7 are popped out with the same process and we obtain final level order traversal

This approach results in a time complexity of O(N) where N is the number of nodes in the binary tree.

Problems based on BFS Approach:

1. Binary Tree Level Order Traversal II - LeetCode
2. Binary Tree Zigzag Level Order Traversal - LeetCode
3. Average of Levels in Binary Tree - LeetCode
4. Maximum Level Sum of a Binary Tree - LeetCode
5. Minimum Depth of Binary Tree - LeetCode
6. Maximum Depth of Binary Tree - LeetCode
7. Populating Next Right Pointers in Each Node - LeetCode
8. Binary Tree Right Side View - LeetCode

BOSSCODER ACADEMY

WHY BOSSCODER?

- **750+** Alumni placed at Top Product-based companies.
- More than **136% hike** for every **2 out of 3** working professional.
- Average package of **24LPA.**

The syllabus is most **up-to-date** and the list of problems provided covers **all important topics.**

Lavanya
Meta

Course is very well **structured** and **streamlined** to crack any **MAANG** company

Rahul .
Google

EXPLORE MORE