# Data Structures & Algorithms (DS&A) for the Technical Job Interview

**Yingquan Li (Wed. 11/8/23 @ 6:00 PM EST)**
**Washington D.C. ACM Meetup**

# Technical Interviews Are:

1. 50% Communication/Likeability.
2. 50% Technical ability and live coding skills.

This talk will be focused **exclusively** on the second part:

**50% Technical ability and live coding skills**

# Think Like a Programmer (V. Anton Spraul, 2012)

**General Problem-Solving Techniques**

- **Always Have a Plan**
- Restate the Problem
- Divide the Problem
- Start with What You Know
- Reduce the Problem
- Look for Analogies
- Experiment
- Don’t Get Frustrated

# Found @ The Gaithersburg Library

**Your Words Change Your Mindset**

| **Fixed** | **Growth** |
|---|---|
| I'll never be as smart as that person. | Everyone is talented in many ways. |
| **I can't do it**. | **I can try a different strategy**. |
| This is too hard. | With more **practice**, it will be easier. |
| I'm not good at this. | This may take some **time** and **effort**. |
| I made a mistake. | **Mistakes help me learn**. |
| I give up. | I'm still learning. I'll keep trying. |

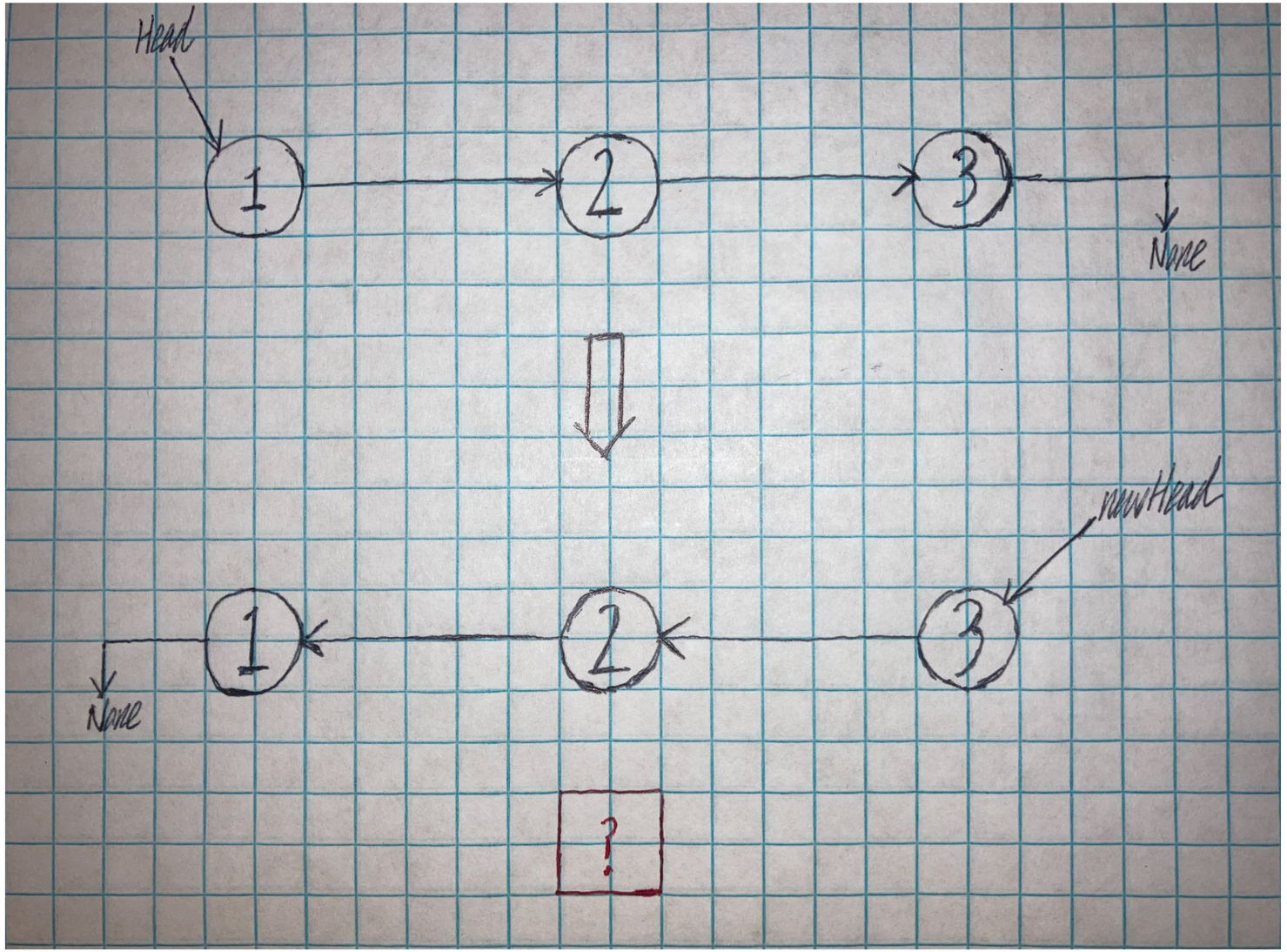
# List of Problem Solving Techniques

## We'll see how many we can do! Source: LeetTrack Mobile App.

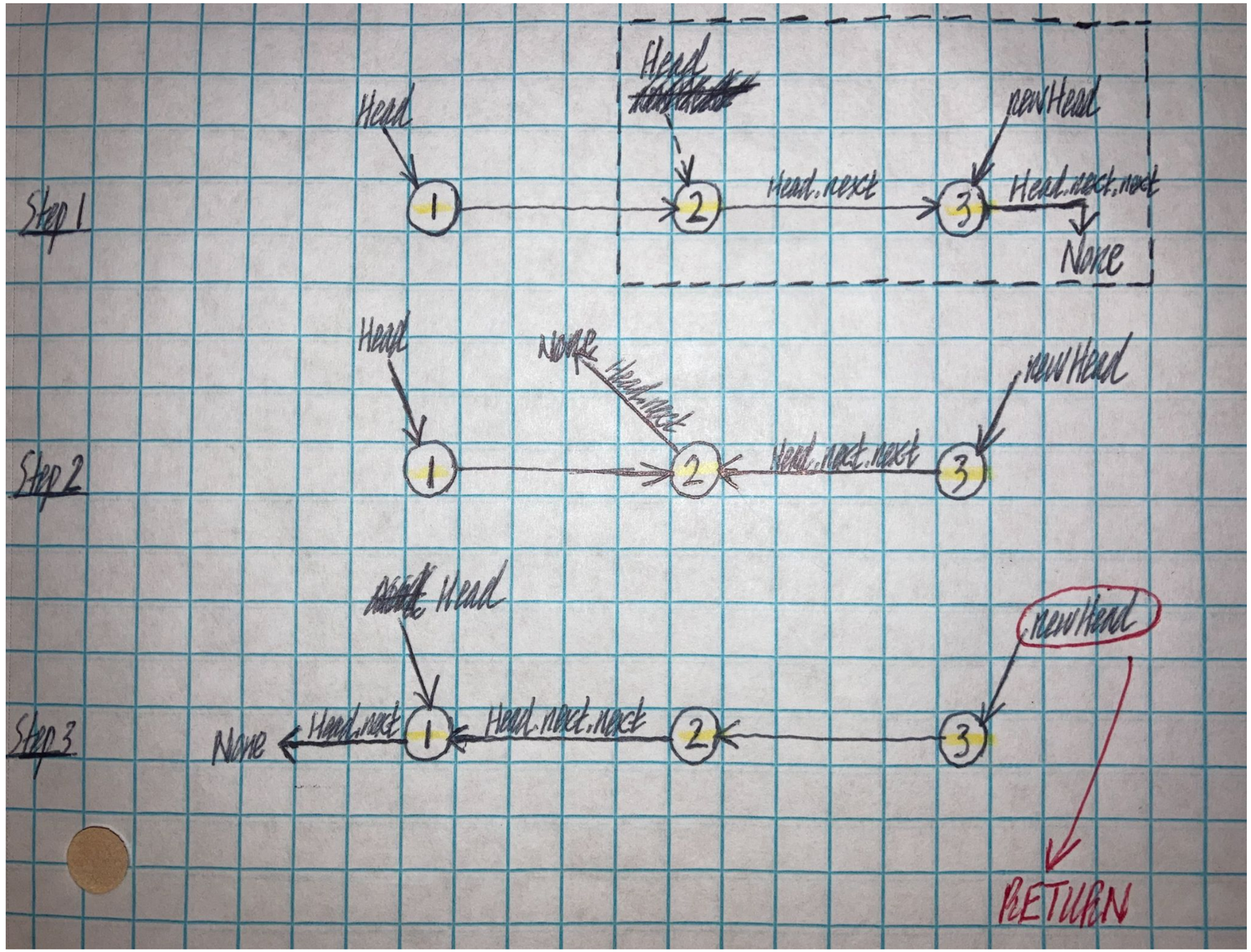
1. In-Place Reversal of Linked List
2. Fast and Slow Pointer
3. Two Pointers/Modified Binary Search
4. Sliding Window
5. Backtracking
6. Tree Depth-First Search (DFS)
7. Tree Breadth-First Search (BFS)
8. Cyclic Sort
9. Merge Intervals
10. Two Heaps
11. Top K Elements
12. K-Way Merge
13. Topological Sort

## We'll practice from the LeetCode platform! If you have an account, pull it up.

# In-Place Reversal of Linked List (Warmup Problem: LC #206)



# In-Place Reversal of Linked List (Intuition)

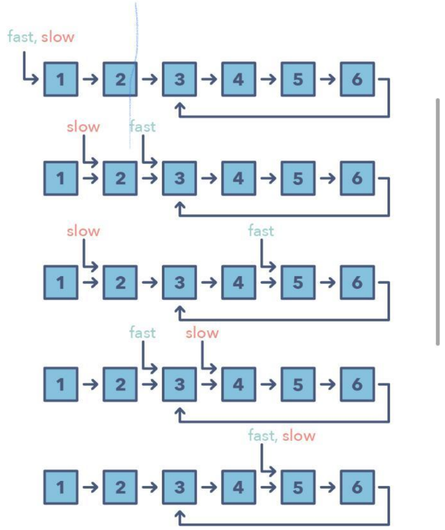



TC: O(N)
SC: O(N)

# Fast and Slow Pointers

The Fast and Slow Pointer approach, also known as the **Hare & Tortoise algorithm**, is a pointer algorithm that uses two pointers which move through the array (or sequence/linked list) at different speeds. This approach is quite useful when dealing with cyclic linked lists or arrays.

By moving at different speeds (say, in a cyclic linked list), the algorithm proves that the two pointers are bound to meet. The fast pointer should catch the slow pointers once with the pointers are in a cyclic loop.




**How do you identify when to use the Fast and Slow pattern**?

- The problem will deal with a loop in a linked list or array.

- When you need to know the position of a certain element or the overall length of the linked list.

**When should I used it over the Two Pointer method mentioned above**?

- There are some cases where you shouldn't use the Two Pointer approach such as in a singly linked list where you can't move in a backwards direction. An example of when to use the Fast and Slow pattern is when you're trying to determine if a linked list is a palindrome.

# Fast and Slow Pointers (Problem: LC #876)

## 876. Middle of the Linked List

Solved

Easy | Topics | Companies

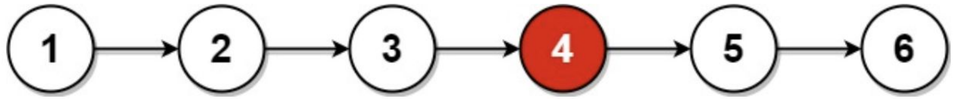
Given the `head` of a singly linked list, return *the middle node of the linked list*.

If there are two middle nodes, return **the second middle** node.

**Example 1:**


```
Input: head = [1,2,3,4,5]
Output: [3,4,5]
Explanation: The middle node of the list is node 3.
```

**Example 2:**

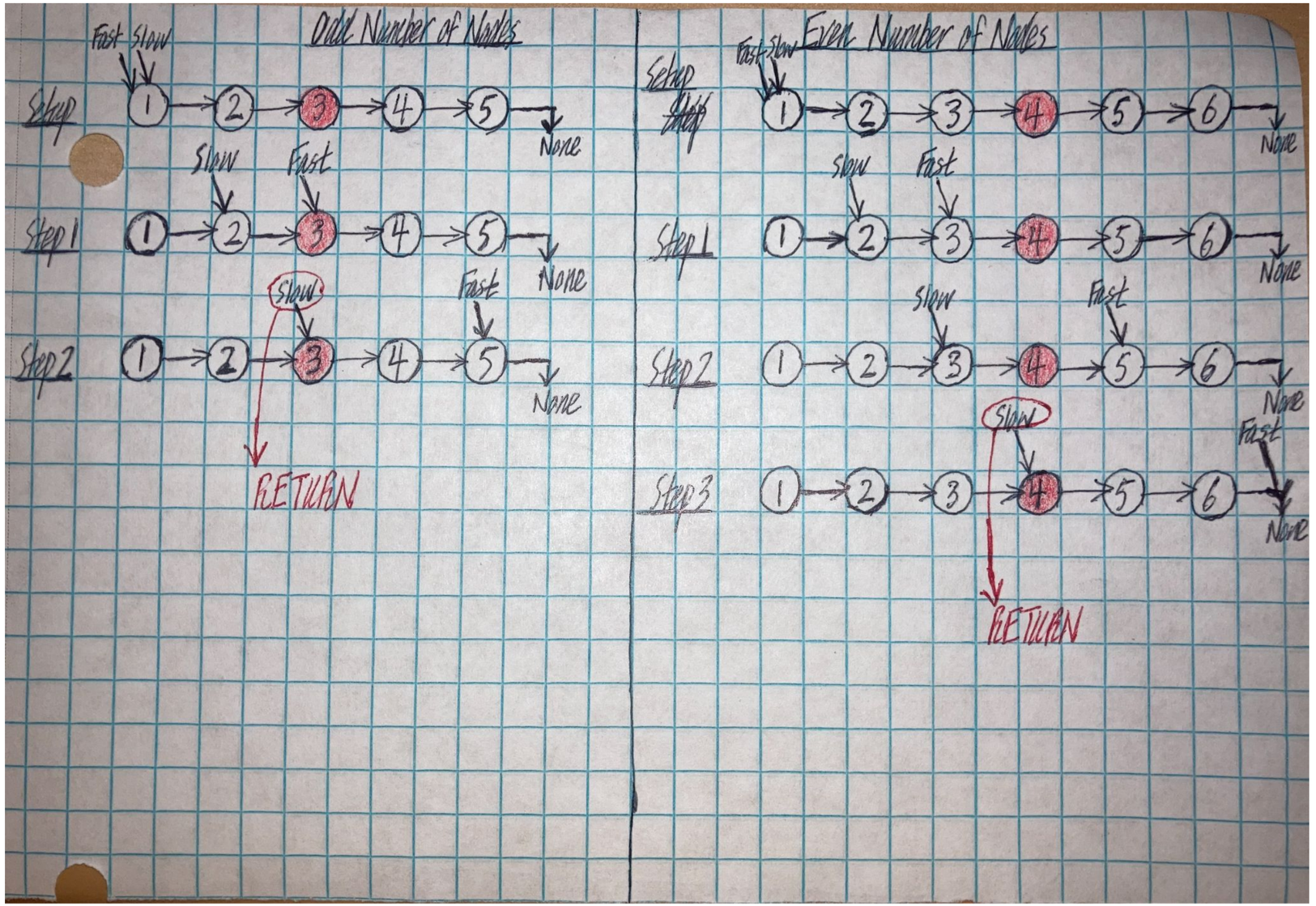



```
Input: head = [1,2,3,4,5,6]
Output: [4,5,6]
Explanation: Since the list has two middle nodes with values 3 and 4,
we return the second one.
```

**Constraints:**

- The number of nodes in the list is in the range `[1, 100]`.
- `1 <= Node.val <= 100`

# Fast and Slow Pointers (Intuition)

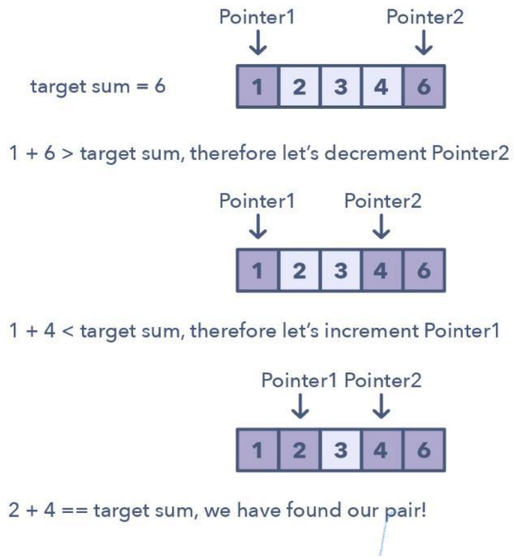



TC: O(N)
SC: O(1)

# Two Pointers

Two Pointers is a pattern where two pointers iterate through the data structure in tandem until one or both of the pointers hit a certain condition. Two Pointers is often useful when searching pairs in a sorted array or linked list; for example, when you have to compare each element of an array to its other elements.

Two Pointers are needed because with just pointer, you would have to continually loop back through the array to find the answer. This back and forth with a single iterator is inefficient for time and space complexity - a concept referred to as **asymptotic analysis**. While the brute force or naive solution with 1 pointer would work, it will produce something along the lines of O(N^2). In many cases, two pointers can help you find a solution with better space or runtime complexity.




**Ways to identify when to use the Two Pointer method**:

- The problem input is a linear data structure such as a linked list, array, or string.

- The set of elements in the array is a pair, a triplet, or even a subarray.

# Two Pointers (Problem: LC #167)

## 167. Two Sum II - Input Array Is Sorted

Solved

Medium | Topics | Companies

Given a **1-indexed** array of integers `numbers` that is already ***sorted in non-decreasing order***, find two numbers such that they add up to a specific `target` number. Let these two numbers be `numbers[index1]` and `numbers[index2]` where `1 <= index1 < index2 < numbers.length`.

Return *the indices of the two numbers,* `index1` *and* `index2`*,* ***added by one*** *as an integer array* `[index1, index2]` *of length 2.*

The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.

Your solution must use only constant extra space.

**Example 1:**

> **Input:** numbers = [2,7,11,15], target = 9
> **Output:** [1,2]
> **Explanation:** The sum of 2 and 7 is 9. Therefore, $index_1$ = 1, $index_2$ = 2. We return [1, 2].

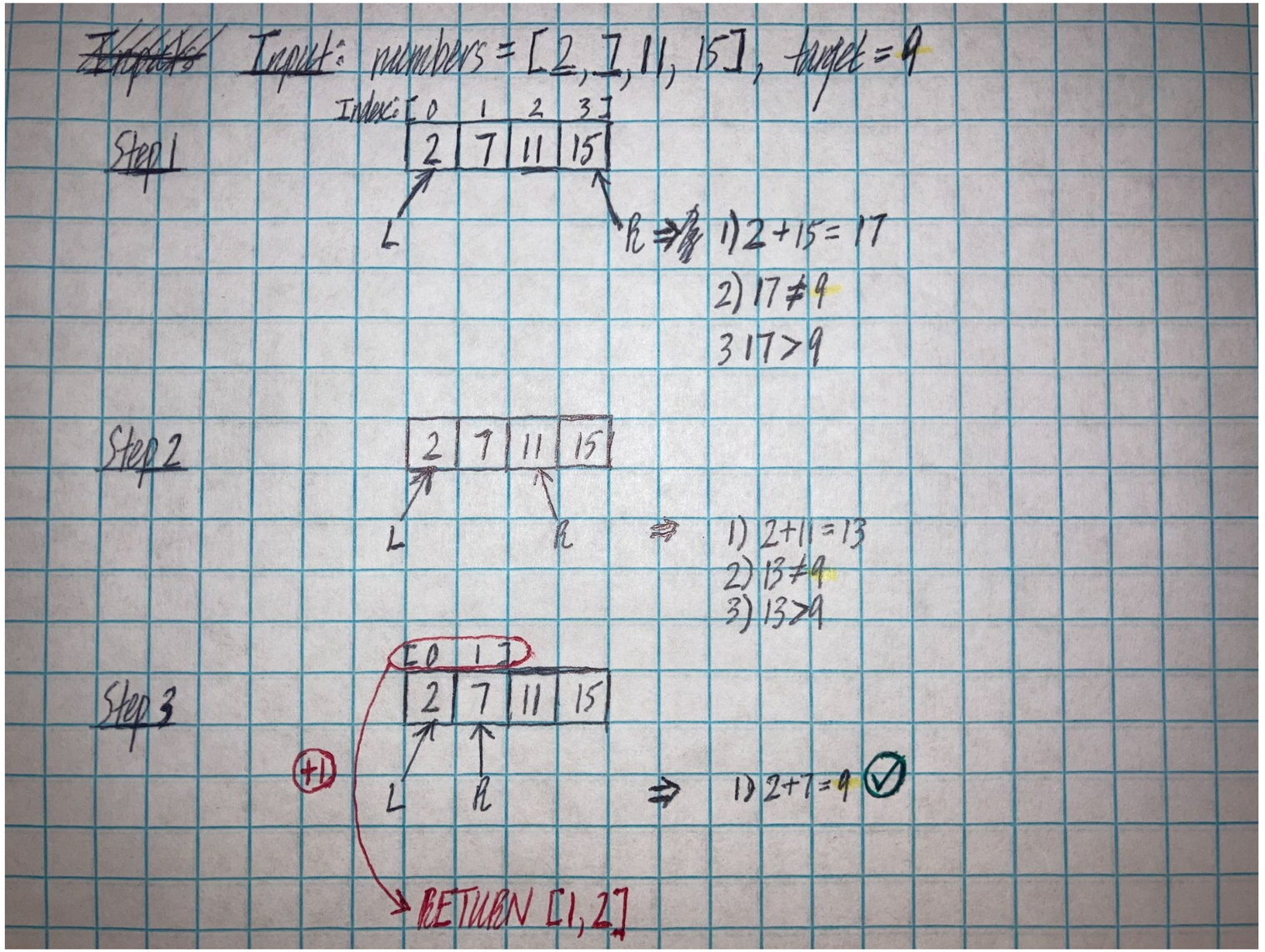
**Example 2:**

> **Input:** numbers = [2,3,4], target = 6
> **Output:** [1,3]
> **Explanation:** The sum of 2 and 4 is 6. Therefore $index_1$ = 1, $index_2$ = 3. We return [1, 3].

**Example 3:**

> **Input:** numbers = [-1,0], target = -1
> **Output:** [1,2]
> **Explanation:** The sum of -1 and 0 is -1. Therefore $index_1$ = 1, $index_2$ = 2. We return [1, 2].

# Two Pointers (Intuition)




TC: O(N)
SC: O(1)

# Two Pointers in Action: Modified Binary Search

Whenever you are given a *sorted array*, *linked list*, or *matrix*, and are asked to find a certain element, the best algorithm you can use is the **Binary Search**. This pattern describes an efficient way to handle all problems involving Binary Search.

**The pattern looks like this for an ascending order set**:

1. First, find the middle of start and end. An easy way do find the middle would be: *middle = (start + end) / 2*. But this has a good chance of producing an integer overflow so it's recommended that you represent the middle as: *middle = start + (end - start) / 2*.

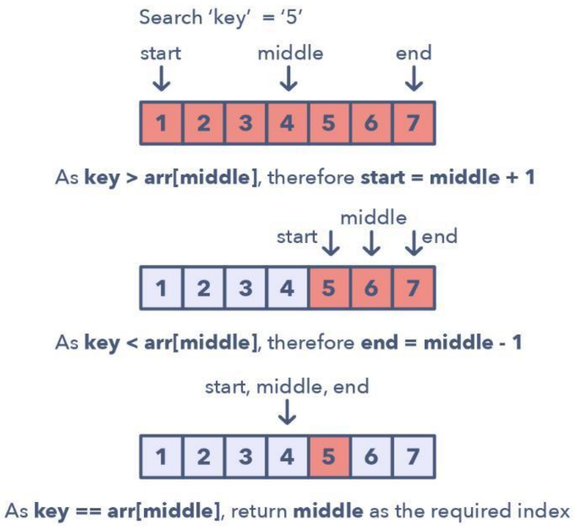
2. If the *'key'* is equal to the number at index middle then return middle.

3. If *'key'* isn't equal to the index middle:

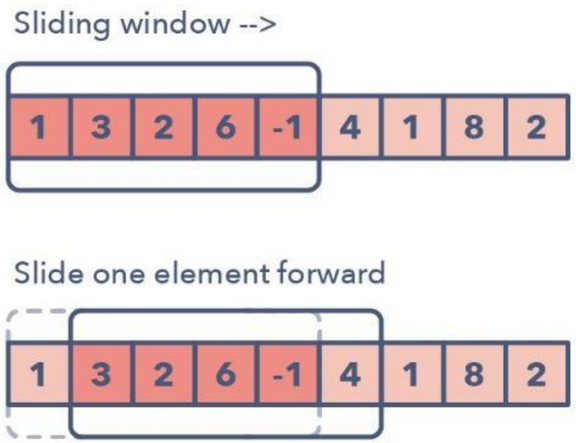
4. Check if *'key'* < arr[middle]. If it is reduce your search to *end = middle - 1*.

5. Check if *'key'* > arr[middle]. If it is reduce your search to *start = middle + 1*.

**Here is a visual representation of the Modified Binary Search pattern**:



# Sliding Window




The Sliding Window pattern is used to perform a required operation on a specific window size of a given array or linked list, such as finding the longest subarray containing all 1s. Sliding Windows start from the 1st element and keep shifting right by one element and adjusting the length of the window according to the problem that you are solving. In some cases, the window size remains constant and in other the sizes grows or shrinks.

**Following are some ways you can identify that the given problem might require a sliding window**:

- The problem input is a linear data structure such as a linked list, array, or string.

- You’re asked to find the longest/shortest substring, subarray, or a desired value.

# Sliding Window (Problem: LC: #239)

## 239. Sliding Window Maximum

Solved

Hard | Topics | Companies | Hint

You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Return *the max sliding window*.

**Example 1:**

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

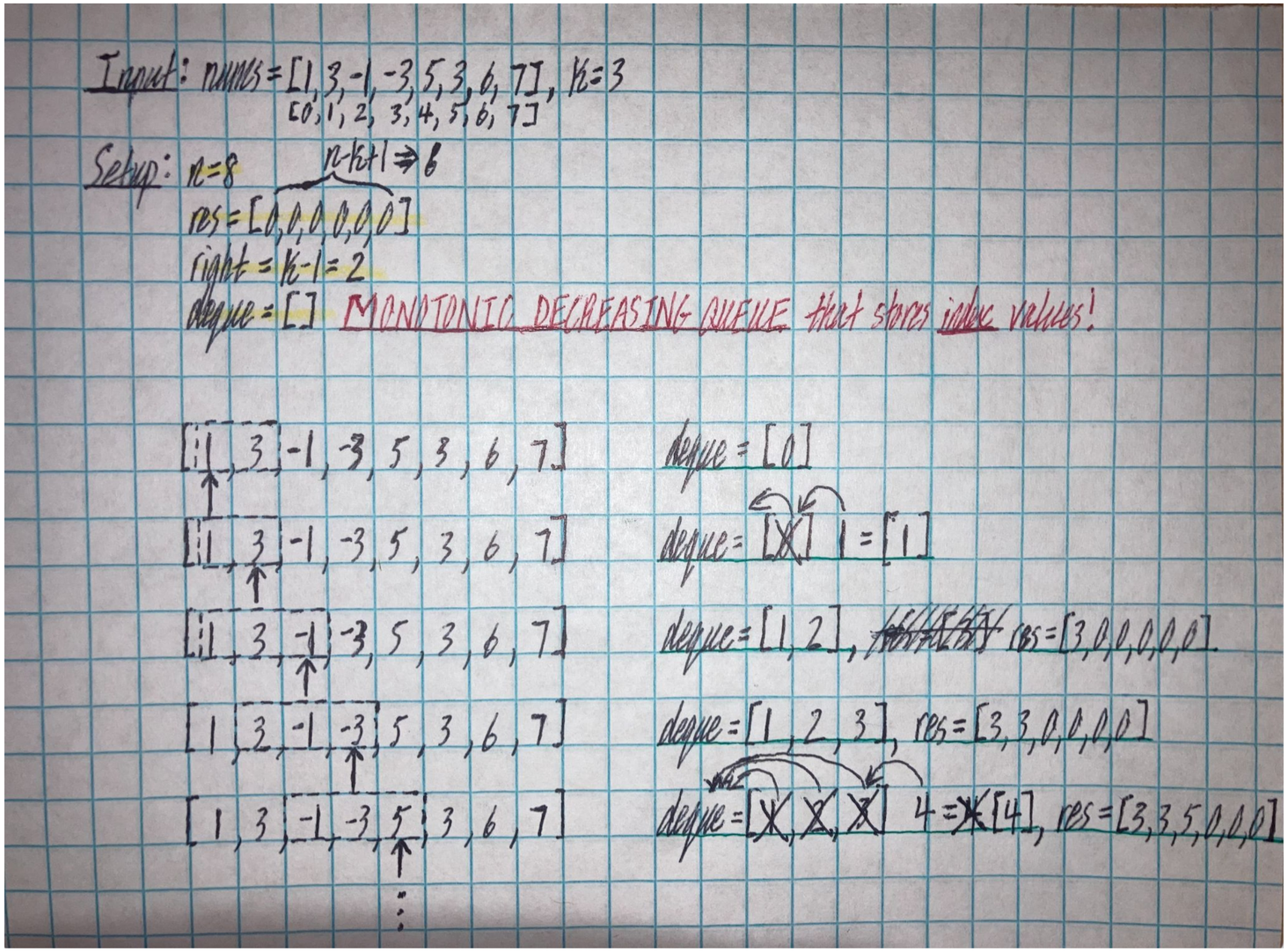
**Example 2:**

```
Input: nums = [1], k = 1
Output: [1]
```

**Constraints:**

- `1 <= nums.length <= 10^5`
- `-10^4 <= nums[i] <= 10^4`
- `1 <= k <= nums.length`

# Sliding Window (Intuition)

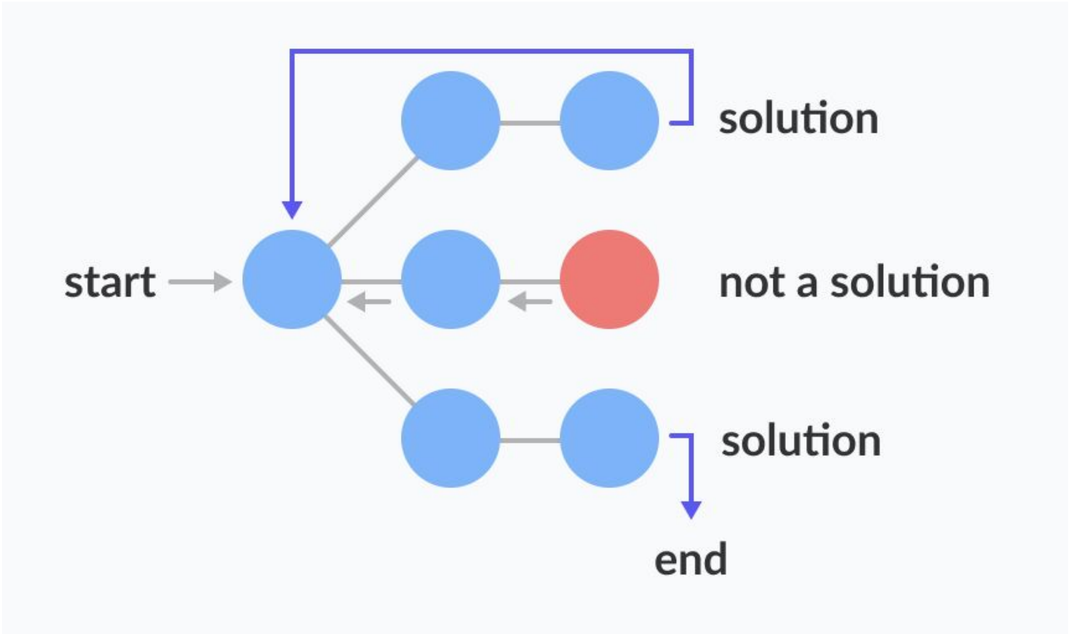



TC: O(N)
SC: O(k)

# Backtracking

**Backtracking** is a brute-force algorithmic approach that often involves recursion that incrementally builds up solutions. If a solution is tried and found to not be a correct solution, then the algorithm “backtracks” and tries to find another solution with different constituent parts and checks the validity of this new solution.

Backtracking is among the *most confusing* patterns to apply correctly because it often involves recursion and the help of an auxiliary function, but it’s not impossible and so the key is just more practice.




**How do you identify when to use the backtracking pattern**?

- Decision problems, optimization problems, as well as enumeration problems are good to apply the backtracking algorithm.
- Basically, searching for a solution among a large set of possibilities is fair game for backtracking.

# Backtracking (Problem: LC #78)

## 78. Subsets

Medium | Topics | Companies

Given an integer array `nums` of **unique** elements, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

**Example 1:**

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

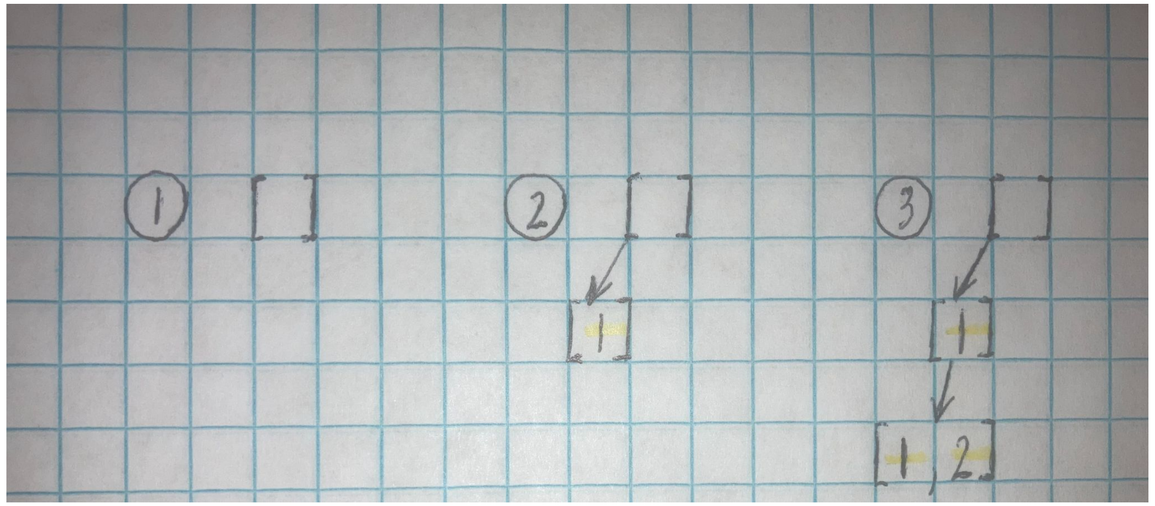
**Example 2:**

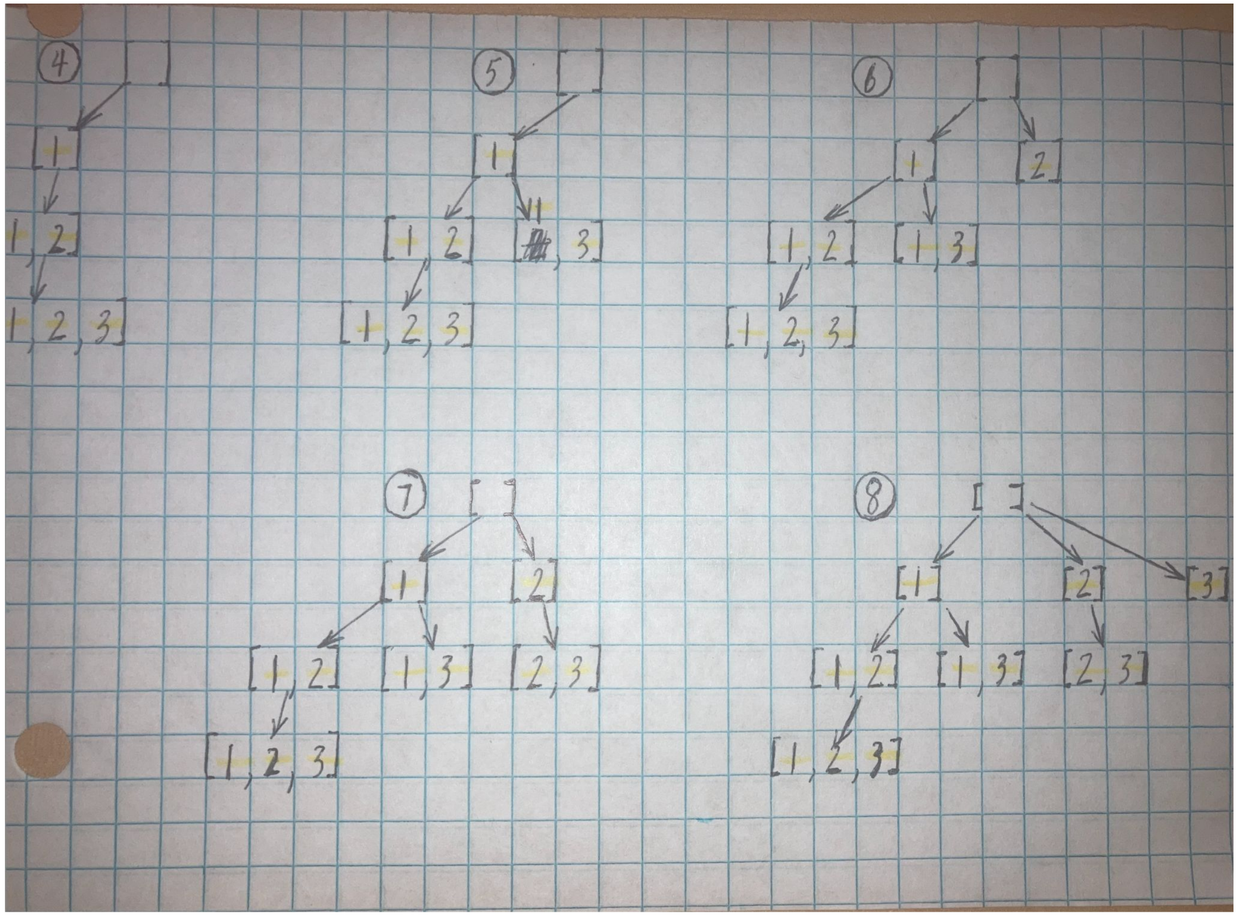
```
Input: nums = [0]
Output: [[],[0]]
```

**Constraints:**

- `1 <= nums.length <= 10`
- `-10 <= nums[i] <= 10`
- All the numbers of `nums` are **unique**.

# Backtracking (Intuition)




TC: $O(N*2^N)$
SC: $O(N*2^N)$

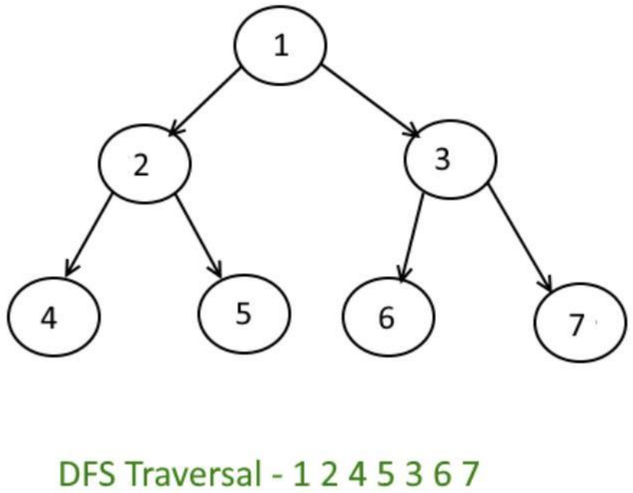
# Backtracking in Action: Tree Depth-First Search (DFS)

Tree DFS is based on the **Depth First Search** (**DFS**) technique to traverse a tree.

You can use *recursion* (or a *stack* for the iterative approach) to keep track of all the previous (parent) nodes while traversing.

The Tree DFS pattern works by starting at the root of the tree, if the node is not a leaf you need to do three things:

1. Decide whether to process the **current node** (**pre-order**), or **between processing two children** (**in-order**) or **after processing both children** (**post-order**).
2. Make two recursive calls for both the children of the current node to process them.




DFS Traversal - 1 2 4 5 3 6 7

**How to identify the Tree DFS pattern**?

- If you're asked to traverse a tree with in-order, pre-order, or post-order DFS.
- If the problem requires searching for something where the node is closer to a leaf.

# Tree Depth-First Search (DFS) (Problem: LC #814)

## 814. Binary Tree Pruning

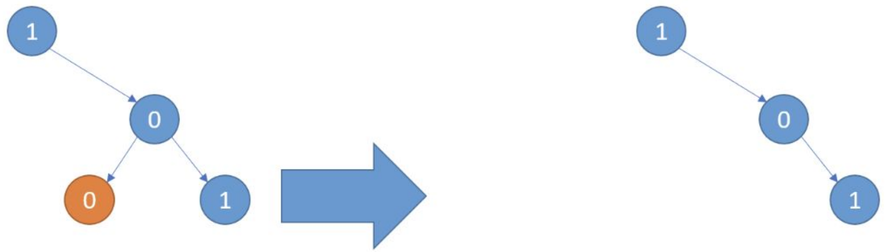
Solved

Medium | Topics | Companies

Given the `root` of a binary tree, return *the same tree where every subtree (of the given tree) not containing a* `1` *has been removed*.

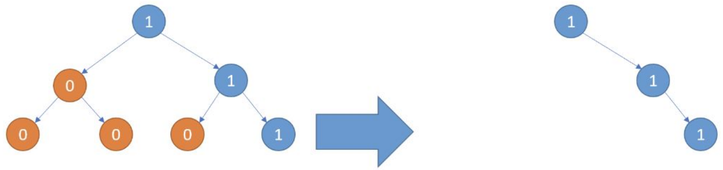
A subtree of a node `node` is `node` plus every node that is a descendant of `node`.

**Example 1:**



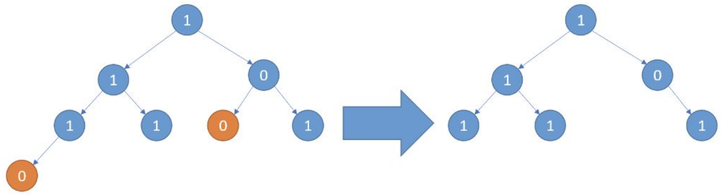

```
Input: root = [1,null,0,0,1]
Output: [1,null,0,null,1]
Explanation:
Only the red nodes satisfy the property "every subtree not containing a 1".
The diagram on the right represents the answer.
```

**Example 2:**




```
Input: root = [1,0,1,0,0,0,1]
Output: [1,null,1,null,1]
```

**Example 3:**

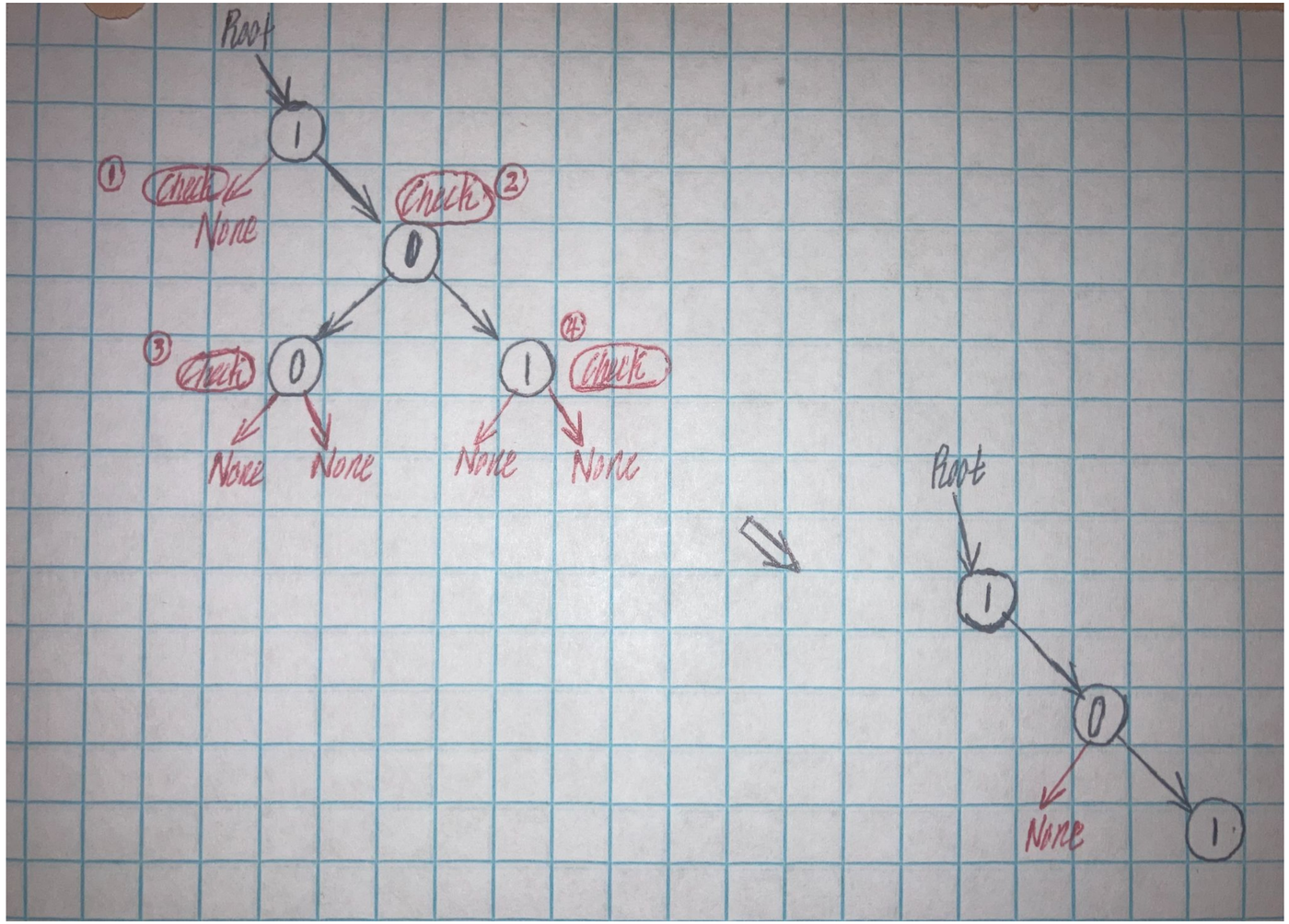



```
Input: root = [1,1,0,1,1,0,1,0]
Output: [1,1,0,1,1,null,1]
```

**Constraints:**

- The number of nodes in the tree is in the range `[1, 200]`.
- `Node.val` is either `0` or `1`.

# Tree Depth-First Search (DFS) (Intuition)

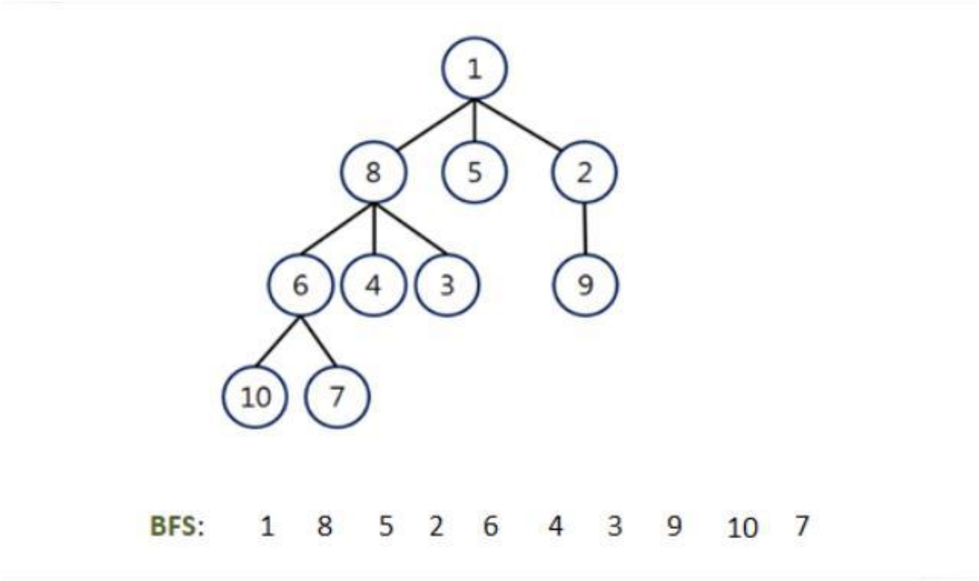



TC: O(N)
SC: O(H)

# Tree Breadth-First Search (BFS)

This pattern is based on the **Breadth First Search** (**BFS**) technique to traverse a tree and uses a *queue* to keep track of all the nodes of a level before jumping onto the next level. Any problem involving the traversal of a tree in a level-by-level order can be efficiently solved using this approach.

The Tree BFS pattern works by pushing the root node to the queue and then continually iterating until the queue is empty. For each iteration, we remove the node at the head of the queue and "visit" that node. After removing each node from the queue, we also insert all of its children into the queue.




**How to identify the Tree BFS pattern**?

- If you're asked to traverse a tree in a level-by-level fashion (or level order traversal).

# Tree Breadth-First Search (BFS) (Problem: LC #102)

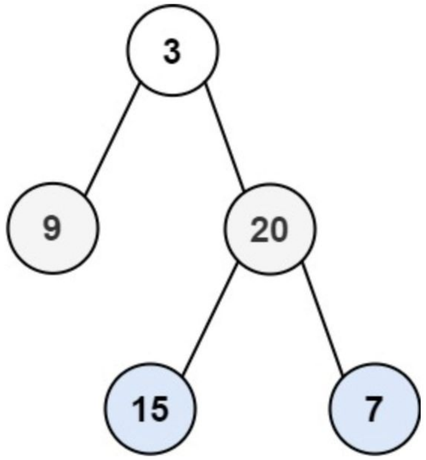
## 102. Binary Tree Level Order Traversal

Solved

Medium | Topics | Companies

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

**Example 1:**




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

**Example 2:**

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

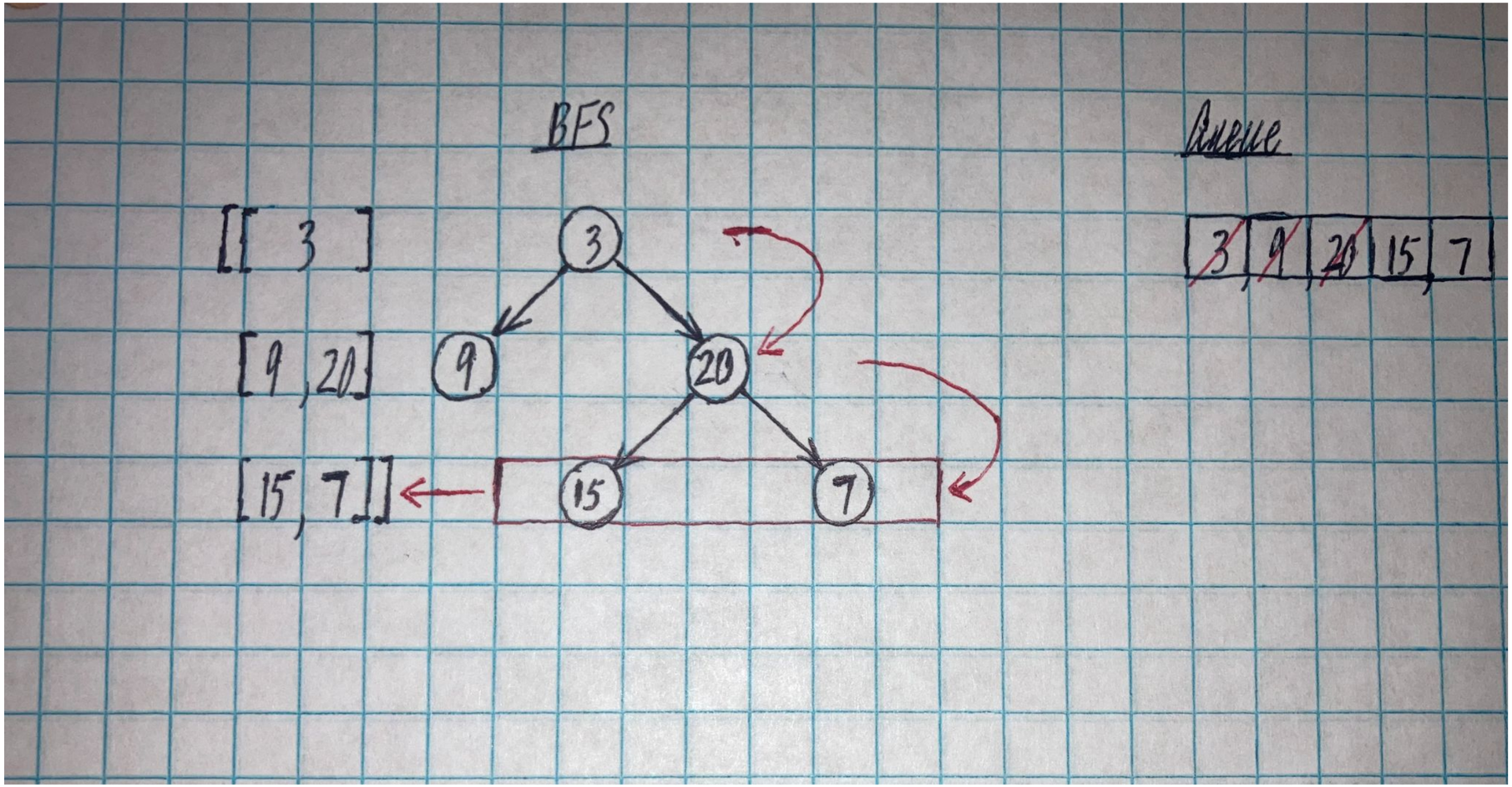
**Example 3:**

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

**Constraints:**

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

# Tree Breadth-First Search (BFS) (Intuition)



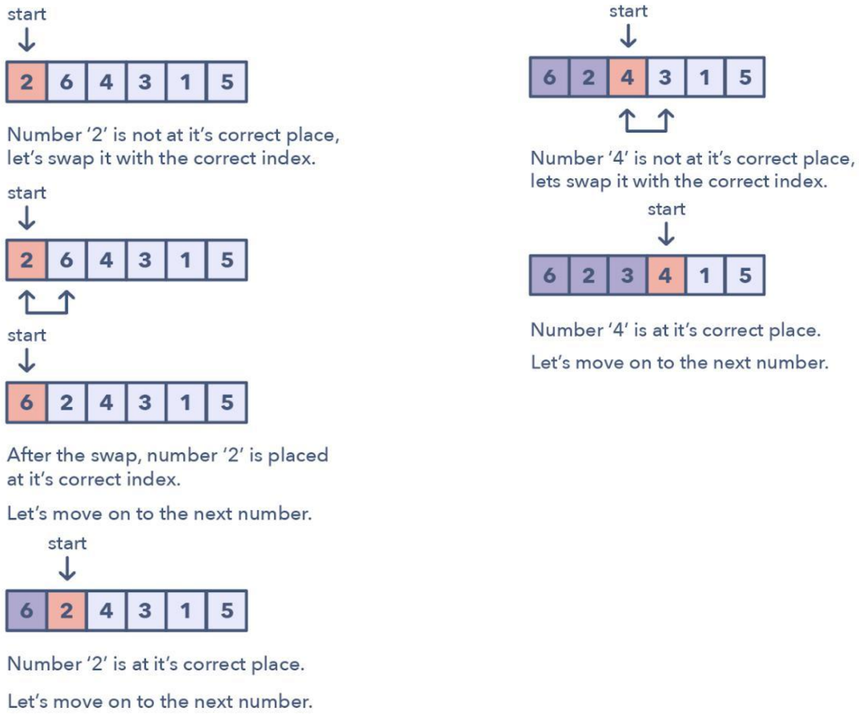

TC: O(N)

SC: O(N/2) -> O(N)

# Advanced Topics

# Cyclic Sort

This pattern describes an interesting approach to deal with problems involving arrays containing numbers in a given range. The Cyclic Sort pattern iterates over the array one number at a time, and if the current number you are iterating is not at the correct index, you swap it with the number at its correct index. You could try placing the number in its correct index, but this will produce a complexity of O(N^2) which is not optimal, hence the Cyclic Sort pattern.



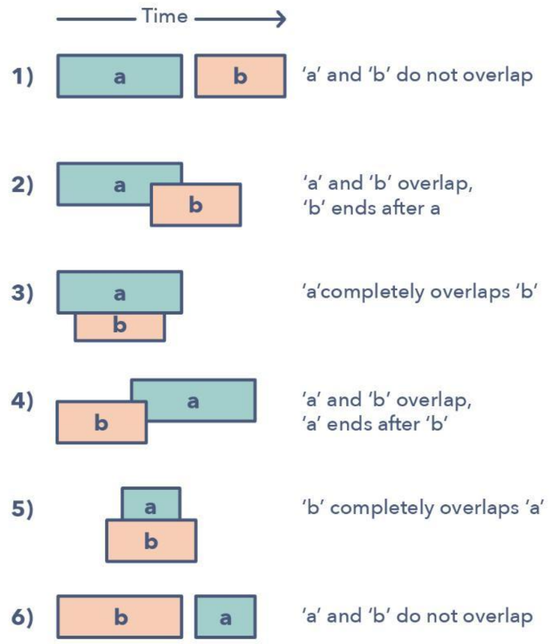

**How do I identify this pattern**?

- They will be problems involving a sorted array with numbers in a given range.
- If the problem asks you to find the missing/duplicate/smallest number in an sorted/rotated array.

# Merge Intervals

The Merge Intervals pattern is an efficient technique to deal with overlapping intervals. In a lot of problems involving intervals, you either need to find overlapping intervals or merge intervals if they overlap. The pattern works like this:

Given two intervals (‘a’ and ‘b’), there will be six different ways the two intervals can relate to each other:




1) ‘a’ and ‘b’ do not overlap
2) ‘a’ and ‘b’ overlap, ‘b’ ends after ‘a’
3) ‘a’ completely overlaps ‘b’
4) ‘a’ and ‘b’ overlap, ‘a’ ends after ‘b’
5) ‘b’ completely overlaps ‘a’
6) ‘a’ and ‘b’ do not overlap; ‘b’ comes before ‘a’

Understanding and recognizing these six cases will help you solve a wide range of problems from inserting intervals to optimizing interval merges.

**How do you identify when to use the Merge Intervals pattern**?

- If you’re asked to produce a list with only mutually exclusive intervals.

- If you hear the term “overlapping intervals”.

# Two Heaps

In many problems, we are given a set of elements such that we can divide them into two parts. To solve the problem, we are interested in knowing the smallest element in one part and the biggest element in the other part. This pattern is an efficient approach to solve such problems.

This pattern uses two heaps; A **Min Heap** to find the smallest element and a **Max Heap** to find the biggest element. The pattern works by storing the *first half of numbers* in a **Max Heap**, this is because you want to find the largest number in the first half. You then store the *second half of numbers* in a **Min Heap**, as you want to find the smallest number in the second half. At any time, the *median* of the current list of numbers can be calculated from the top element of the two heaps.

**Ways to identify the Two Heaps pattern**:

- Useful in situations like Priority Queue, Scheduling.

- If the problem states that you need to find the **smallest**/**largest**/**median** elements of a set.

- Sometimes, useful in problems featuring a binary tree data structure.

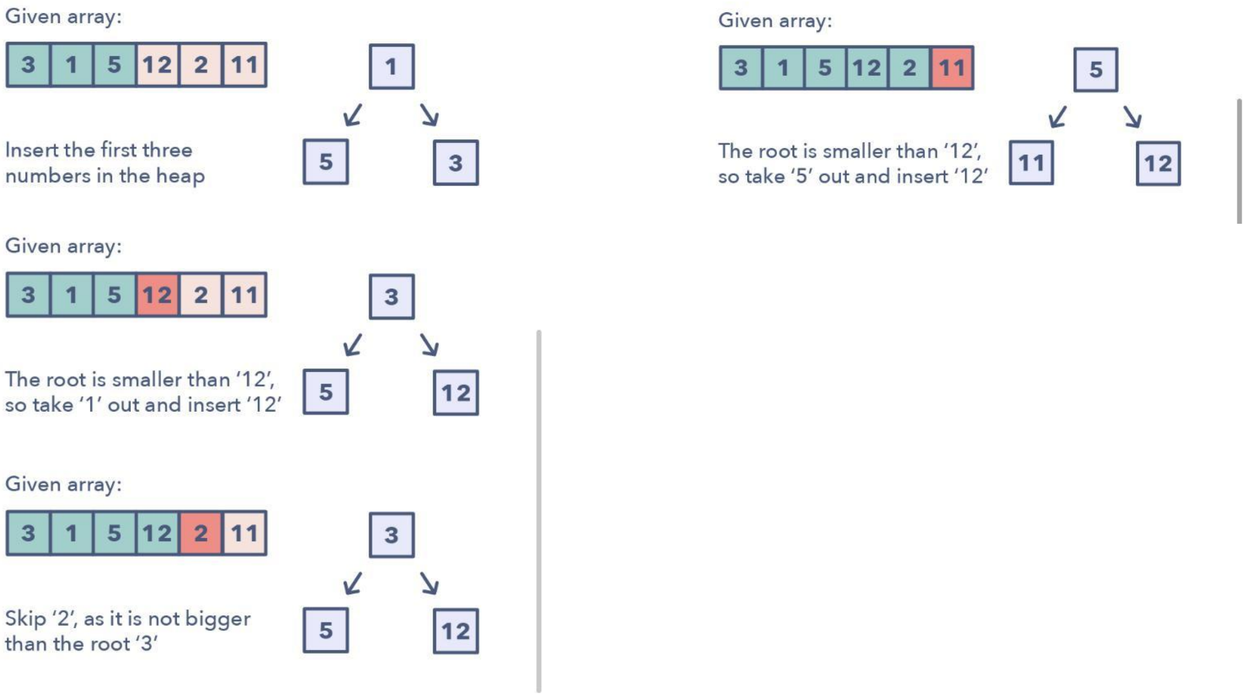
# Top K Elements

Any problem that asks us to find the top/smallest/frequent ‘K’ elements among a given set falls under this pattern.

The best data structure to keep track of ‘K’ elements is Heap. This pattern will make use of the Heap to solve multiple problems dealing with ‘K’ elements at a time from a set of given elements.

**The pattern looks like this**:

1. Insert ‘K’ elements into the *min-heap* or *max-heap* based on the problem.
2. Iterate through the remaining numbers and if you find one that is larger than what you have in the heap, then remove that number and insert the larger one.

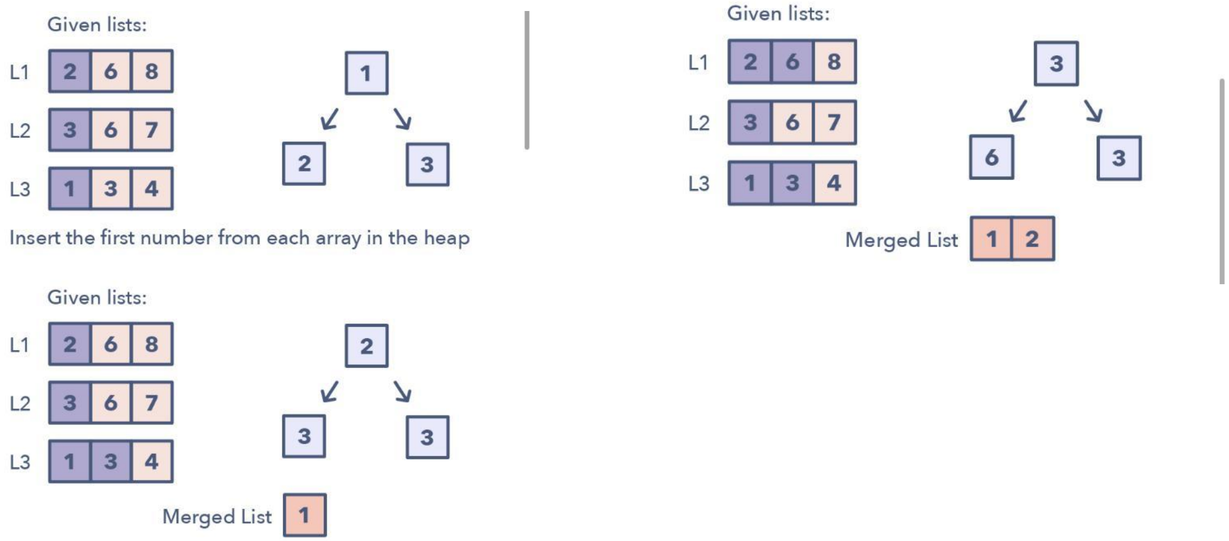



There is no need for a sorting algorithm because the heap will keep track of the elements for you.

# K-Way Merge

**K-Way Merge** helps you solve problems that involve a *set of sorted arrays*.

Whenever you're given 'K' sorted arrays, you can use a **Heap** to efficiently perform a sorted traversal of all the elements of all arrays. You can push the smallest element of each array in a **Min Heap** to get the overall minimum. After getting the overall minimum, push the next element from the same array to the heap. Then, repeat this process to make a sorted traversal of all elements.




**The pattern looks like this**:

1. Insert the first element of each array in a Min Heap.
2. After this, take out the smallest (top) element from the heap and add it to the merged list.
3. After removing the smallest element from the heap, insert the next element of the same list into the heap.
4. Repeat steps 2 and 3 to populate the merged list in sorted order.

**How to identify the K-Way Merge pattern**:

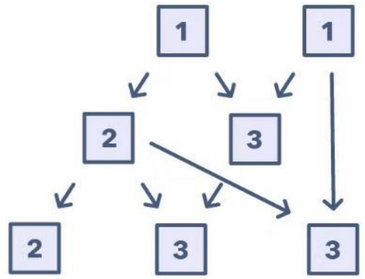
- The problem will feature *sorted arrays*, *lists*, or *a matrix*.
- If the problem asks you to merge sorted lists, find the smallest element in a sorted list.

# Topological Sort

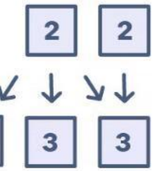
**Topological Sort** is used to find a linear ordering of elements that have dependencies on each other. For example, if event ‘B’ is dependent on event ‘A’, ‘A’ comes before ‘B’ in topological ordering.

This pattern defines an easy way to understand the technique for performing topological sorting of a set of elements.

**The pattern works like this**:




Add all sources to the sorted list.
Remove all sources and their edges to find new sources




Sources: [3,4]
Topological Sort: “5,6”

Add all sources to the sorted list.
Remove all sources and their edges to find new sources


Sources: [0, 1, 2]
Topological Sort: “5, 6, 3, 4”

All remaining vertices are source,
so we will add them in the sorted list

Sources: []
Topological Sort: “5, 6, 3, 4, 0 ,1, 2”

# Last Thought!

Roman **Gen. Julius Caesar** had the right mindset in 55 B.C.E.: **Take the island by burning the boats**!

**Thank you! My contact information**:


- yli12313@umd.edu
- 301-204-3957

# Slides That Will Not be Presented

# Subsets

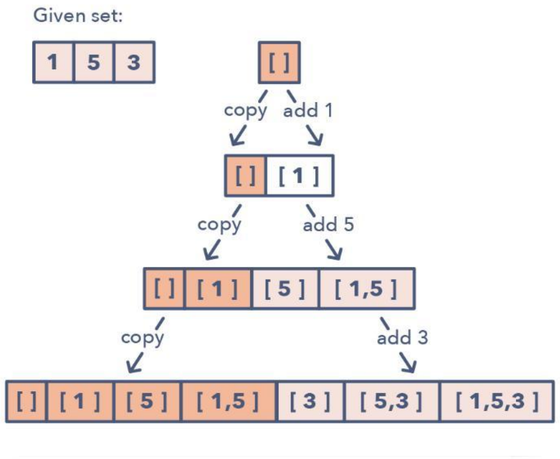
A huge number of coding interview problems involve dealing with ***Permutations*** and ***Combinations*** of a given set of elements. The pattern **Subsets** describes an efficient **Breadth First Search** (**BFS**) approach to handle all these problems.

**The pattern looks like this**:

Given a set of [1,5,3]

1. Start with an empty set: [[]]
2. Add the first number (1) to all the existing subsets to create new subsets: [[], [1]];
3. Add the second number (5) to all the existing subsets: [[], [1], [5], [1,5]];
4. Add the third number (3) to all the existing subsets: [[], [1], [5], [1,5], [3], [1,3], [5,3], [1,5,3]];

**Here is a visual representation of the Subsets pattern**:




**How to identify the Subsets pattern**:

- Problems where you need to find the *Combinations* or *Permutations* of a given set.