LeetCode

1.twoSum

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

You may assume that each input would have *exactly* one solution, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].

Example 2:

Input: nums = [3,2,4], target = 6

Output: [1,2] Example 3:

General Solutions

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        int len = nums.length;
        int indices[] = new int[2];;
        for (int i = 0; i < len; i++) {
            for (int j = i+1; j < len; j++) { //// Fix: Start j from i+1
                if (nums[i] + nums[j] == target) {
                    indices[0] = i;
                    indices[1] = j;
                     return indices;

                }
```

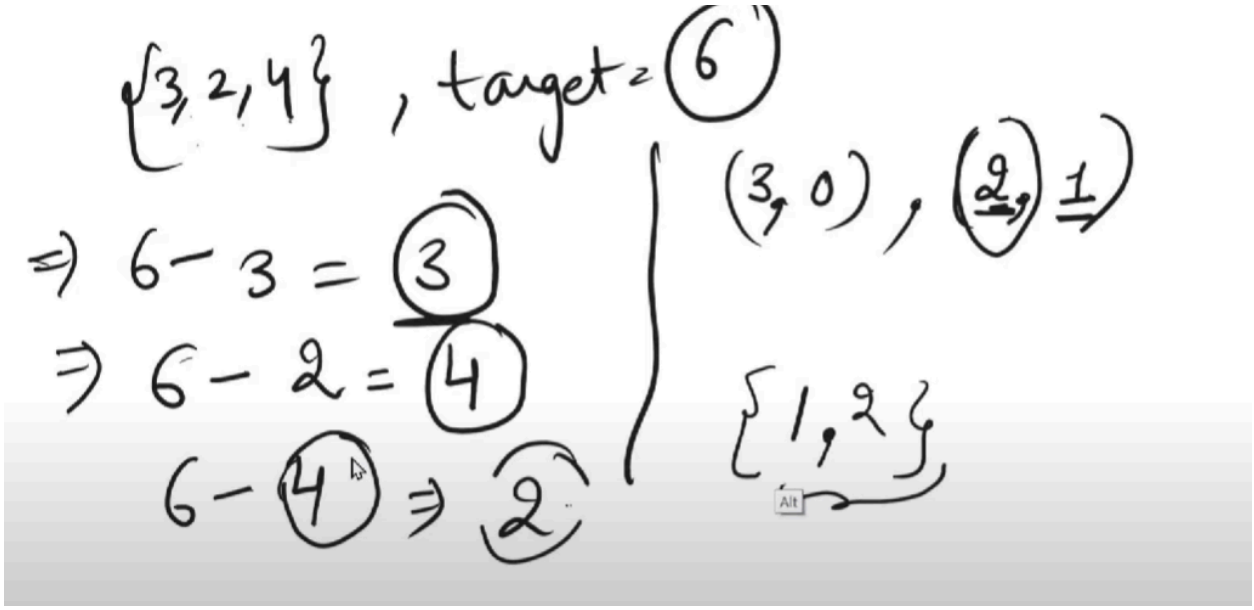
```
            }
        }
        return new int[0];//// Return an empty array if no solution is found (though guaranteed to have one)
    }
}
```

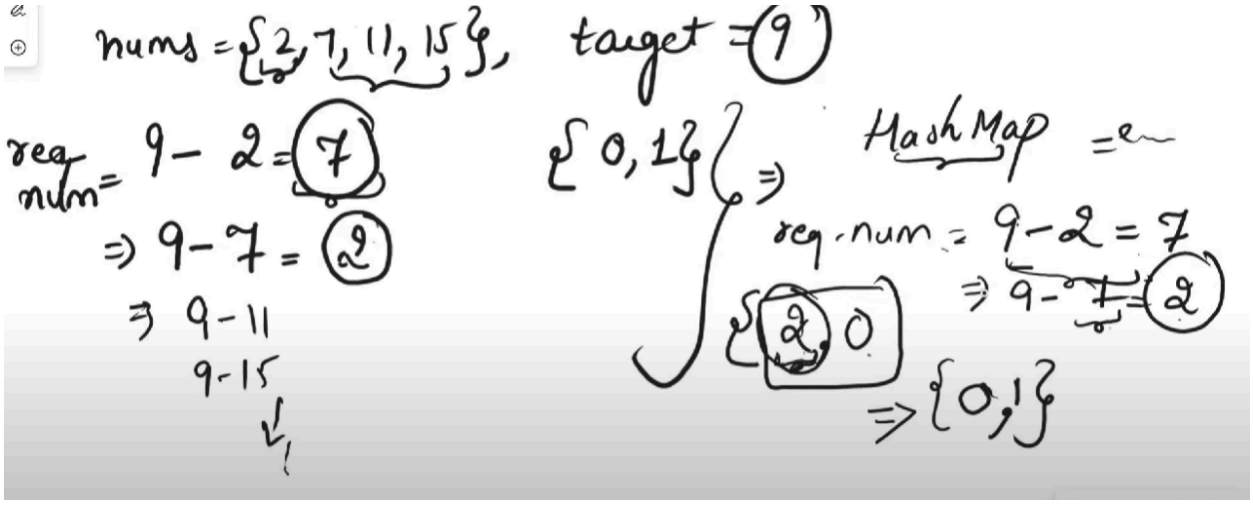
Or

```
package com.leetcode150;
import java.util.Arrays;
//Online Java Compiler
//Use this editor to write, compile and run your Java code online
public class TwoSumExample {
	public static void main(String[] args) {
		int nums[] = { 1, 2, 3, 4, 5, 6, 7 };
		int target = 5;
		int[] result = twoSum(nums, target);
		System.out.println(Arrays.toString(result));
	}
	public static int[] twoSum(int[] arr, int target) {
		int length = arr.length;
		for (int i = 0; i < length; i++) {
			for (int j = i + 1; j < length; j++) {
				if ((target == arr[i] + arr[j])) {
					return new int[] { i, j };
				}
			}
		}
		return new int[] {};
	}
}
```

[0, 3]

Optimized Solution





Concept of Two Sum:

We need to find two numbers in an array that add up to a given target:

target=num[i]+num[j]

Ex: 9 = 2+7;

Instead of iterating through the array using two nested loops ($O(n^2)$), we use a HashMap for O(1) lookups.
target- num[i] = nums[j]

Will search nums[j] in hashmap if it is there it will return nums[j] index and num[i] index (current index).

Initialize a HashMap:

- The HashMap (map) will store numbers as keys and their indices as values.

Iterate through the array:

- For each element nums[i], compute target - nums[i]= complement.
- Check if complement exists in the HashMap:
  If Yes → Return its index and i (current index).

  If No → Store nums[i] and its index in the HashMap and continue the next element.

```
package com.leetcode150;
import java.util.Arrays;
import java.util.HashMap;
//Online Java Compiler
//Use this editor to write, compile and run your Java code online
public class TwoSumExample {
	public static void main(String[] args) {
		int nums[] = { 1, 2, 3, 4, 5, 6, 7 };
		int target = 5;
		int[] result = twoSum(nums, target);
		System.out.println(Arrays.toString(result));
	}
	public static int[] twoSum(int[] nums, int target) {
		HashMap<Integer, Integer> hm = new HashMap<Integer, Integer>();
		for (int i = 0; i < nums.length; i++) {
			int req_num = target - nums[i];
			if (hm.containsKey(req_num)) {
				int arr[] = { hm.get(req_num), i };
				return arr;
			} else {
				hm.put(nums[i], i);
			}
		}
		return null;
	}
```

}

Output

[1, 2]

Why O(n) Time?

- We iterate through the array only once (single loop).
- Each lookup in the HashMap (containsKey()) and insertion (put()) is O(1) on average.
- So, the overall time complexity is O(n).

Why O(n) Space?

- We use a HashMap to store seen numbers and their indices.
- In the worst case, we may store all n elements, making the space complexity O(n).

2. Find the Duplicate Number

Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only **one repeated number** in nums, return *this repeated number*.

You must solve the problem **without** modifying the array nums and using only constant extra space.

Example 1:

Input: nums = [1,3,4,2,2]
Output: 2

Example 2:

Input: nums = [3,1,3,4,2]
Output: 3

Example 3:

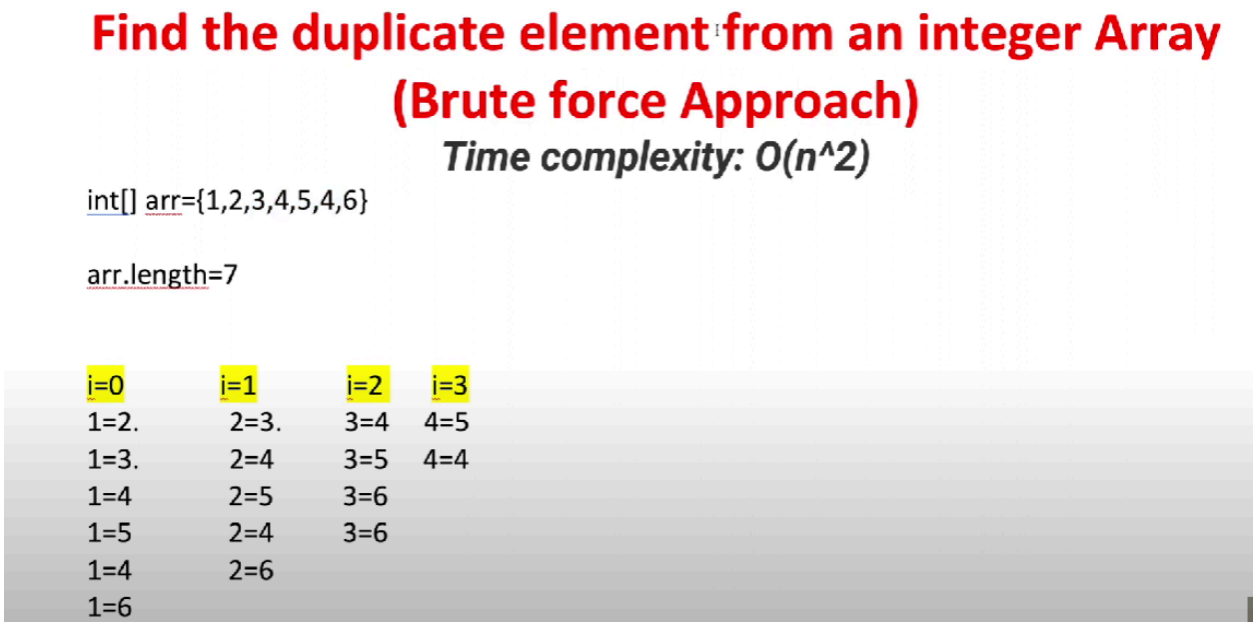
Input: nums = [3,3,3,3,3]
Output: 3

General Solutions

1) **brute-force** approach

When we say your approach is **brute-force**, we mean this:

You're checking **each element against every other element** that comes after it in the array.



- **Time Complexity**: O(n^2) — this nested loop approach is inefficient for large arrays.
- **Space Complexity**: O(1) — good, as it doesn't use extra space.

```
class Solution {
    public int findDuplicate(int[] nums) {
        for (int i = 0; i < nums.length - 1; i++) {
            for (int j = i + 1; j < nums.length; j++) {
```

```
            if (nums[i] == nums[j]) {
                return nums[i];
            }
        }
    }
    return -1;
  }
}
```

2) Sorting-Based Approach

1. **Step 1: Sort the Array**
    - The given array [1, 7, 3, 4, 3, 3, 5] is sorted into [1, 3, 3, 3, 4, 5, 7].
2. **Step 2: Check Adjacent Elements**
    - Since duplicates will be next to each other after sorting, iterate through the array and compare adjacent elements.
    - If arr[i] == arr[i+1], you've found the duplicate.

```
package com.leetcode150;
import java.util.Arrays;
public class FindDuplicateNumber {
	public static void main(String[] args) {
		int nums[] = { 1, 3, 4, 2, 2 };
		System.out.println(findDuplicate(nums));
	}
	public static int findDuplicate(int[] nums) {
		Arrays.sort(nums); // Step 1: Sort the array
		for (int i = 0; i < nums.length - 1; i++) {
			if (nums[i] == nums[i + 1]) { // Step 2: Check adjacent elements
				return nums[i];
			}
		}
		return -1; // Shouldn't reach here if a duplicate exists
	}
```

}

Output

2

Sorting takes O(n log n), and **the loop takes** O(n).

Total Complexity → O(n log n).

Space Complexity → O(1) (modifies input array).

3) Using HashSet to Find Duplicates (O(n) Time Complexity)

Instead of sorting or using nested loops, we can use a **HashSet** to store elements while iterating through the array. If we encounter a number that is already in the set, it's the duplicate.

1. Create an empty **HashSet**.
2. Iterate through the array:
    - If the number is already in the set, return it (it's the duplicate).
    - Otherwise, add the number to the set.
3. If no duplicate is found (which shouldn't happen based on the problem constraints), return -1.

```
package com.leetcode150;
import java.util.HashSet;
public class FindDuplicateNumber {
	public static void main(String[] args) {
		int nums[] = { 1, 3, 4, 2, 2 };
		System.out.println(findDuplicate(nums));
	}
	public static int findDuplicate(int[] nums) {
		HashSet<Integer> seen = new HashSet<>();
		for (int num : nums) {
```

```
                if (seen.contains(num)) { // If num is already in the set, it's a duplicate
                    return num;
                }
                seen.add(num);
            }
            return -1; // Should not reach here if a duplicate exists
        }
}
```

Output

2

Time Complexity: O(n), as we iterate through the array once.

Space Complexity: O(n), because we store elements in a HashSet.

Much faster than brute-force ($O(n^2)$) and sorting ($O(n \log n)$).
Requires **extra space** O(n), which may not be allowed in some problems.

Optimized Solution

Efficient solution using **Floyd's Cycle Detection Algorithm (Tortoise and Hare method)** to find the duplicate in an array.

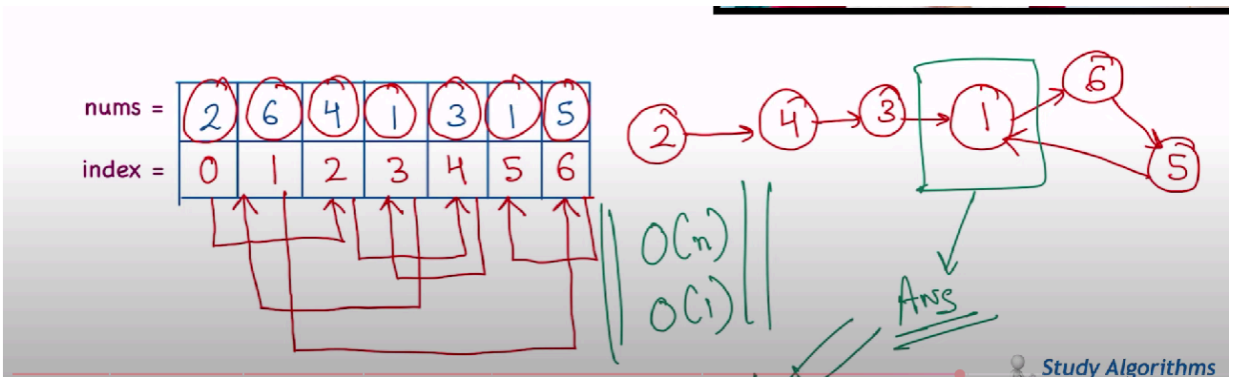
Floyd's Cycle Detection Algorithm, also known as the **Tortoise and Hare Algorithm**, is used to detect **cycles in a sequence** (typically in linked lists or arrays). It does so **without using extra space (O(1) memory)** and runs in **O(n) time**.

It uses **two pointers** that traverse the sequence at **different speeds**:

- **Tortoise (slow pointer)** → Moves **one step at a time**.
- **Hare (fast pointer)** → Moves **two steps at a time**.

How Does It Work?

1. **Phase 1: Detect Cycle**
    - Move both pointers at their respective speeds.
    - If a cycle exists, **they will eventually meet** inside the cycle.
2. **Phase 2: Find Cycle Start**
    - Reset **one pointer to the start** of the sequence.
    - Move both pointers **one step at a time**.
    - The point where they meet again is the **start of the cycle**.



This method treats the array like a **linked list** where:

- The **index** is like a **node**.
- The **value** at each index is the **next pointer**.

Since there's a duplicate, a cycle (loop) will exist in this linked list. The goal is to detect the **starting point of the cycle**, which is the duplicate number.

nums = [2, 6, 4, 1, 3, 1, 5]

Index Mapping (As a Linked List)

Index	Value (Next Pointer)
0	2
1	6
2	4
3	1
4	3
5	1 (Cycle Here)
6	5

Cycle Formation

lua Copy Edit

```
2 → 4 → 3 → 1 → 6 → 5
            ↑   ↓
            ←---
```

- **Duplicate:** `1` is causing the cycle.

Steps (Tortoise and Hare Algorithm)

1. **Slow and Fast Pointers**:
    - Use two pointers: slow moves **one step** at a time, fast moves **two steps**.
    - They will eventually meet inside the cycle.
2. **Find the Start of the Cycle (Duplicate Number)**:
    - Reset slow to the start (nums[0]).
    - Move both slow and fast **one step at a time** until they meet again.
    - The meeting point is the **duplicate number**.

Finding the Duplicate Number using Slow and Fast Pointers (Floyd's Cycle Detection Algorithm)

nums = [2, 6, 4, 1, 3, 1, 5]

Each index acts as a node, and its value points to the next index.

Index Mapping:

Index	0	1	2	3	4	5	6
Value	2	6	4	1	3	1	5

Each number points to the next **index** in the list.
For example, `nums[0] = 2` means index **0** points to index **2**.

We treat the array as a **linked list**, where:

- The **index** acts as a **node**.
- The **value** at each index points to the **next node**.

Step 1: Define Movement Rules

- slow moves **one step** at a time: slow = nums[slow]
- fast moves **two steps** at a time: fast = nums[nums[fast]]

Step 2: Detecting the Cycle (Finding Meeting Point)

Initial Pointers:

- slow = nums[0] = 2
- fast = nums[nums[0]] = nums[2] = 4

Pointer Movements:

Step	Slow Pointer (**slow = nums[slow]**)	Fast Pointer (**fast = nums[nums[fast]]**)
1	nums[2] = 4	nums[nums[4]] = nums[3] = 1

2	nums[4] = 3	nums[nums[1]] = nums[6] = 5
3	nums[3] = 1	nums[nums[6]] = nums[5] = 1
4	nums[1] = 6	nums[nums[5]] = nums[1] = 6

Meeting Point: slow and fast meet at **index 6 (value = 6)**.

Step 3: Finding the Start of the Cycle (Duplicate Number)

- Reset slow to nums[0] = 2
- Keep fast where it is (nums[6] = 6)
- Move both **one step at a time** until they meet again.

Pointer Movements (Cycle Entry)

Step	Slow (**slow = nums[slow]**)	Fast (**fast = nums[fast]**)
1	nums[2] = 4	nums[6] = 5
2	nums[4] = 3	nums[5] = 1
3	nums[3] = 1	nums[1] = 6
4	nums[1] = 6	nums[6] = 5
5	nums[5] = 1	nums[5] = 1

Meeting Point (Start of Cycle) = 1 (Duplicate Number) ✅

```
package com.leetcode150;
import java.util.HashSet;
public class FindDuplicateNumber {
	public static void main(String[] args) {
		int nums[] = { 1, 3, 4, 2, 2 };
		System.out.println(findDuplicate(nums));
	}
	public static int findDuplicate(int[] nums) {
		int slow = nums[0];
		int fast = nums[0];
```

```
                // Step 1: Detect the cycle
                while (true) {
                        slow = nums[slow]; // Moves one step
                        fast = nums[nums[fast]]; // Moves two steps
                        if (slow == fast) { // Cycle detected
                                break;
                        }
                }
                // Step 2: Find the entry point of the cycle (duplicate number)
                slow = nums[0]; // Reset slow to start
                while (slow != fast) {
                        slow = nums[slow]; // Move one step
                        fast = nums[fast]; // Move one step
                }
                return slow; // Duplicate number found
        }
}
```

Output
2

Time Complexity: O(n) (Both loops iterate at most n times)
Space Complexity: O(1) (Uses only two pointers, no extra memory)

3.Maximum Subarray

Given an integer array nums, find the subarray with the largest sum, and return *its sum*.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.

Example 2:

Input: nums = [1]

Output: 1
Explanation: The subarray [1] has the largest sum 1.

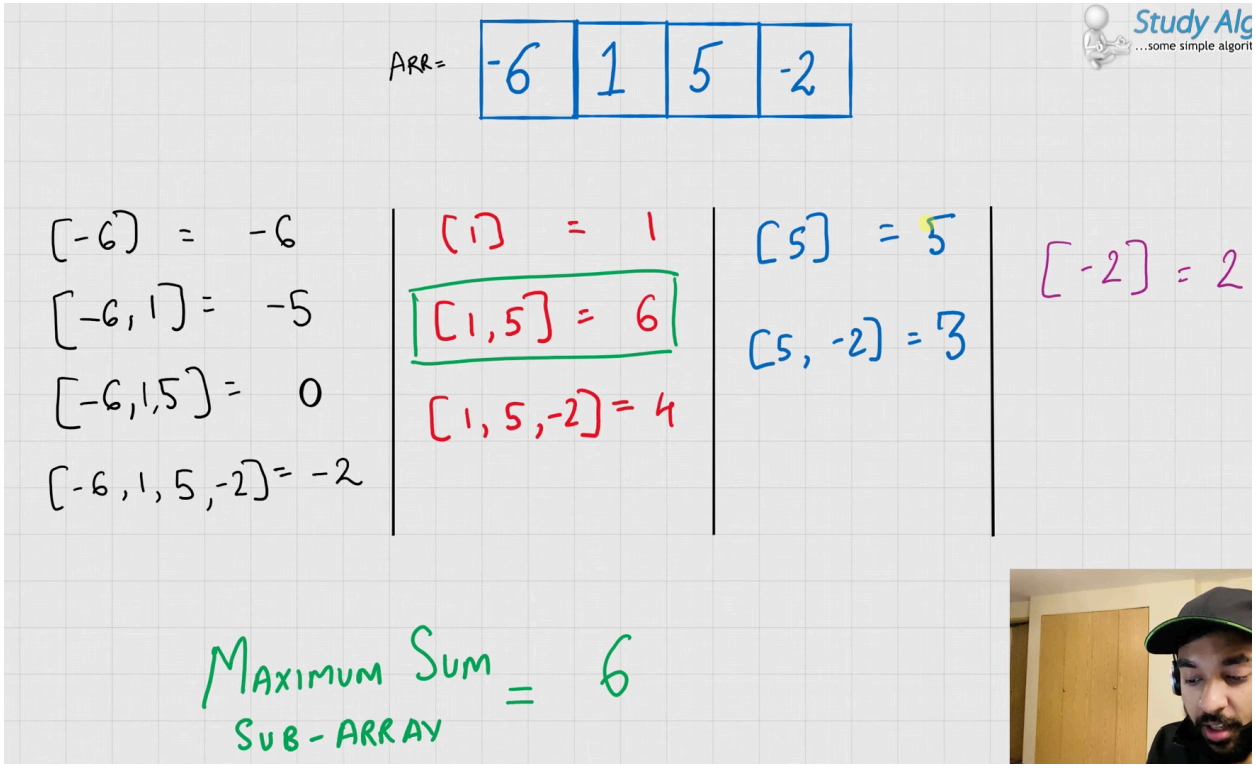
Example 3:

Input: nums = [5,4,-1,7,8]
Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

General Solutions

brute force approach (O(n²)) for **Maximum Subarray Sum** in **Java**:

Possible sub arrays for the below array




```
package com.leetcode150;
public class MaximumSubarray {
	public static void main(String[] args) {
		int[] nums = { -6, 1, 5, -2 };
		System.out.println("Maximum Subarray Sum: " + maxSubarraySum(nums));
	}
```

```
    public static int maxSubarraySum(int[] nums) {
        int maxSum = Integer.MIN_VALUE;; // Initialize with smallest integer value
        int n = nums.length;
        for (int i = 0; i < n; i++) { // Start index of subarray
            int currentSum = 0;
            for (int j = i; j < n; j++) { // End index of subarray
                currentSum += nums[j]; // Calculate sum
                maxSum = Math.max(maxSum, currentSum); // Update max sum
            }
        }
        return maxSum;
    }
}
```

Output
6

O(n²) because we iterate through all possible subarrays (nested loops).

Space Complexity: **O(1)** (no extra storage used).

Optimized Solution

Kadane's Algorithm is a **dynamic programming approach** used to find the **maximum sum of a contiguous subarray** in an array of numbers. It solves the **Maximum Subarray Problem** in **O(n) time complexity**.

Concept Behind Kadane's Algorithm

The algorithm works by iterating through an array **only once** and maintaining:

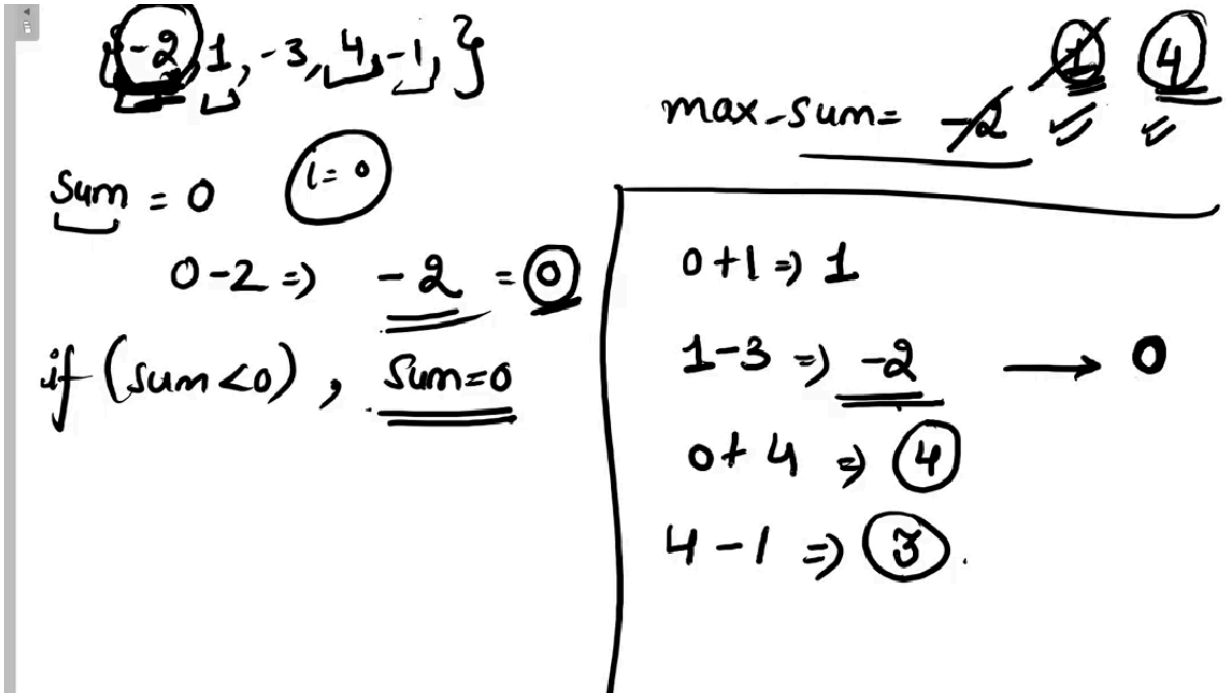
1. **A running sum (currentSum)** – Tracks the sum of the current subarray.
2. **A maximum sum (maxSum)** – Keeps track of the highest sum encountered so far.

Kadane's Algorithm in Simple Terms

Imagine moving through an array:

1. Start from the first number.

2. Keep adding numbers as long as the sum is positive.
3. If the sum turns negative, reset the sum (start fresh from the next number).
4. Keep track of the highest sum found during the process.




```
package com.leetcode150;
public class MaximumSubarray {
	public static void main(String[] args) {
		int[] nums = { -6, 1, 5, -2 };
		System.out.println("Maximum Subarray Sum: " + maxSubarraySum(nums));
	}
	public static int maxSubarraySum(int[] nums) {
		int sum = 0;
		int max_sum = nums[0];// as first element in array
		for (int i = 0; i < nums.length; i++) {
			sum = sum + nums[i];
			if (sum > max_sum) {
				max_sum = sum;
			}
			if (sum < 0) {
```

```
                    sum = 0;
                }
            }
            return max_sum;
        }
}
```

Output

Maximum Subarray Sum: 6

Time Complexity: O(n) Iterates through the array only once.

Space Complexity: O(1) Uses only two extra variables.

4. Remove Duplicates from Sorted Array

Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return *the number of unique elements in* nums. And it return length of array.

Example 1:

Input: nums = [1,1,2]
Output: 2, nums = [1,2,_]
Explanation: Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.
It does not matter what you leave beyond the returned k (hence they are underscores).

Example 2:

Input: nums = [0,0,1,1,1,2,2,3,3,4]
Output: 5, nums = [0,1,2,3,4,_,_,_,_,_]
Explanation: Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively.
It does not matter what you leave beyond the returned k (hence they are underscores).

{1, 1, 2, 2, 2, 3, 3, 4, 5, 5, 6} ⟶ sorted

(i) if current ele = next ele
continue

(ii) if (cur ≠ next)
Save ele.

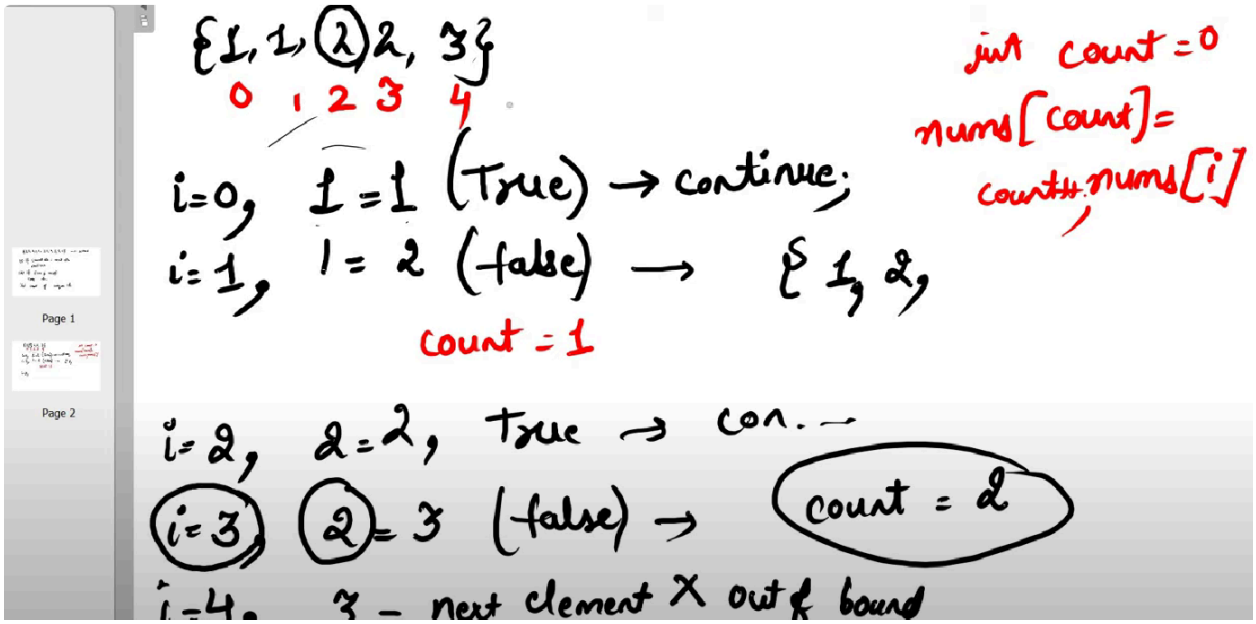
(iii) count of unique ele.

{1, 1, 2, 2, 3}
0 1 2 3 4

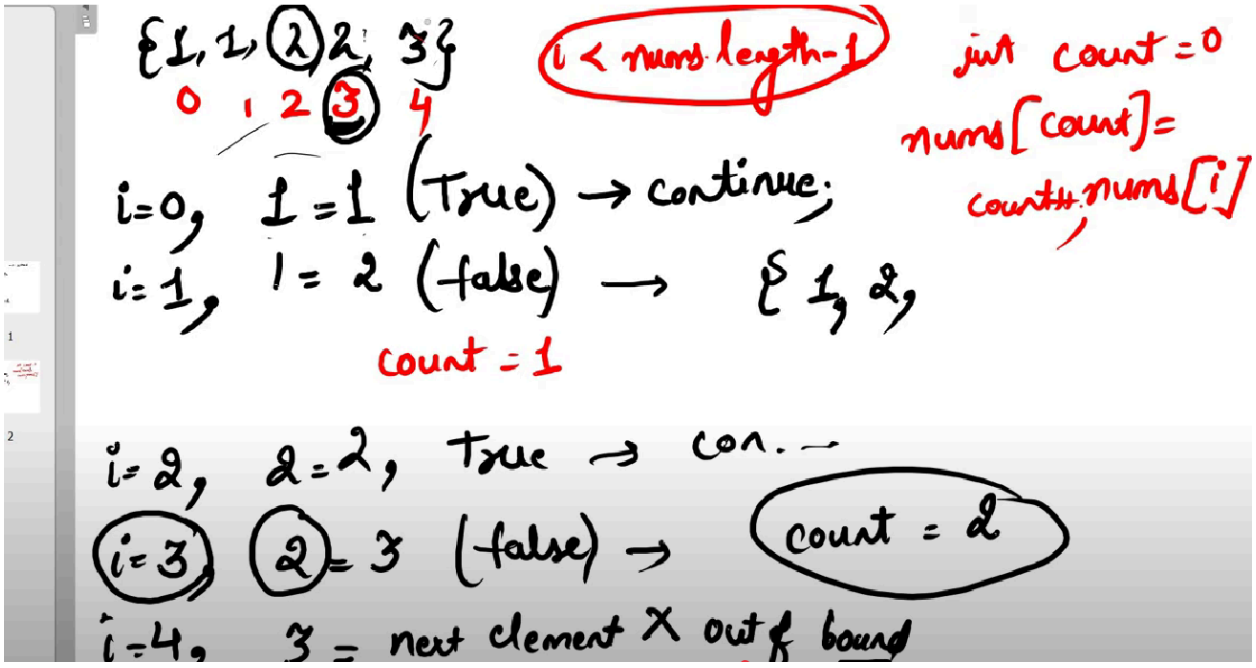
just count = 0
nums[count] = count++, nums[i]

i=0, 1 = 1 (True) → continue;
i=1, 1 = 2 (false) → {1,
count = 1

i=2, 2=2, true → con.—
i=3, 2=3 (fo



If we run the loop to the end it will get out of bound exceptions.
So we will run a loop till the second largest element.



Optimized Solution

```
package com.leetcode150;
```

```java
public class RemoveDuplicatesfromSortedArray {
	public static void main(String[] args) {
		int nums[] = { 0, 0, 1, 1, 1, 2, 2, 3, 3, 4 };// 1,1,2
		int k = removeDuplicates(nums);
		System.out.println(k);
	}
	public static int removeDuplicates(int[] nums) {
		int count = 0;
		for (int i = 0; i < nums.length; i++) {
			if (i < nums.length - 1 && nums[i] == nums[i + 1]) {
				continue;
			}
			nums[count] = nums[i];
			count++;
		}
		return count;
	}
}
```

Output

5

Why O(n) Time?

- We are using a single loop that iterates over the array once to check and remove duplicates.
- So, time complexity is O(n).

Why O(1) Space?

- We are not using any extra array.
- We are modifying the same input array in-place using a count pointer to place unique elements.
- So, space complexity is O(1).

5.Remove Duplicates from Sorted Array II

Given an integer array nums sorted in non-decreasing order, remove some duplicates in-place such that each unique element appears at most twice(one or twice not more than that). The relative order of the elements should be kept the same.

Example 1:

Input: nums = [1,1,1,2,2,3]
Output: 5, nums = [1,1,2,2,3,_]
Explanation: Your function should return k = 5, with the first five elements of nums being 1, 1, 2, 2 and 3 respectively.
It does not matter what you leave beyond the returned k (hence they are underscores).

Example 2:

Input: nums = [0,0,1,1,1,1,2,3,3]
Output: 7, nums = [0,0,1,1,2,3,3,_,_]
Explanation: Your function should return k = 7, with the first seven elements of nums being 0, 0, 1, 1, 2, 3 and 3 respectively.
It does not matter what you leave beyond the returned k (hence they are underscores).

Do not allocate extra space for another array. You must do this by modifying the input array in-place with O(1) extra memory.

Optimized Solution

- Since the array is sorted, duplicates appear consecutively.
- If an element appears more than twice, its third occurrence (or more) can be identified by checking the value at i and i-2.

  Implementation Steps

  - Start with count = 0 to track valid elements.
  - Iterate through the array (starting from i = 0).
  - For each nums[i]:

    If count < 2 → Always include the first two elements.

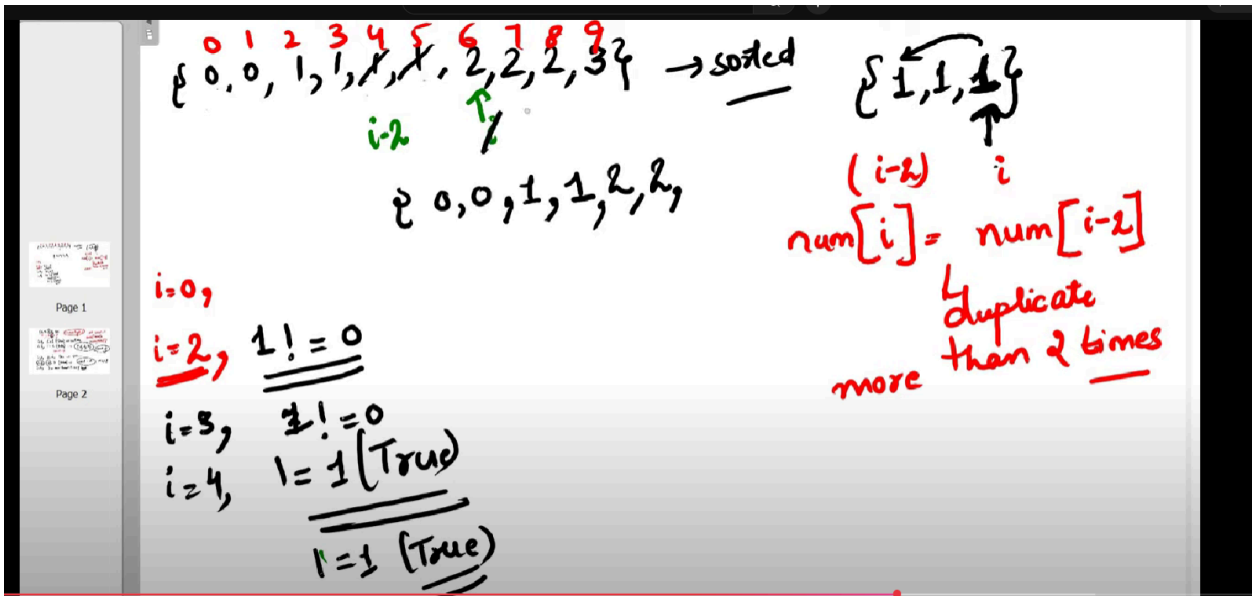
If nums[i] != nums[count - 2], then store nums[i] at nums[count] and increment count.

Otherwise, ignore nums[i] (more than twice occurrence).

- Finally, return count, which represents the new array length.

Time and Space Complexity

- Time Complexity = O(n) → Single pass through the array.
- Space Complexity = O(1) → In-place modification, no extra space.




```
package com.leetcode150;
public class RemoveDuplicatesfromSortedArray2 {
	public static void main(String[] args) {
		int nums[] = { 0, 0, 1, 1, 1, 3, 4 };// 1,1,2
		int k = removeDuplicates(nums);
		System.out.println(k);
	}
	public static int removeDuplicates(int[] nums) {
		int i = 0;
		for (int n : nums) {
```

```
                if (i < 2 || n != nums[i - 2]) {
                    nums[i] = n;
                    i++;
                }
            }
            return i;
        }
}
```

Output

6

6. Valid Anagram

Two strings are said to be anagram if we can form one string by arranging the characters of another string. For example, Race and Care. Here, we can form Race by arranging the characters of Care.

Example 1:

Input: s = "anagram", t = "nagaram"

Output: true

Example 2:

Input: s = "rat", t = "car"

Output: false

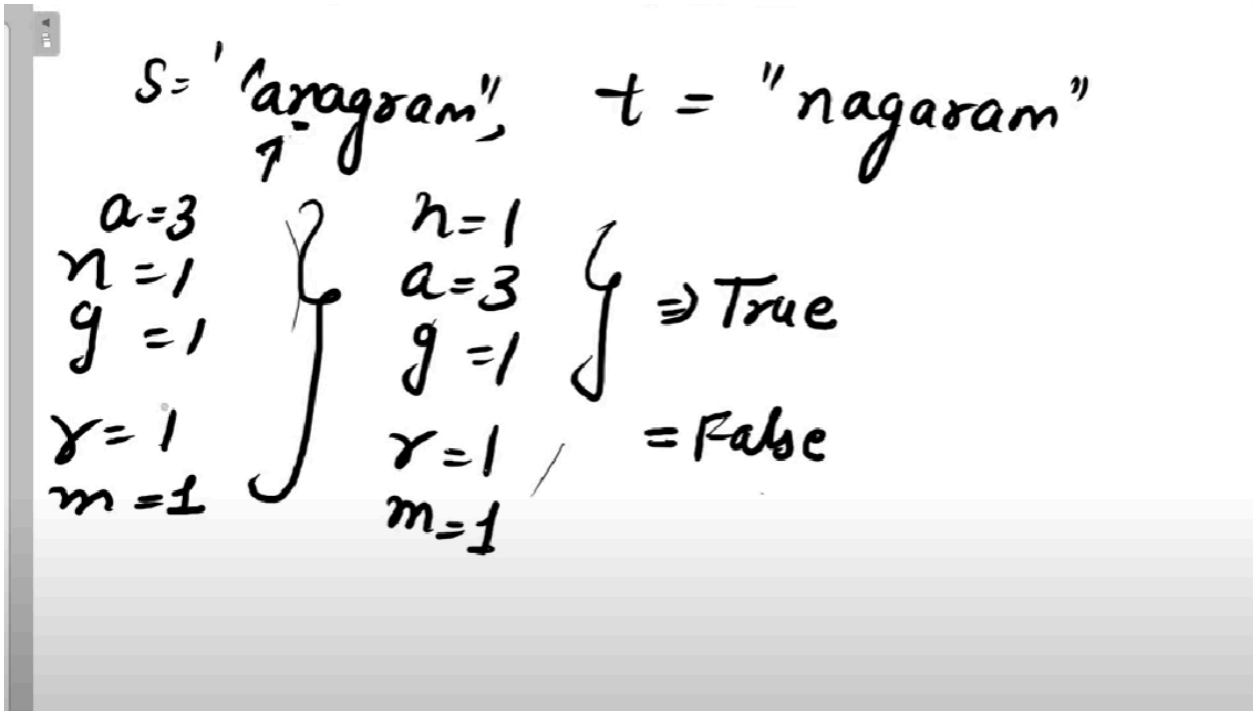
General Solutions

```
package com.leetcode150;
import java.util.Arrays;
public class Anagram {
    public static void main(String[] args) {
        boolean result = isAnagram("anagram", "nagaram");// ("rat", "car");
```

```
            if (result) {
                System.out.println("Both the Strings are anagrams");
            }
        }
        public static boolean isAnagram(String s, String t) {
            char[] arr1 = s.toCharArray();
            char[] arr2 = t.toCharArray();
            Arrays.sort(arr1);
            Arrays.sort(arr2);
            return Arrays.equals(arr1, arr2);
        }
}
```

Other way

By counting characters



As we know we have 26 alphabets so we will store in one array

S = "anagram", t = "nagaram"

⇒ 26

= {0, 1, 2 - - - - - 25}

S = "anagram", t = "nagaram"

⇒ 26

= {0, 1, 2 - (g) (12) (13) - - - 25}

1 1

(14) ⇒ 2

⇒ (3)

'n' - 'a'

⇒ 110 - 97 ⇒ 13

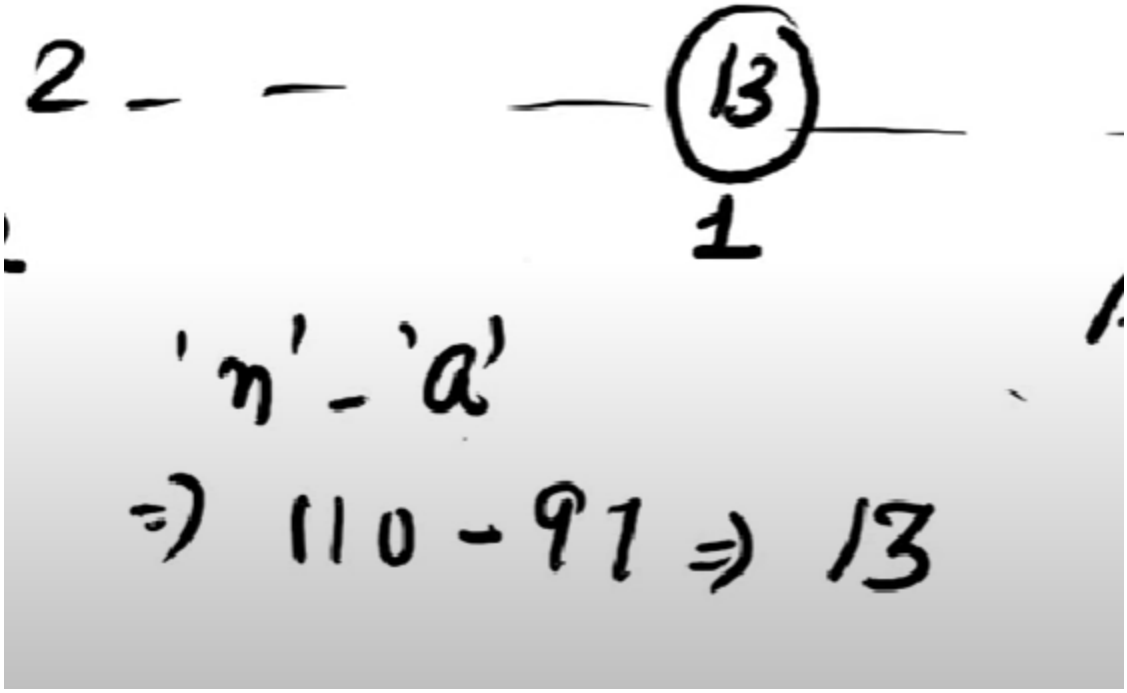
1 - 1 = 0

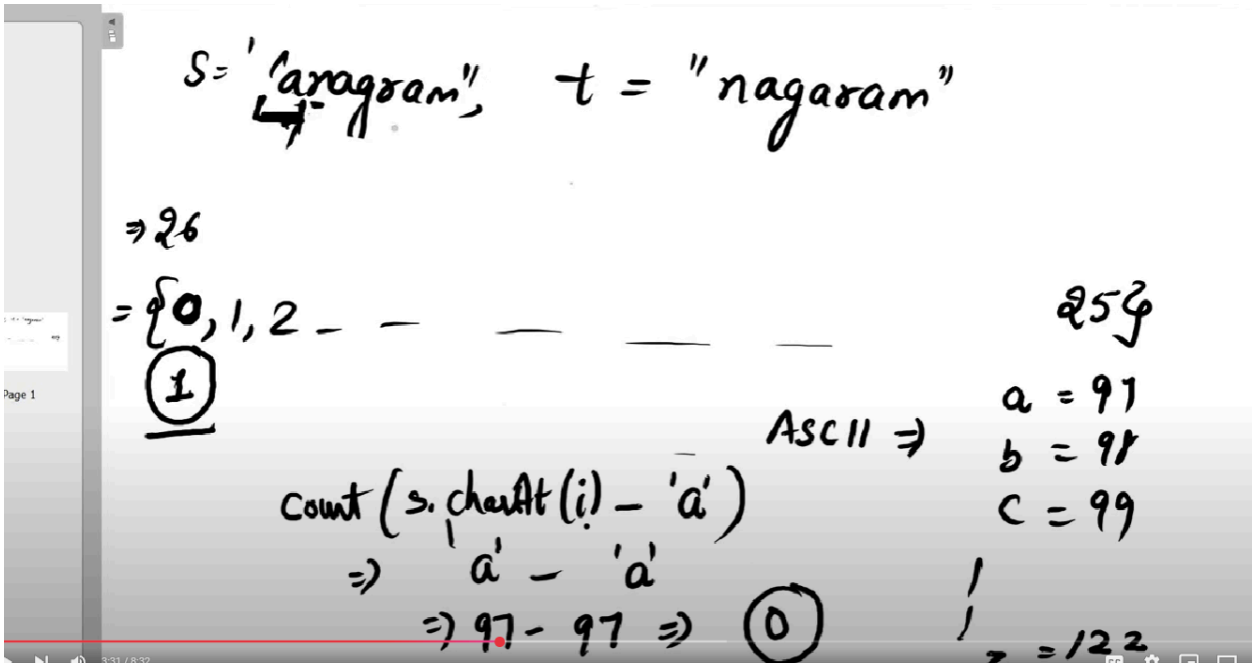
ASCII ⇒

a = 97

b = 98

c = 99





Anagram Check Using ASCII Values (Character Counting)

We can determine if two strings are anagrams using a character frequency array.

- Steps to Check for an Anagram

1. Use an integer array (count[26]) to store the frequency of each letter.
2. Ex ascii values of characters a = 91 to …….z = 122,

3. Process the first string:
    - Convert each character to its index: index = s.charAt(i) - 'a'
    - Increment count[index] for each character in s
4. Process the second string:
    - Convert each character to its index.
    - Decrement count[index] for each character in t
5. Final check:
    - If all values in count[] are 0, the strings are anagrams.

🔹 **Example**

Input:

`s = "listen"`, `t = "silent"`

Processing:

Character	ASCII Value	Index (`char - 'a'`)	`count[]` after "listen"	`count[]` after "silent"
`'l'`	108	`11`	`count[11]++` → `1`	`count[11]--` → `0`
`'i'`	105	`8`	`count[8]++` → `1`	`count[8]--` → `0`
`'s'`	115	`18`	`count[18]++` → `1`	`count[18]--` → `0`
`'t'`	116	`19`	`count[19]++` → `1`	`count[19]--` → `0`
`'e'`	101	`4`	`count[4]++` → `1`	`count[4]--` → `0`
`'n'`	110	`13`	`count[13]++` → `1`	`count[13]--` → `0`

After processing both strings, **all counts are zero** → ✅ **They are anagrams.**

Optimized Solution

```
public class Anagram {
        public static void main(String[] args) {
                boolean result = isAnagram("anagram", "nagaram");
                if (result) {
```

```
                System.out.println("Both the Strings are anagrams");
        }
    }
    public static boolean isAnagram(String s, String t) {
        int length1 = s.length();
        int length2 = t.length();
        if (length1 != length2) {
            return false;
        } else {
            int count[] = new int[26];
            // string 1
            for (int i = 0; i < length1; i++) {
                count[s.charAt(i) - 'a']++;
            }
            // string 2
            for (int i = 0; i < length1; i++) {
                count[t.charAt(i) - 'a']--;
            }
            // checking if any value of count is =0
            for (int i = 0; i < count.length; i++) {
                if (count[i] != 0) {
                    return false;
                }
            }
            return true;
        }
    }
}
```

7.Longest Substring Without Repeating Characters

Given a string s, find the length of the **longest** **substring** without duplicate characters.

Example 1:

Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.

Example 2:

Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.

Example 3:

Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a

General Solutions

the **brute force approach (O(n²))** in Java for finding the length of the longest substring without repeating characters.

Approach:

- Use **two nested loops** to generate all possible substrings.
- Use a **Set** to check if the substring has unique characters.
- Keep track of the **maximum length** encountered.

```
import java.util.HashSet;
public class LongestSubstringBruteForce {
  public static void main(String[] args) {
    String s = "pwwkew";
    System.out.println("Length of Longest Substring Without Repeating Characters: " +
longestSubstring(s));
  }
  public static int longestSubstring(String s) {
    int maxLength = 0;
    int n = s.length();
    for (int i = 0; i < n; i++) { // Start of substring
      HashSet<Character> set = new HashSet<>();
      for (int j = i; j < n; j++) { // End of substring
        if (set.contains(s.charAt(j))) {
          break; // Stop if we find a duplicate character
        }
        set.add(s.charAt(j)); // Add character to set
```

```
            maxLength = Math.max(maxLength, j - i + 1); // Update max length
        }
    }
    return maxLength;
  }
}
```

j - i + 1:
Index: 0 1 2
String: a b c
If i = 0 and j = 2, the substring is "abc".

The length of "abc" should be 3, but j - i = 2 - 0 = 2 (which is incorrect).

To count all characters, we do j - i + 1 = 2 - 0 + 1 = 3.

Time Complexity: $O(n^3)$
Space Complexity: O(min(n, m))
Here, n is the length of the string, and m is the size of the character set

Note: This approach is inefficient for large strings due to its cubic time complexity.

Optimized Solution

1) Two-Pointer Approach

Concept: This method uses two pointers to create a sliding window that represents the current substring without duplicate characters.

Steps:

n this approach, we utilize two pointers, often referred to as **start** and **end**, to represent the current window (substring) within the string that contains unique characters.

- **start Pointer:** Marks the beginning of the current substring.
- **end Pointer:** Expands the substring by moving through each character in the string.

Detailed Steps

1. **Initialization:**
    - Set both start and end pointers to the beginning of the string (0).
    - Create a HashSet to keep track of characters within the current window.
    - Initialize a variable maxLength to store the maximum length of substrings found.
2. **Expanding the Window:**
    - Iterate through the string using the end pointer.
    - For each character at position end:
        - If the character is **not** in the HashSet:
            - Add it to the HashSet.
            - Update maxLength as the maximum of its current value and the size of the window (end - start + 1).
            - Move the end pointer to the right (end++).
        - If the character **is** already in the HashSet:
            - Remove the character at the start pointer from the HashSet.
            - Move the start pointer to the right (start++).
            - Repeat this process until the duplicate character is removed from the window.
3. **Continue the Process:**
    - Repeat step 2 until the end pointer reaches the end of the string.
4. **Result:**
    - After the loop concludes, maxLength contains the length of the longest substring without repeating characters

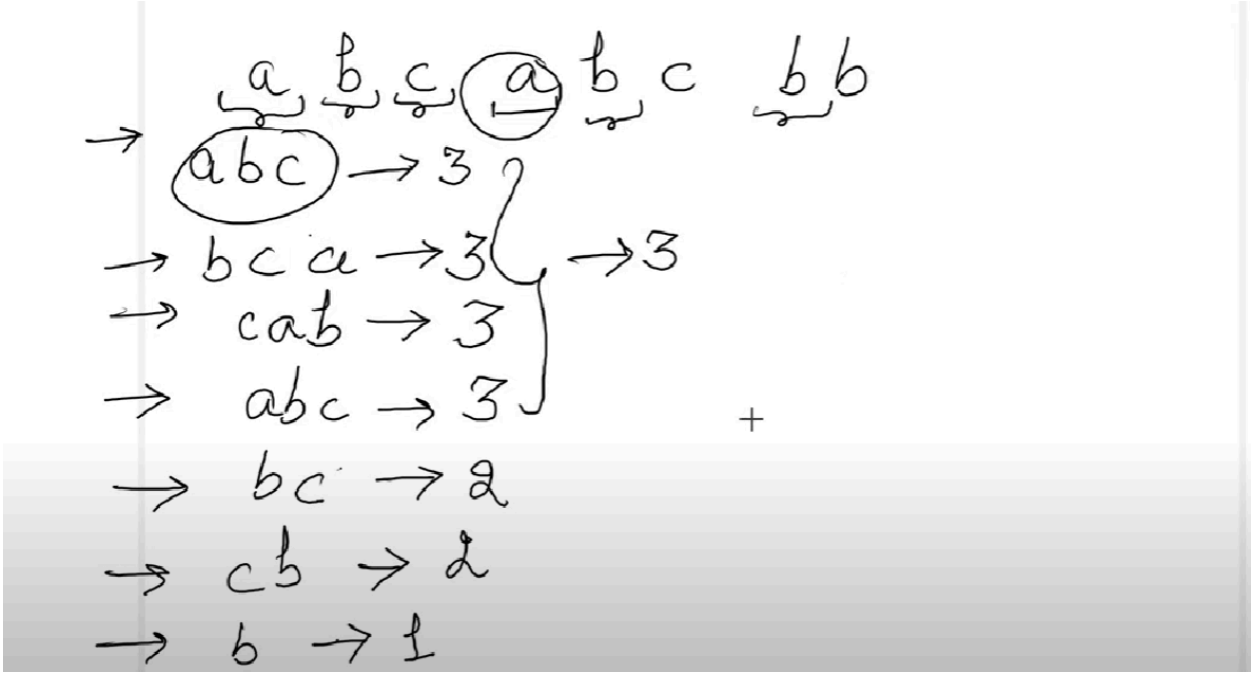

```
import java.util.HashSet;
public class LongestSubstringWithoutRepeating {
  public static int lengthOfLongestSubstring(String s) {
    int start = 0, end = 0, maxLength = 0;
    HashSet<Character> uniqueChars = new HashSet<>();
    while (end < s.length()) {
      char currentChar = s.charAt(end);
      if (!uniqueChars.contains(currentChar)) {
        uniqueChars.add(currentChar);
        maxLength = Math.max(maxLength, end - start + 1);
        end++;
      } else {
        uniqueChars.remove(s.charAt(start));
        start++;
      }
    }
    return maxLength;
  }
  public static void main(String[] args) {
    String s = "pwwkew";
    System.out.println("Length of Longest Substring Without Repeating Characters: " +
lengthOfLongestSubstring(s));
  }
}
```

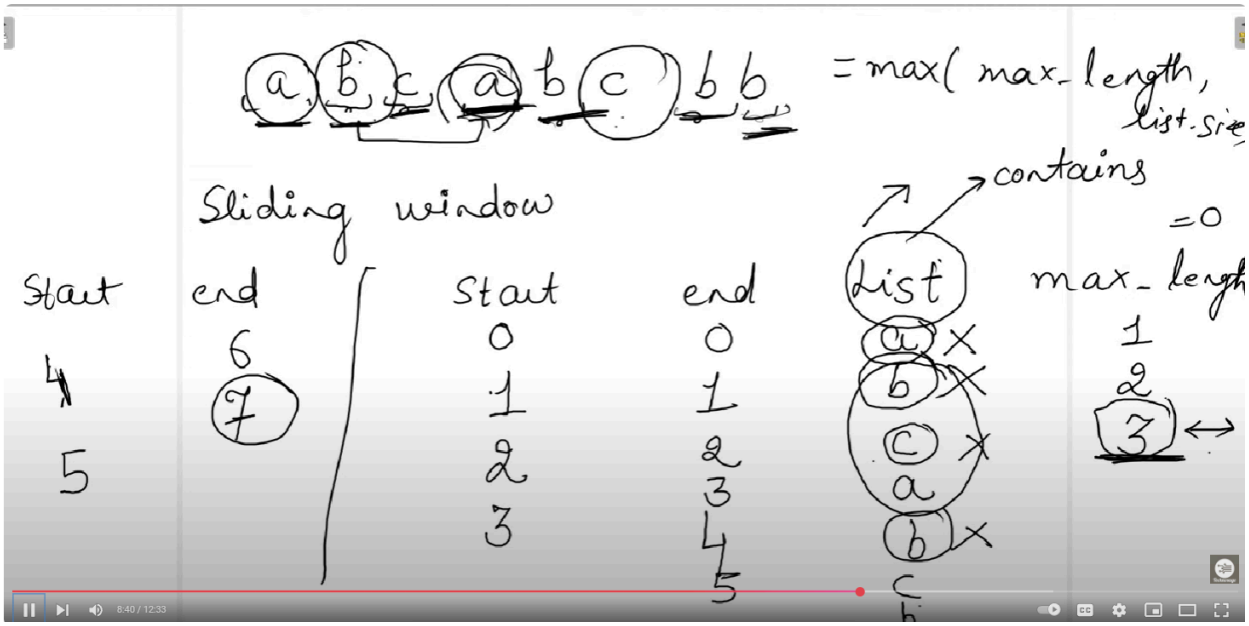
Time Complexity: O(n)
Space Complexity: O(min(n, m))
Here, n is the length of the string, and m is the size of the character set.

Note: This approach is more efficient than the brute force method, as each character is processed at most twice.

2) Sliding window

Possible sub strings for below example





Sliding Window Approach

Approach:

1. Use a **List** to store characters and track unique characters in the current window.
2. Initialize start = 0 , end = 0 and maxLength =0.
3. Expand the end pointer until a duplicate character is found.
4. When a duplicate is found:
    - Remove characters from start until the duplicate is removed.
    - Increment start accordingly.
5. Update maxLength whenever a new valid window is found.

```
package com.leetcode150;
import java.util.ArrayList;
import java.util.List;
public class LongestSubstringWithoutRepeatingCharacters {
	public static void main(String[] args) {
		String s = "abcabcbb";
```

```
		System.out.println("max length is:" + longestSWRC(s));
	}
	public static int longestSWRC(String s) {
		int start = 0;
		int end = 0;
		int max_length = 0;
		List<Character> list = new ArrayList<>(); // Stores characters in the current
window
		while (end < s.length()) {
			if (!list.contains(s.charAt(end))) {
				list.add(s.charAt(end));
				end++;
				max_length = Math.max(max_length, list.size());
			} else {
				list.remove(Character.valueOf(s.charAt(start)));
				start++;
			}
		}
		return max_length;
	}
}
```

Output

max length is:3

Time Complexity: O(n)
Space Complexity: O(min(n, m))

Key Differences:

- **Data Structure Used:**
    - **Two-Pointer Approach:** Typically employs a HashSet to efficiently check for the presence of characters and ensure all characters in the current window are unique.
    - **Sliding Window Approach:** In the provided implementation, a List is used to store characters in the current window. However, this can lead to less efficient

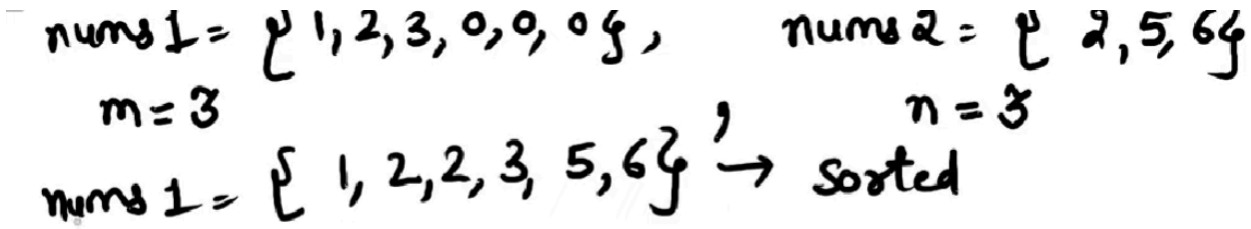
operations when checking for duplicates and removing characters, as these operations may require linear time in a list.

8. Merge Sorted Array

nums1: It has a length of m + n. The first m elements contain the actual sorted values, and the last n elements are set to 0 as placeholders to accommodate elements from nums2.

nums2: It contains n sorted elements.

Goal: Merge nums2 into nums1 so that nums1 becomes a fully sorted array of length m + n.



Example 1:

Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]
Explanation: The arrays we are merging are [1,2,3] and [2,5,6].
The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from nums1.

Example 2:

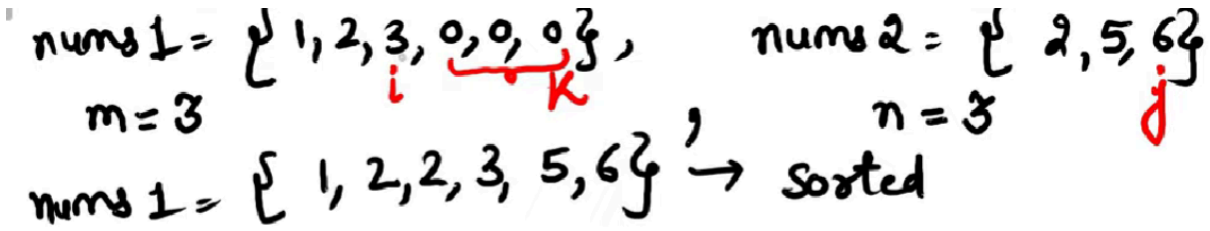
Input: nums1 = [1], m = 1, nums2 = [], n = 0
Output: [1]
Explanation: The arrays we are merging are [1] and [].
The result of the merge is [1].

General Solutions

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        for (int j = 0; j < n; j++) {
```

```
            nums1[m+j] = nums2[j];
        }
         Arrays.sort(nums1);
        System.out.println(Arrays.toString(nums1));

    }
}
```

Optimized Solution: Two Pointer Approach



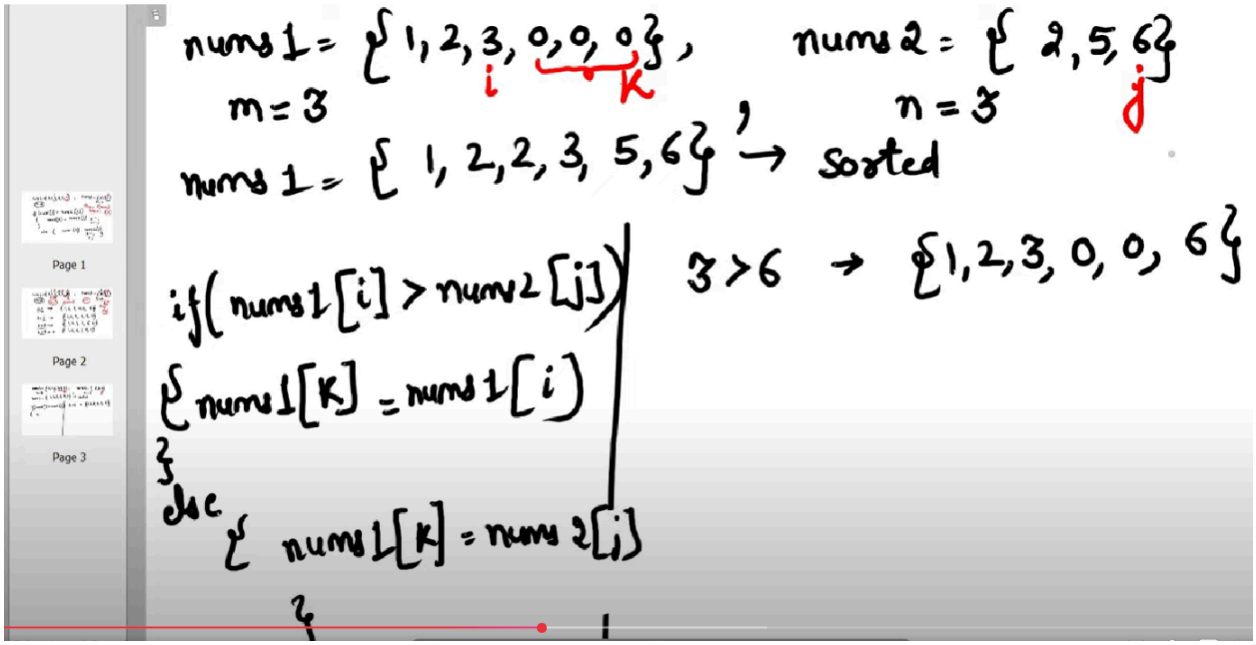
AS we need a sorted array so will three pointers . As we already know, the nums1 array has size of m+n ; so we will compare the elements in nums1 till size (m) and nums 2 , place the elements in nums1 array after size the nums1 array(m).

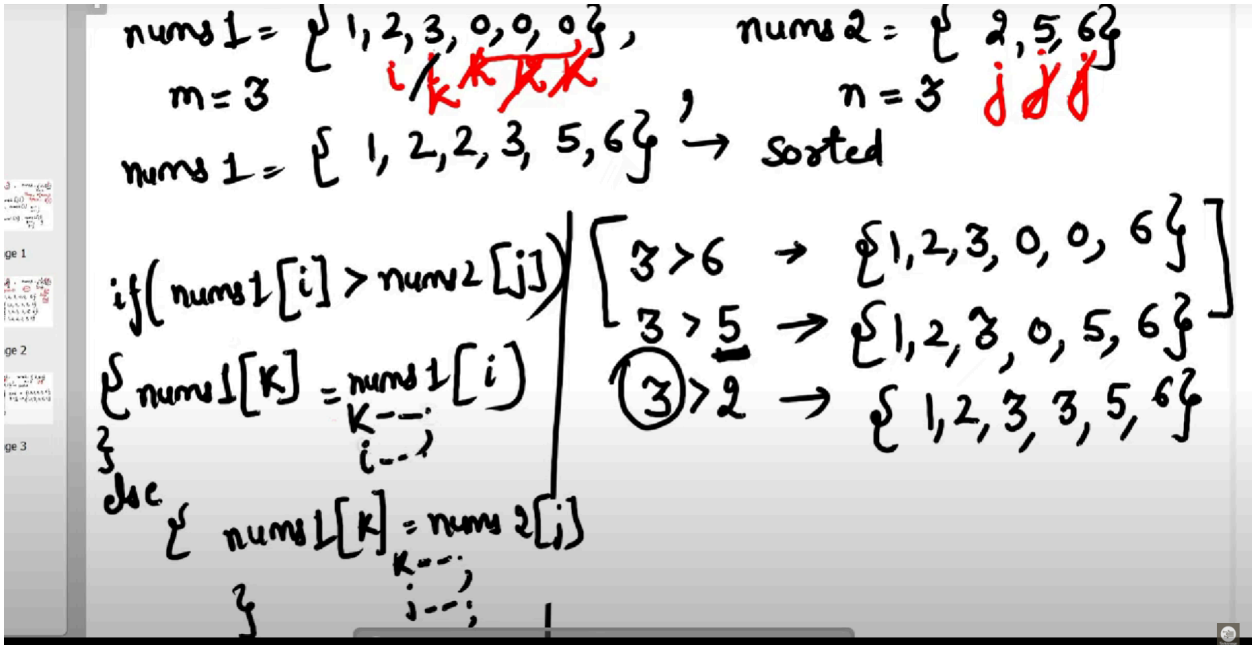
Key Idea (Three Pointers, Start from End)

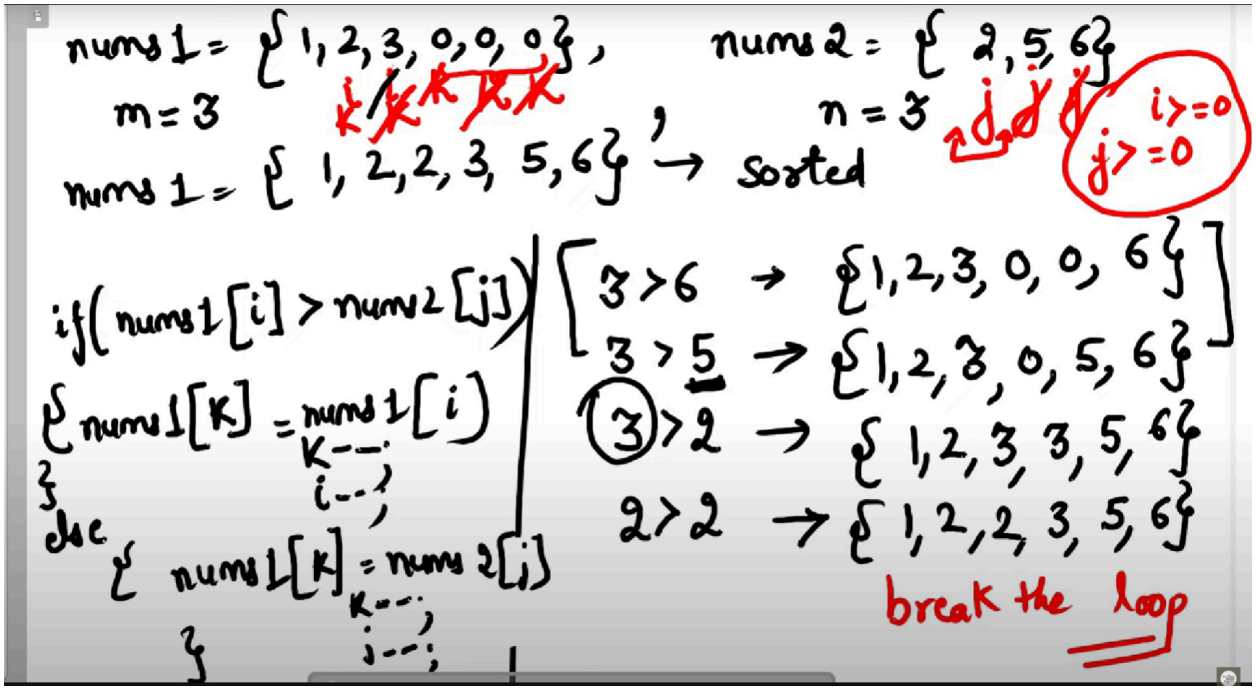
1. Use three pointers:
    - i = m - 1 → Last valid element in nums1.
    - j = n - 1 → Last element in nums2.
    - k = m + n - 1 → Last position in nums1 (end of the array).

2. Compare elements from nums1[i] and nums2[j]:
    - If nums1[i] > nums2[j], place nums1[i] at nums1[k], then decrement i and k.

- Else, place nums2[j] at nums1[k], then decrement j and k.







Optimized Solution

```
package com.leetcode150;
public class MergeSortedArray {
	public static void main(String[] args) {
		int nums1[] = { 1, 2, 3, 0, 0, 0 };
		int nums2[] = { 2, 3, 6 };
		int m = 3;
		int n = 3;
		mergeArrays(nums1, m, nums2, n);
		for (int arr : nums1) {
			System.out.print(arr + " ");
		}
	}
	public static void mergeArrays(int[] nums1, int m, int[] nums2, int n) {
		int i = m - 1;
		int j = n - 1;
		int k = m + n - 1;
		while (j >= 0) {
			if (i >= 0 && nums1[i] > nums2[j]) {
				nums1[k] = nums1[i];
				k--;
```

```
                    i--;
                } else {
                    nums1[k] = nums2[j];
                    k--;
                    j--;
                }
            }
        }
}
```

Output

1 2 2 3 3 6

9.Remove Element

Given an integer array **nums** and an integer **val**, remove all occurrences of **val** in **nums** in-place. The order of the elements may be changed. Then return the number of elements in **nums** which are not equal to **val**. **Do not take a new array to store.**

Example 1:

Input: nums = [3,2,2,3], val = 3
Output: 2, nums = [2,2,_,_]
Explanation: Your function should return k = 2, with the first two elements of nums being 2.
It does not matter what you leave beyond the returned k (hence they are underscores).

Example 2:

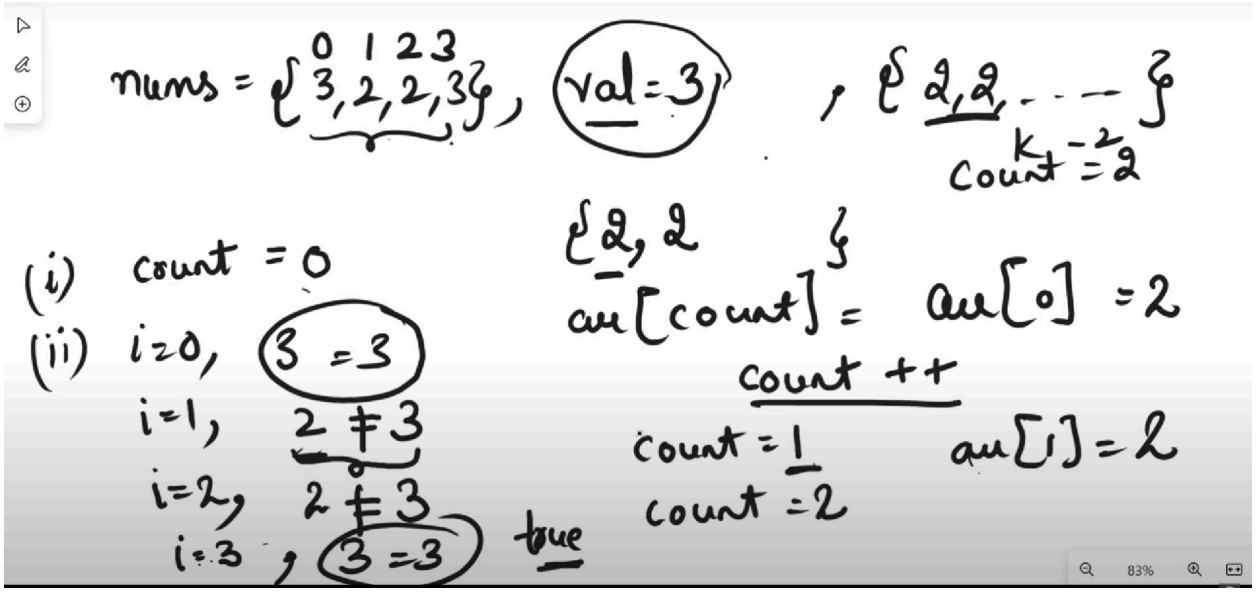
Input: nums = [0,1,2,2,3,0,4,2], val = 2
Output: 5, nums = [0,1,4,0,3,_,_,_]
Explanation: Your function should return k = 5, with the first five elements of nums containing 0, 0, 1, 3, and 4.
Note that the five elements can be returned in any order.
It does not matter what you leave beyond the returned k (hence they are underscores).

General Solutions

```java
package com.leetcode150;
import java.util.Arrays;
public class RemoveElementEX {
	public static void main(String[] args) {
		int nums[] = { 2, 5, 6, 8, 9, 2 };
		System.out.println("intially length of array: " + nums.length);
		int val = 2;
		int newLength = removeElement(nums, val);
		System.out.println("after removing val length of array: " + newLength);
		// Print only valid elements
		System.out.print("Modified array: ");
		for (int i = 0; i < newLength; i++) {
			System.out.print(nums[i] + " ");
		}
	}
	public static int removeElement(int nums[], int val) {
		int index = 0; // Tracks valid elements
		for (int i = 0; i < nums.length; i++) {
			if (nums[i] != val) {
				nums[index] = nums[i]; // Move valid elements forward
				index++;
			}
		}
		return index; // New valid length of the array
	}
}
```

initially length of array: 6
after removing val length of array: 4
Modified array: 5 6 8 9

Optimized Solution



Key Observations from the Image

1. Input Array
    - nums = {3, 2, 2, 3}
    - val = 3 (value to be removed)
2. Two-Pointer Approach (Efficient In-Place Modification)
    - A count (or index) variable is used to track the position where valid elements (not equal to val) should be placed.
    - If the current element is not equal to val, it is moved to nums[count], and count is incremented.
3. Step-by-Step Execution
    - count = 0
    - Iterate over nums:
        - i = 0: nums[i] = 3 → Skip (not stored)

        - i = 1: nums[i] = 2 → Store at nums[count], count++
        - i = 2: nums[i] = 2 → Store at nums[count], count++
        - i = 3: nums[i] = 3 → Skip (not stored)
    - Final modified array: {2, 2, _, _} (_ represents ignored elements)
    - New length = count = 2

4. Final Output
    - New Length = 2
    - Updated Array = {2, 2, _, _} (only first count elements matter)

```
public class RemoveElementEX {
	public static void main(String[] args) {
		int[] nums = { 3, 2, 2, 3 };
		int val = 3;
		System.out.println(removeElement(nums, val));
	}
	public static int removeElement(int nums[], int val) {
		int count = 0;
		for (int i = 0; i < nums.length; i++) {
			if (nums[i] != val) {
				nums[count] = nums[i];
				count++;
			}
		}
		return count;
	}
}
```

Output

2

10.Palindrome

Given an integer x, return true *if* x *is a palindrome, and* false *otherwise*.

Example 1:

Input: x = 121
Output: true
Explanation: 121 reads as 121 from left to right and from right to left.

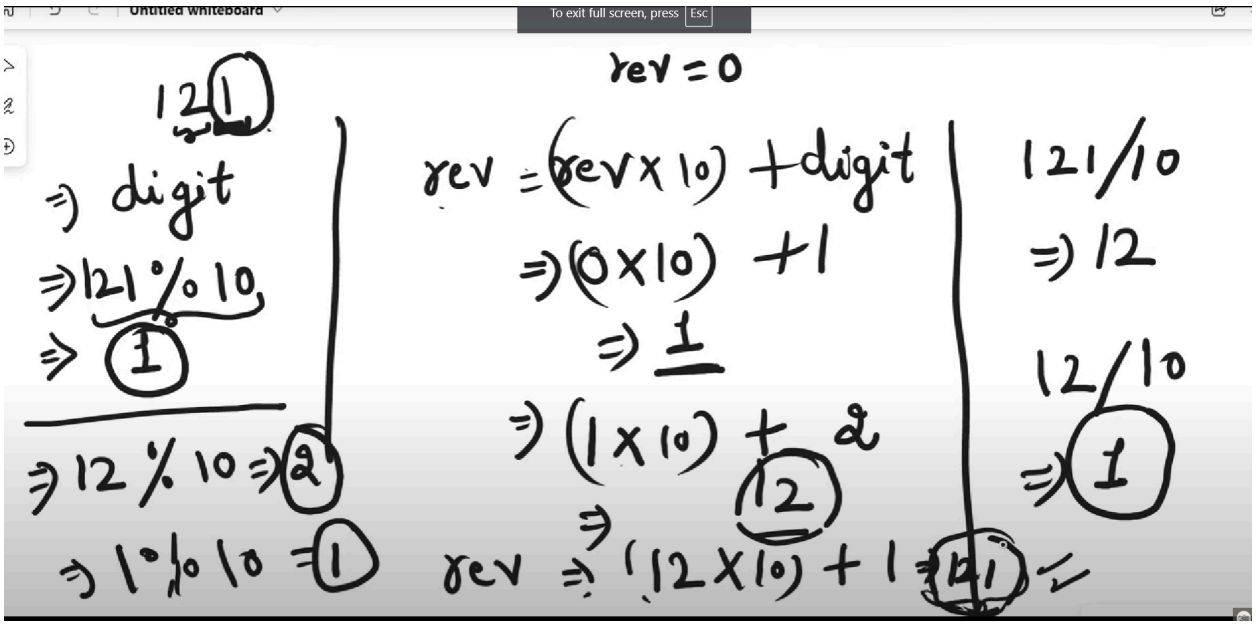
Example 2:

Input: x = -121
Output: false
Explanation: From left to right, it reads -121. From right to left, it becomes 121-. Therefore it is not a palindrome.

Example 3:

Input: x = 10
Output: false
Explanation: Reads 01 from right to left. Therefore it is not a palindrome.

Optimized Solution



```
package com.leetcode150;
public class Palindrome {
	public static void main(String[] args) {
		int num = 121;
		System.out.println(isPalindrome(num));
	}
	public static boolean isPalindrome(int x) {
		int original = x;
		int rem, rev = 0;
		while (x > 0) {
			rem = x % 10;
			rev = rev * 10 + rem;
			x = x / 10;
		}
		if (original == rev) {
			return true;
		} else {
			return false;
		}
	}
}
```

Output
true

Time & Space Complexity

- Time Complexity: $O(\log_{10}(n))$ (since x is divided by 10 in each step).
- Space Complexity: O(1) (only a few integer variables are used).

  No extra space used → O(1).

11.Longest Palindromic Substring

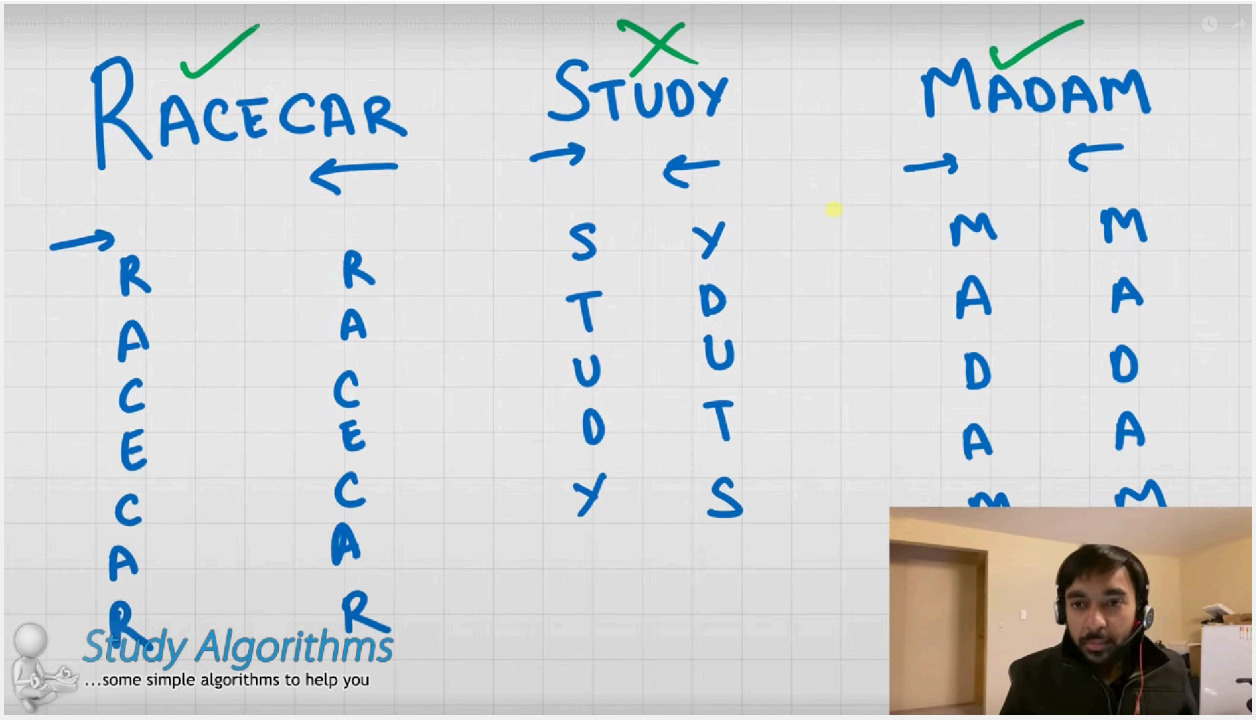
Given a string s, return *the longest palindromic substring* in s.

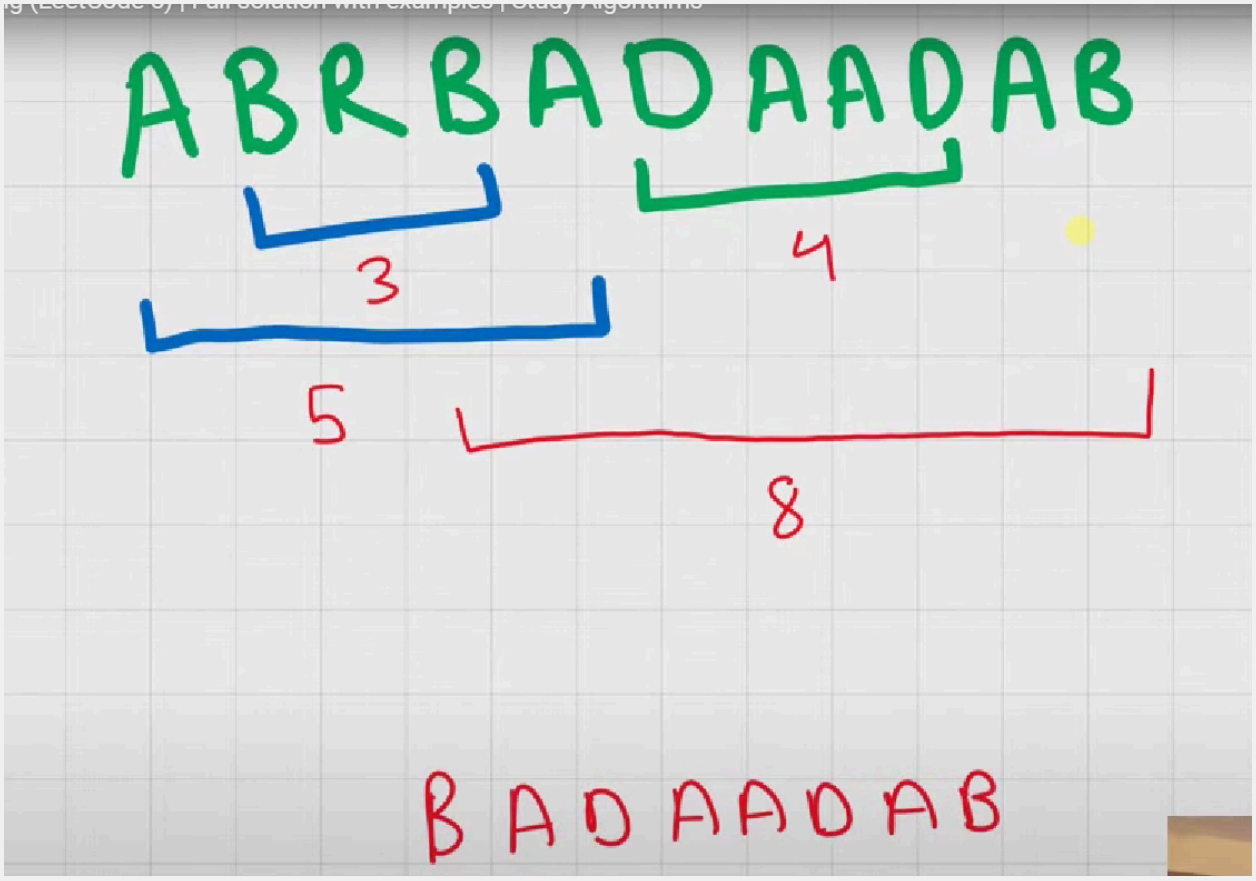
Example 1:

Input: s = "babad"
Output: "bab"
Explanation: "aba" is also a valid answer.

Example 2:

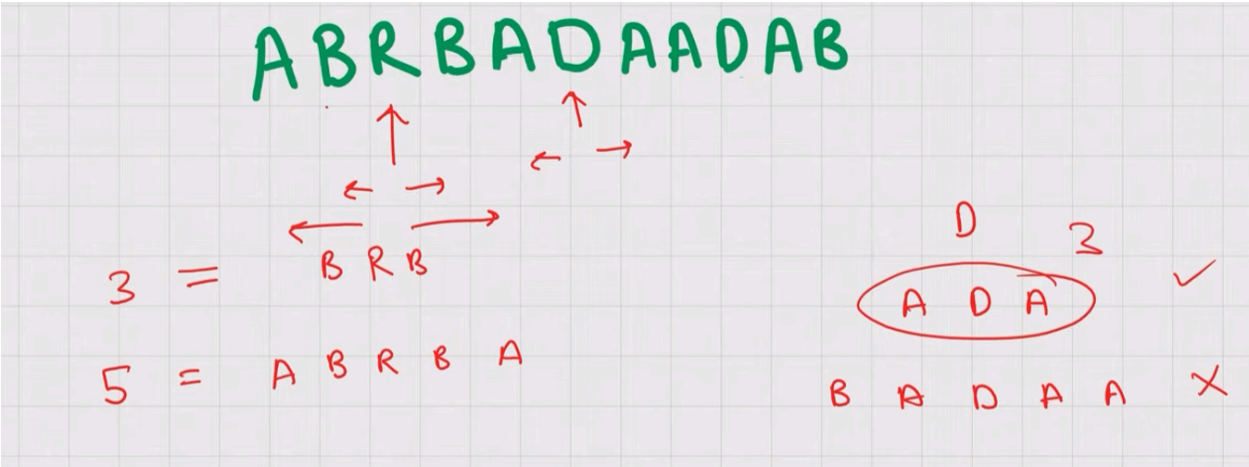
Input: s = "cbbd"
Output: "bb"





Single letter can also a palindrome

Going left and right side from character to find substring of palindrome from string



the **Expand Around Center** approach to finding the **Longest Palindromic Substring** in a given string. The key concept of this approach is:

- **Odd-length palindromes:** Expand outward from a single center character.
- **Even-length palindromes:** Expand outward from two adjacent characters.

1. **Odd-length palindrome:**
    - Start with left = i, right = i (single character as center).
      Expand outward while s[left] == s[right].
    - Example: "aba" → expand from 'b' → 'aba'.

2. **Even-length palindrome:**
    - Start with left = i, right = i + 1 (two adjacent characters as center).
    - Expand outward while s[left] == s[right].
    - Example: "cbbd" → expand from 'bb' → 'bb'.

Check for edge cases:

- If the string is null or has a length of 0 or 1, return the string itself as it is already a palindrome.

2️⃣ Initialize variables:

- longestPalindrome = "" (stores the longest palindrome found).

3️⃣ Loop through the string starting from index 1:

- Treat each character as a center to expand around.

- Use **two-pointer expansion** to check palindromes.

4️⃣ **Expand for odd-length palindromes:**

- Set left = i, right = i.
- Expand outward while s[left] == s[right].
- Stop when characters do not match or go out of bounds.
- Update longestPalindrome if a longer palindrome is found.

5️⃣ **Expand for even-length palindromes:**

- Set left = i, right = i + 1 (since an even-length palindrome has two adjacent characters as the center).
- Expand outward while s[left] == s[right].
- Stop when characters do not match or go out of bounds.
- Update longestPalindrome if a longer palindrome is found.

Return the longest palindromic substring found.

```
package com.leetcode150;
public class LongestPalindromicSubstring {
	public static void main(String[] args) {
		String s = "babad";
		System.out.println("Longest Palindromic Substring: " + longestPalindrome(s));
	}
	public static String longestPalindrome(String str) {
		if (str.length() <= 1)
			return str;
		String LPS = "";
		for (int i = 0; i < str.length(); i++) {
			// Consider odd length palindromes
			String oddPalindrome = expandAroundCenter(str, i, i);
			if (oddPalindrome.length() > LPS.length()) {
				LPS = oddPalindrome;
			}
			// Consider even length palindromes
```

```
                String evenPalindrome = expandAroundCenter(str, i, i + 1);
                if (evenPalindrome.length() > LPS.length()) {
                    LPS = evenPalindrome;
                }
            }
            return LPS;
        }
        private static String expandAroundCenter(String str, int left, int right) {
            while (left >= 0 && right < str.length() && str.charAt(left) == str.charAt(right)) {
                left--;
                right++;
            }
            return str.substring(left + 1, right); // Extract valid palindrome
        }
}
```

Output

Longest Palindromic Substring: bab

12. Climbing Stairs

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example 1:

Input: n = 2
Output: 2

Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps

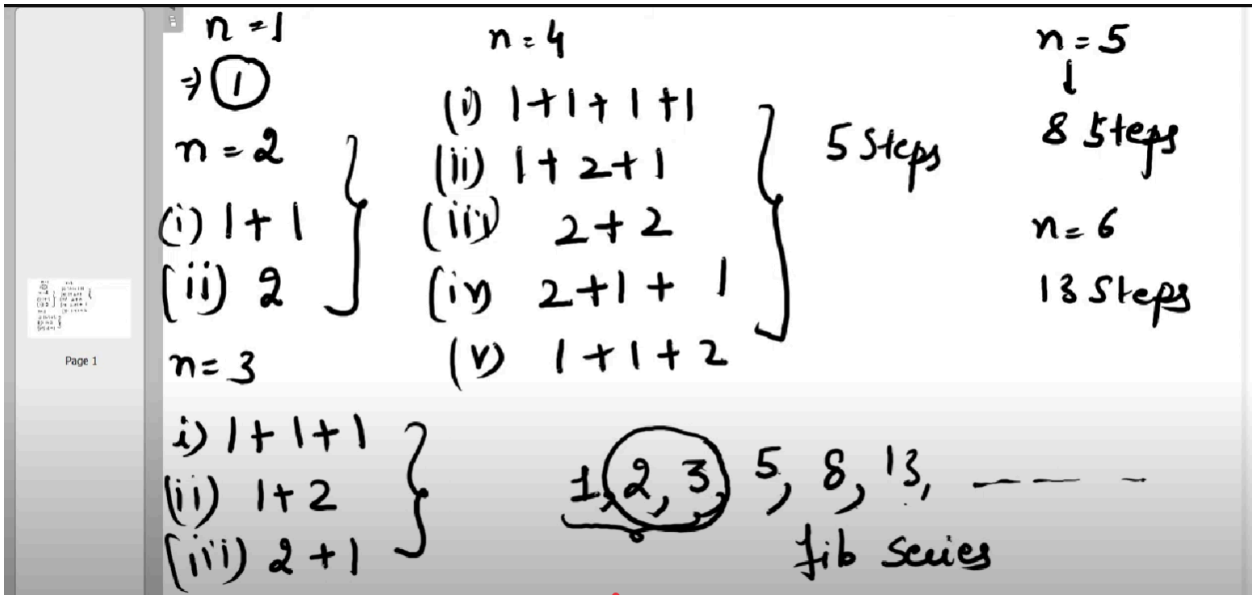
Example 2:

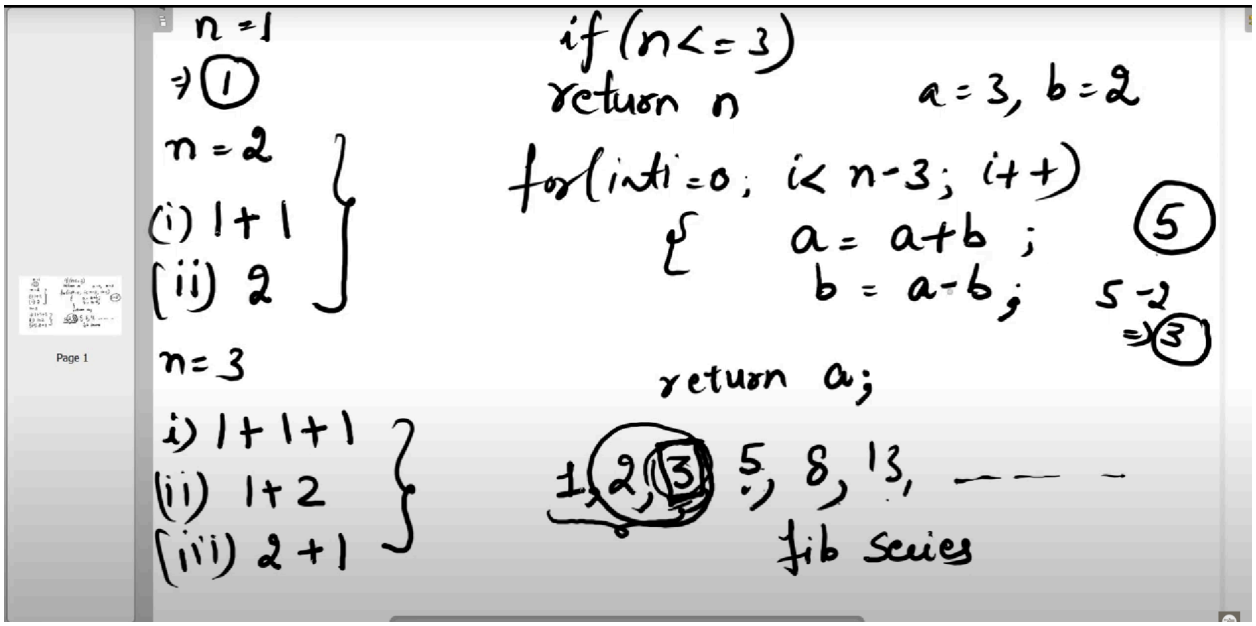
Input: n = 3
Output: 3

Explanation: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

Optimized Solution





Flow

The Climbing Stairs problem follows the Fibonacci sequence concept!

- Handle base cases: If n <= 3, return n directly since the number of ways is equal to n till 3.
- Use two variables (a, b) to store previous results instead of an array (saving space).

- **Iterate n - 3 times using for (int i = 0; i < n - 3; i++)**:

  Compute the new number of ways as b = a + b.

  Update a using a = b - a

Return b as the final count of distinct ways.

```
package com.leetcode150;
public class ClimbingStairsEx {
	public static void main(String[] args) {
		System.out.println(climbingStairs(5));
	}
	public static int climbingStairs(int n) {
		if (n <= 3) {
			return n;
		}
		int a = 2, b = 3;
		for (int i = 0; i < n - 3; i++) {
			b = a + b; // Compute new Fibonacci number
			a = b - a; // Update `a` using difference
			/*
			 * int temp = a + b; // Compute next Fibonacci number a = b; // Move `a` to `b`
			 * b = temp; // Move `b` to `temp`
			 */
		}
		return b;
	}
}
```

Output

8

Ex; for 3 ->3
4->5,
5->8

13.Valid Parentheses

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: s = "()"

Output: true

Example 2:

Input: s = "()[]{}"

Output: true

Example 3:

Input: s = "(]"

Output: false

Example 4:

Input: s = "([])"

Output: true

Optimized Solution

(i) ( ), { }, [ ]

(ii) ([ ]) ✓ correct → Valid

(([ ])]) → Incorrect

(iii) { [} ✗

(i) S = "( )" → valid String

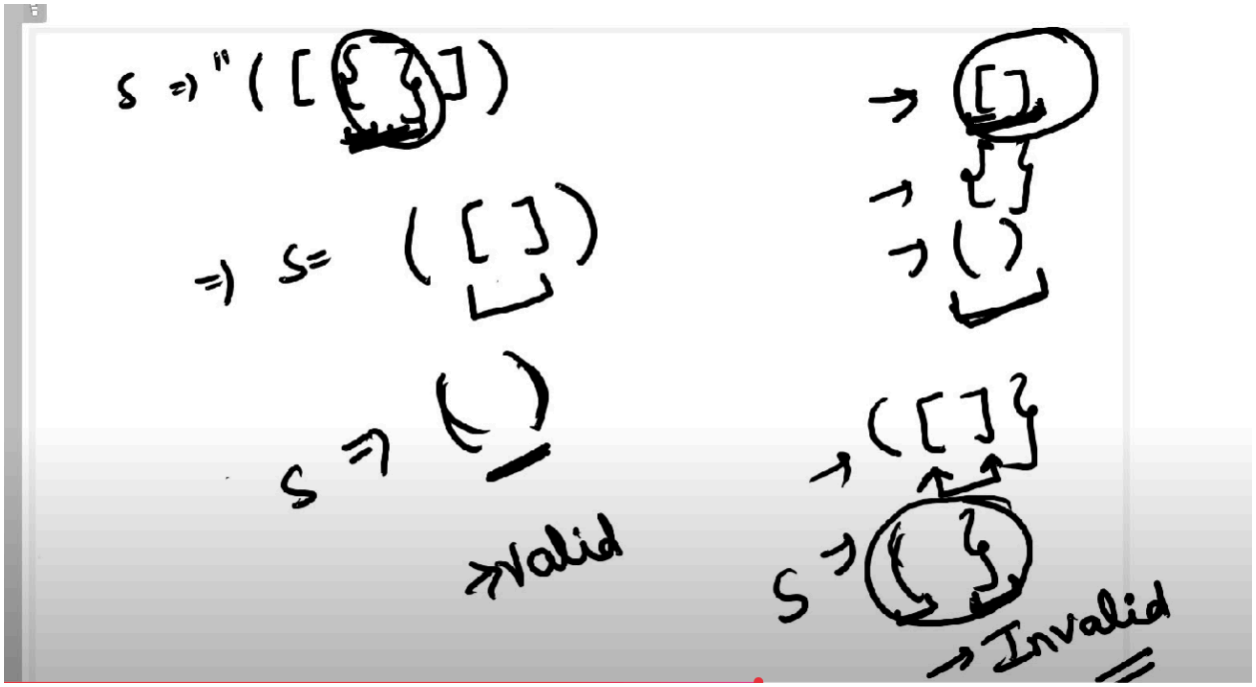
Yes

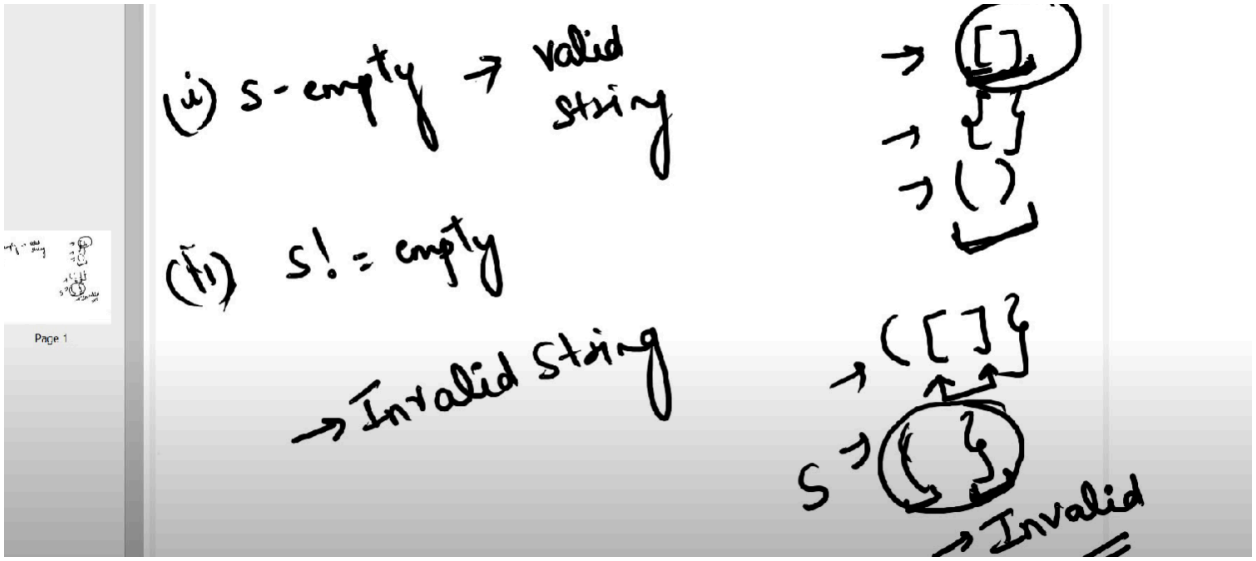
(ii) "( ) [ ] { }" → valid String

(iii) "( ]" → Invalid String

diff

(iv) ([ ]) → valid String





How This Works

Keep checking for valid pairs ("()", "[]", "{}"**)**

- If found, remove them using s.replace("()", ""), s.replace("[]", ""), etc.
- Repeat until no more valid pairs exist.
  If the string is empty after all removals, return true (valid).

Ex: below S1=([{}])

([{}]) - > {} valid so will assign empty(“”) .
([]) - > [] valid so will assign empty(“”)
() - > () valid so will assign empty(“”) .

If characters remain, return false (invalid).

S2 = ([]}
([]} - > [] valid so will assign empty(“”) .
(} no opening and closing brackets fail.

```
package com.leetcode150;
public class ValidParentheses {
	public static void main(String[] args) {
		String s = "{}";
		System.out.println(validParentheses(s));
	}
	public static boolean validParentheses(String s) {
		while (true) {
			if (s.contains("()")) {
				s = s.replace("()", "");
			} else if (s.contains("[]")) {
				s = s.replace("[]", "");
			} else if (s.contains("{}")) {
				s = s.replace("{}", "");
			} else {
				return s.isEmpty();
			}
		}
	}
}
```

14. Reverse Array or Reverse String

Write a function that reverses a string. The input string is given as an array of characters s.

You must do this by modifying the input array in-place with O(1) extra memory.

Example 1:

Input: s = ["h","e","l","l","o"]
Output: ["o","l","l","e","h"]

Example 2:

Input: s = ["H","a","n","n","a","h"]
Output: ["h","a","n","n","a","H"]

Optimized Solution

To **reverse an array**, you can use the **two-pointer approach**, which swaps elements from the **beginning** and **end** while moving toward the **center**.

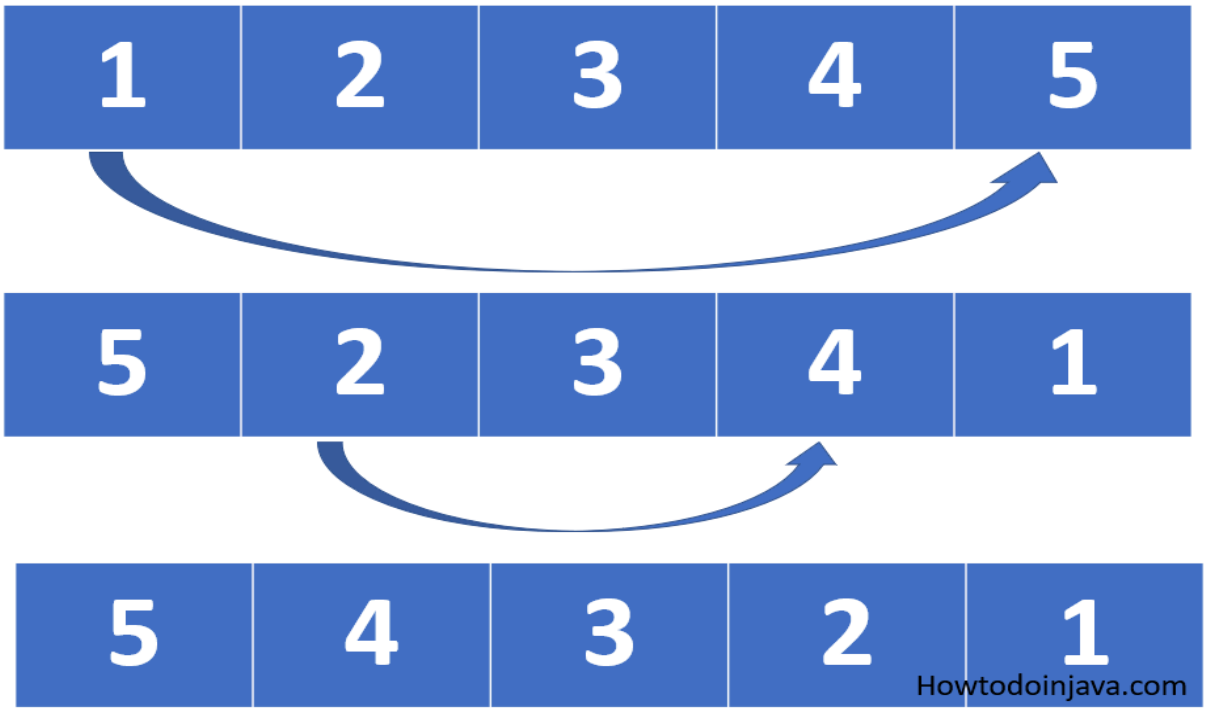
The reversal of an array **does not sort it in ascending or descending order**—it only flips the elements' positions. It **mirrors** the array from front to back.

Not Sorted: The order of elements is simply reversed but not arranged in ascending or descending order.
Preserves Relative Order: The first element moves to the last, the second moves to the second-last, and so on.
Does Not Change Values: It only swaps positions.

Method: Using Two-Pointer Approach

Algorithm

1. Initialize two pointers:
    - start = 0 (beginning of the array).
    - end = n-1 (end of the array).
2. Swap nums[start] and nums[end].
3. Move the pointers inward:
    - start++
    - end--
4. Repeat until start >= end.



Initial Array:

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

Step	Start Index	End Index	Swap Elements	After Swap
1	0	6	Swap (2, 7)	`[7, 5, 3, 4, 6, 1, 2]`
2	1	5	Swap (5, 1)	`[7, 1, 3, 4, 6, 5, 2]`
3	2	4	Swap (3, 6)	`[7, 1, 6, 4, 3, 5, 2]`
4	3	3	**Stop (No Swap Needed)**	`[7, 1, 6, 4, 3, 5, 2]`

Final Reversed Array:

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

Reverse string

```
package com.leetcode150;
import java.util.Arrays;
public class ReverseString {
	public static void main(String[] args) {
		char[] nums = { 'h', 'e', 'l', 'l', 'o' };
		reverseString(nums);
		System.out.println("Reversed: " + Arrays.toString(nums));
```

```
	}
	public static void reverseString(char[] nums) {
		int start = 0;
		int end = nums.length - 1;
		while (start < end) {
			char temp = nums[start];
			nums[start] = nums[end];
			nums[end] = temp;
			start++;
			end--;
		}
	}
}
```

Output

Reversed: [o, l, l, e, h]

Reverse numbers (same code just **char[] nums is** replaced **by int[] nums , char temp** is replaced by **int temp) as** because it is numbers.

```
package com.leetcode150;
import java.util.Arrays;
public class ReverseArray {
	public static void main(String[] args) {
		int[] nums = { 3, 5, 6, 1, 2, 7 };
		reverseArray(nums, 0, nums.length - 1);
		System.out.println("reversed: " + Arrays.toString(nums));
	}
	public static void reverseArray(int[] nums, int start, int end) {
		while (start < end) {
			int temp = nums[start];
			nums[start] = nums[end];
			nums[end] = temp;
			start++;
			end--;
		}
	}
}
```

reversed: [7, 2, 1, 6, 5, 3]

15.Rotate Array

Given an integer array nums, rotate the array to the right by k steps, where k is non-negative.

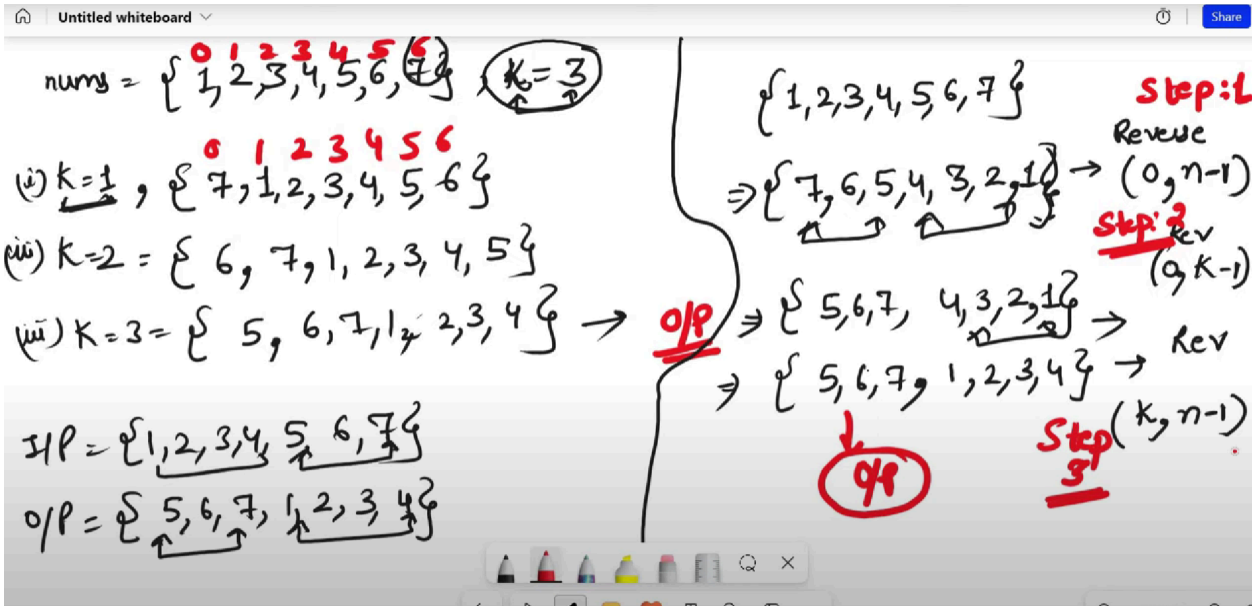
Example 1:

Input: nums = [1,2,3,4,5,6,7], k = 3
Output: [5,6,7,1,2,3,4]
Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]

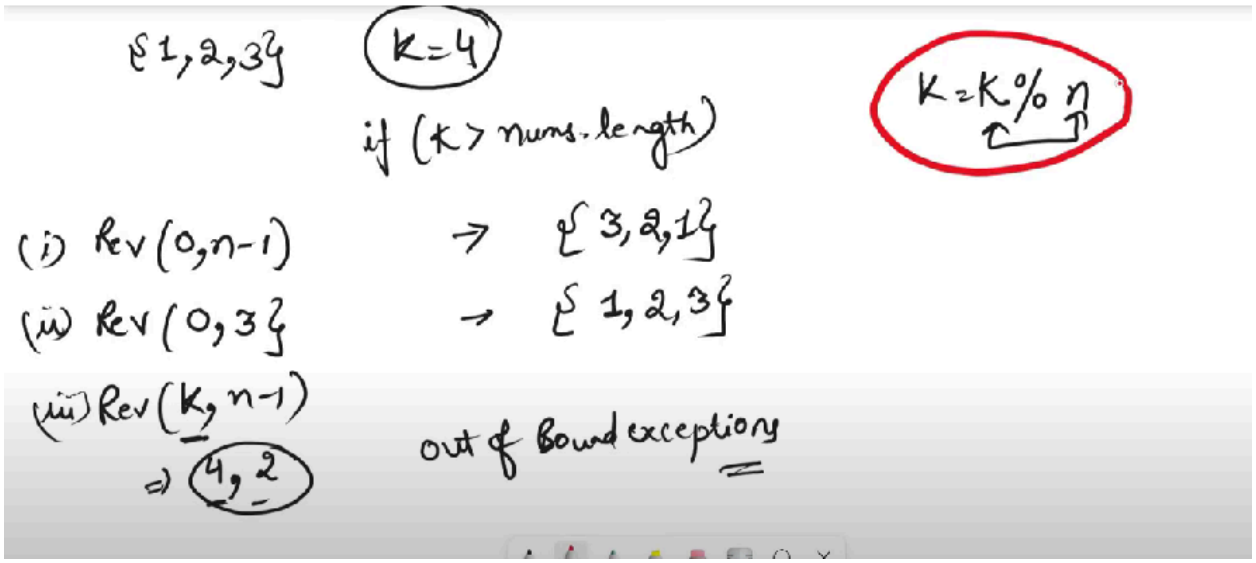
Example 2:

Input: nums = [-1,-100,3,99], k = 2
Output: [3,99,-1,-100]
Explanation:
rotate 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]

Optimized Solution



If K values > size of array (n)



The left side one is how it actually rotates the array.
The right side one is how we achieve the same using reverse .

Ex: nums ={1,2,3,4,5,6,7};,k=3;

Steps to Rotate an Array by k Positions

- **Reverse the entire array** → This moves the last k elements to the front but in reverse order.

Ex: nums ={1,2,3,4,5,6,7};

nums={7,6,5,4,3,2,1};

- **Reverse the first k elements** → This restores their correct order.

  For example k =3 , then will reverse k elements using (0 , k-1)

  Before: nums={7,6,5,4,3,2,1};

  After: nums={5,6,7,4,3,2,1};

- **Reverse the remaining (n-k) elements** → This restores their correct order.

  Before:nums={5,6,7,4,3,2,1};

  Will reverse remaining elements using (k,n-1)

  After: nums={5,6,7,1,2,3,4};

Handling k > n

```
1  package com.leetcode150;
2
3  import java.util.Arrays;
4
5  public class RotateArray {
6
7      public static void main(String[] args) {
8          int nums[] = { 1, 2, 3, 4, 5, 6, 7 };
9          int k = 13;
10         System.out.println("rotated: " + Arrays.toString(rotateArray(nums, k)));
11     }
```

Javadoc | Declaration | Console

```
<terminated> RotateArray [Java Application] C:\ProgramData\chocolatey\lib\springtoolsuite\tools\sts-4.23.1.RELEASE\plugins\org.eclipse.justj.openjdk.ho
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 12 out of bounds for length 7
        at com.leetcode150.RotateArray.reverse(RotateArray.java:24)
        at com.leetcode150.RotateArray.rotateArray(RotateArray.java:16)
        at com.leetcode150.RotateArray.main(RotateArray.java:10)
```

If k is greater than the size of the array (n), we **only need to rotate by k % n** positions because rotating by n or its multiples results in the same array.

Example:

- Given nums = {1,2,3,4,5,6,7} and k = 10.
- Since k > n, update k = 10 % 7 (k%n)= 3.
- Now, follow the above same 3-step process above with k = 3.

- This means rotating 10 times is the same as rotating k= **3 times**.

Reverse the Entire Array(0,n-1)

Before: {1, 2, 3, 4, 5, 6, 7}
After: {7, 6, 5, 4, 3, 2, 1}

Reverse the First k = 3 Elements(0,k-1)

Before: {7, 6, 5, 4, 3, 2, 1}

After: {5, 6, 7, 4, 3, 2, 1}

Reverse the Remaining Elements(k,n-1)

Before: {5, 6, 7, 4, 3, 2, 1}

After: {5, 6, 7, 1, 2, 3, 4}

Final Rotated Array: {5, 6, 7, 1, 2, 3, 4}

```
package com.leetcode150;
import java.util.Arrays;
public class RotateArray {
	public static void main(String[] args) {
		int nums[] = { 1, 2, 3, 4, 5, 6, 7 };
		int k = 3;
		System.out.println("rotated: " + Arrays.toString(rotateArray(nums, k)));
	}
	public static int[] rotateArray(int[] nums, int k) {
		int n = nums.length;
		k = k % n; // it for when k>n (k = 3 % 7 = 3 (same, since k < n) , k = 13%7 =6
		reverse(nums, 0, n - 1);
		reverse(nums, 0, k - 1);
		reverse(nums, k, n - 1);
		return nums;
	}
	public static void reverse(int[] nums, int start, int end) {
		while (start <= end) {
			int temp = nums[start];
			nums[start] = nums[end];
			nums[end] = temp;
			start++;
			end--;
```

```
                }
        }
}
```

Output

rotated: [5, 6, 7, 1, 2, 3, 4]

Since we are not using any additional data structures, the space complexity is O(1).

Time Complexity

The algorithm consists of three main steps:

1. Reverse the entire array → O(n)
2. Reverse the first k elements → O(k)
3. Reverse the remaining (n-k) elements → O(n-k)

Total time complexity:

$O(n)+O(k)+O(n-k)=O(n)+O(n)=O(2n)=O(n)$

In Big-O notation, constant factors like 2 are ignored, so O(2n) is simplified to O(n).

16.Reverse Integer

Given a signed 32-bit integer x, return x *with its digits reversed*. If reversing x causes the value to go outside the signed 32-bit integer range [-231, 231 - 1], then return 0.

Assume the environment does not allow you to store 64-bit integers (signed or unsigned).

Example 1:

Input: x = 123
Output: 321

Example 2:

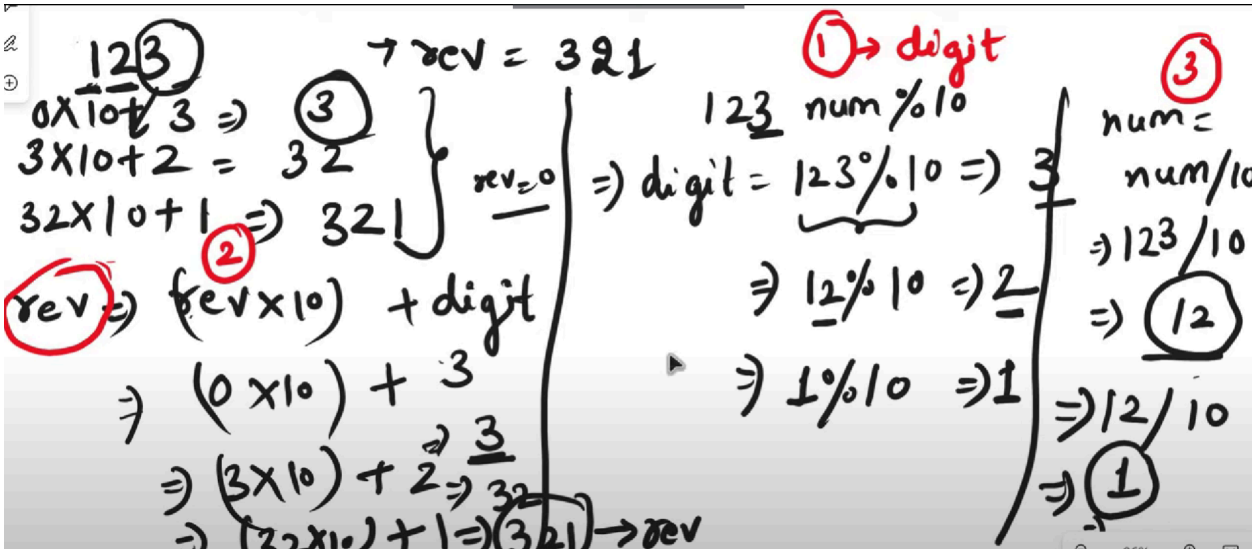
Input: x = -123

Output: -321

Example 3:

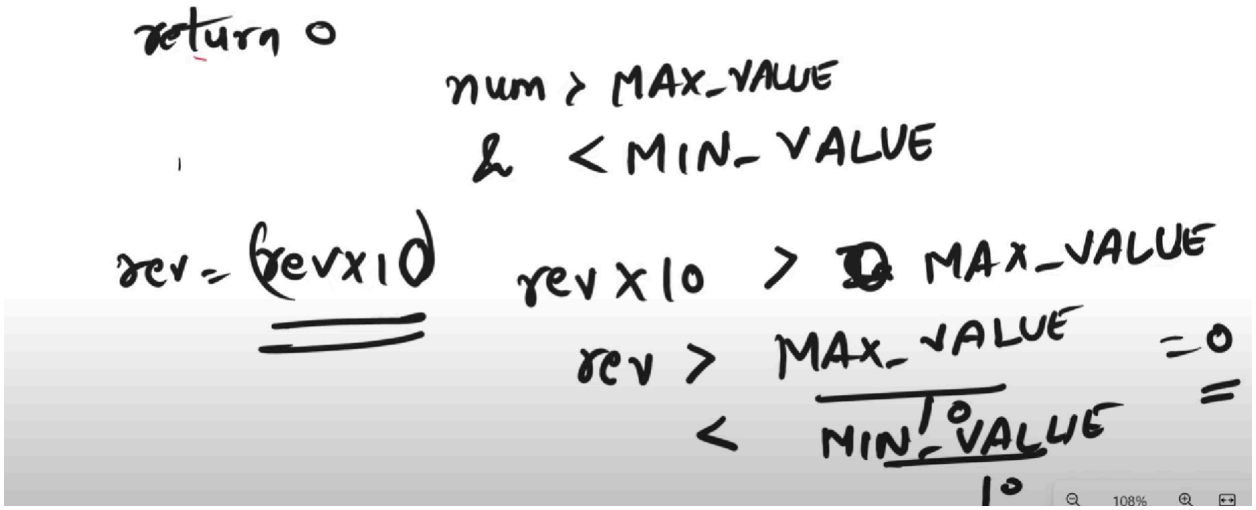
Input: x = 120
Output: 21

Optimized Solution



Given a signed 32-bit integer `x`, return `x` *with its digits reversed*. If reversing `x` causes the value to go outside the signed 32-bit integer range `[-2^31, 2^31 - 1]`, then return `0`.

you need to ensure that reversing the integer does **not** exceed the **signed 32-bit integer range**,



Checking if rev * 10 exceeds MAX_VALUE or MIN_VALUE.

Returning 0 if overflow occurs.

If rev > Integer.MAX_VALUE / 10, return 0 (positive overflow).

If rev < Integer.MIN_VALUE / 10, return 0 (negative overflow).

```
package com.leetcode150;
public class ReverseInteger {
	public static void main(String[] args) {
		int x = -189;
		System.out.println(reverse(x));
	}
	public static int reverse(int x) {
		int rev = 0, rem;
		while (x != 0) {
			rem = x % 10;
			if (rev > Integer.MAX_VALUE / 10 || rev < Integer.MIN_VALUE / 10) {
				return 0;
			}
			rev = rev * 10 + rem;
			x = x / 10;
		}
		return rev;
	}
}
```

Output

-981

If x= 187 , output = 781
x= -98 , output = -89(even negative values also reversing)
x= 19900000000000 , output = 0 (it return o has it exceed the 32 bit integer range)

17. **Majority Element**

Given an array nums of size n, return *the majority element*.

The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

Example 1:

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

n = 3, so ⌊n / 2⌋ = ⌊3 / 2⌋ = 1

The element 3 appears **twice**, which is more than 1

Output: 3

Example 2:

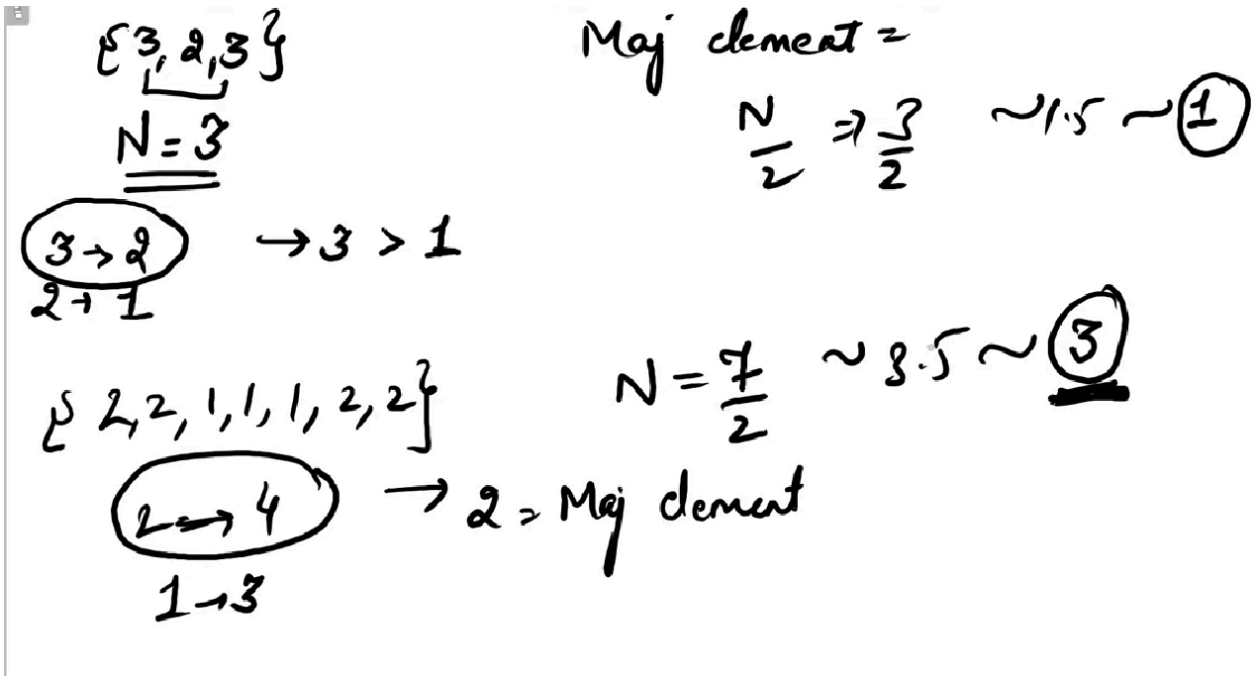
Input: nums = [2,2,1,1,1,2,2]
Output: 2

n = 7, so ⌊n / 2⌋ = ⌊7 / 2⌋ = 3

The element 2 appears **four times**, which is more than 3

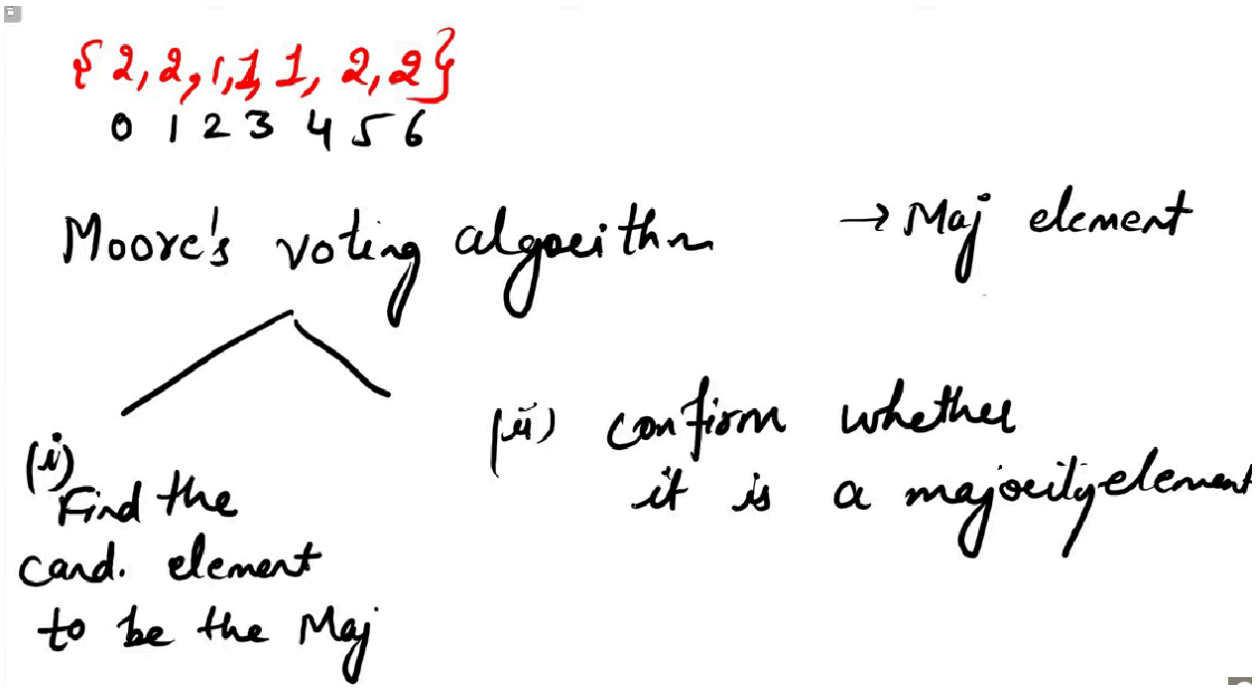
Output: 2

Optimized Solution



We can achieve it using hashmap, also like storing numbers and their count but it takes more time and space complexity so that's why we will use the below way.

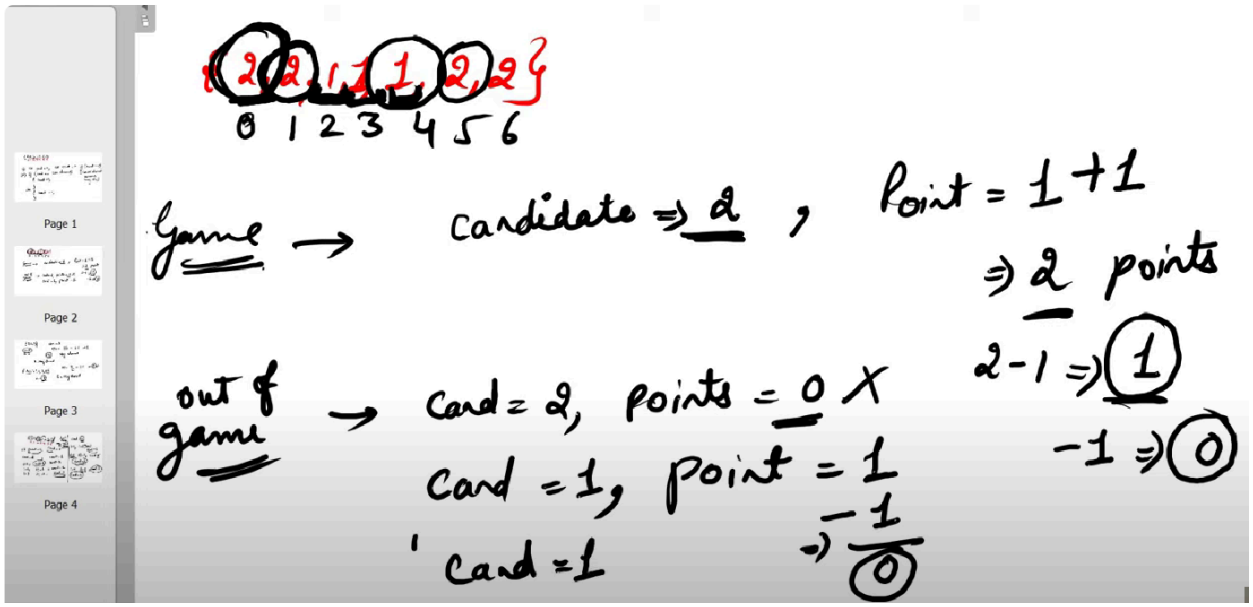
Moore's Voting Algorithm is an efficient algorithm used to find the **majority element** in an array in **O(n) time and O(1) space**. It works in two phases:

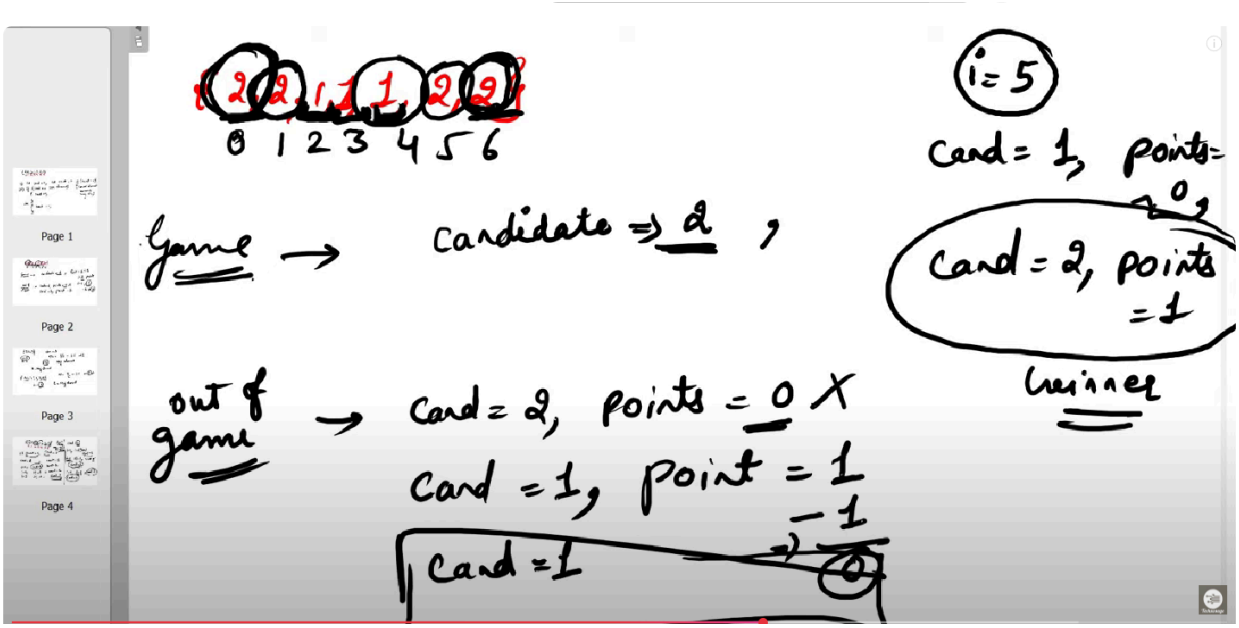


Ex1:

Candidate Selection Phase

- Initialize a candidate variable and a count variable (set to 1).
- Traverse through the array:
    - If count is 0, set the candidate to the current element.
    - If the current element matches the **candidate**, increase count.
    - Otherwise, decrease count.





Example Input:

plaintext Copy Edit

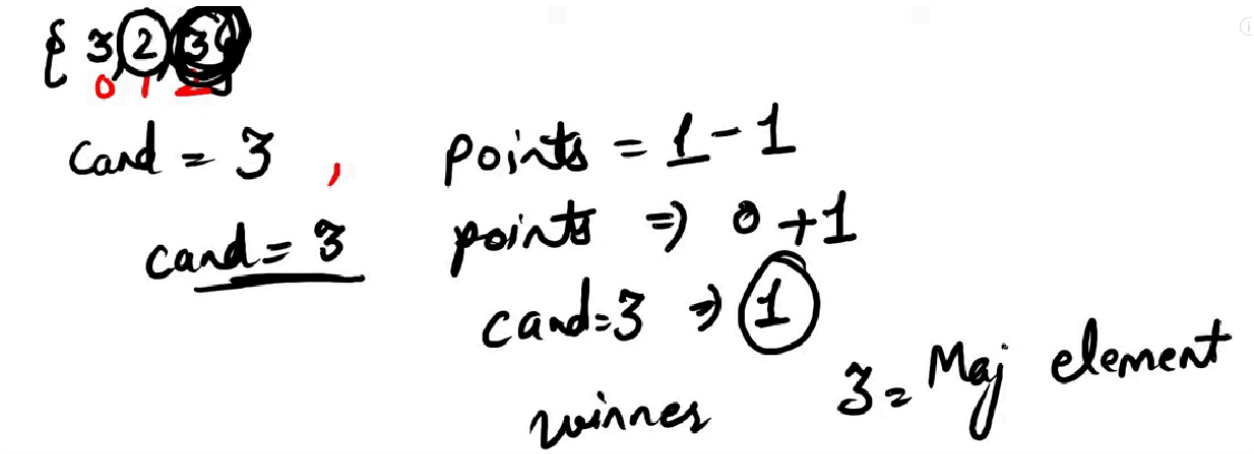
```
nums = [2,2,1,1,1,2,2]
```

Initialization:

- `candidate = 0` (default value)
- `count = 0`

Step	Element	Condition (`count == 0` ?)	Candidate Update	Count Update
1	2	✅ Yes	candidate = 2	count = 1
2	2	❌ No	- (unchanged)	count = 2
3	1	❌ No	- (unchanged)	count = 1
4	1	❌ No	- (unchanged)	count = 0
5	1	✅ Yes	candidate = 1	count = 1
6	2	❌ No	- (unchanged)	count = 0
7	2	✅ Yes	candidate = 2	count = 1

Ex2:




```
package com.leetcode150;
public class MajorityElement {
	public static void main(String[] args) {
		int nums[] = { 2, 2, 1, 1, 1, 2, 2 };
		System.out.println(majorityElement(nums));
	}
	public static int majorityElement(int[] nums) {
		int cand = 0;
		int count = 0;
		for (int i = 0; i < nums.length; i++) {
			if (count == 0) {
				cand = nums[i];
			}
			if (cand == nums[i]) {
				count++;
			} else {
				count--;
			}
		}
		return cand;
	}
}
```

Output

2

Time Complexity:

- **O(n)** → Since there is a **single loop** iterating through the array of size n, the time complexity is **O(n)**.

Space Complexity:

- **O(1)** → The algorithm uses only **two extra variables** (cand and count), which remain constant regardless of the input size. No additional data structures (like arrays, hash maps, etc.) are used, so the space complexity is **O(1)**.

Otherway-general

```
package com.leetcode150;
import java.util.HashMap;
public class MajorityElement {
	public static void main(String[] args) {
		int nums[] = { 2, 2, 1, 1, 1, 2, 2 };
		System.out.println(majorityElement(nums));
	}
	public static int majorityElement(int[] nums) {
		HashMap<Integer, Integer> map = new HashMap<>();
		int n = nums.length;
		for (int num : nums) {
			map.put(num, map.getOrDefault(num, 0) + 1);
			if (map.get(num) > n / 2) {
				return num;
			}
		}
		return -1; // Should never reach here as majority element is guaranteed
	}
}
```

Output

2

The **key** is the number from nums[].

The **value** is the **count** of occurrences of that number.
map.put(num, map.getOrDefault(num, 0) + 1);
This updates the count of the number num in the map:

map.getOrDefault(num, 0):

If num exists in the map, it returns its current count.

If num does not exist, it returns 0.

+1: Increments the count.

map.put(num, newCount): Updates the count in the HashMap.

18.Find the index of the first occurrence in a string

Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

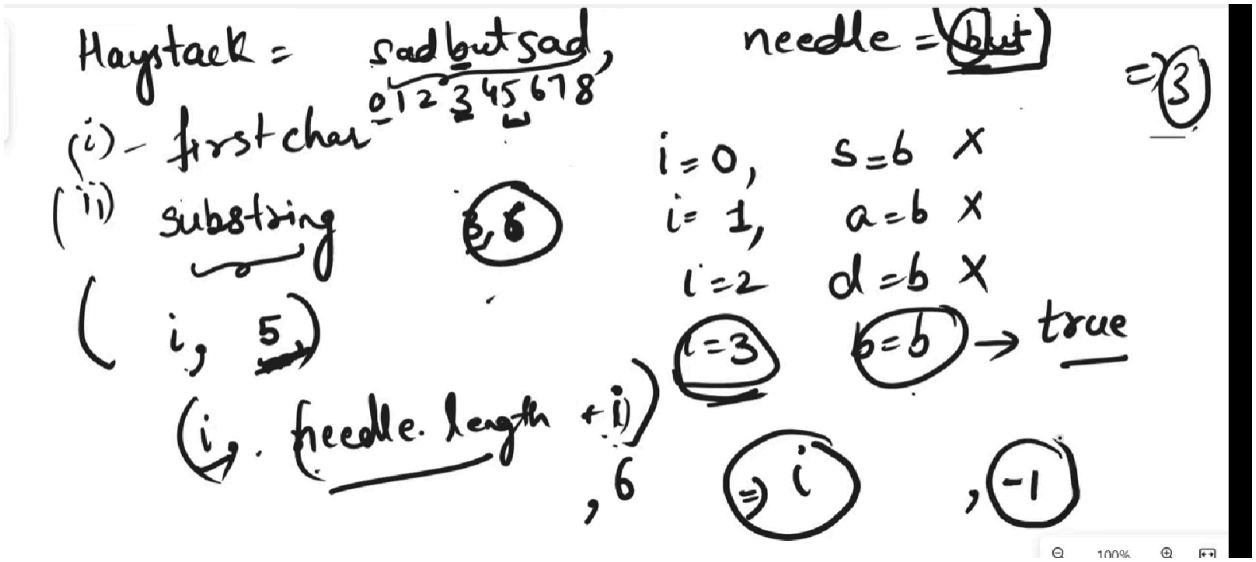
Example 1:

Input: haystack = "sadbutsad", needle = "sad"
Output: 0
Explanation: "sad" occurs at index 0 and 6.
The first occurrence is at index 0, so we return 0.

Example 2:

Input: haystack = "leetcode", needle = "leeto"
Output: -1
Explanation: "leeto" did not occur in "leetcode", so we return -1.

Optimized Solution



We use two conditions

1) **Check for the First Character:**

- First, check if the **first character** of string2 exists in string1.
- If the first character **doesn't exist** in string1, then **string2 cannot be a substring** of string1, so return -1.
  If the first character **is found**, note down its **index** in string1

  string1 = "sadbutsad", string2 = "sad"

  1st Condition: Check for the first character of string2 ('s') in string1 → Found at index 0.

2) **Extract and Compare the Substring:**

- Using the index from the first condition, extract a substring from string1 starting from that index with the length of string2.

- Compare the extracted substring with string2.

  2nd Condition:
  substring(index , string.length()+index) -> substring(0,0+needle.length())
  ->substring(0,3)

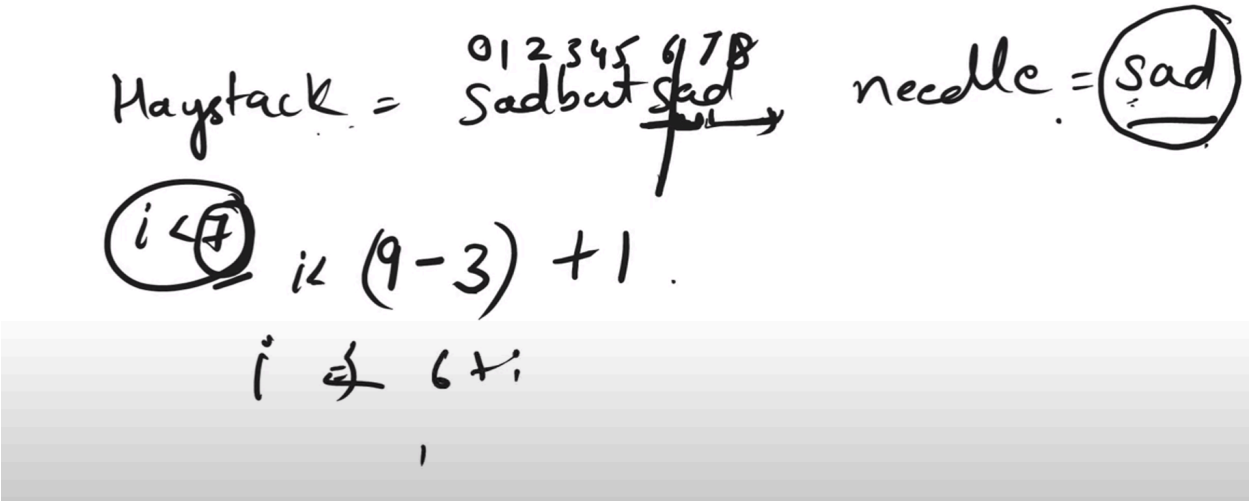
  Extract substring (0, 3) from string1 → substring = "sad". Compare with string2 → Match found.

3) **Return the Index or -1:**

- If the extracted substring matches string2, return the **index**.
- If no match is found after checking all possibilities, return -1

  Ex: Extract substring (0, 3) from string1 → substring = "sad". Compare with string2 → Match found.

  Return index: 0.



In above

// we can use for (int i = 0; i < str1.length(); i++) also will get the same results but if we below one will reduce some time complexity.

By using str1.length() - str2.length() + 1, we ensure that the loop only iterates up to the last possible index where str2 can fit inside str1.

Breaking It Down:

- str1.length() - str2.length(): This gives the last starting index where str2 can fit completely inside str1.
- +1: Ensures we include that last valid index in the iteration.

Ex: String str1 = "abcdef";

String str2 = "cd";

str1.length() = 6

str2.length() = 2

str1.length() - str2.length() + 1 = 6 - 2 + 1 = 5

This means we **only** check indices 0, 1, 2, 3, 4 because:

- Index 4 is the last position where a substring of length 2 can be extracted ("ef").
- Checking beyond index 4 (i.e., 5) would try to extract characters beyond str1.length().

If we add +1 i < str1.length() - str2.length() + 1

if we don't want add +1 we can use also i <= str1.length() - str2.length()

```
package com.leetcode150;

public class FindTheIndexOfTheFirstOccurrenceInAString {
	public static void main(String[] args) {
		System.out.println(firstOccurrenceInAString("sadbutsad", "sad"));
	}
	public static int firstOccurrenceInAString(String str1, String str2) {
for (int i = 0; i < str1.length() - str2.length() + 1; i++) { // we can use  for (int i = 0; i < str1.length(); i++)
			if (str1.charAt(i) == str2.charAt(0)) {
				if (str1.substring(i, str2.length() + i).equals(str2)) {
					return i;
				}
			}
		}
		return -1;
	}
}
```

Output

0

Space Complexity: **O(1)**Since we are not using any additional data structures, the space complexity is O(1).

According leet code string variables instead of str1 , str2

```
class Solution {
    public int strStr(String haystack, String needle) {
        for (int i = 0; i < haystack.length() - needle.length() + 1; i++) {
            if (haystack.charAt(i) == needle.charAt(0)) {
                if (haystack.substring(i, needle.length() + i).equals(needle)) {
                    return i;
                }
            }

        }
        return -1;
    }
}
```

19.**Best Time to Buy and Sell Stock**

You are given an array of prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

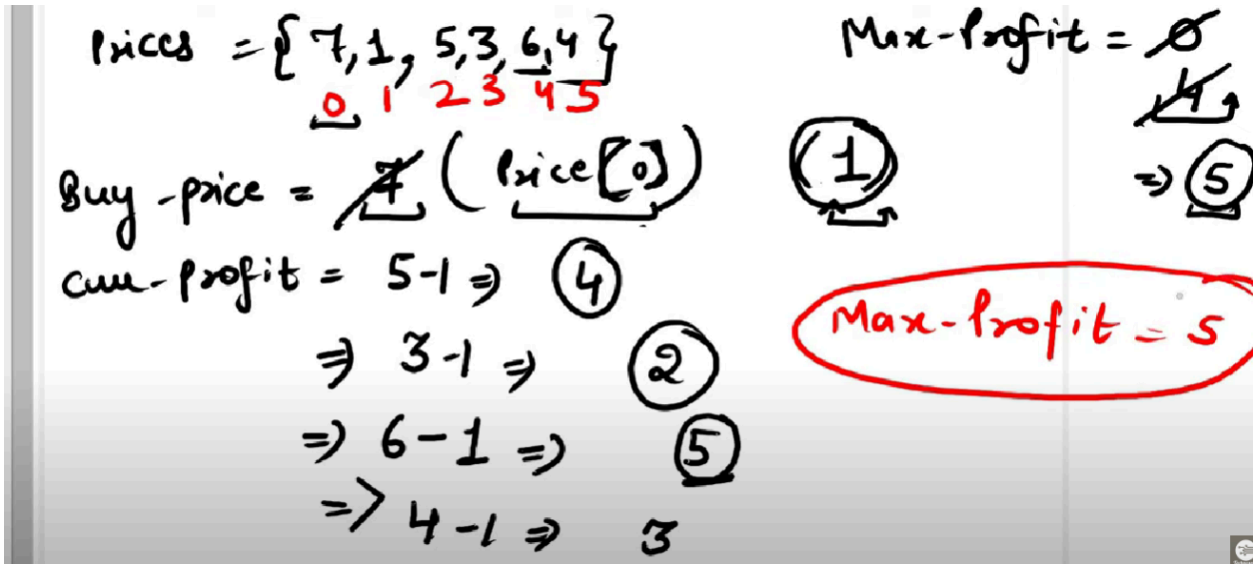
Example 1:

Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transactions are done and the max profit = 0.

Optimized Solution



Steps to Find Maximum Profit:

1. **Initialize** maximum profit = 0 as we don't know what the maximum profit is.

2. **Assume buy_price starts at index 0** (price at the 0th index) and compare it with other prices in the array to get the maximum profit.

3. Let's assume the **current profit** by buying at index 0 and selling at index 1:

    - **Example Input:** prices = [7,1,5,3,6,4]

    - Initially, buy_price = 7 (0th index), selling_price = 1 (1st index)

    - **Current profit** = 1 - 7 = -6 (Loss), so no profit, and we **won't update maximum profit**.

4. Since index 1 has a lower price than index 0 (prices[1] (1) < buy_price (7)), we **update** buy_price = prices[i] (buy_price = 1.)
    - Now, to know the **current profit**, we compare it with the next value.
    - **Current profit** = 5 - 1 = 4 (Profit), so we update:
        - maximum profit = 0 → maximum profit = 4
    - Now, since index 2 has a **higher price** than index 1((prices[2] (5) > buy_price (1)), we **don't update** buy_price.
5. **Buy price remains = 1**, and the next selling price is 3
    - **Current profit** = 3 - 1 = 2, which is **less than maximum profit**, so we don't update maximum profit.
6. **Buy price remains = 1**, and the next selling price is 6
    - **Current profit** = 6 - 1 = 5
    - Since current profit > maximum profit, we update:
        - maximum profit = 4 → maximum profit = 5
7. **Buy price remains = 1**, and the next selling price is 4
    - **Current profit** = 4 - 1 = 3, which is **less than maximum profit**, so we don't update maximum profit.
8. **No more elements** in the array to compare, so the **final maximum profit = 5**.

Input: `prices = [7,1,5,3,6,4]`

Step	Buy Price	Sell Price	Current Profit	Max Profit	Action
1	7 (0th index)	1 (1st index)	**1 - 7 = -6** (Loss)	0	No update
2	**Update Buy Price = 1**	5 (2nd index)	**5 - 1 = 4**	**Update Max Profit = 4**	
3	Buy Price = 1	3 (3rd index)	**3 - 1 = 2**	No update	
4	Buy Price = 1	6 (4th index)	**6 - 1 = 5**	**Update Max Profit = 5**	
5	Buy Price = 1	4 (5th index)	**4 - 1 = 3**	No update	

✅ **Final Maximum Profit = 5** 🎯

```java
package com.leetcode150;
public class BestTimetoBuyandSellStock {
	public static void main(String[] args) {
		int prices[] = { 7, 1, 5, 3, 6, 4 };// 7,6,4,3,1
		System.out.println(maxProfit(prices));
	}
	public static int maxProfit(int[] prices) {
		int n = prices.length;
		int max_Profit = 0;
		int buy_price = prices[0]; // Initially index 0 will be buy_price
		for (int i = 1; i < n; i++) { // starting from second index
			int current_profit = prices[i] - buy_price;
			if (current_profit > max_Profit) {
				max_Profit = current_profit;
			}
			if (prices[i] < buy_price) {
				buy_price = prices[i];
			}
		}
		return max_Profit;
```

```
        }
}
```

Output
5

Time Complexity: O(n) → Single loop iterating over the array once.
Space Complexity: O(1) → Only a few integer variables, no extra data structures.

20.Jump Game

You are given an integer array **nums**. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return true *if you can reach the last index, or* false *otherwise*.

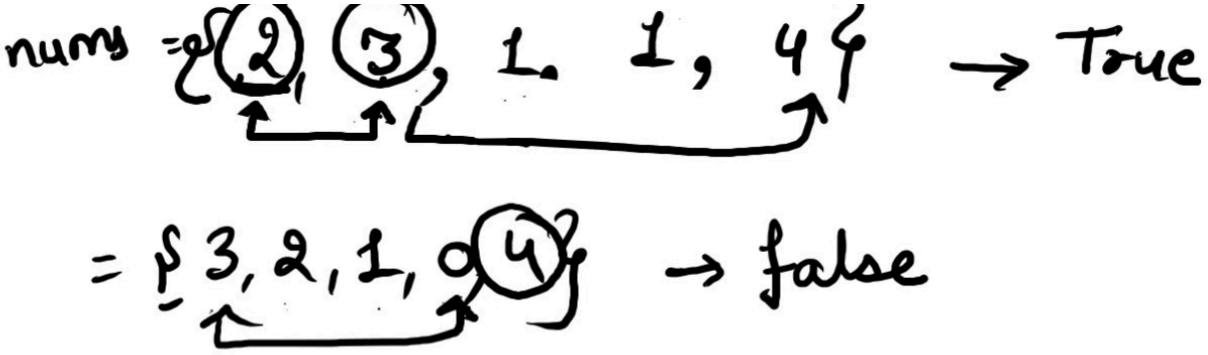
Example 1:

Input: nums = [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2:

Input: nums = [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.

Optimized Solution



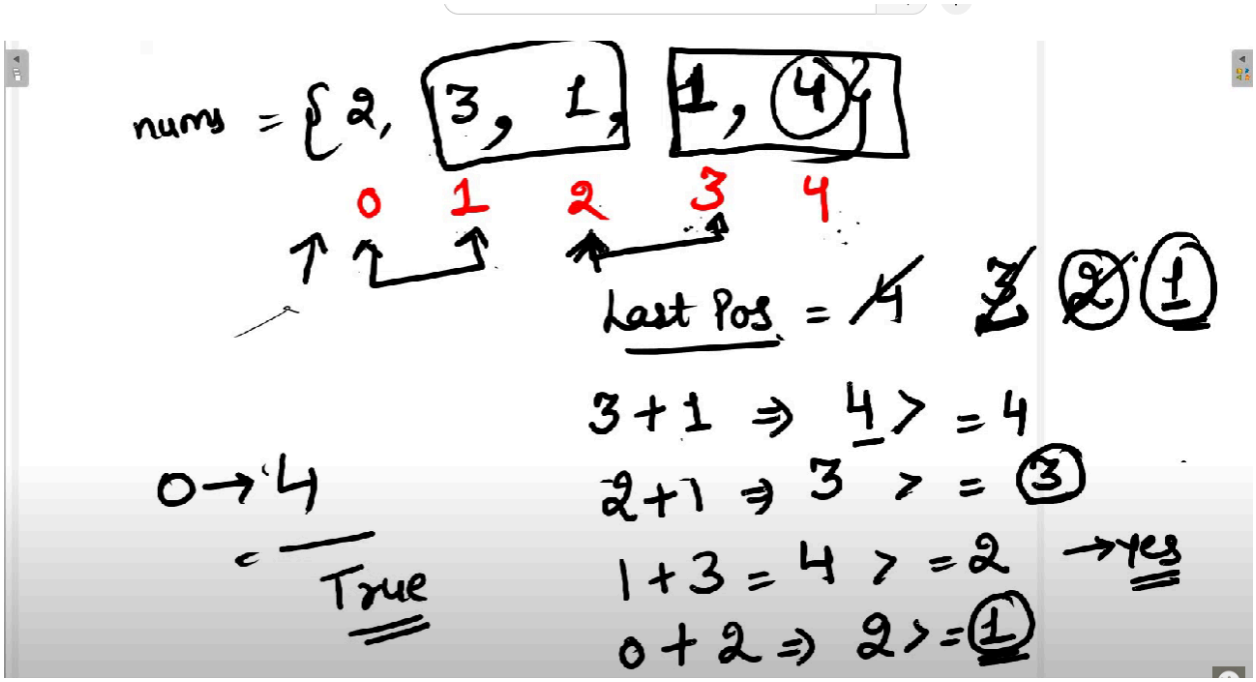
Basic understanding

Example 1: nums = [2,3,1,1,4]

- Start at index 0, max jump 2, so maxReach = 2. It means we can jump max 2 jump (means we can also jump to index 1 also)
- Move to index 1, max jump 3, so maxReach = 4 (we can reach the last index).
- Return True.

Example 2: nums = [3,2,1,0,4]
Start at index 0, max jump 3, so maxReach = 3.

- Move through indices 1, 2, 3, but nums[3] = 0 prevents further movement.
- We never reach index 4, so return False.



Reverse Greedy Approach to Jump Game

We traverse backward from the last index to determine if we can reach the first index.

Steps:

1. **Initialize lastPos to the last index** (i.e., n - 1).
2. **Start from the second last index (n-2)** and move backward.
3. **Check if we can reach lastPos from the current index:**
    - If index + nums[index] >= lastPos, update lastPos = index.
4. **Repeat this process until we reach index 0**.

Example: nums = [2,3,1,1,4]

- Start from lastPos = 4 (last index).
- **Index 3:** Can reach 4 (3 + 1 >= 4), so update lastPos = 3.
- **Index 2:** Can reach 3 (2 + 1 >= 3), so update lastPos = 2.

- **Index 1:** Can reach 2 (1 + 3 >= 2), so update lastPos = 1.
- **Index 0:** Can reach 1 (0 + 2 >= 1), so update lastPos = 0.

Since lastPos becomes 0, **we can reach the last index → return True**.

```
package com.leetcode150;
public class JumpGame {
	public static void main(String[] args) {
		int nums[] = { 2, 3, 1, 1, 4 };
		System.out.println(jumpGame(nums));
	}
	public static boolean jumpGame(int[] nums) {
		int las_Pos = nums.length - 1;
		for (int i = nums.length - 1; i >= 0; i--) {
			if (i + nums[i] >= las_Pos) {
				las_Pos = i;
			}
		}
		return las_Pos == 0;
	}
}
```

output

true

Time : O(n): We traverse the array **once** from right to left.

O(1) Space: Only a single variable (lastPos) is used.

21.Ransom Note

Given two strings ransomNote and magazine, return true *if* ransomNote *can be constructed by using the letters from* magazine *and* false *otherwise*.

Each letter in the magazine can only be used once in ransomNote.

Example 1:

Input: ransomNote = "a", magazine = "b"
Output: false

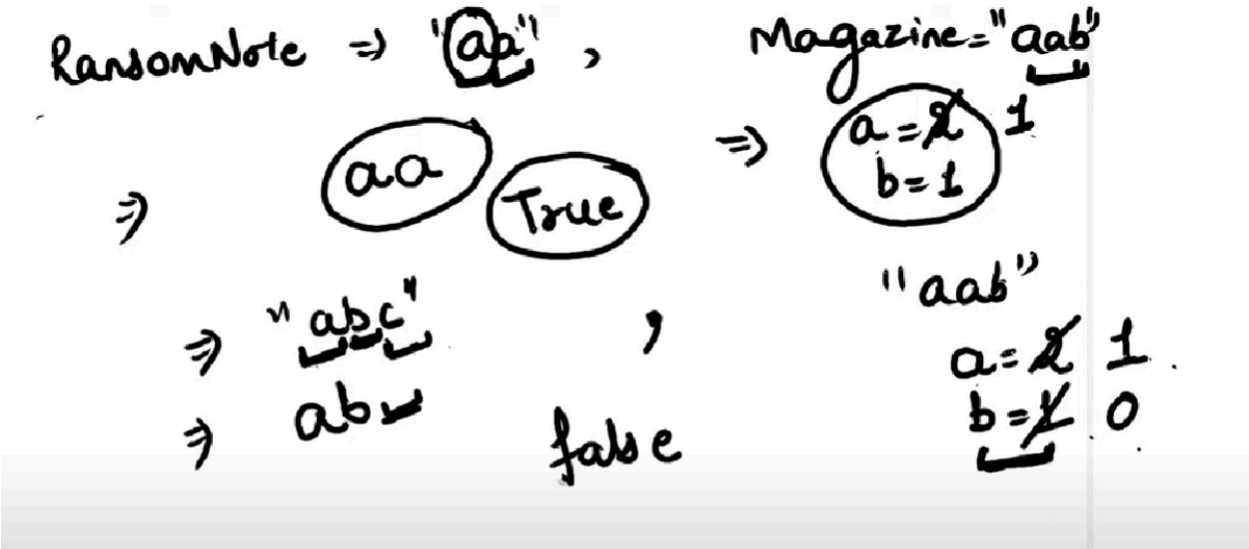
Example 2:

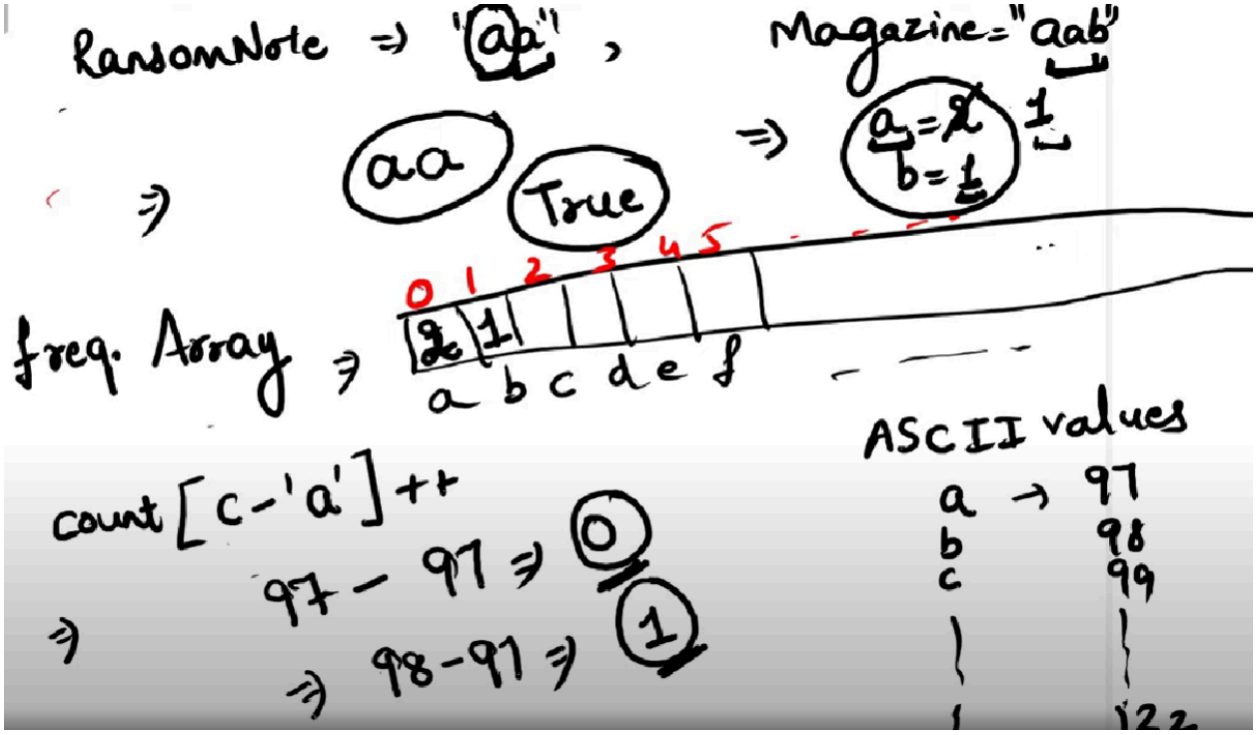
Input: ransomNote = "aa", magazine = "ab"
Output: false

Example 3:

Input: ransomNote = "aa", magazine = "aab"
Output: true

Optimized Solution





1. **Use a frequency count**
    - Count occurrences of each character in magazine.
    - Check if ransomNote can be formed using these counts.
2. **Steps:**
    - Create an array charCount[26] (since we only have lowercase letters a-z).
    - Iterate through magazine and increase the count of each character.
    - Iterate through ransomNote and decrease the count of required characters.
    - If any character count goes **below zero**, return false.

Example Walkthrough

Input:
ransomNote = "ab";
magazine = "aab";

Execution:

Step 1: Count frequencies from magazine:

a → 2
b → 1
Step 2: Check ransomNote:

a (available) → decrement count (1 left)
b (available) → decrement count (0 left)

Result: **Return true** (All characters are available).

```
package com.leetcode150;
public class RansomNote {
	public static void main(String[] args) {
		String ransomNote = "ab";
		String magazine = "aab";
		System.out.println(canConstruct(ransomNote, magazine));
	}
	public static boolean canConstruct(String ransomNote, String magazine) {
		int[] charCount = new int[26];
		for (char c : magazine.toCharArray()) {
			charCount[c - 'a']++;
		}
		for (char d : ransomNote.toCharArray()) {
			if (charCount[d - 'a'] == 0) {
				return false;
			}
			charCount[d - 'a']--;
		}
		return true;
	}
}
```

Output
true

22.Length of Last Word

Given a string s consisting of words and spaces, return *the length of the **last** word in the string.*

A **word** is a maximal substring consisting of non-space characters only.

Example 1:

Input: s = "Hello World"
Output: 5
Explanation: The last word is "World" with length 5.

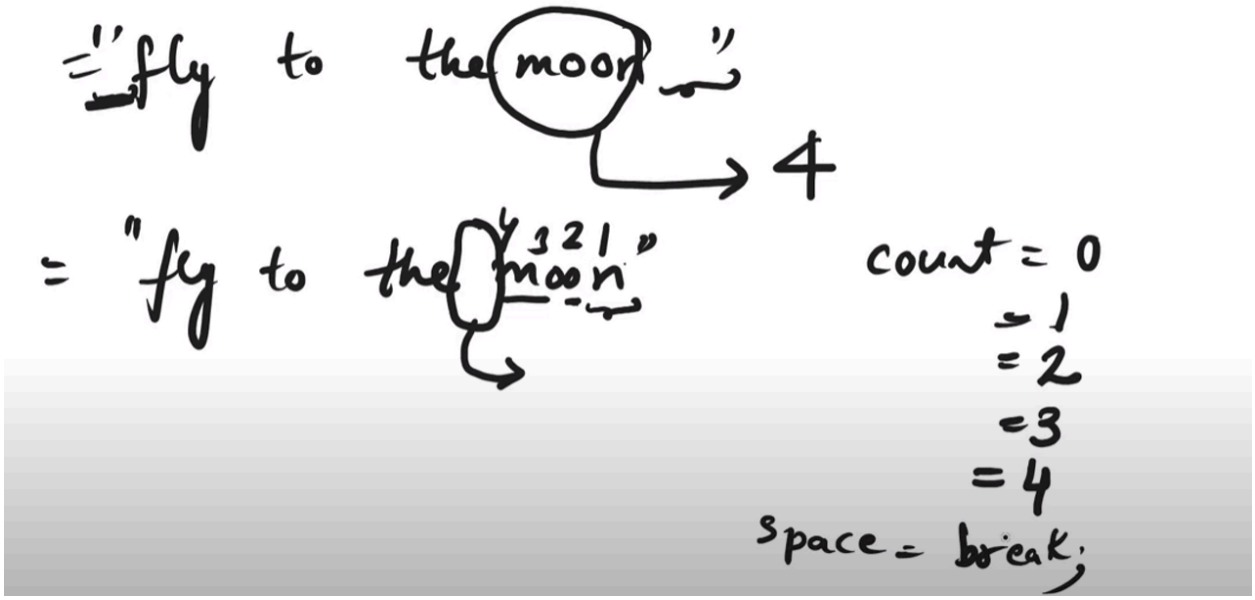
Example 2:

Input: s = " fly me to the moon "
Output: 4
Explanation: The last word is "moon" with length 4.

Example 3:

Input: s = "luffy is still joyboy"
Output: 6
Explanation: The last word is "joyboy" with length 6.

Optimized Solution



Steps to Find the Length of the Last Word in a String

1. **Remove leading and trailing spaces**

- Trim the input string to remove extra spaces from both ends.
  trim() method will remove extra spaces from both ends.

  Ex: String s = " fly me to the moon ";

  s.trim();

  String s = "fly me to the moon" // leading and trailing spaces removed.

2. **Start traversing from the last character**

   - Initialize a counter count = 0.
   - Start iterating from the end of the string.

3. **Count the length of the last word**

   - If the character is not a space, increment count.
   - If a space is encountered after counting at least one character, break the loop.

4. **Return the count**

   - The count gives the length of the last word.

```
package com.leetcode150;
public class LengthofLastWord {
	public static void main(String[] args) {
		String input = "  fly me  to  the moon  ";
		System.out.println(lengthofLastWord(input));
	}
	public static int lengthofLastWord(String input) {
		String str = input.trim();
		int count = 0;
		for (int i = str.length() - 1; i >= 0; i--) {
			if (str.charAt(i) != ' ') {
				count++;
			} else {
				break;
			}
		}
		return count;
```

```
        }
}
```

Output

4

Time Complexity: O(n) → We traverse at most once.

Space Complexity: O(1) → No extra space used.

23.Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

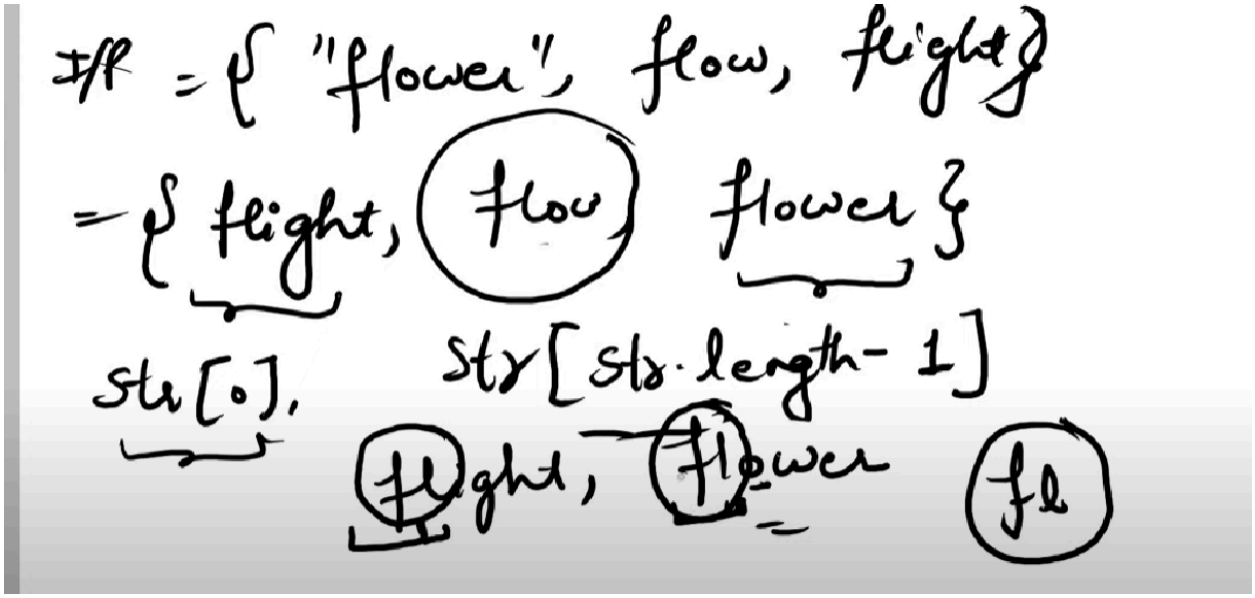
Example 1:

Input: strs = ["flower","flow","flight"]
Output: "fl"

Example 2:

Input: strs = ["dog","racecar","car"]
Output: ""
Explanation: There is no common prefix among the input strings.

Optimized Solution



Steps to Find the Longest Common Prefix

1. **Sort the array**
    - Sorting the array ensures that the smallest and largest strings (lexicographically) are at the extremes.
    - If a common prefix exists, it must be present in both the first and last strings.
2. **Compare only the first and last strings**
    - Since the array is sorted, the middle strings automatically have the same prefix range as the first and last.
3. **Character-by-character comparison**
    - Compare the characters of the first and last string.
    - Stop when characters don't match.
4. **Return the matched prefix**
    - If all characters match up to a certain point, return the prefix.

- If no characters match, return an empty string.

```
package com.leetcode150;
import java.util.Arrays;
public class LongestCommonPrefix {
	public static void main(String[] args) {
		String strs[] = { "flower", "flow", "flight" };
		System.out.println(longestCommonPrefix(strs));
	}
	public static String longestCommonPrefix(String[] input) {
		Arrays.sort(input);
		String first = input[0];
		String last = input[input.length - 1];
		int index = 0;
		while (index < first.length()) {
			if (first.charAt(index) == last.charAt(index)) {
				index++;
			} else {
				break;
			}
		}
		return index == 0 ? "" : first.substring(0, index);
	}
}
```

Output

fl

Sorting the array → O(n log n), where n is the number of strings.

Comparing characters of the first and last string → O(m), where m is the length of the shortest string.

Total Complexity = O(n log n) + O(m)

- The sorting step dominates, so the overall complexity is **O(n log n)**.

Space Complexity: O(1) (no extra space used).

24.Reverse words in a string

Given an input string s, reverse the order of the **words**.

A **word** is defined as a sequence of non-space characters. The **words** in s will be separated by at least one space.

Return *a string of the words in reverse order concatenated by a single space.*

Note that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

Example 1:

Input: s = "the sky is blue"
Output: "blue is sky the"

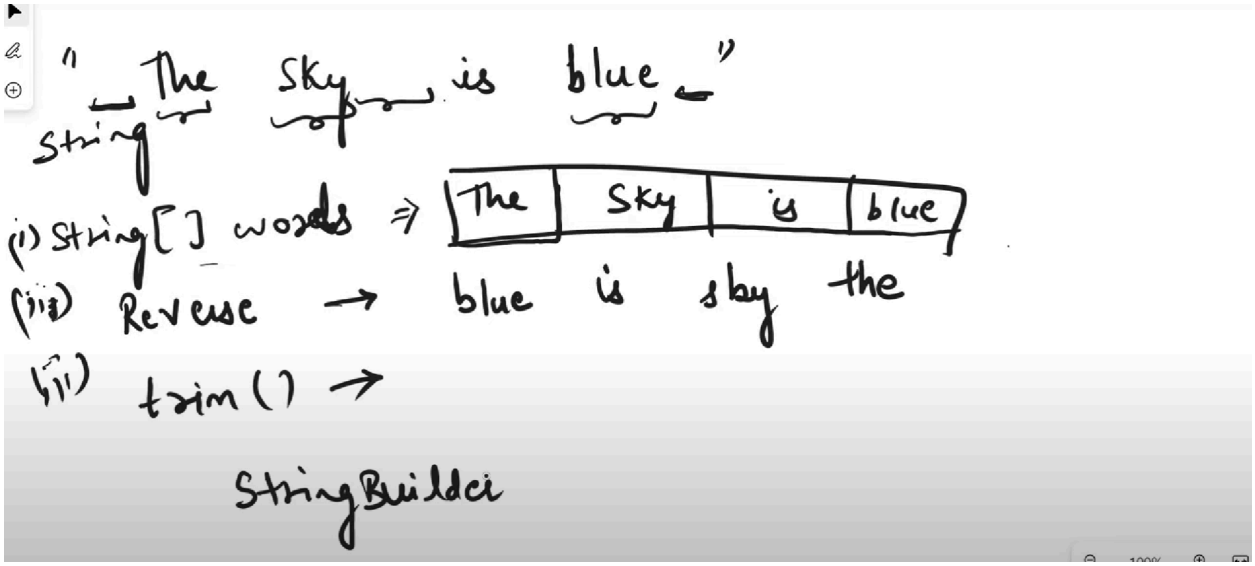
Example 2:

Input: s = " hello world "
Output: "world hello"
Explanation: Your reversed string should not contain leading or trailing spaces.

Example 3:

Input: s = "a good   example"
Output: "example good a"
Explanation: You need to reduce multiple spaces between two words to a single space in the reversed string.

Optimized Solution



Step 1: Split the String into Words

- Since there might be **multiple spaces** between words, use s.split(" +").

  " +" is a **regular expression (regex)**.

  The + **(plus) means "one or more" spaces**.

- This ensures that **all spaces** (single or multiple) are treated as **one separator**.

- Store the words in an **array**.

Step 2: Reverse the Words Order

- Use a **loop** to iterate through the array from **end to start**.

- Append each word to a **StringBuilder**.

- **Add a space** after each word, except for the **last one**.

Step 3: Convert to String and Trim

- Convert StringBuilder to **String**.

- **Trim** any leading or trailing spaces (**extra safety measure**).

```
package com.leetcode150;
```

```java
public class ReverseWordsInaString {
	public static void main(String[] args) {
		String s = "the sky is blue";
		System.out.println(reverseWordsInaString(s));
	}
	public static String reverseWordsInaString(String input) {
		String[] words = input.split(" +");
		StringBuilder sb = new StringBuilder();
		for (int i = words.length - 1; i >= 0; i--) {
			sb.append(words[i]);
			sb.append(" ");
		}
		return sb.toString().trim();
	}
}
```

Output

blue is sky the

Time Complexity: O(n), since we **iterate the string once**.
Space Complexity: O(n), as we **store words in an array** and build a new string.

25.Product of Array Except Self

Given an integer array **nums**, return *an **array*** answer *such that* answer[i] *is equal to the **product of all the elements** of* nums ***except*** **nums[i]**.

The product of any prefix or suffix of nums is **guaranteed** to fit in a **32-bit** integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

Example 1:

Input: nums = [1,2,3,4]
Output: [24,12,8,6]

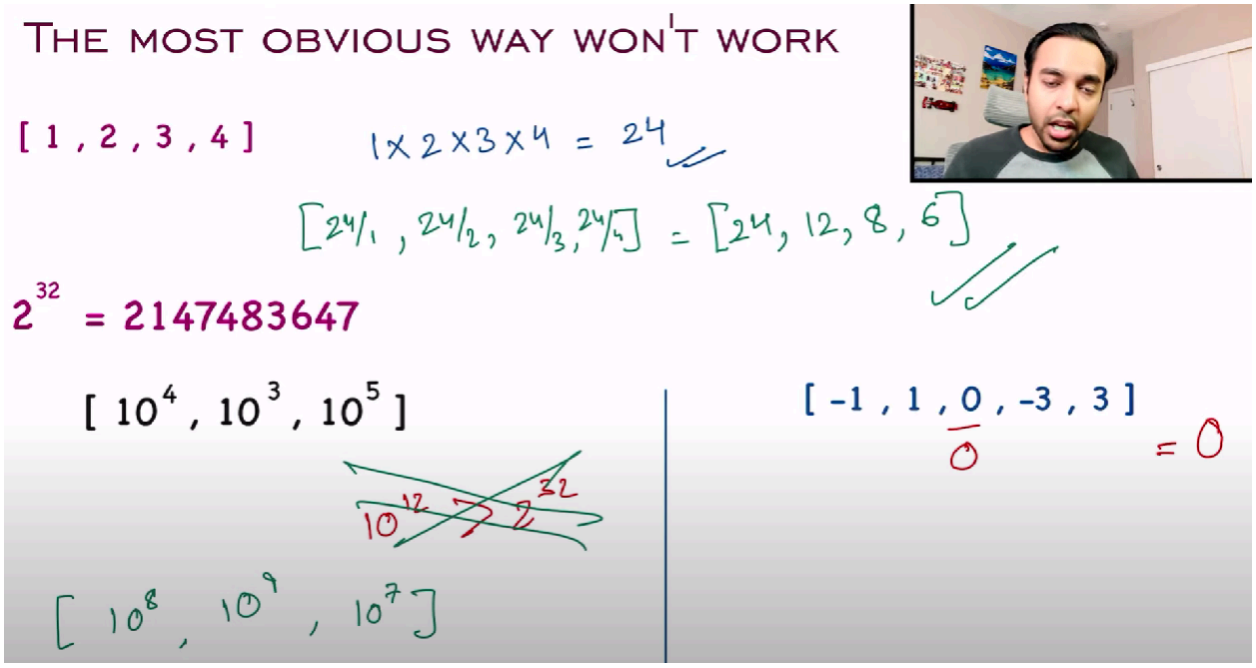
Example 2:

Input: nums = [-1,1,0,-3,3]

Output: [0,0,9,0,0]

General solution X

We don't use this general approach as it has few disadvantages.



The Division Approach (which seems soooo easy at first):

You take the **product of the entire array** and then, for each index, divide that total by the value at that index.

Example from the image:

[1, 2, 3, 4] → total = 1×2×3×4 = 24
Then output = [24/1, 24/2, 24/3, 24/4] = [24, 12, 8, 6]

So elegant, so clean. Almost makes you feel clever.

But here comes the villain: Zero.

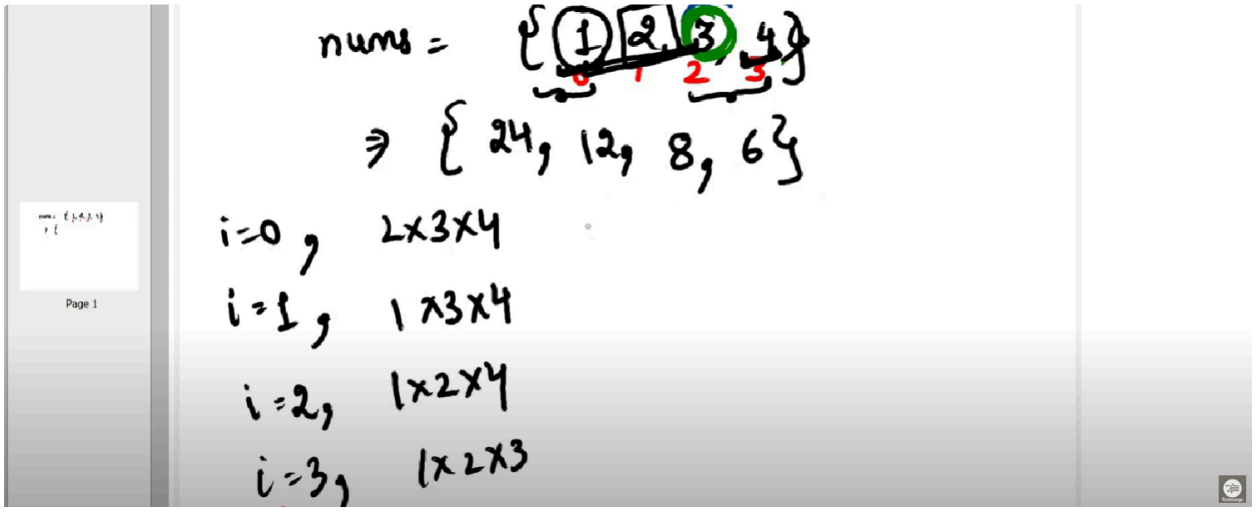
When the input array contains a zero—like in [-1, 1, 0, -3, 3]—division suddenly becomes an **existential crisis**.

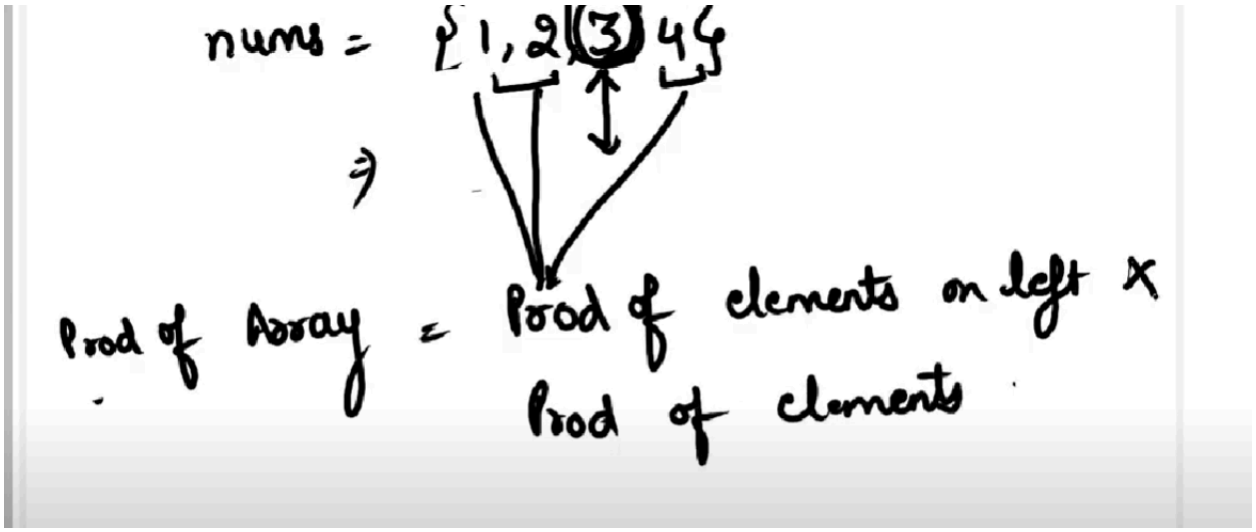
Here's what happens:

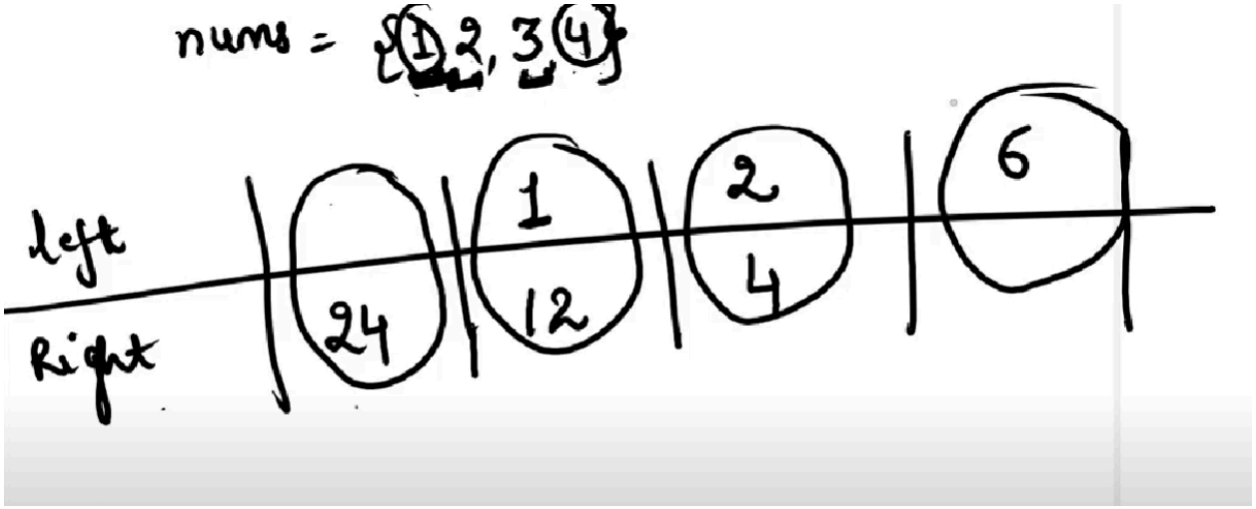
- Total product = 0 (because 0 × anything = 0)
- Now you try to divide by the elements:
    - 0 / -1 = 0
    - 0 / 1 = 0
    - 0 / 0 = 🎉 Undefined! Let's break the universe.

You can't divide by zero unless you're in a philosophy class.

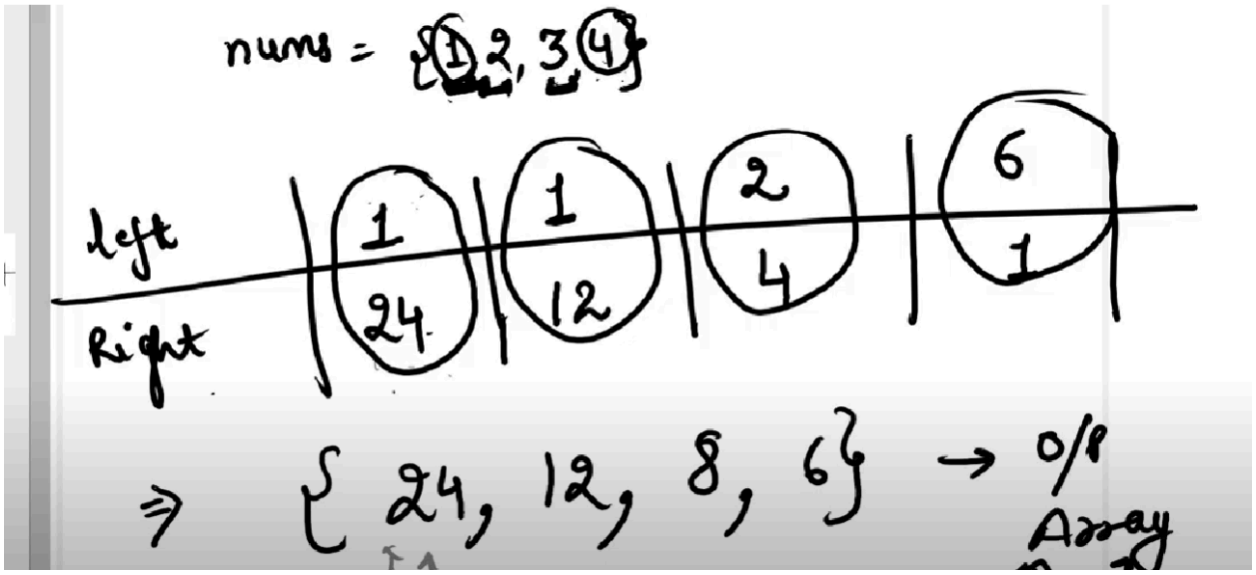
Optimized Solution







If there is no element on the right/left side element will take it as 1 as , product of any element with 1 is that element only.



We use two auxiliary arrays:

1. **Left product array**: left[i] is the product of all elements to the **left** of index i.
2. **Right product array**: right[i] is the product of all elements to the **right** of index i.

Then:
output[i] = left[i] * right[i]

Example: nums = [1, 2, 3, 4]

Step 1: Left Product

- left[0] = 1 (no elements to the left)
- left[1] = nums[0] = 1
- left[2] = nums[0] * nums[1] = 1 * 2 = 2
- left[3] = nums[0] * nums[1] * nums[2] = 1 * 2 * 3 = 6

So, left = [1, 1, 2, 6]

Step 2: Right Product

- right[3] = 1 (no elements to the right)
- right[2] = nums[3] = 4
- right[1] = nums[2] * nums[3] = 3 * 4 = 12
- right[0] = nums[1] * nums[2] * nums[3] = 2 * 3 * 4 = 24

So, right = [24, 12, 4, 1]

Step 3: Final Output

- output[0] = left[0] * right[0] = 1 * 24 = 24
- output[1] = 1 * 12 = 12
- output[2] = 2 * 4 = 8
- output[3] = 6 * 1 = 6

Final result: **[24, 12, 8, 6]**

```
package com.leetcode150;
import java.util.Arrays;
public class ProductofArrayExceptSelf {
	public static void main(String[] args) {
```

```
        int nums[] = { 1, 2, 3 };
        System.out.println(Arrays.toString(productofArrayExceptSelf(nums)));
    }
    public static int[] productofArrayExceptSelf(int[] nums) {
        int len = nums.length;
        int ans[] = new int[len];
        int ProductRight = 1;
        int ProductLeft = 1;
        for (int i = len - 1; i >= 0; i--) {
            ans[i] = ProductRight;
            ProductRight = ProductRight * nums[i];
        }
        for (int i = 0; i < len; i++) {
            ans[i] = ans[i] * ProductLeft;
            ProductLeft = ProductLeft * nums[i];
        }
        return ans;
    }
}
```

Output

[6, 3, 2]

Time Complexity:

O(n)

- First loop: Builds right products → O(n)
- Second loop: Multiplies by left products → O(n)
  ✅ Total: **O(n)**

26.Merge Intervals

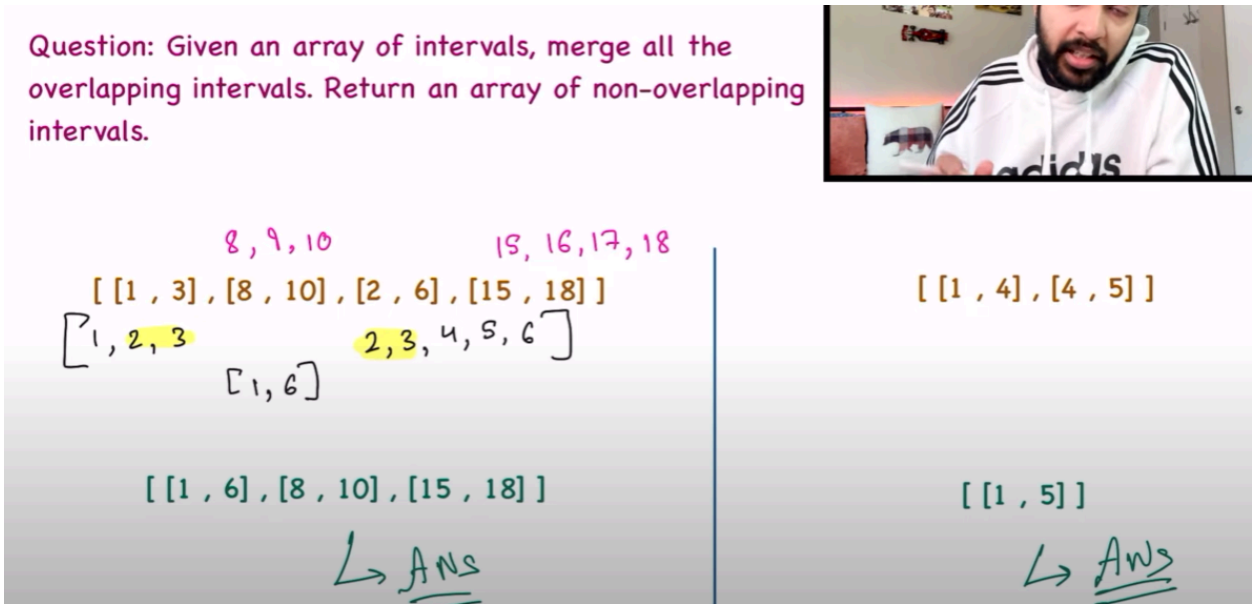
Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

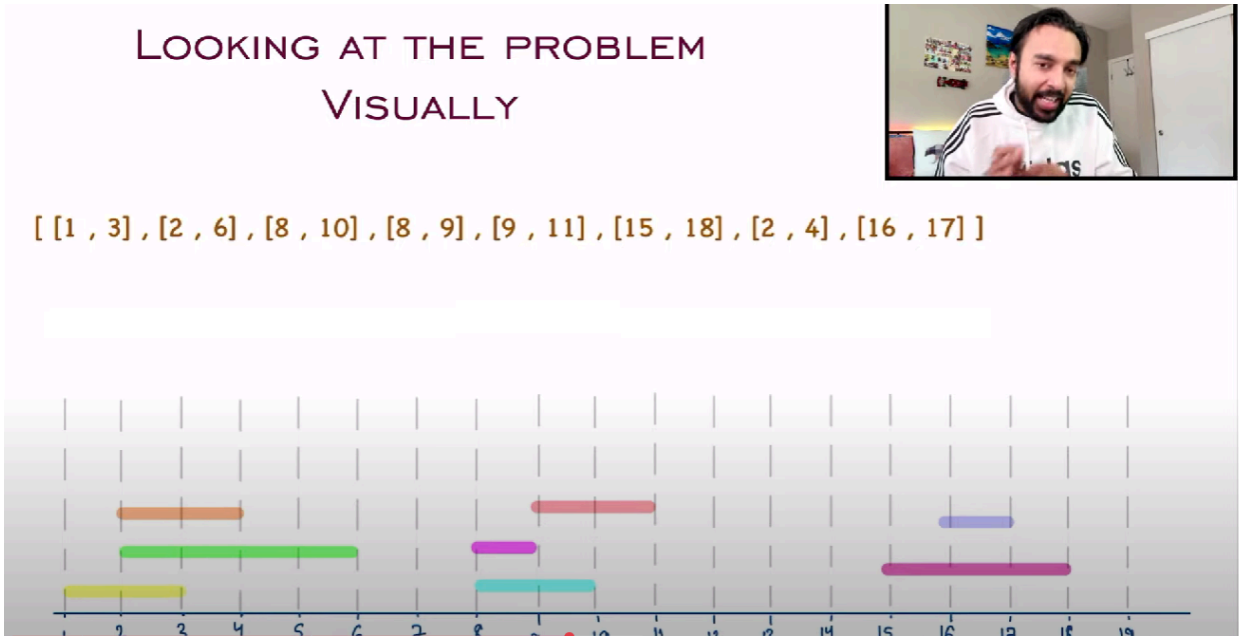
Example 1:

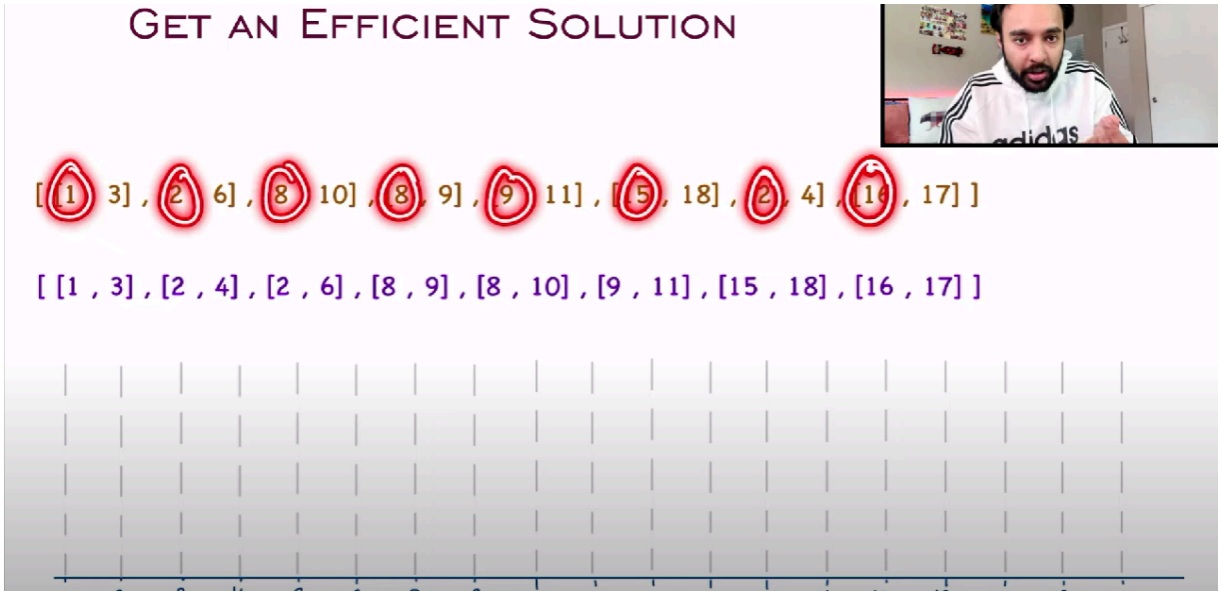
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

Example 2:

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





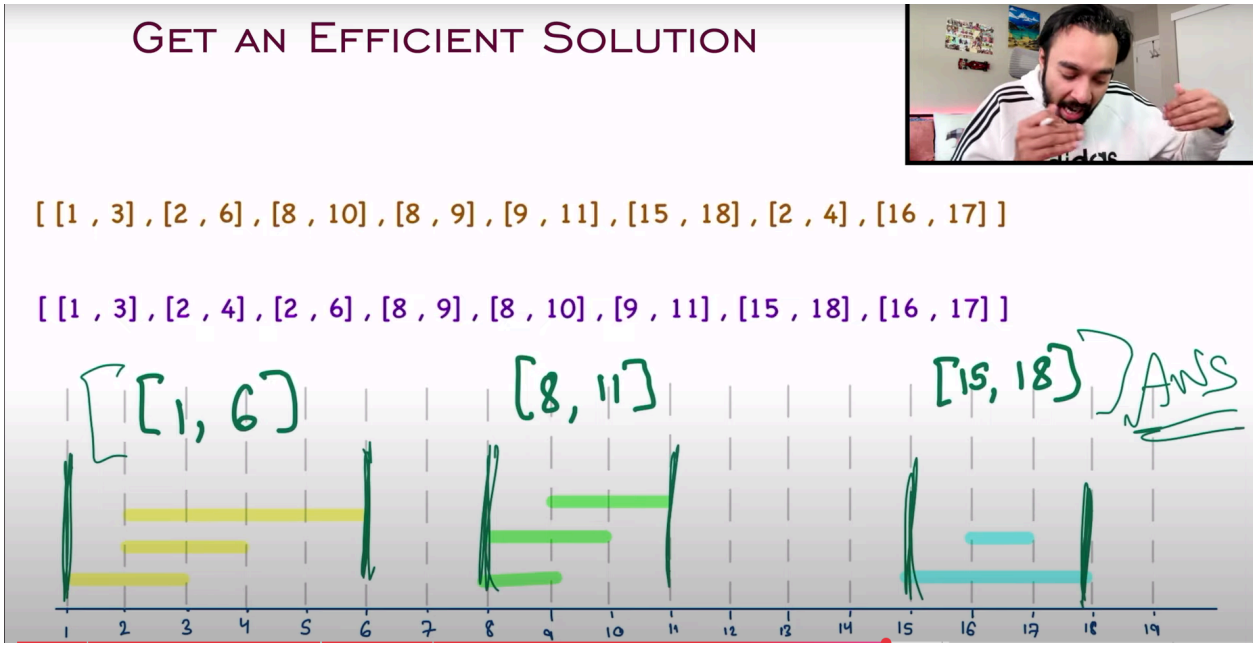


Sort intervals by start time.

Traverse and compare:

- If current start > last merged end → new interval
- Else → merge by updating end

Return merged list



Time Complexity:

- **O(n log n)** — for sorting the intervals
- **O(n)** — for traversing once to merge
- **Total: O(n log n)**

Space Complexity:

- **O(n)** — in worst case, if no merges happen, output list stores all interval

```
package com.leetcode150;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
public class MergeIntervals {
	public static void main(String[] args) {
		int[][] intervals = { { 1, 3 }, { 2, 6 }, { 8, 10 }, { 15, 18 } };
		// int[][] intervals = { {1,4},{4,5} };
		System.out.println(Arrays.deepToString(merge(intervals)));
```

```
		/*
		 * int[][] result = merge(intervals); for (int[] arr : result) {
		 * System.out.println(Arrays.toString(arr)); }
		 */
	}
	public static int[][] merge(int[][] intervals) {
		if (intervals.length <= 1)
			return intervals;
		// Step 1: Sort intervals by starting value
		Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
		List<int[]> merged = new ArrayList<>();
		int[] current = intervals[0];
		merged.add(current);
		for (int[] interval : intervals) {
			int currStart = current[0];
			int currEnd = current[1];
			int nextStart = interval[0];
			int nextEnd = interval[1];
			if (currEnd >= nextStart) {
				// Overlapping intervals: merge them
				current[1] = Math.max(currEnd, nextEnd);
			} else {
				// Non-overlapping interval: move to next
				current = interval;
				merged.add(current);
			}
		}
		return merged.toArray(new int[merged.size()][]);
	}
}
```

Output

[[1, 6], [8, 10], [15, 18]]

Ex:

int[][] intervals = { {1, 3}, {8, 10}, {2, 6}, {15, 18} };

Step 1: Sort intervals by starting value

After sorting by interval[0] (start of each interval), we get

[[1, 3], [2, 6], [8, 10], [15, 18]]

Step 2: Start with first interval

We initialize:

current = [1, 3]

merged = [[1, 3]]

Step 3: Iterate through the rest

1. interval = [2, 6]
    - current end (3) ≥ next start (2) → **overlap**
    - Merge: current[1] = max(3, 6) = 6

      intervals[0] = [1, 3]

      current = intervals[0]

      merged = [[1, 3]] ← current points here too
      Then, overlap detected with [2, 6], so you do:

      current[1] = Math.max(3, 6); // becomes [1, 6]
    - Now current = [1, 6], merged = [[1, 6]]
2. interval = [8, 10]
    - current end (6) < next start (8) → **no overlap**
    - Move to new interval: current = [8, 10]

- merged = [[1, 6], [8, 10]]

3. interval = [15, 18]
    - current end (10) < next start (15) → **no overlap**
    - Move to new interval: current = [15, 18]
    - merged = [[1, 6], [8, 10], [15, 18]]

Final Output:

[[1, 6], [8, 10], [15, 18]]

merged is our result list. This list will store all the merged intervals — our final answer.

int[] current = intervals[0]; We assume the first interval (after sorting) is the one we'll start comparing others with. We name it current.

merged.add(current); We immediately add the current interval to the result list *because*:

- We *will* either merge the upcoming intervals into this one,
- Or we'll determine that the upcoming interval does *not* overlap and needs to be added separately.

Let's walk through the first two intervals:

Imagine the input is: [[1,3], [2,6], [8,10], [15,18]]

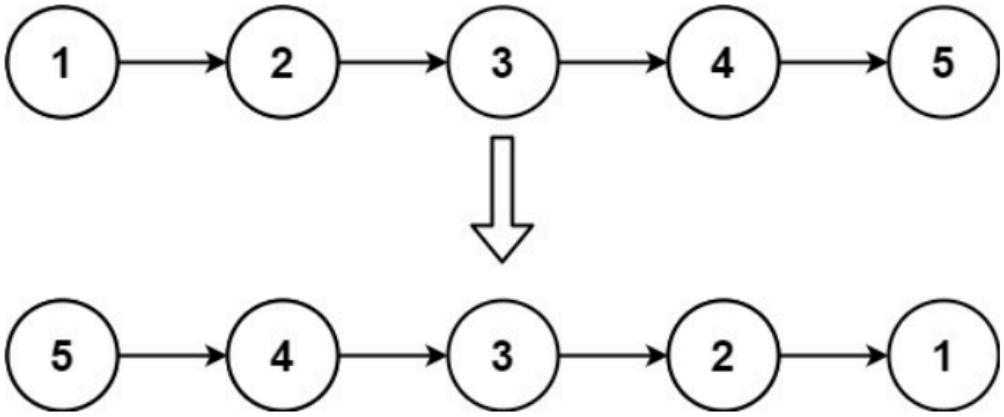
1. Start: current = [1,3] → added to merged
2. Next interval: [2,6]
    - Since current[1] (3) >= 2, they **overlap**
    - Merge by updating: current[1] = max(3,6) → becomes [1,6]
    - Because current is already merged, this update reflects in the result!

LinkedLists

27.Reverse Linked List

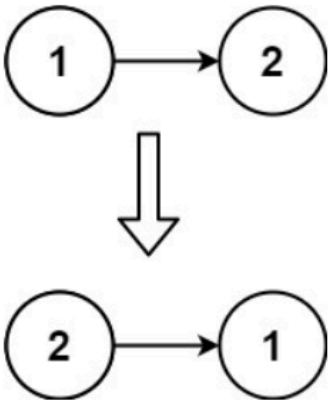
Given the head of a singly linked list, reverse the list, and return *the reversed list*.

Example 1:




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

Example 2:




```
Input: head = [1,2]
Output: [2,1]
```

Example 3:

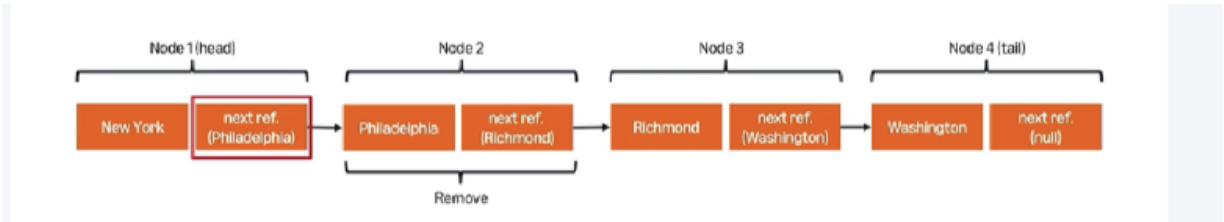
```
Input: head = []
Output: []
```

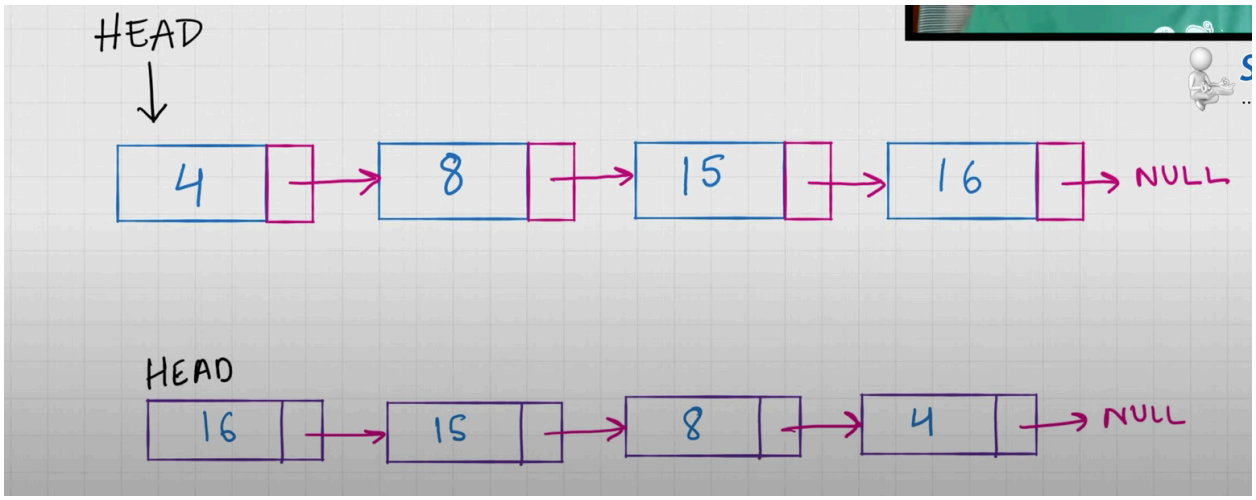
What is a Linked List?

A **Linked List** is a linear data structure where elements (nodes) are connected using **pointers**. Each node contains:

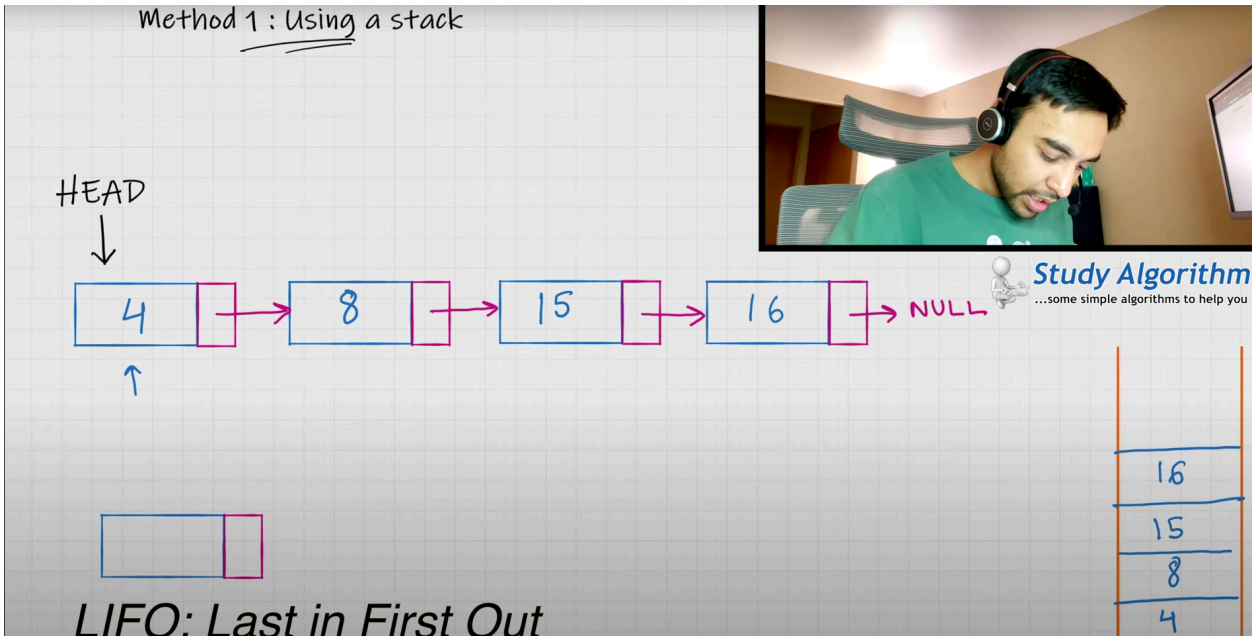
1. **Data** – Stores the value.
2. **Next** – A pointer/reference to the next node in the list.

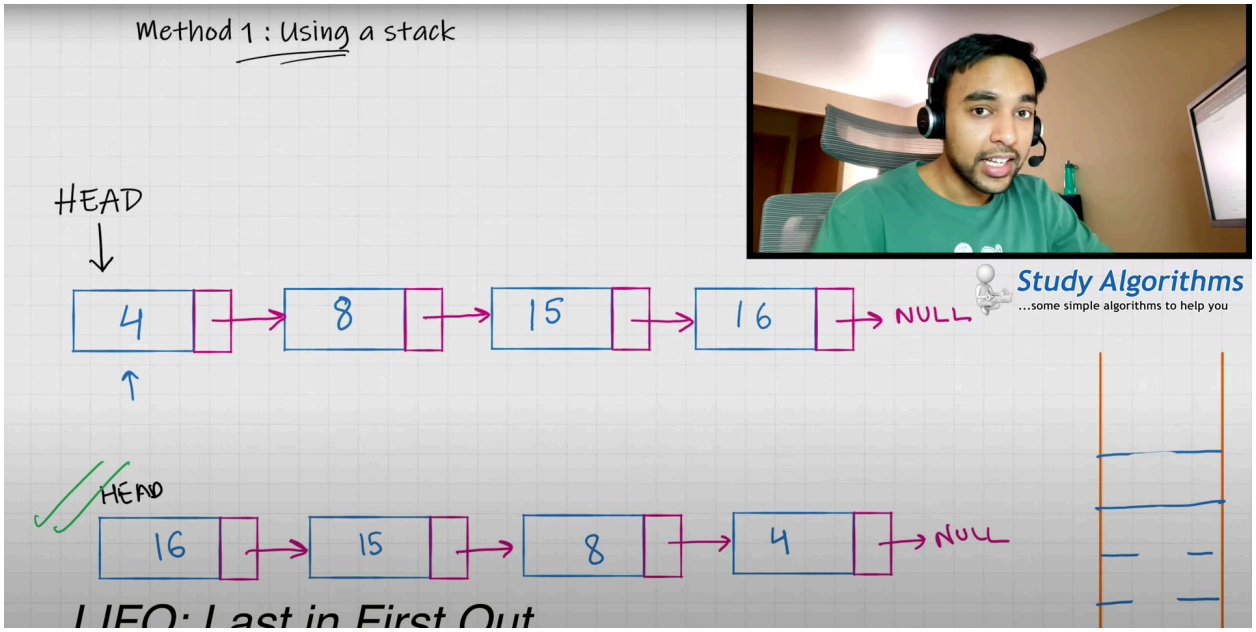
Unlike arrays, linked lists **do not require contiguous memory** allocation, making insertions and deletions more efficient.



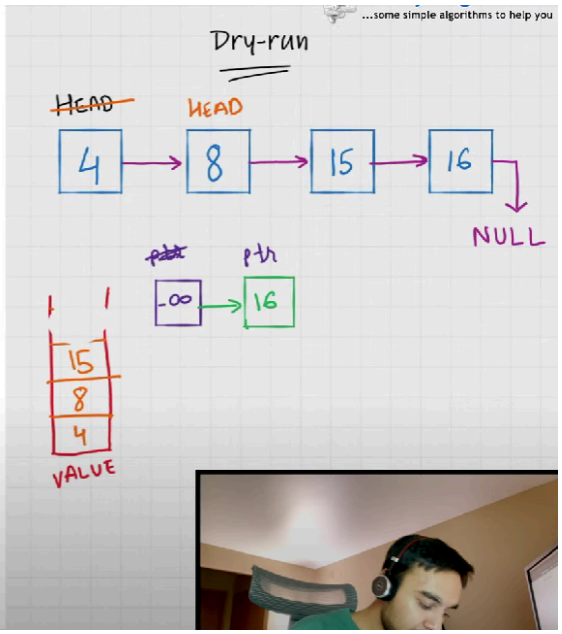


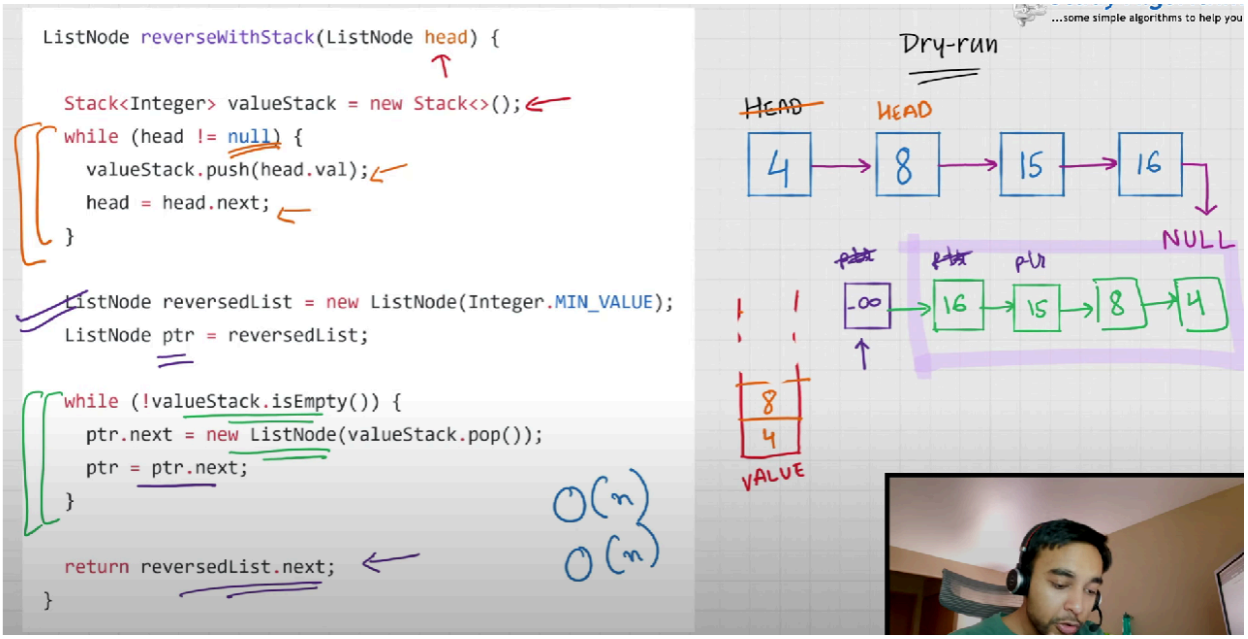
Using stacks






```
ListNode reverseWithStack(ListNode head) {

  Stack<Integer> valueStack = new Stack<>();
  while (head != null) {
    valueStack.push(head.val);
    head = head.next;
  }

  ListNode reversedList = new ListNode(Integer.MIN_VALUE);
  ListNode ptr = reversedList;

  while (!valueStack.isEmpty()) {
    ptr.next = new ListNode(valueStack.pop());
    ptr = ptr.next;
  }

  return reversedList.next;
}
```






```
package com.leetcode150;
import java.util.Stack;
class ListNode {
	int val;
	ListNode next;
	ListNode() {
	}
	ListNode(int val) {
		this.val = val;
		this.next = null;
	}
	ListNode(int val, ListNode next) {
		this.val = val;
		this.next = next;
	}
}
public class ReverseLinkedList {
	public static void main(String[] args) {
		ListNode head = new ListNode(16);
		head.next = new ListNode(15);
		head.next.next = new ListNode(8);
		head.next.next.next = new ListNode(4);
		// Optional: print original list
		System.out.print("Original List: ");
```

```
        printList(head);
        // Print reversed list
        // Reverse the list
        ListNode reversedHead = reverseList(head);
        System.out.print("Reversed List: ");
        printList(reversedHead);
    }
    // Utility method to print linked list
    public static void printList(ListNode head) {
        while (head != null) {
            System.out.print(head.val + " ");
            head = head.next;
        }
        System.out.println();
    }
    // Method to reverse linked list
    public static ListNode reverseList(ListNode head) {
        Stack<Integer> valueStack = new Stack<>();
        // Step 1: Push all node values into the stack
        while (head != null) {
            valueStack.push(head.val);
            head = head.next;
        }
        // Step 2: Use a dummy node to build reversed list
        ListNode reversedList = new ListNode(Integer.MIN_VALUE);
        ListNode ptr = reversedList;
        while (!valueStack.isEmpty()) {
            ptr.next = new ListNode(valueStack.pop());
            ptr = ptr.next;
        }
        // Step 3: Return the head of the reversed list (skip dummy)
        return reversedList.next;
    }
}
```

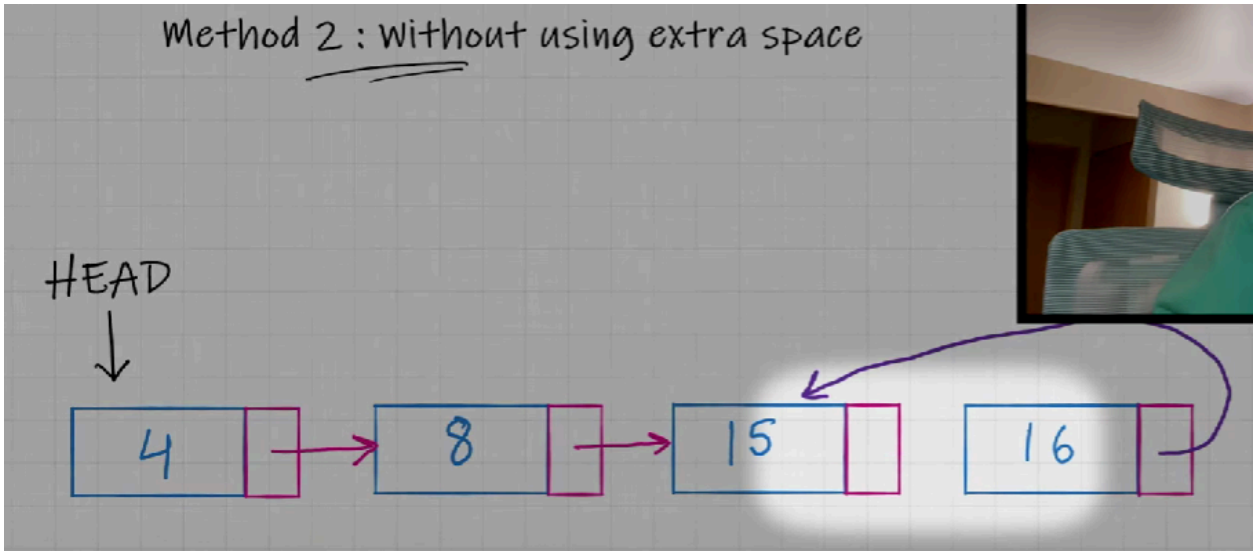
Time: O(n) – One pass to push, another to pop.

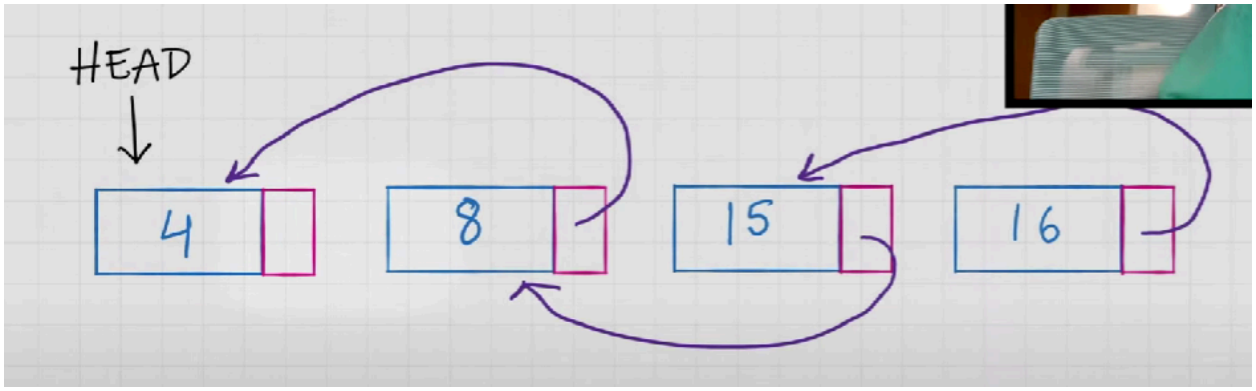
Space: O(n) – Extra space for the stack. A stack is used to store n values → **O(n)** extra space

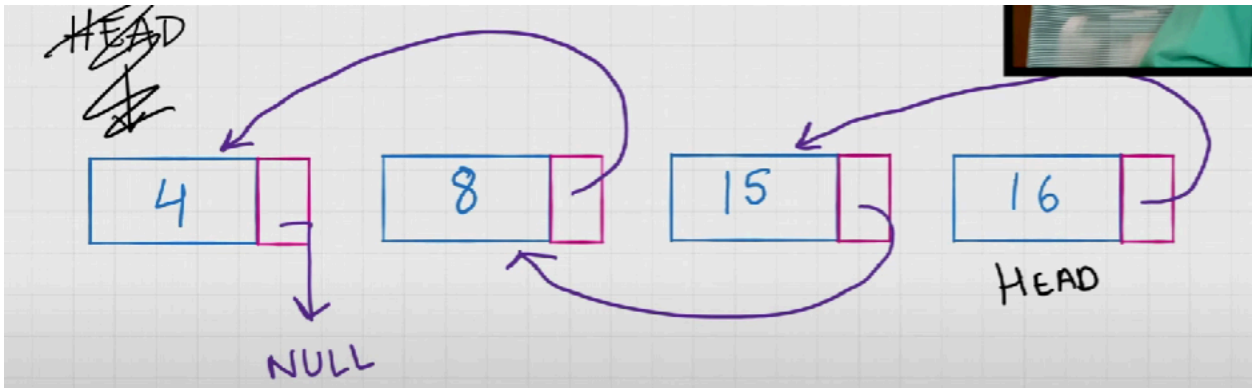
Optimized solution

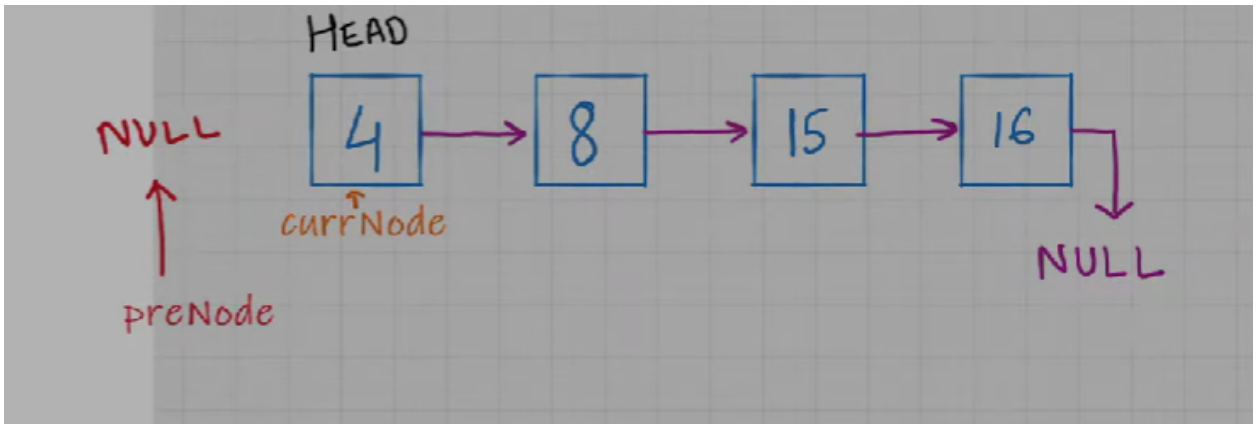
This way uses space complexity is O(1) only

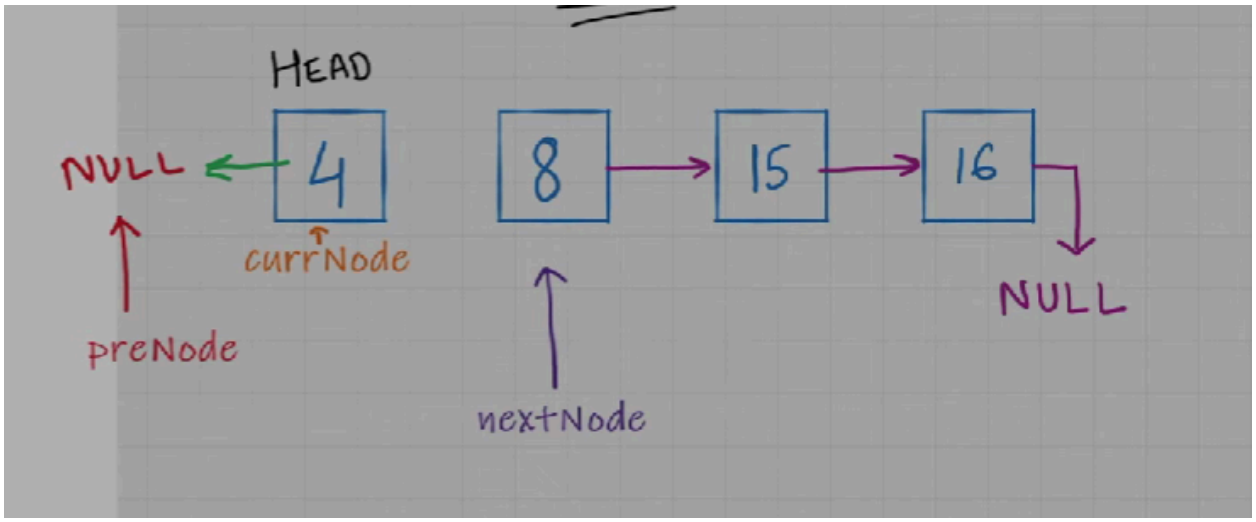
Without using extra space:

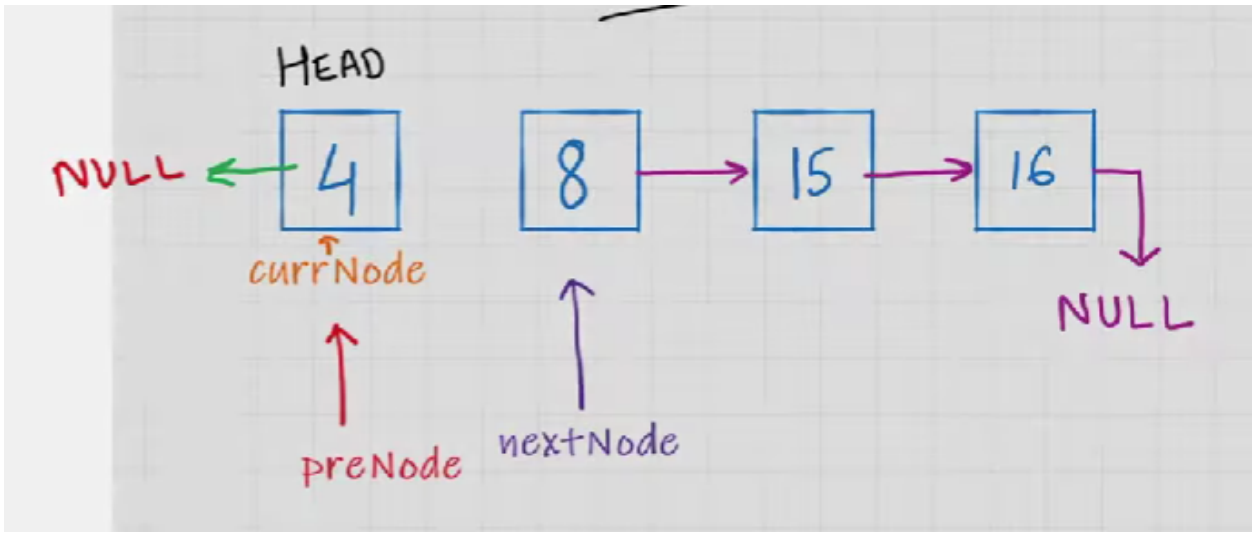


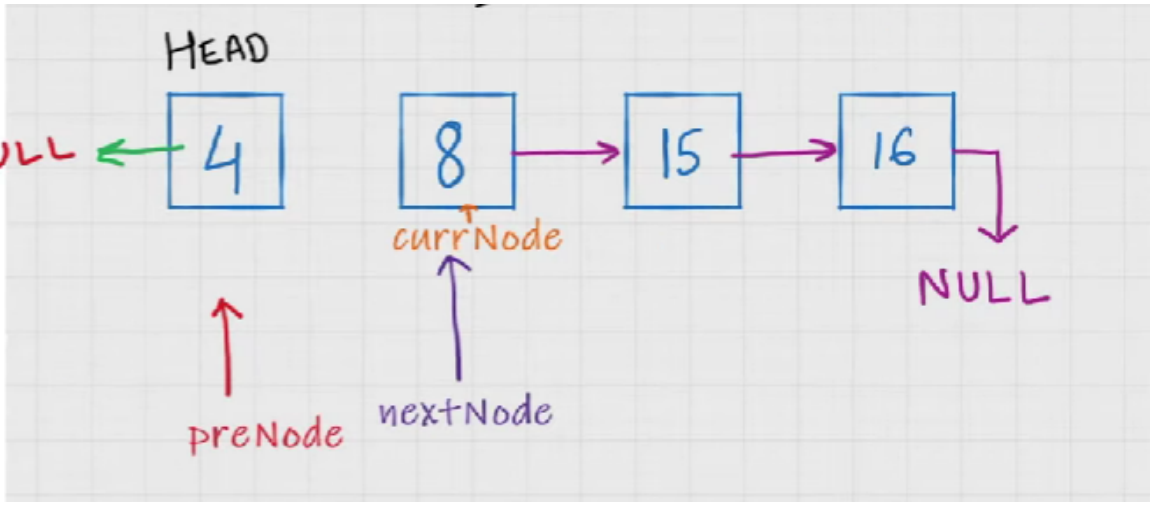





HEAD
NULL
4
8
15
16
currNode
NULL
preNode

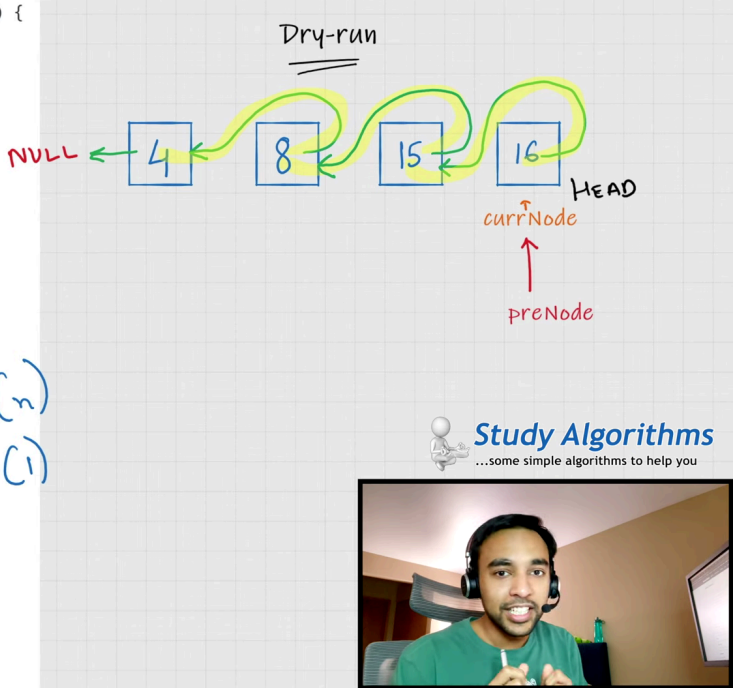

HEAD
NULL
4
8
15
16
currNode
NULL
preNode
nextNode


HEAD
NULL
4
8
15
16
currNode
NULL
preNode
nextNode




```
ListNode reverseWithoutExtraSpace(ListNode head) {
  if (head == null) {
    return null;
  }

  if (head.next == null) {
    return head;
  }

  ListNode preNode = null;
  ListNode currNode = head;

  while (currNode != null) {
    ListNode nextNode = currNode.next;
    currNode.next = preNode;
    preNode = currNode;
    currNode = nextNode;
  }

  head = preNode;

  return head;
}
```

O(n)
O(1)




```
package com.leetcode150;
class ListNode {
        int val;
        ListNode next;
        ListNode() {
        }
        ListNode(int val) {
                this.val = val;
                this.next = null;
        }
```

```
        ListNode(int val, ListNode next) {
                this.val = val;
                this.next = next;
        }
}
public class ReverseLinkedList {
        public static void main(String[] args) {
                ListNode head = new ListNode(16);
                head.next = new ListNode(15);
                head.next.next = new ListNode(8);
                head.next.next.next = new ListNode(4);
                // Optional: print original list
                System.out.print("Original List: ");
                printList(head);
                // Print reversed list
                // Reverse the list
                ListNode reversedHead = reverseList(head);
                System.out.print("Reversed List: ");
                printList(reversedHead);
        }
        // Utility method to print linked list
        public static void printList(ListNode head) {
                while (head != null) {
                        System.out.print(head.val + " ");
                        head = head.next;
                }
                System.out.println();
        }
        // Method to reverse linked list
        public static ListNode reverseList(ListNode head) {
                if (head == null) {
                        return null;
                }
                if (head.next == null) {
                        return head;
                }
                ListNode preNode = null;
                ListNode currNode = head;
                while (currNode != null) {
                        ListNode nextNode = currNode.next;
```

```
                currNode.next = preNode;
                preNode = currNode;
                currNode = nextNode;
            }
            head = preNode;
            return head;
        }
}
```

Time Complexity: O(n)

- The loop runs once for each node in the list.
- Each node is visited and its .next pointer is reversed exactly once.
- So for a list of n nodes, time complexity is **O(n)**.

  Space Complexity: O(1)

- No extra space or data structures (like stack or recursion) are used.

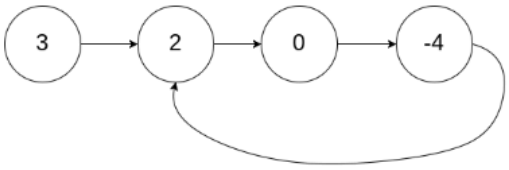
28.Linked List Cycle or Cycle Detection in Linked List

Given head, the head of a linked list, determines if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter**.

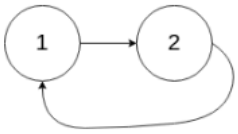
Return true *if there is a cycle in the linked list*. Otherwise, return false.

Example 1:




```
Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the
1st node (0-indexed).
```

Example 2:



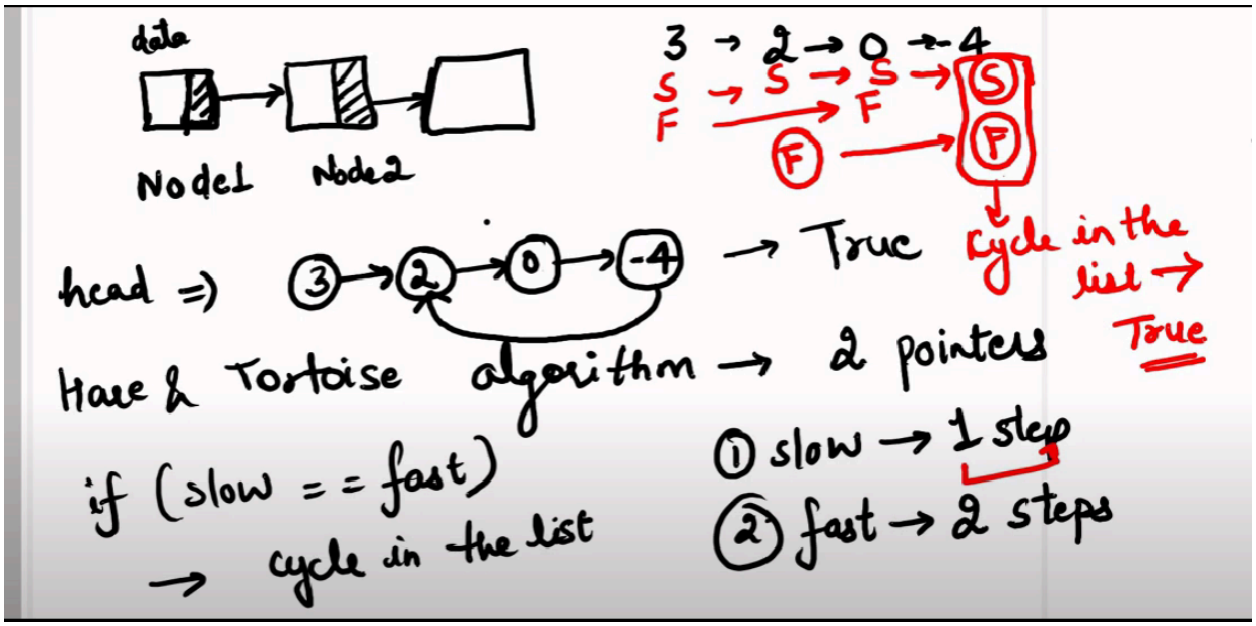

```
Input: head = [1,2], pos = 0
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the
0th node.
```

Example 3:


```
Input: head = [1], pos = -1
Output: false
Explanation: There is no cycle in the linked list.
```

Floyd's Cycle Detection Algorithm (Tortoise and Hare)

- Use **two pointers**:
    - slow: moves **1 step** at a time.
    - fast: moves **2 steps** at a time.
- If there is **no cycle**, fast will reach the end (null).
- If there **is a cycle**, slow and fast will eventually meet inside the loop.



In two conditions we can break the loop , where fast=null or fast.next = null


```
public class Solution {

    public boolean hasCycle(ListNode head) {

        ListNode slow = head;

        ListNode fast = head;

        while (fast != null && fast.next != null) {

            fast = fast.next.next;

            slow = slow.next;
```

```
            if (slow == fast) {
                return true;
            }
        }
        return false;
    }
}
```

Or

```
package com.leetcode150;
class Node {
        int val;
        Node next;
        Node() {
        }
        Node(int val) {
                this.val = val;
                this.next = null;
        }
        Node(int val, Node next) {
                this.val = val;
                this.next = next;
        }
}
public class LinkedListCycle {
        public static void main(String[] args) {
                // Create nodes
                Node head = new Node(3);
                Node node2 = new Node(2);
                Node node3 = new Node(0);
                Node node4 = new Node(-4);
                // Link nodes
                head.next = node2;
                node2.next = node3;
                node3.next = node4;
                node4.next = node2; // Creates a cycle back to node2 (pos = 1)
```

```
                // Call the static method directly
                System.out.println(hasCycle(head)); // Output: true
        }
        public static boolean hasCycle(Node head) {
                Node slow = head;
                Node fast = head;
                while (fast != null && fast.next != null) {
                        fast = fast.next.next;
                        slow = slow.next;
                        if (slow == fast) {
                                return true;
                        }
                }
                return false;
        }
}
```

Complexity	Value
Time Complexity	O(n)
Space Complexity	O(1)

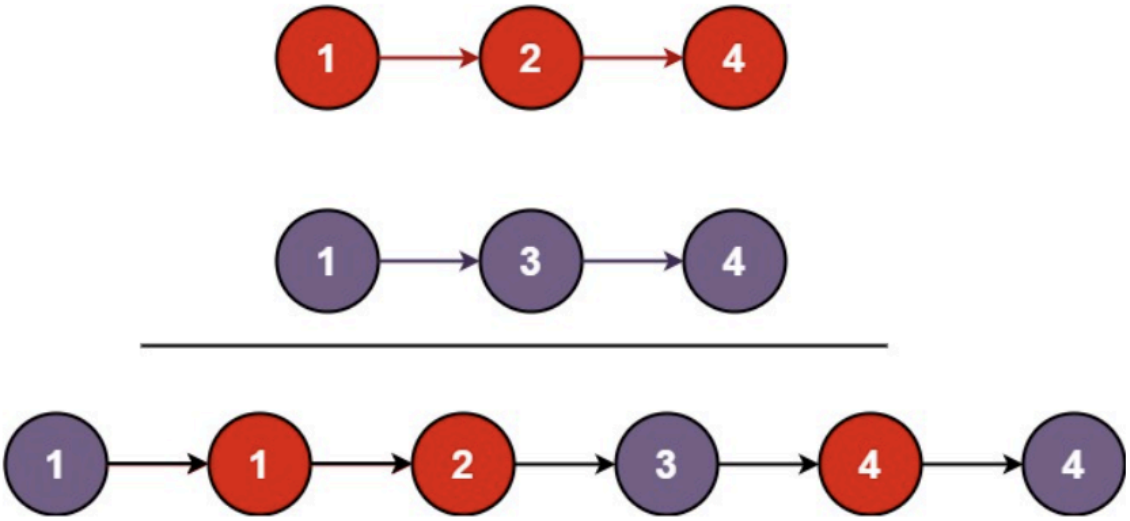
29.Merge Two Sorted Lists

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one **sorted** list. The list should be made by splicing(not creating new nodes, but rearranging the existing ones.) together the nodes of the first two lists.

Return *the head of the merged linked list*.

Example 1:



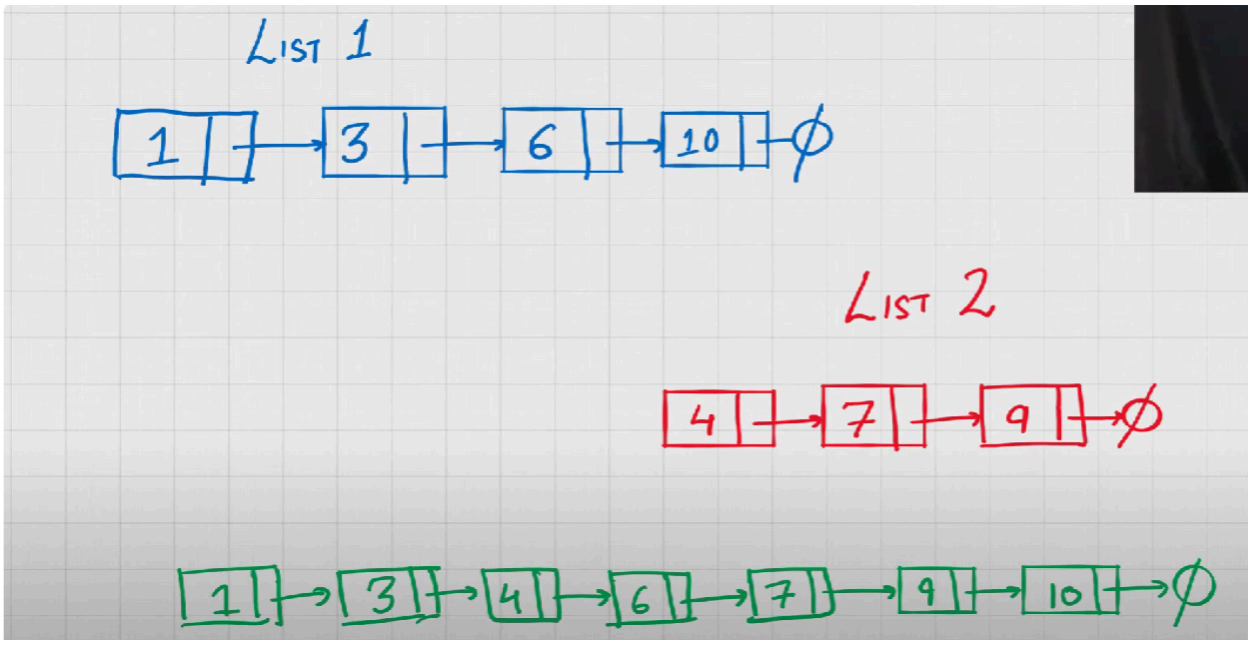

```
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]
```

Example 2:

```
Input: list1 = [], list2 = []
Output: []
```

Example 3:

```
Input: list1 = [], list2 = [0]
Output: [0]
```

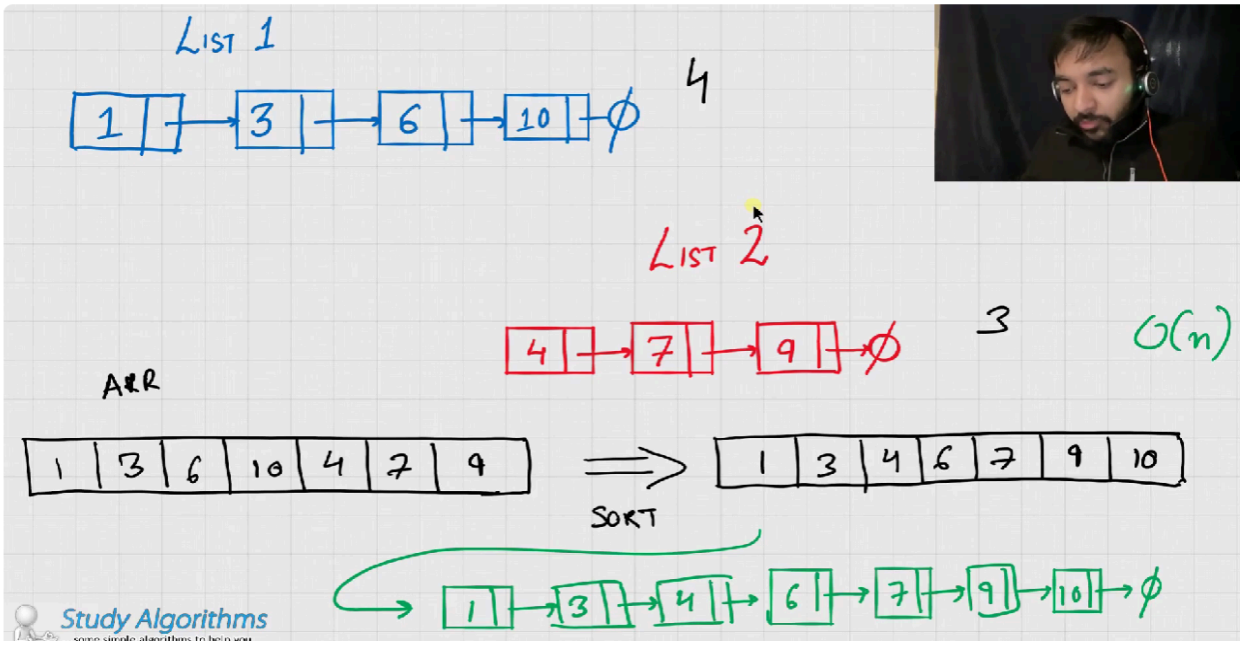


General solutions

- Convert both linked lists into an array.
- Sort the array.
- Build a new linked list from the sorted array.

This method is simple but **not optimal in terms of space**, because:

- It uses **O(n)** extra space (where n = total number of nodes in both lists).
- But it can be easier to implement, especially if optimal performance isn't required.



The code follows this flow:

1. Traverse both linked lists (list1 and list2).
2. Collect all node values into a List<Integer>.
3. Sort the list using Collections.sort().
4. Build a new sorted linked list from the sorted values.
5. Return the actual head of the result list using dummy.next.

   ListNode dummy = new ListNode(Integer.MIN_VALUE); //Use **any value** for dummy (-1, 0, Integer.MIN_VALUE) — it won't matter.

   dummy is a pointer to the first node (Integer.MIN_VALUE), which is just a **helper**.

dummy.next is the **real head** of the final list.

So, returning dummy.next means you're **ignoring the dummy** and starting from the first real node.

```
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        List<Integer> values = new ArrayList<>();
        // Collect values from list1
        while (list1 != null) {
            values.add(list1.val);
            list1 = list1.next;
        }
        // Collect values from list2
        while (list2 != null) {
            values.add(list2.val);
            list2 = list2.next;
        }
        // Sort the array
        Collections.sort(values);
        // Build new sorted linked list
        ListNode dummy = new ListNode(Integer.MIN_VALUE);
        ListNode current = dummy;
        for (int val : values) {
            current.next = new ListNode(val);
            current = current.next;
        }
        // Skip the dummy and return the actual head
        return dummy.next;
    }
}
```

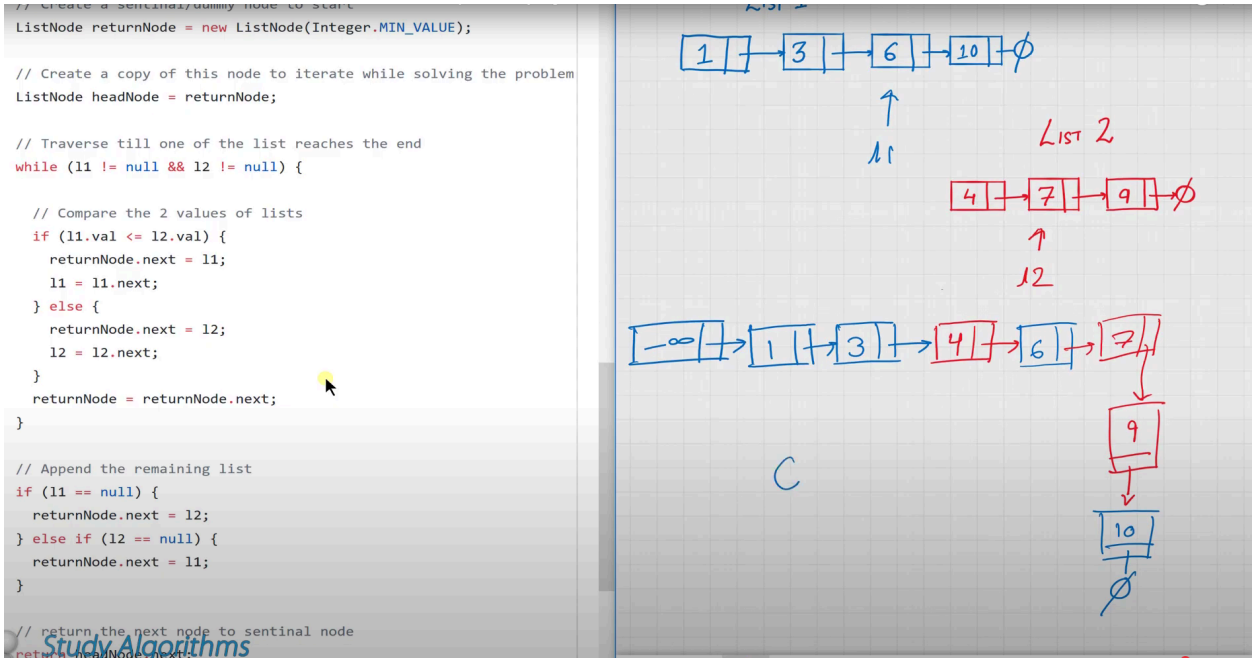
Or

```
package com.leetcode150;
import java.util.ArrayList;
import java.util.Collections;
```

```
import java.util.List;
class ListNodes {
	int val;
	ListNodes next;
	ListNodes() {
	}
	ListNodes(int val) {
		this.val = val;
		this.next = null;
	}
	ListNodes(int val, ListNodes next) {
		this.val = val;
		this.next = next;
	}
}
public class MergeTwoSortedLists {
	public static void main(String[] args) {
		// list1: 1 -> 3 -> 5
		ListNodes list1 = new ListNodes(1, new ListNodes(3, new ListNodes(5)));
		/*
		 * ListNode head = new ListNode(16); head.next = new ListNode(15);
head.next.next
		 * = new ListNode(8); head.next.next.next = new ListNode(4); 🔍 What it does:
		 * Manually connects each node, step by step.
		 *
		 * This will create: 16 → 15 → 8 → 4
		 */
		// list2: 2 -> 4 -> 6
		ListNodes list2 = new ListNodes(2, new ListNodes(4, new ListNodes(6)));
		MergeTwoSortedLists obj = new MergeTwoSortedLists();
		ListNodes merged = obj.mergeTwoLists(list1, list2);
		// Print result
		printList(merged); // Output: 1 2 3 4 5 6
	}
	public ListNodes mergeTwoLists(ListNodes list1, ListNodes list2) {
		List<Integer> values = new ArrayList<>();
		// Collect values from list1
		while (list1 != null) {
			values.add(list1.val);
			list1 = list1.next;
```

```
                }
                // Collect values from list2
                while (list2 != null) {
                        values.add(list2.val);
                        list2 = list2.next;
                }
                // Sort the array
                Collections.sort(values);
                // Build new sorted linked list
                ListNodes dummy = new ListNodes(Integer.MIN_VALUE);
                ListNodes current = dummy;
                for (int val : values) {
                        current.next = new ListNodes(val);
                        current = current.next;
                }
                // Skip the dummy and return the actual head
                return dummy.next;
        }
        // Utility method to print linked list
        public static void printList(ListNodes merged) {
                while (merged != null) {
                        System.out.print(merged.val + " ");
                        merged = merged.next;
                }
                System.out.println();
        }
}
```

Optimized Solution



The approach shown in the image is called the **Two-Pointer / In-Place Merge** using a **Dummy Node**. It's the optimal and most common way to **merge two sorted linked lists** in **O(n + m)** time and **O(1)** space.

```
public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        // Step 1: Create dummy node to start building merged list
        ListNode returnNode = new ListNode(Integer.MIN_VALUE);
        ListNode headNode = returnNode;
        // Step 2: Traverse both lists until either becomes null
        while (list1 != null && list2 != null) {
            if (list1.val <= list2.val) {
             returnNode.next = list1;
                list1 = list1.next;
            } else {
                returnNode.next = list2;
                list2 = list2.next;
            }
            returnNode = returnNode.next;
        }
        // Step 3: Attach remaining elements of the non-null list
        if (list1 == null) {
```

```
            returnNode.next = list2;
        } else if (list2 == null) {
         returnNode.next = list1;
        }
        // Step 4: Return merged list (skip dummy)
        return headNode.next;
    }
```

Dummy Node:

- A dummy node (e.g., new ListNode(Integer.MIN_VALUE)) acts as a starting placeholder.
- This avoids handling special cases for the head of the new list.

Two Pointers (l1, l2):

- These pointers traverse the two input linked lists.
- At each step, compare l1.val and l2.val.

Build the Result List:

- Attach the smaller node to the result list using returnNode.next.
- Move the pointer (l1 or l2) from which the node was taken.
- Advance returnNode to the new last node.

Attach Remaining Nodes:

- When one of the lists is fully traversed, attach the remaining part of the other list — it's already sorted.

Return the Real Head:

- Skip the dummy node and return headNode.next.

Why This Dummy Node Doesn't Violate O(1):

- The dummy node is **just one node** → constant space → **O(1)**.
- The rest of the merged list is built by **reusing the nodes from list1 and list2**, not by **creating new nodes**.

You're just **reassigning next pointers** of existing nodes — no new memory allocation for each data value.

List 1 (L1):

1 -> 3 -> 6 -> 10

List 2 (L2):

4 -> 7 -> 9

We want to merge them into one **sorted** linked list.

📑 Step 1: Compare L1 and L2 nodes

L1.val	L2.val	Action	Result List
1	4	1 < 4 → attach L1 to result	dummy -> 1
3	4	3 < 4 → attach L1 to result	dummy -> 1 -> 3
6	4	4 < 6 → attach L2 to result	dummy -> 1 -> 3 -> 4
6	7	6 < 7 → attach L1 to result	dummy -> ... -> 6
10	7	7 < 10 → attach L2 to result	dummy -> ... -> 7
10	9	9 < 10 → attach L2 to result	dummy -> ... -> 9

Now, **L2 is empty**, but **L1 still has 10**.

Step 2: Attach remaining nodes

Since L2 == null, attach remaining L1:

returnNode.next = l1; // 10

Final Merged List:

1 -> 3 -> 4 -> 6 -> 7 -> 9 -> 10

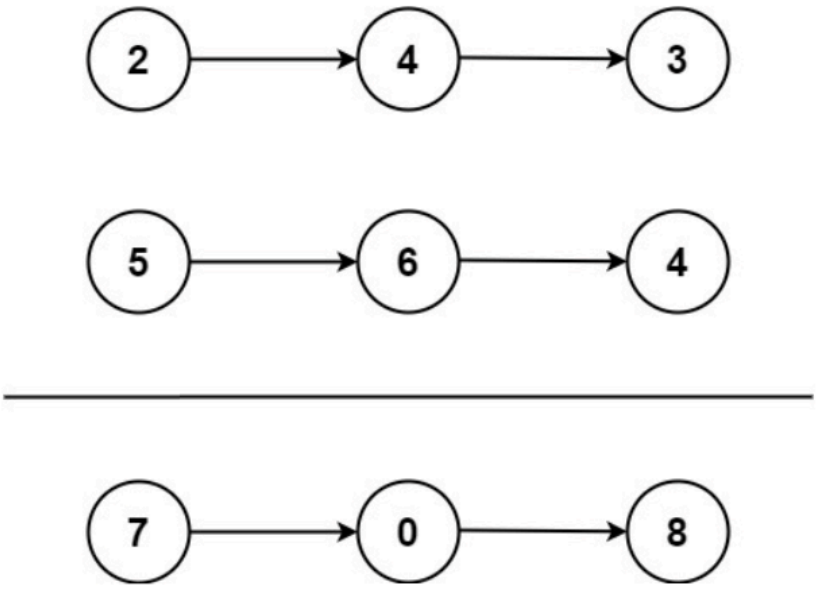
Return dummy.next as the head of the merged list.

30.Add Two Numbers

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example 1:




```
Input: l1 = [2,4,3], l2 = [5,6,4]
Output: [7,0,8]
Explanation: 342 + 465 = 807.
```

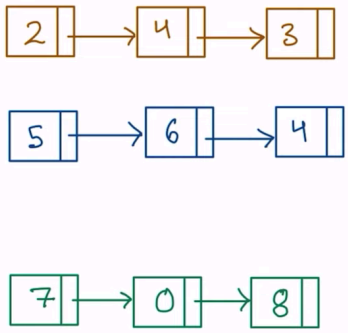
Example 2:

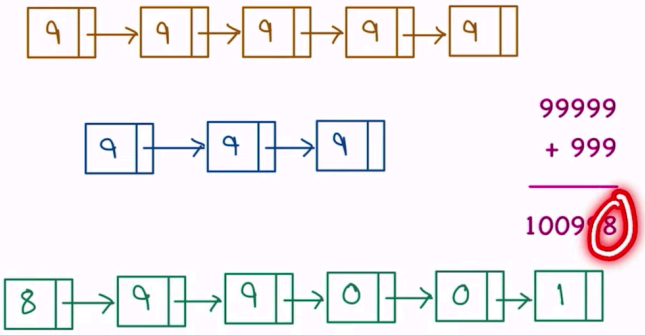
Input: l1 = [0], l2 = [0]
Output: [0]

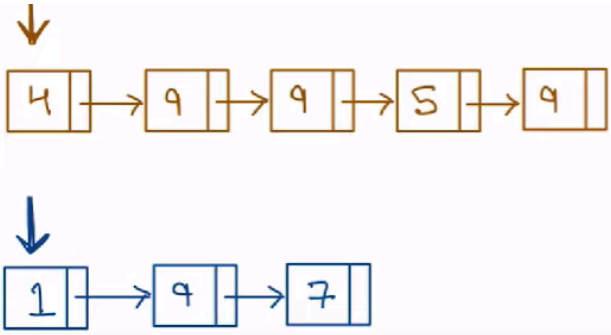
Example 3:

Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
Output: [8,9,9,9,0,0,0,1]

Optimized solution

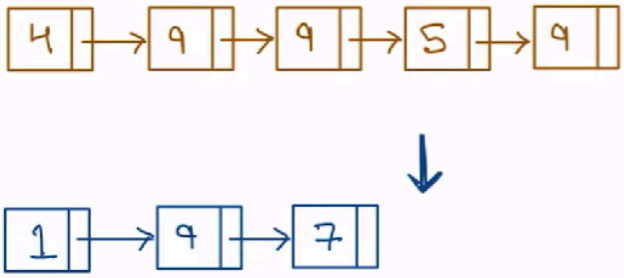


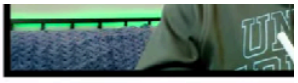




Sum = 5

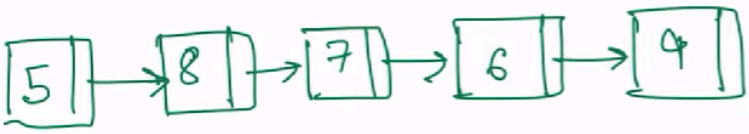
Carry = 0



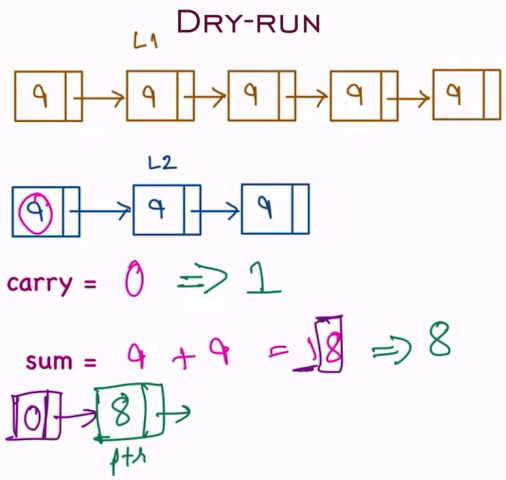


Sum = 9

Carry = 0



```java
ListNode addTwoNumbers(ListNode l1, ListNode l2) {

  ListNode result = new ListNode( x: 0);
  ListNode ptr = result;

  int carry = 0;    // Set default carry

  while (l1 != null || l2 != null) {

    int sum = 0 + carry;    // Initialize sum

    if (l1 != null) {    // Use number from first list
      sum += l1.val;
      l1 = l1.next;
    }

    if (l2 != null) {    // Use number from 2nd list
      sum += l2.val;
      l2 = l2.next;
    }

    carry = sum / 10;    // Get sum and carry
    sum = sum % 10;
    ptr.next = new ListNode(sum);
    ptr = ptr.next;
  }

  if (carry == 1) ptr.next = new ListNode( x: 1);

  return result.next;
}
```



1. Initialize:
    - A dummy node to build the result.
    - carry = 0.

2. Loop through both lists:
    - Sum corresponding nodes + carry.
    - Extract digit = sum % 10, and carry = sum / 10.

3. If any carry remains after loop, append it as a new node.

Ex:

l1 = [9,9,9,9]
l2 = [9,9]

Walkthrough:

- First Iteration:
    - 9 + 9 + 0 = 18 → carry = 1, node = 8
- Second:
    - 9 + 9 + 1 = 19 → carry = 1, node = 9
- Third:
    - 9 + 0 + 1 = 10 → carry = 1, node = 0

- Fourth:
  - 9 + 0 + 1 = 10 → carry = 1, node = 0
- End:
  - Carry = 1 → add extra node 1

Result Linked List:
[8 → 9 → 0 → 0 → 1]

→ Which represents: 19998 + 99 = 20097

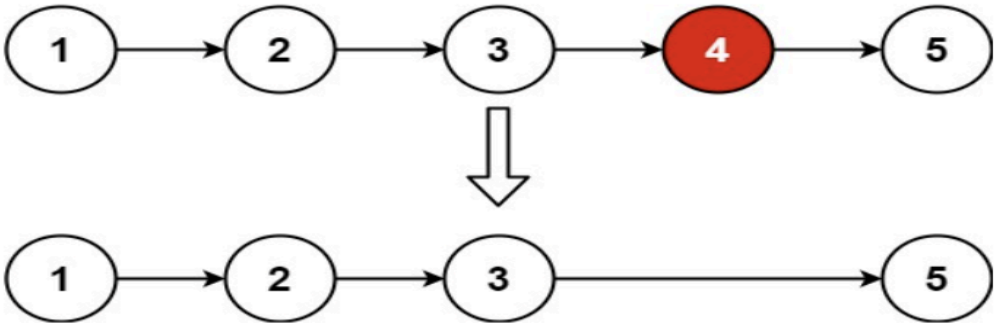
```
package com.leetcode150;
class ListNod {
	int val;
	ListNod next;
	ListNod() {
	}
	ListNod(int val) {
		this.val = val;
		this.next = null;
	}
	ListNod(int val, ListNod next) {
		this.val = val;
		this.next = next;
	}
}
public class AddTwoNumbers {
	public static void main(String[] args) {
		// l1 = [2,4,3] represents 342
		ListNod l1 = new ListNod(2, new ListNod(4, new ListNod(3)));
		// l2 = [5,6,4] represents 465
		ListNod l2 = new ListNod(5, new ListNod(6, new ListNod(4)));
		ListNod result = addTwoNumbers(l1, l2); // Should return 7 -> 0 -> 8 (807)
		printList(result); // Helper method to print the list
	}
	public static ListNod addTwoNumbers(ListNod l1, ListNod l2) {
		ListNod result = new ListNod(0); // Dummy node
		ListNod ptr = result; // Pointer to traverse result list
		int carry = 0; // Set default carry
		while (l1 != null || l2 != null) {
			int sum = 0 + carry; // Initialize sum with carry
			if (l1 != null) {
```

```
                sum += l1.val; // Add value from first list
                l1 = l1.next;
            }
            if (l2 != null) {
                sum += l2.val; // Add value from second list
                l2 = l2.next;
            }
            carry = sum / 10; // Update carry
            sum = sum % 10; // Take remainder for digit
            ptr.next = new ListNod(sum); // Create new node with current digit
            ptr = ptr.next; // Move pointer forward
        }
        if (carry == 1) {
            ptr.next = new ListNod(1); // Add leftover carry if exists
        }
        return result.next; // Return actual result (skipping dummy head)
    }
    // Utility method to print linked list
    public static void printList(ListNod head) {
        while (head != null) {
            System.out.print(head.val + " ");
            head = head.next;
        }
        System.out.println();
    }
}
```

31.Remove Nth Node From End of List

Given the head of a linked list, remove the **nth** node from the **end of the list** and return its head.

Example 1:



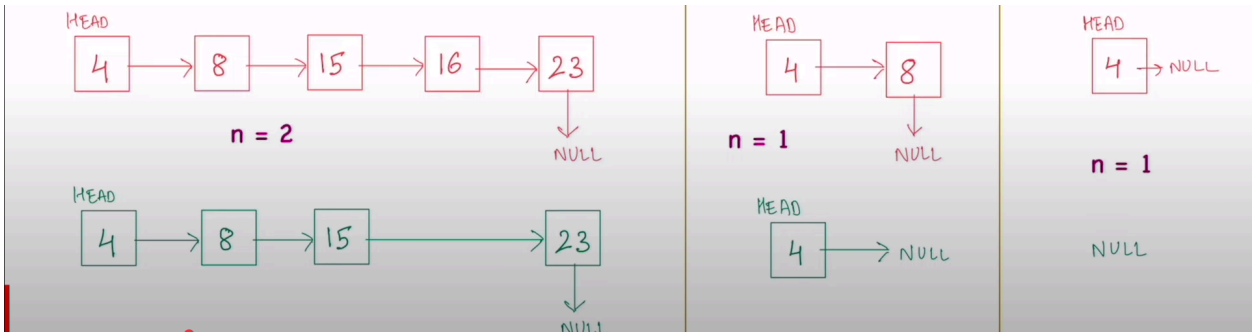

```
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]
```

Example 2:

```
Input: head = [1], n = 1
Output: []
```

Example 3:

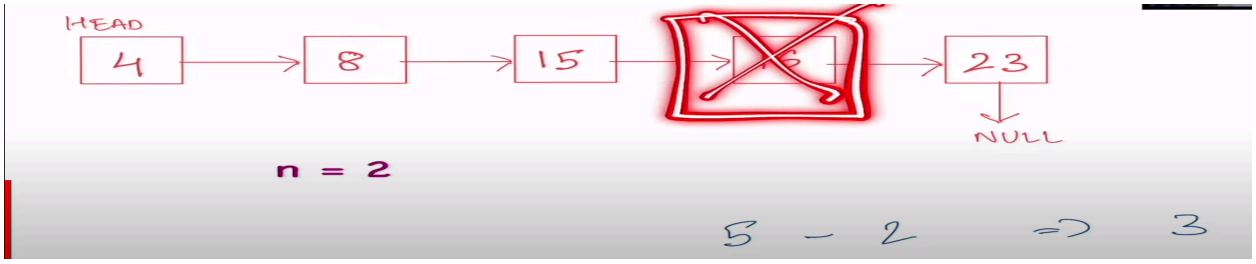
```
Input: head = [1,2], n = 1
Output: [1]
```



Approach 1: **Two Passes (Length-Based)**

1. **First Pass**: Traverse the linked list to calculate its total length.
2. **Second Pass**: Traverse again to reach the (length - n)th node and remove the next node.

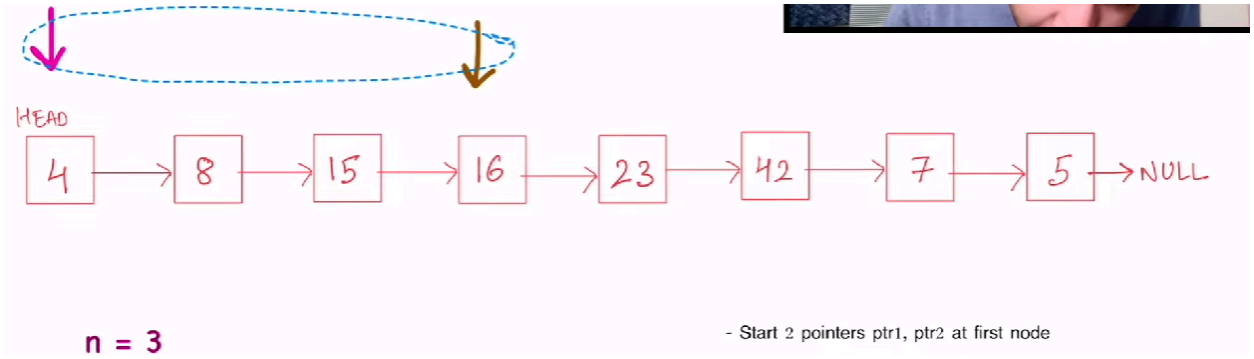
Time Complexity: O(2n)

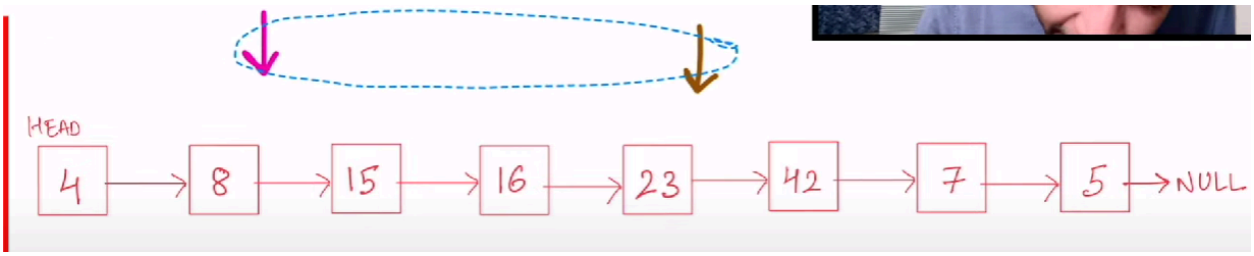


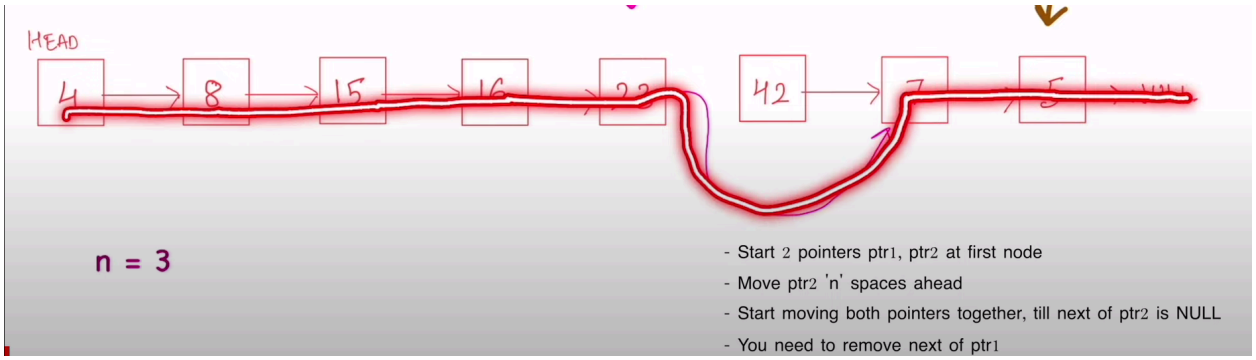
Optimized solution

Approach 2: Single Pass (**Two Pointer – Fast and Slow)**

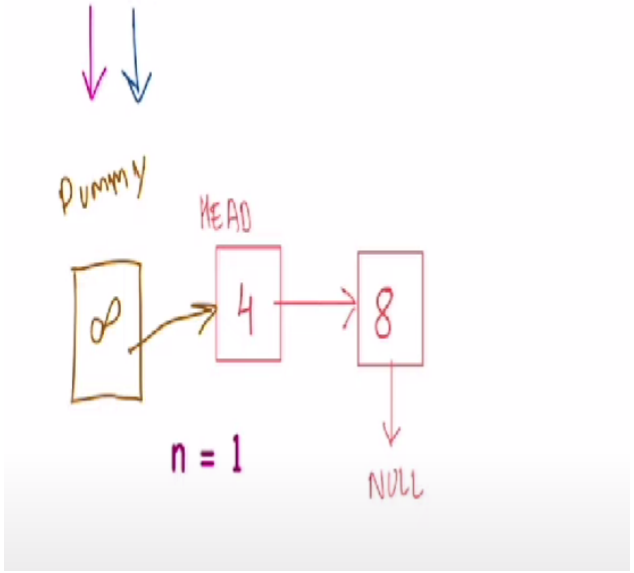
- Create a dummy node and set dummy.next = head.
  This helps to handle edge cases (like deleting the head node).
- Start two pointers: firstPtr and secondPtr at the dummy node.
- Move secondPtr ahead by n nodes.
- Then move both pointers simultaneously until secondPtr.next == null.
- Now firstPtr is at the node just before the one to be deleted.
- Update firstPtr.next = firstPtr.next.next to skip the target node.
- Return dummy.next as the new head of the list.

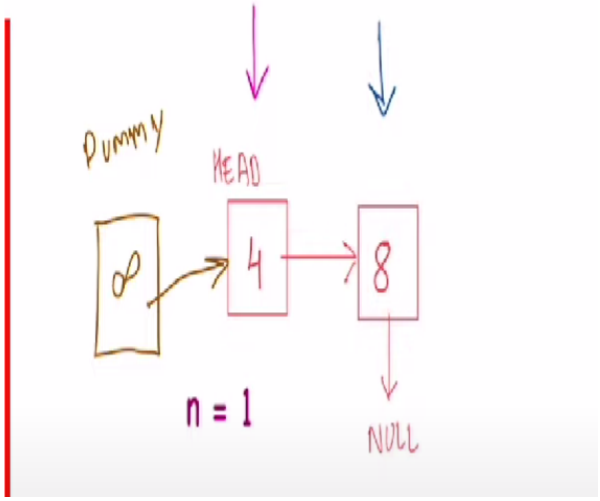


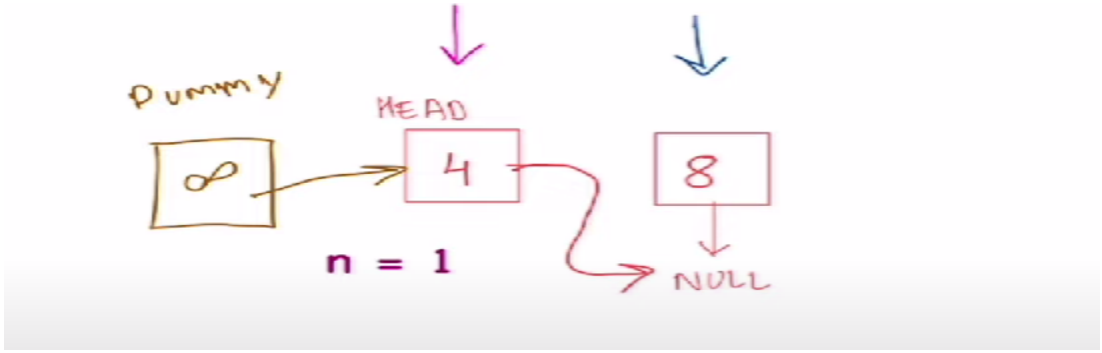



HEAD
4
8
15
16
22
42
7
5
n = 3
- Start 2 pointers ptr1, ptr2 at first node
- Move ptr2 'n' spaces ahead
- Start moving both pointers together, till next of ptr2 is NULL
- You need to remove next of ptr1

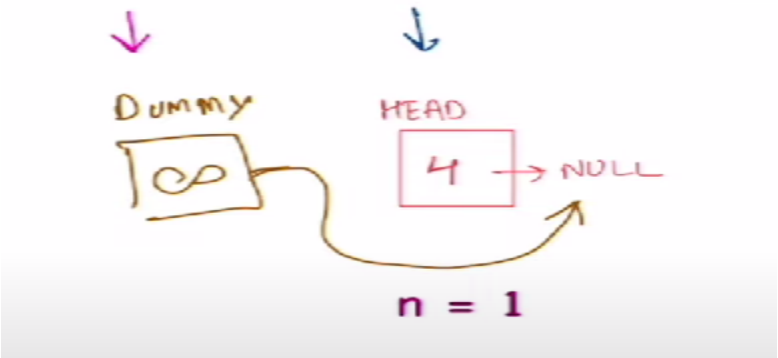
EX1:

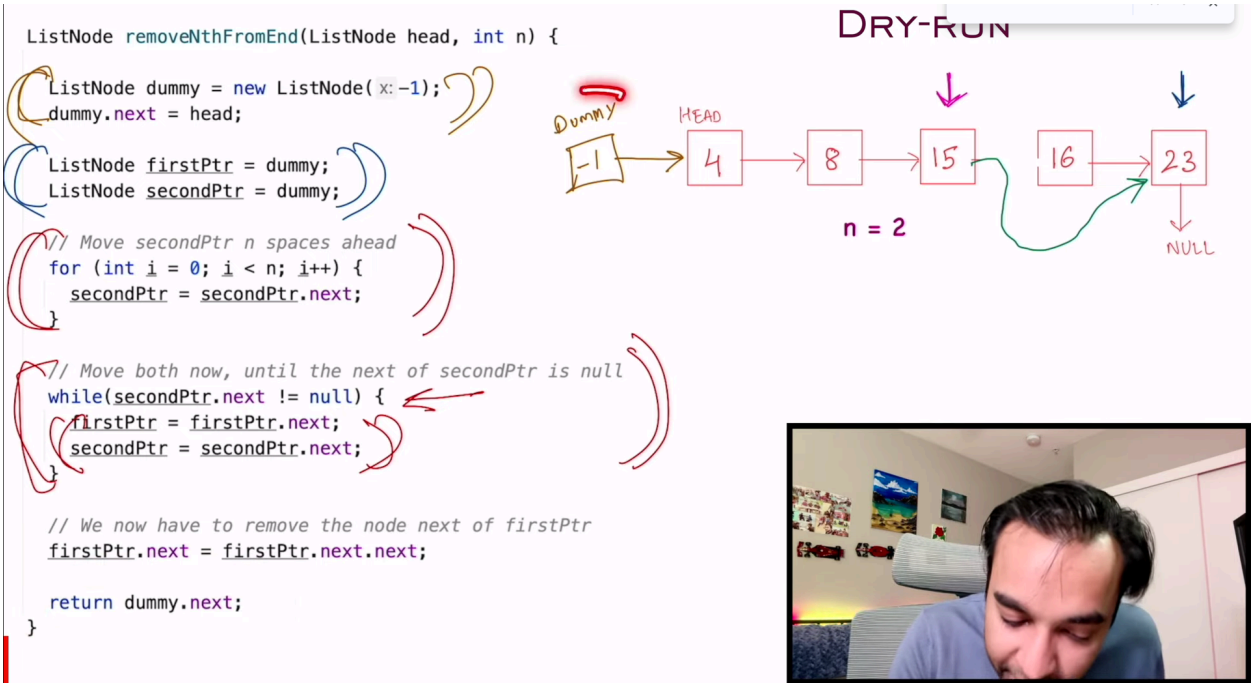

Dummy
HEAD
4
8
n = 1
NULL


Dummy
HEAD
4
8
n = 1
NULL


Dummy
HEAD
4
8
n = 1
NULL

Ex3






```
package com.leetcode150;
class ListNodee {
	int val;
	ListNodee next;
	ListNodee() {
	}
	ListNodee(int val) {
		this.val = val;
		this.next = null;
	}
```

```
        ListNodee(int val, ListNodee next) {
                this.val = val;
                this.next = next;
        }
}
public class RemoveNthNodeFromEndofList {
        public static void main(String[] args) {
                ListNodee l1 = new ListNodee(2, new ListNodee(4, new ListNodee(3, new
ListNodee(9, new ListNodee(19)))));
                /*
                 * // Creating the linked list: 4 -> 8 -> 15 -> 16 -> 23 ListNodee head = new
                 * ListNodee(4); head.next = new ListNodee(8); head.next.next = new
                 * ListNodee(15); head.next.next.next = new ListNodee(16);
                 * head.next.next.next.next = new ListNodee(23);
                 *
                 */
                System.out.println("Original List:");
                printList(l1);
                int n = 2; // Remove the 2nd node from the end
                l1 = removeNthFromEnd(l1, n);
                System.out.println("List after removing " + n + "th node from end:");
                printList(l1);
        }
        public static ListNodee removeNthFromEnd(ListNodee head, int n) {
                ListNodee dummy = new ListNodee(Integer.MIN_VALUE); // -1,or 0 any value
                dummy.next = head;
                ListNodee firstPtr = dummy;
                ListNodee secondtPtr = dummy;
                for (int i = 0; i < n; i++) {
                        secondtPtr = secondtPtr.next;
                }
                while (secondtPtr.next != null) {
                        firstPtr = firstPtr.next;
                        secondtPtr = secondtPtr.next;
                }
```

```
                // removing the node next of firstPtr
                firstPtr.next = firstPtr.next.next;
                return dummy.next;
        }
        // Utility method to print linked list
        public static void printList(ListNodee head) {
                while (head != null) {
                        System.out.print(head.val + " ");
                        head = head.next;
                }
                System.out.println();
        }
}
```

Original List:
2 4 3 9 19
List after removing 2th node from end:
2 4 3 19

Stacks and Queues

Stacks and Queues **- done**

32)Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.
Implement the MinStack class:

- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on the top of the stack.
- int top() gets the top element of the stack.
- int getMin() retrieves the minimum element in the stack.

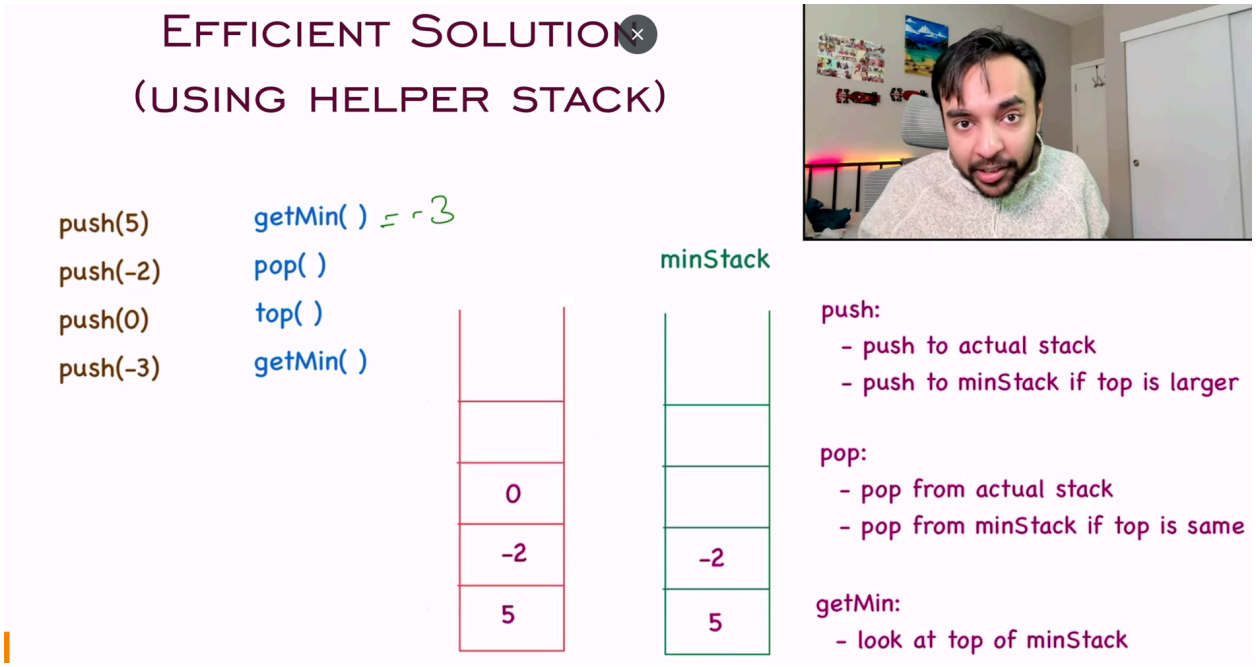
You must implement a solution with O(1) time complexity for each function.

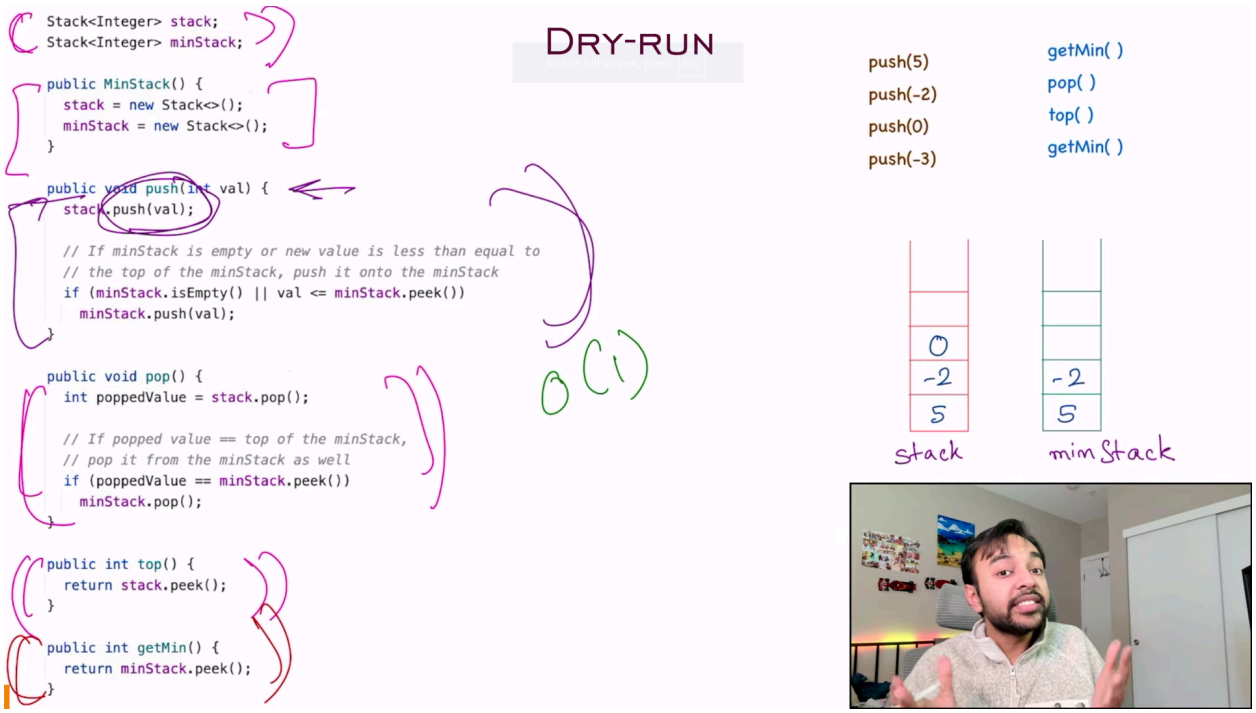
Example 1:

Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output
[null,null,null,null,-3,null,0,-2]

Explanation
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top(); // return 0
minStack.getMin(); // return -2





How It Works (Min Stack using Two Stacks):

We use two stacks:

stack – holds all pushed values.

minStack – holds the minimum values seen so far.

stack stores **all values** pushed into the stack (normal stack behavior).

minStack keeps track of **minimum values** so that the **top of minStack is always the current minimum**.

On push(val):

- Push val to stack.
- If minStack is empty **or** val <= minStack.peek(), also push val to minStack.

On pop():

- Pop the top from the **stack**.
- If the popped value is **equal to** minStack.peek(), also pop from minStack.

On top():

- Return the top element from the **stack**.

On getMin():

- Return the top element from minStack (current minimum).

Operation	Time Complexity	Space Complexity
`push`	O(1)	O(1)
`pop`	O(1)	O(1)
`top`	O(1)	O(1)
`getMin`	O(1)	O(1)
Overall Space	–	O(n)

```
	package com.leetcode150;
import java.util.Stack;
public class MinStack {
	Stack<Integer> stack;
	Stack<Integer> minStack;
	// constructor
	public MinStack() {
		stack = new Stack<>();
		minStack = new Stack<>();
	}
```

```
	public static void main(String[] args) {
		MinStack minStack = new MinStack();
		minStack.push(5);
		minStack.push(-1);
		minStack.push(2);
		minStack.push(-2);
		System.out.println("Stack: " + minStack.stack);
		System.out.println("MinStack: " + minStack.minStack);
		System.out.println(minStack.getMin()); // Output: 1
		minStack.pop(); // Removes 1
		System.out.println(minStack.getMin()); // Output: 2
		minStack.pop(); // Removes 2
		System.out.println(minStack.getMin()); // Output: 2
		System.out.println("Stack: " + minStack.stack);
		System.out.println("MinStack: " + minStack.minStack);
	}
	public void push(int val) {
		stack.push(val);
		// If minStack is empty or new value is added is less than equal to the top of
		// the minStack , push it onto the minStack
		if (minStack.isEmpty() || val <= minStack.peek()) {
			minStack.push(val);
		}
	}
	public void pop() {
		int poppedValue = stack.pop();
		// IF the popped value== top of the minStack,
		// pop it from the minStack as well
		if (poppedValue == minStack.peek()) {
			minStack.pop();
		}
	}
	public int top() {
		return stack.peek();
	}
	public int getMin() {
		return minStack.peek();
	}
}
```

Output

Stack: [5, -1, 2, -2]
MinStack: [5, -1, -2]
-2
-1
-1
Stack: [5, -1]
MinStack: [5, -1]

33.Implement Queue using Stacks

Implement a first in first out (FIFO) **queue** using only **two stacks**. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).
Implement the MyQueue class:

- void push(int x) Pushes element x to the back of the queue.
- int pop() Removes the element from the front of the queue and returns it.
- int peek() Returns the element at the front of the queue.
- boolean empty() Returns true if the queue is empty, false otherwise.

Notes:

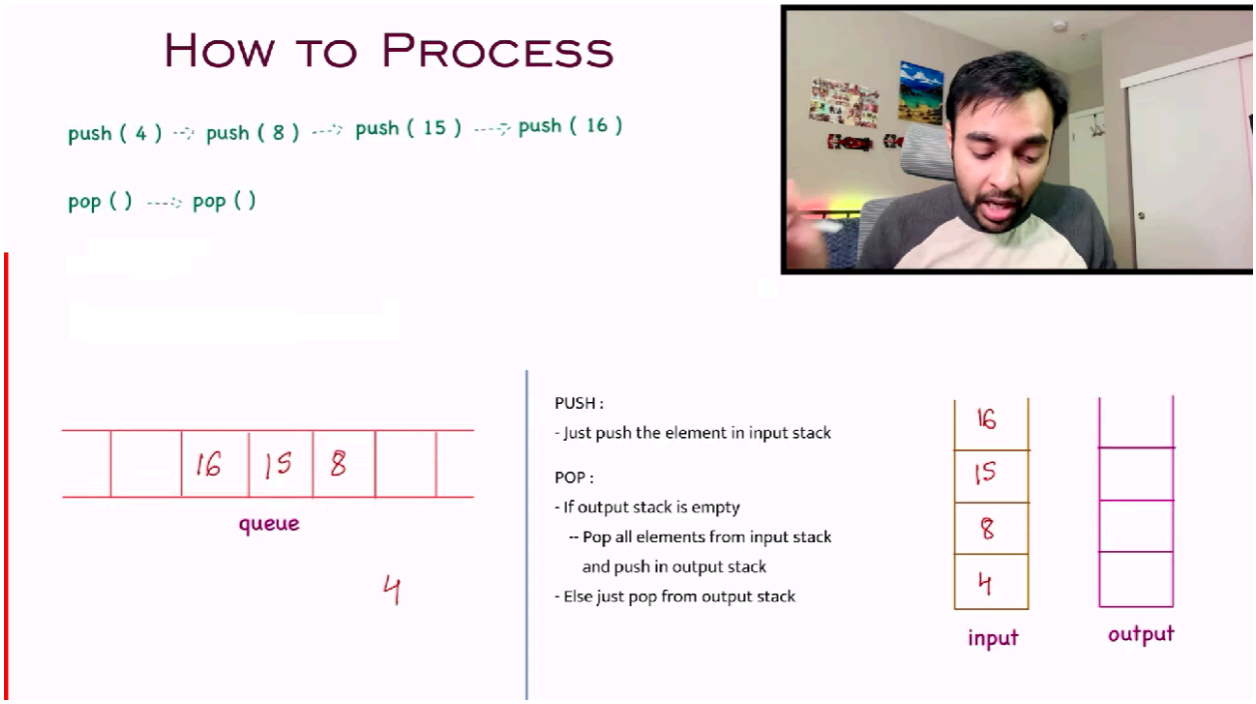
- You must use **only** standard operations of a stack, which means only push to top, peek/pop from top, size, and is empty operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

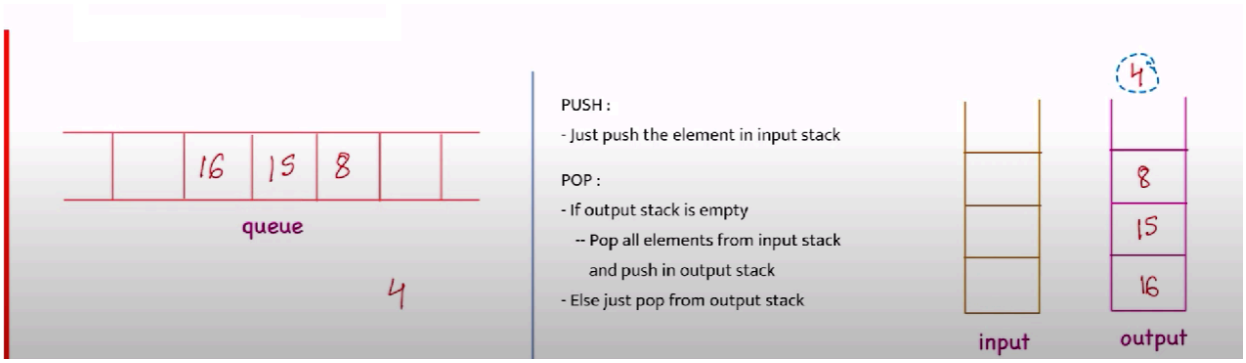
Example 1:

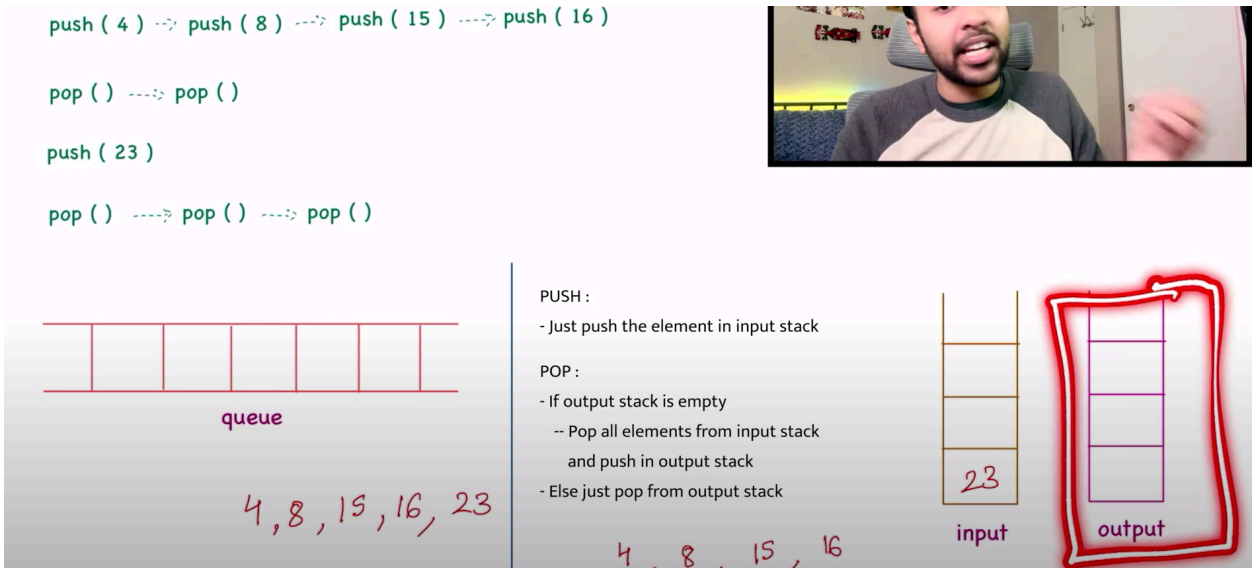
Input
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
Output
[null, null, null, 1, 1, false]

Explanation
MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)

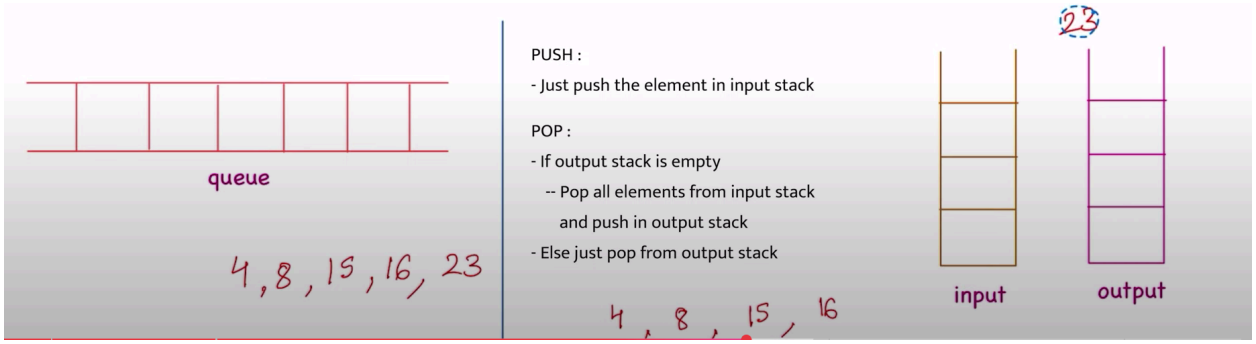
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false







Whenever stackOut is empty, we transfer all elements from stackIn to stackOut



1. **Two stacks:**
    - input stack: for pushing new elements (enqueue)
    - output stack: for removing elements from the front (dequeue/peek)
2. **push(x)**:
    - Push element x to the input stack.
    - **Time:** O(1)
3. **peek()**:
    - If output is empty:
        - Transfer all elements from input to output.
    - Then return the top of the output.
    - **Time:** Amortized O(1)

4. **pop()**:
    - Calls peek() to ensure the correct element is on top of output.
    - Then pops the top from the output.
    - **Time:** Amortized O(1)
5. **empty()**:
    - Returns true only if both stacks are empty.
    - **Time:** O(1)

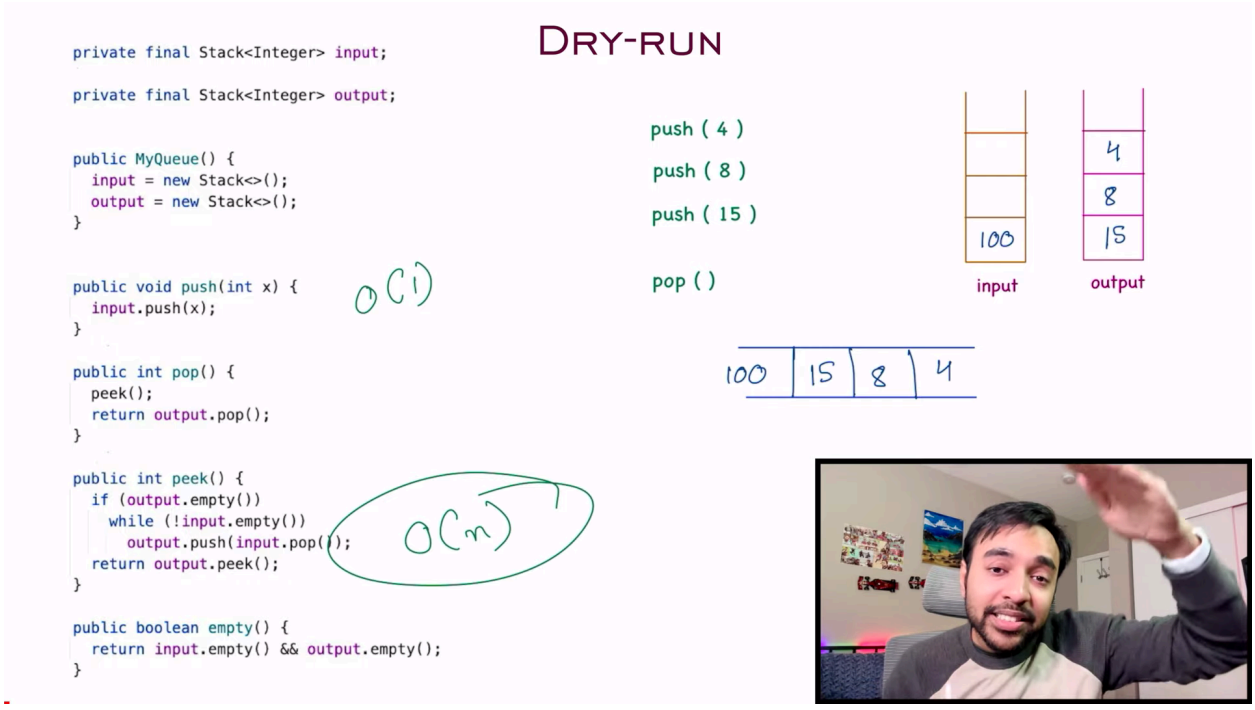
⏱️ Time Complexity (Per Operation)

Operation	Time Complexity
push	O(1)
pop	**Amortized** O(1)
peek	**Amortized** O(1)
empty	O(1)

> **Amortized O(1): Although a full transfer from `input` to `output` takes O(n), each element is moved at most once — hence, over many operations, the average is O(1).**

Space Complexity:

- **O(n)** – where n is the number of elements in the queue.
- All elements are either in input or output — never duplicated.




```
        package com.leetcode150;
import java.util.Stack;
public class ImplementQueueusingStacks {
        Stack<Integer> input;
        Stack<Integer> output;
        public ImplementQueueusingStacks() {
                input = new Stack<>();
                output = new Stack<>();
        }
        public static void main(String[] args) {
                ImplementQueueusingStacks queue = new ImplementQueueusingStacks();
                // Push elements into the queue
                queue.push(10);
                queue.push(20);
                queue.push(30);
                System.out.println("input stack :" + queue.input);
                System.out.println("output stack :" + queue.output);
                System.out.println("pop operation :" + queue.pop());
                System.out.println("input stack :" + queue.input);
                System.out.println("output stack :" + queue.output);
                System.out.println("empty operation :" + queue.empty());
        }
```

```
        public void push(int x) {
                input.push(x);
        }
        public int pop() {
                peek();
                return output.pop();
        }
        public int peek() {
                if (output.isEmpty()) {
                        while (!input.isEmpty()) {
                                output.push(input.pop());
                        }
                }
                return output.peek();
        }
        public boolean empty() {
                return input.isEmpty() && output.isEmpty();
        }
}
```

Output
input stack :[10, 20, 30]
output stack :[]
pop operation :10
input stack :[]
output stack :[30, 20]
empty operation :false

34.Implement Stack using Queues

Implement a last-in-first-out (LIFO) stack **using only two queues**. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

Implement the MyStack class:

- void push(int x) Pushes element x to the top of the stack.
- int pop() Removes the element on the top of the stack and returns it.
- int top() Returns the element on the top of the stack.
- boolean empty() Returns true if the stack is empty, false otherwise.

Notes:

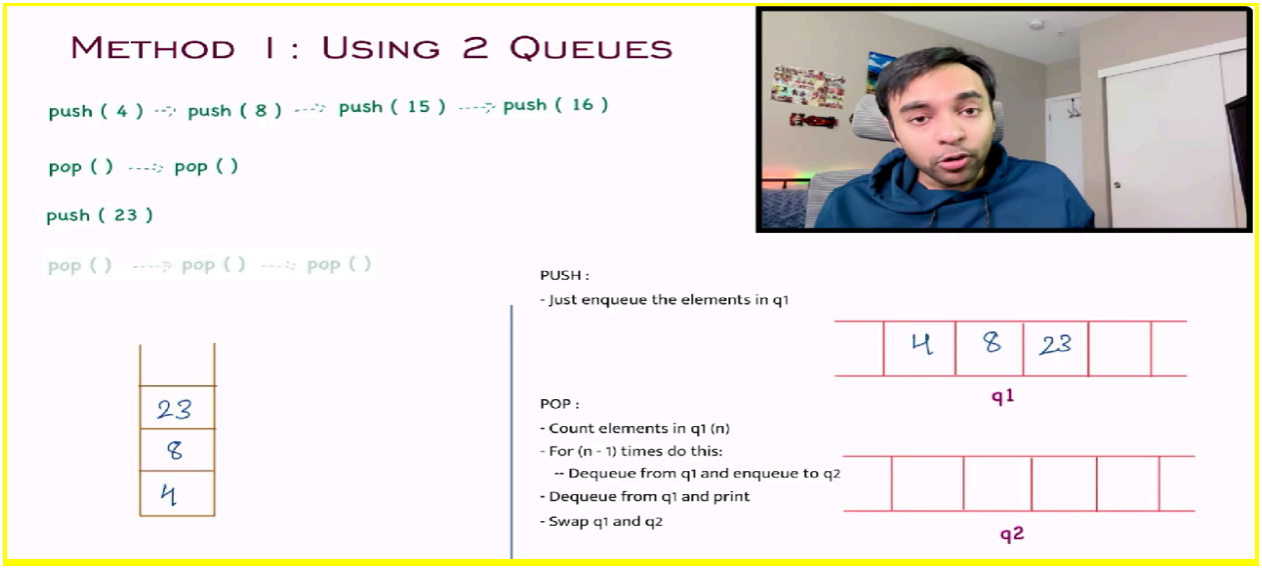
- You must use **only** standard operations of a queue, which means that only push to back, peek/pop from front, size and is empty operations are valid.
- Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

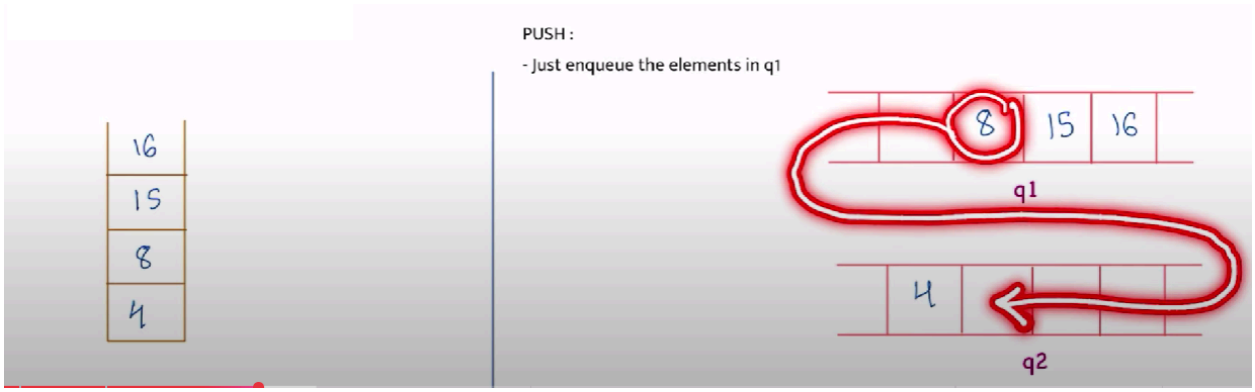
Example 1:

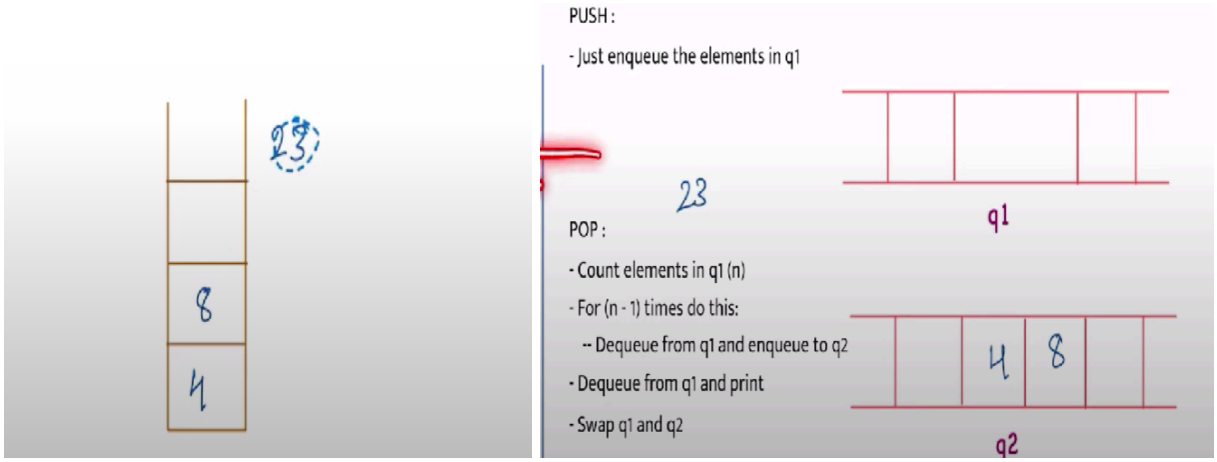
Input
["MyStack", "push", "push", "top", "pop", "empty"]
[[], [1], [2], [], [], []]
Output
[null, null, null, 2, 2, false]

Explanation
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // return 2
myStack.pop(); // return 2
myStack.empty(); // return False

Optimized solution
Using two queues (First Approach)







Stack Using Two Queues – Key Steps:

- Take **two queues**: q1 (main), q2 (helper).
- For push(x):
  → Simply enqueue(add) the element into q1.
- For pop():
  - Remove n - 1 elements from q1 and enqueue into q2.
  - The **last element** left in q1 is the top of the stack → remove and return it.
  - **Swap** q1 and q2.
- For top():

- Remove n - 1 elements from q1 and enqueue into q2.
- The **last element** is the top → store it, enqueue it into q2.
- **Swap** q1 and q2.

- For empty():
  - Return true if q1 is empty.

⏱ Time and Space Complexity Summary:

Operation	Time Complexity	Space Complexity
push	O(1)	O(n) (queue holds elements)
pop	O(n)	O(n)
top	O(n)	O(n)
empty	O(1)	O(1)

```
package com.leetcode150;
import java.util.LinkedList;
import java.util.Queue;
public class ImplementStackusingQueues {
	Queue<Integer> q1;
	Queue<Integer> q2;
	public ImplementStackusingQueues() {
		q1 = new LinkedList<>();
		q2 = new LinkedList<>();
	}
	public static void main(String[] args) {
		ImplementStackusingQueues myStack = new ImplementStackusingQueues();
		myStack.push(1);
		myStack.push(2);
		System.out.println(myStack.top()); // Output: 2
		System.out.println(myStack.pop()); // Output: 2
		System.out.println(myStack.empty()); // Output: false
	}
	// Push element x onto stack.
```

```
        public void push(int x) {
                q1.add(x); // Just enqueue to q1
        }
        // Removes the element on top of the stack and returns that element.
        public int pop() {
                // Move n-1 elements from q1 to q2
                while (q1.size() > 1) {
                        q2.add(q1.remove());
                }
                // Last element is the top of the stack
                int popped = q1.remove();
                // Swap q1 and q2
                Queue<Integer> temp = q1;
                q1 = q2;
                q2 = temp;
                return popped;
        }
        // Get the top element.
        public int top() {
                while (q1.size() > 1) {
                        q2.add(q1.remove());
                }
                int top = q1.remove();
                q2.add(top); // Put it back into q2
                // Swap q1 and q2
                Queue<Integer> temp = q1;
                q1 = q2;
                q2 = temp;
                return top;
        }
        // Returns whether the stack is empty.
        public boolean empty() {
                return q1.isEmpty();
        }
}
```

Output

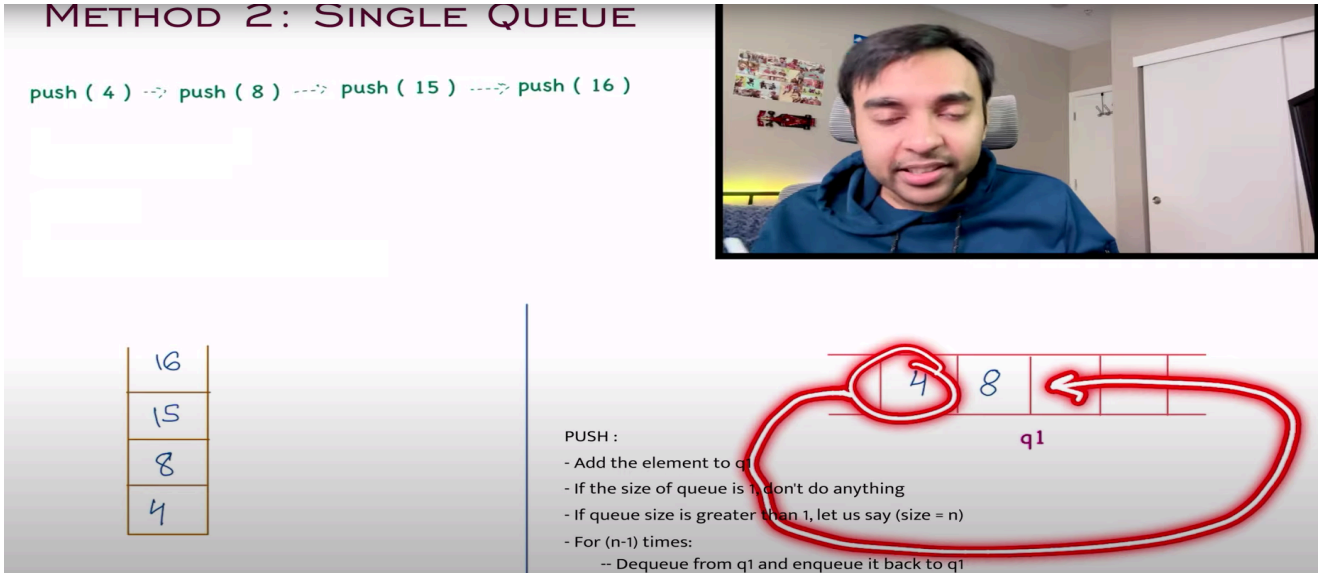
2

2

false

Using single queue (second approach)

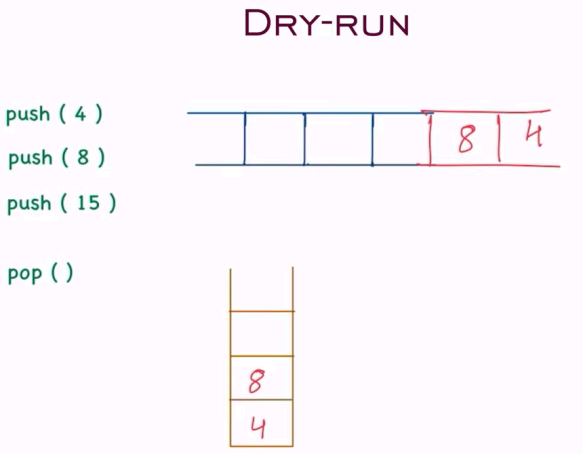
But don't use when we have more push operation.**implementing a stack using a single queue** by making the **push** **operation costly**,



```java
private Queue<Integer> queue;

public MyStack() {
  queue = new LinkedList<>();
}

public void push(int x) {;
  queue.add(x);
  for (int i = 1; i < queue.size(); i++)
    queue.add(queue.remove());
}

public int pop() {
  return queue.remove();
}

public int top() {
  return queue.peek();
}

public boolean empty() {
  return queue.isEmpty();
}
```

O(n)

O(n)



35.Sliding Window Maximum

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:

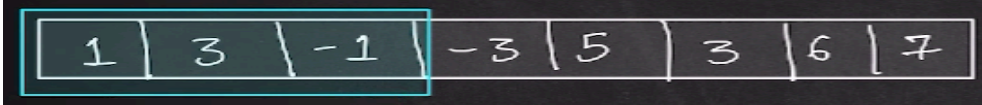
```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3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

Example 2:

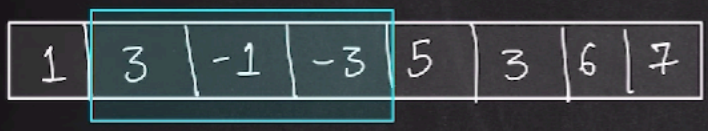
```
Input: nums = [1], k = 1
Output: [1]
```

Sliding Window Maximum

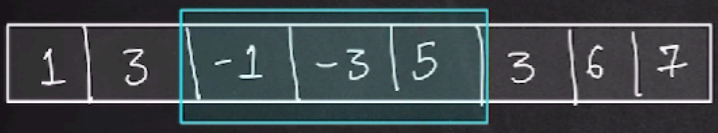
nums = [1,3,-1,-3,5,3,6,7] $k = 3$ (window size)



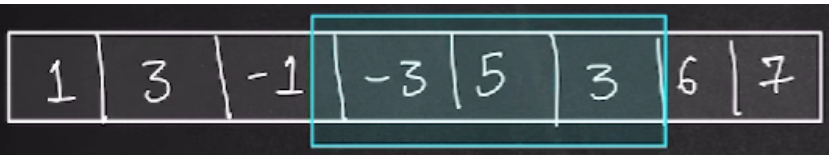
3,



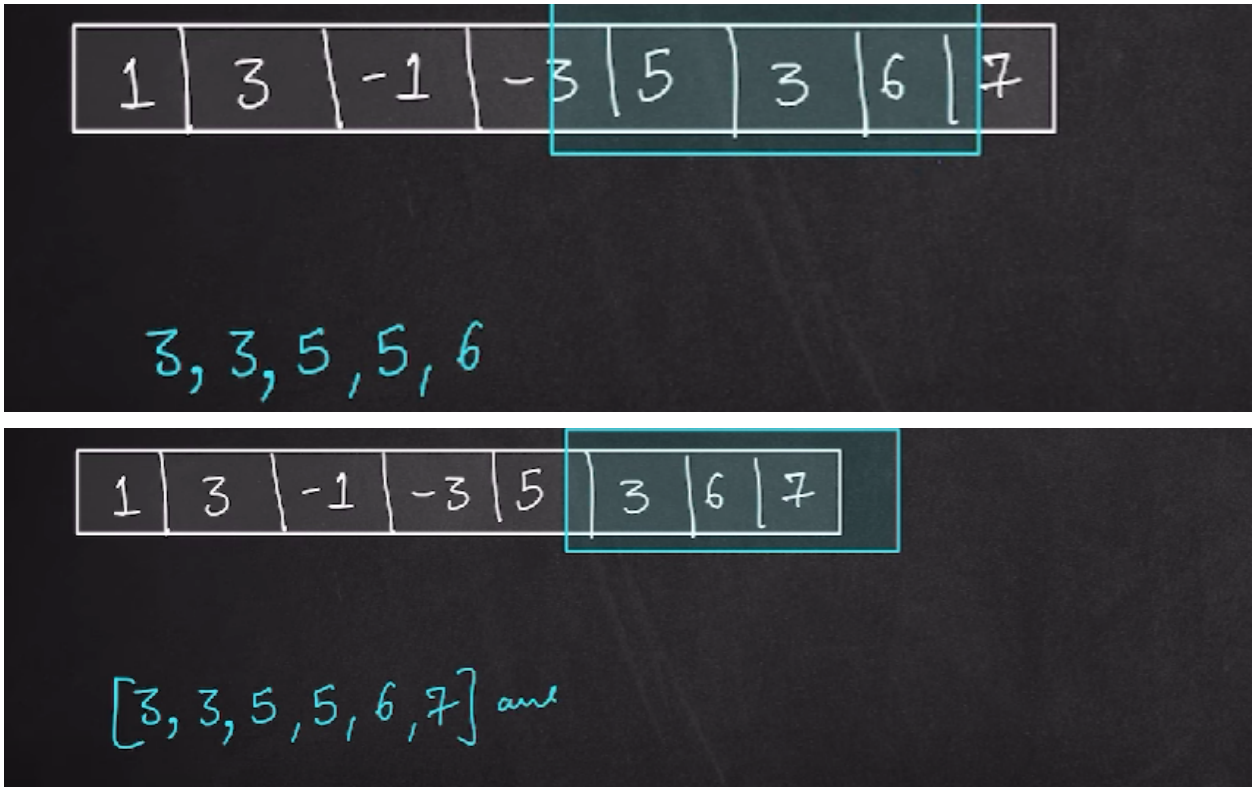
3, 3,



3, 3, 5

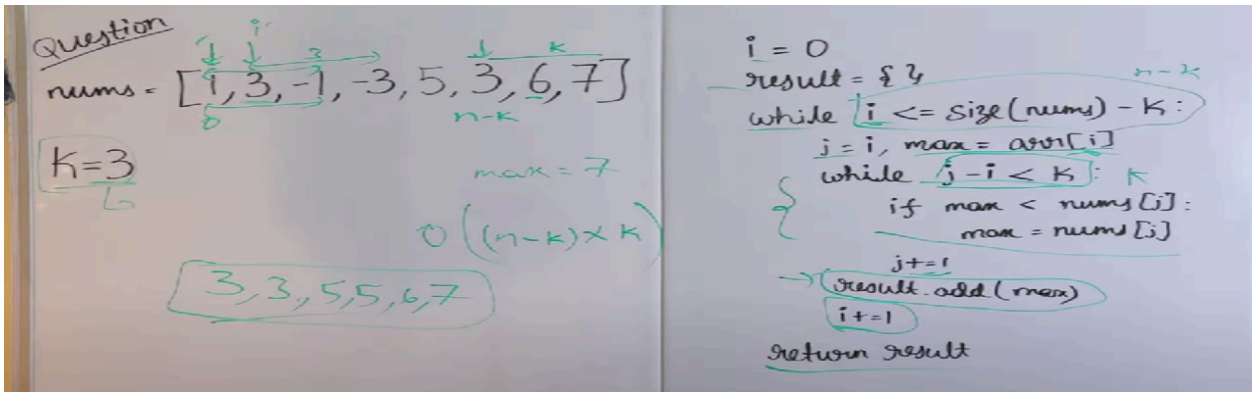


3, 3, 5, 5



General Solution

brute-force approach to solving the **Sliding Window Maximum** problem



Steps (Brute Force Approach):

1. **Initialize** an empty result list.
2. **Loop** through the array from index i = 0 to i <= nums.length - k:

- This ensures we process each possible window of size k.

3. For each window starting at index i:
    - Initialize max as the first element in the window → max = nums[i].
    - Loop j from i to j < i + k;:
        - Compare and update max with nums[j] if nums[j] > max.
4. After the inner loop, **add max to the result** list.
5. After processing all windows, **return the result** list.

```
package com.leetcode150;
import java.util.*;
public class SlidingWindowMaximum {
	public List<Integer> maxSlidingWindow(int[] nums, int k) {
		List<Integer> result = new ArrayList<>();
		int n = nums.length;
		for (int i = 0; i <= n - k; i++) { //for (int i = 0; i < n - k+1; i++)
			int max = nums[i];
			for (int j = i; j < i + k; j++) {
				if (nums[j] > max) {
					max = nums[j];
				}
			}
			result.add(max);
		}
		return result;
	}
	public static void main(String[] args) {
		SlidingWindowMaximum sw = new SlidingWindowMaximum();
		int[] nums = { 1, 3, -1, -3, 5, 3, 6, 7 };
		int k = 3;
		System.out.println(sw.maxSlidingWindow(nums, k)); // Output: [3, 3, 5, 5, 6, 7]
	}
}
```

Or

```java
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        List<Integer> result = new ArrayList<>();
        int n = nums.length;
        for (int i = 0; i <= n - k; i++) { // for (int i = 0; i < n - k+1; i++) {
            int max = nums[i];
            for (int j = i; j < i + k; j++) {
                if (nums[j] > max) {
                    max = nums[j];
                }
            }
            result.add(max);
        }
        // Convert List<Integer> to int[]
        int[] resArray = new int[result.size()];
        for (int i = 0; i < result.size(); i++) {
            resArray[i] = result.get(i);
        }
        return resArray;
    }
}
```

Output

[3, 3, 5, 5, 6, 7]

- Outer loop: runs **(n - k + 1)** times
- Inner loop: runs **k** times per outer loop iteration

Total comparisons = (n - k + 1) * k

- **Time Complexity: O((n - k + 1) * k)**

Space Complexity: O(1)

Approach 2 :Using Map (don't use,just for idea)

```java
package com.leetcode150;
import java.util.*;
public class SlidingWindowMaximum {
	public List<Integer> maxSlidingWindow(int[] nums, int k) {
		List<Integer> result = new ArrayList<>();
		TreeMap<Integer, Integer> map = new TreeMap<>();
		// Build the first window
		for (int i = 0; i < k; i++) {
			map.put(nums[i], map.getOrDefault(nums[i], 0) + 1);
		}
		result.add(map.lastKey());
		for (int i = k; i < nums.length; i++) {
			// Remove the element going out of the window
			int outNum = nums[i - k];
			map.put(outNum, map.get(outNum) - 1);
			if (map.get(outNum) == 0) {
				map.remove(outNum);
			}
			// Add the new element coming into the window
			int inNum = nums[i];
			map.put(inNum, map.getOrDefault(inNum, 0) + 1);
			// Max is always the last key
			result.add(map.lastKey());
		}
		return result;
	}
	public static void main(String[] args) {
		SlidingWindowMaximum sw = new SlidingWindowMaximum();
		int[] nums = { 1, 3, -1, -3, 5, 3, 6, 7 };
		int k = 3;
		System.out.println(sw.maxSlidingWindow(nums, k)); // Output: [3, 3, 5, 5, 6, 7]
	}
}
```

Array:

```java
int[] nums = {3, 3, 5};
```

You want to count how many times each number appears.

Step-by-step breakdown:

```
Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++) {
   int num = nums[i];
   map.put(num, map.getOrDefault(num, 0) + 1);
}
```

Iteration 1: num = 3

- map.getOrDefault(3, 0) → not found, so return 0
- 0 + 1 = 1
- map.put(3, 1) → map = {3=1}

Iteration 2: num = 3 again

- map.getOrDefault(3, 0) → found 1
- 1 + 1 = 2
- map.put(3, 2) → map = {3=2}

Iteration 3: num = 5

- map.getOrDefault(5, 0) → not found, return 0
- 0 + 1 = 1
- map.put(5, 1) → map = {3=2, 5=1}

map = {3=2, 5=1}

getOrDefault(num, 0) gives us the **current count**, or 0 if it doesn't exist.

We do +1 because we've just seen that number again.

put(num, newCount) updates the map.

A TreeMap<K, V> in Java is a **sorted map** — it keeps its keys in **ascending order**.

Let’s say inside the window your TreeMap looks like this:

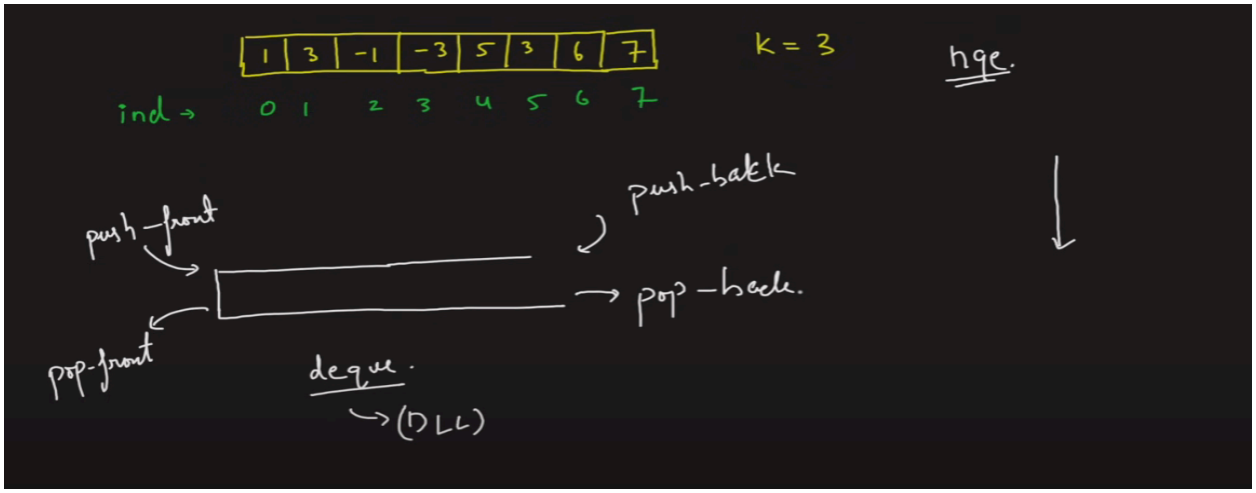
map = {1=1, 3=1, 5=1}

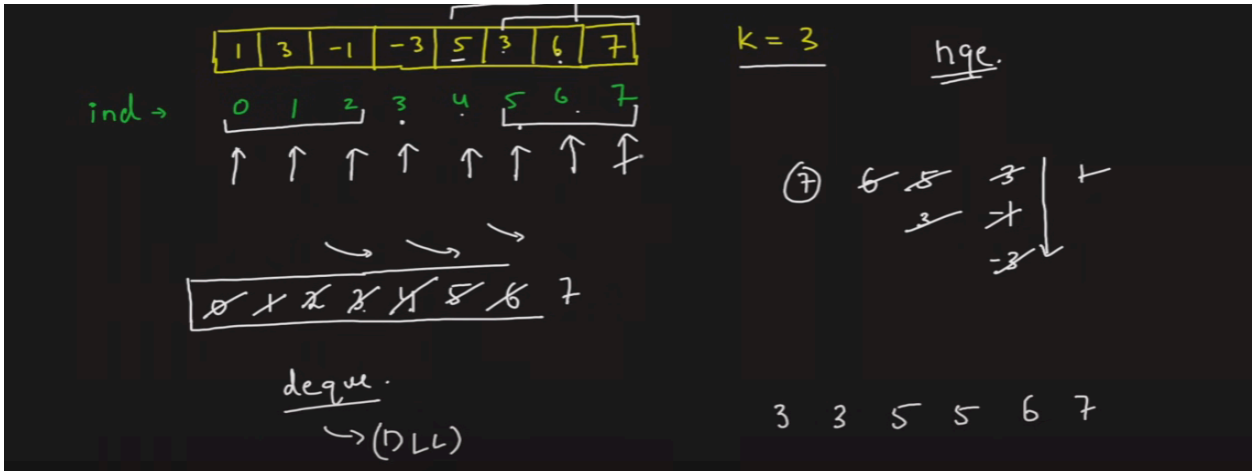
map.lastKey() → 5

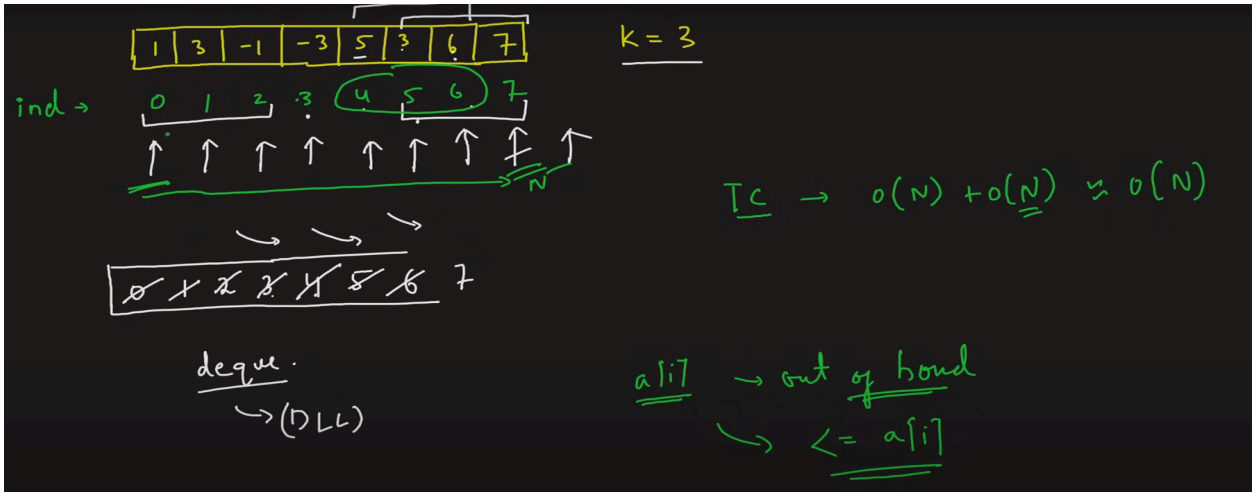
nums = [1, 3, -1, -3, 5], k = 3
First window (handled before this loop): [1, 3, -1] → map = { -1=1, 1=1, 3=1 }
Now i = 3 → window becomes [3, -1, -3]
outNum = nums[0] = 1
Remove 1 from the map by decreasing its count

Optimized Solution

Using dequeue







Steps (Using Deque):

1. Use a deque to store **indices** of elements, not the elements themselves.

2. Ensure elements in the deque are **in decreasing order** of values (the front always has the max of the current window).
3. For each index i in nums:
    - **Remove out-of-bound indices** (i.e., those outside the current window [i - k + 1, i]).
    - **Remove smaller elements** from the back since they can never be the max if a larger element comes.
    - **Add current index to the deque**.
    - **Add the max** (element at front of deque) to the result once the first window of size k is formed.

Time and Space Complexity

- **Time Complexity**: O(N)
    - Each element is **added and removed at most once** from the deque.
- **Space Complexity**: O(k)
    - The deque stores up to k indices.

```
package com.leetcode150;

import java.util.*;
public class SlidingWindowMaximum {
```

```
    public int[] maxSlidingWindow(int[] nums, int k) {
        if (nums == null || k == 0)
            return new int[0];
        int n = nums.length;
        int[] result = new int[n - k + 1];
        Deque<Integer> dq = new ArrayDeque<>();
        for (int i = 0; i < n; i++) {
            // Remove indices out of the current window
            while (!dq.isEmpty() && dq.peekFirst() < i - k + 1) {
                dq.pollFirst();
            }
            // Remove smaller values from back
            while (!dq.isEmpty() && nums[dq.peekLast()] < nums[i]) {
                dq.pollLast();
            }
            // Add current index
            dq.offerLast(i);
            // Add max to result when window is fully within range
            if (i >= k - 1) {
                result[i - k + 1] = nums[dq.peekFirst()];
            }
        }
        return result;
    }

    public static void main(String[] args) {
        SlidingWindowMaximum sw = new SlidingWindowMaximum();
        int[] nums = { 1, 3, -1, -3, 5, 3, 6, 7 };
        int k = 3;
        int[] result = sw.maxSlidingWindow(nums, k);
        System.out.println(Arrays.toString(result));
    }
}
```

Output

[3, 3, 5, 5, 6, 7]

Example Input:

nums = [1, 3, -1, -3, 5, 3, 6, 7]

k = 3

We need to find the **maximum in each sliding window** of size 3

Step-by-step Execution:

🔁 **i = 0 → nums[0] = 1**

- dq = [0]
  (No window of size k yet)

🔁 **i = 1 → nums[1] = 3**

- Remove 1 from back (since 3 > 1)
- dq = [1]

🔁 **i = 2 → nums[2] = -1**

- No removal (since -1 < 3)
- dq = [1, 2]
- Now i >= k - 1, so record result:
  result[0] = nums[1] = 3

✅ First window [1, 3, -1] → max = **3**

🔁 **i = 3 → nums[3] = -3**

- dq front (1) still in window (1 >= 1)
- -3 < -1, so keep both
- dq = [1, 2, 3]
- Window is valid → result[1] = nums[1] = 3

✅ Second window [3, -1, -3] → max = **3**

🔁 **i = 4 → nums[4] = 5**

- Remove from front: 1 (index 1) is out of range (i - k + 1 = 2)
- Remove from back: -3 and -1 (both < 5)
- dq = [4]
- Window is valid → result[2] = nums[4] = 5

✅ Third window [-1, -3, 5] → max = **5**

🔁 **i = 5 → nums[5] = 3**

- No need to remove front
- 3 < 5, so just add
- dq = [4, 5]
- Window valid → result[3] = nums[4] = 5

✅ Fourth window [-3, 5, 3] → max = **5**

🔁 **i = 6 → nums[6] = 6**

- Remove back: 3 < 6, 5 < 6
- dq = [6]
- Window valid → result[4] = nums[6] = 6

✅ Fifth window [5, 3, 6] → max = **6**

🔁 **i = 7 → nums[7] = 7**

- Remove back: 6 < 7
- dq = [7]
- Window valid → result[5] = nums[7] = 7

✅ Sixth window [3, 6, 7] → max = **7**

✅ **Final Output:**

result = [3, 3, 5, 5, 6, 7]

Trees

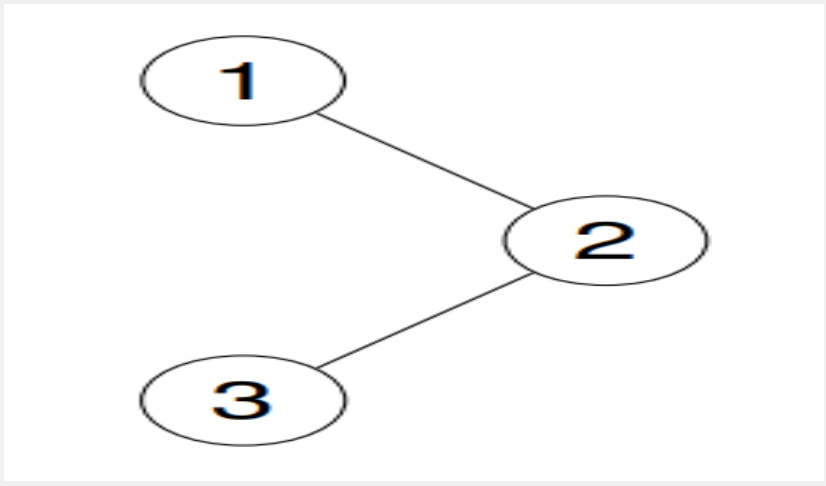
36.Binary Tree Inorder Traversal

Given the root of a binary tree, return *the inorder traversal of its nodes' values*.

Example 1:

Input: root = [1,null,2,3]

Output: [1,3,2]

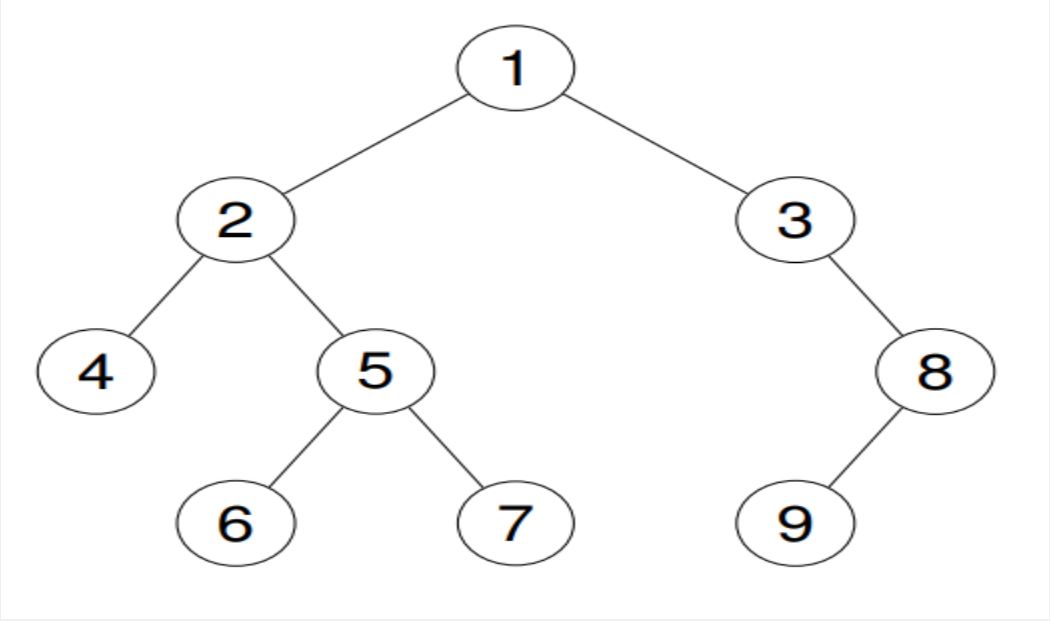


Example 2:

Input: root = [1,2,3,4,5,null,8,null,null,6,7,9]

Output: [4,2,6,5,7,1,3,9,8]

Explanation:



Example 3:

Input: root = []

Output: []

Example 4:

Input: root = [1]

Output: [1]

Above one is

What kind of tree is it then?

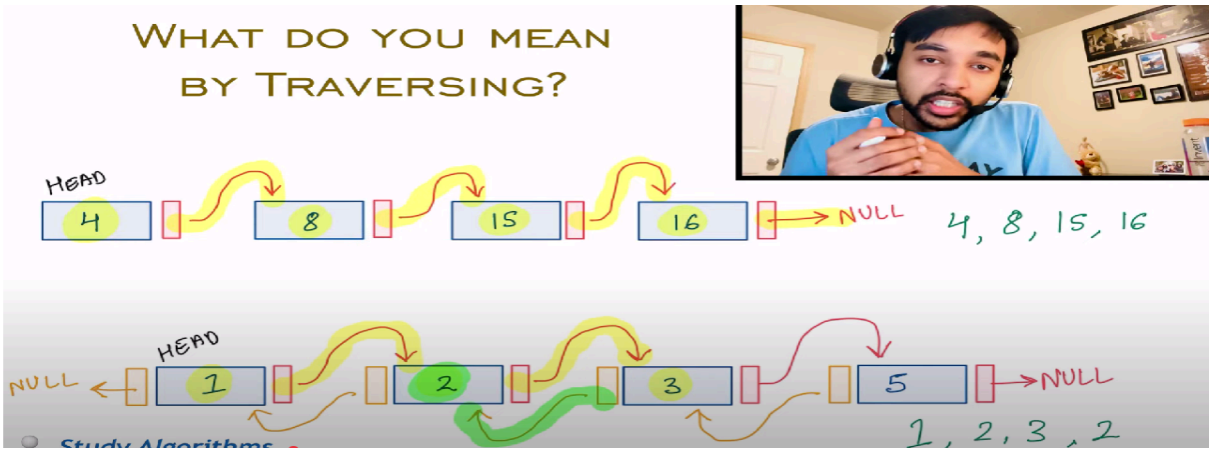
It's just a **binary tree**, meaning:

- Each node can have **at most 2 children**
- But no ordering rules like a BST(Left subtree values < node ,Right subtree values > node).

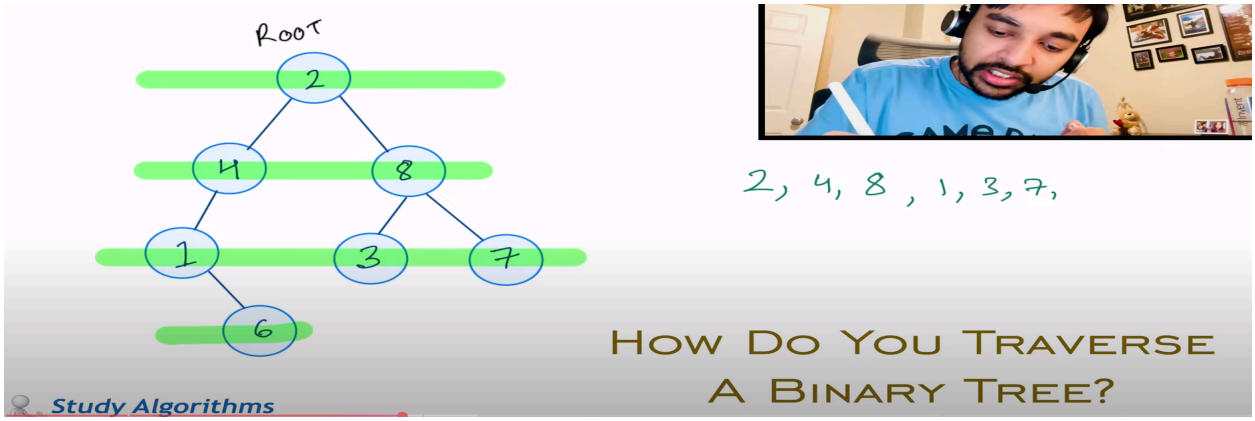
Inorder (Left → Root → Right)

Preorder (Root → Left → Right)

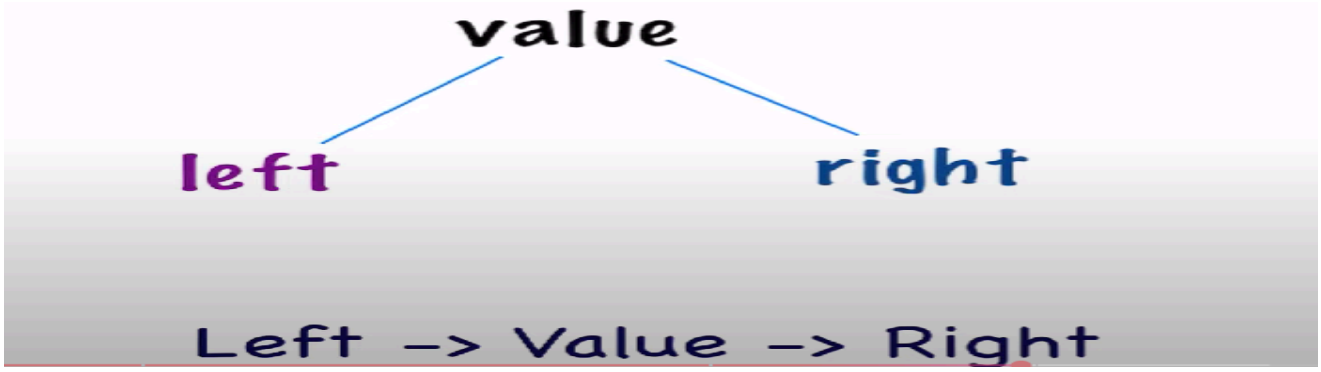
Postorder (Left → Right → Root)



Level order traversing



Inorder (Left → Root → Right)



General Solutions

Sol 1:recursive approach

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList();
        helper(root, list);
        return list;
    }
    public void helper(TreeNode root, List<Integer> list) {
        if(root != null) {
            helper(root.left, list);
            list.add(root.val);
            helper(root.right, list);
        }
    }
}
```

```
package com.leetcode150;
import java.util.ArrayList;
import java.util.List;
//Definition for a binary tree node.
class TreeNode {
	int val;
	TreeNode left;
	TreeNode right;
	TreeNode() {
	}
	TreeNode(int val) {
		this.val = val;
	}
	TreeNode(int val, TreeNode left, TreeNode right) {
		this.val = val;
		this.left = left;
		this.right = right;
	}
}
public class RecursiveInorderTraversal {
	public static void main(String[] args) {
		// Create tree: [1, null, 2, 3]
		TreeNode root = new TreeNode(1);
```

```
        root.right = new TreeNode(2);
        root.right.left = new TreeNode(3);
        List<Integer> result = inorderTraversal(root);
        System.out.println(result); // Output should be [1, 3, 2]
    }
    public static List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList<>();
        helper(root, list);
        return list;
    }
    public static void helper(TreeNode root, List<Integer> list) {
        if (root != null) {
            helper(root.left, list); // 1. Go to left child
            list.add(root.val); // 2. Visit current node
            helper(root.right, list); // 3. Go to right child
        }
    }
}
```

[1, 3, 2]

Step-by-Step Execution:

- Start from the root.
- Traverse left subtree recursively.
- When the left is null, add the current node to the result list.
- Traverse right subtree recursively.
- Backtrack automatically through recursion.

```
1
 \
  2
 /
3
```

Traversal:

Go left from 1 → null → add 1

Go right to 2 → go left to 3 → left null → add 3 → right null

Back to 2 → add 2

Output: [1, 3, 2]

Time & Space Complexity:

- **Time:** O(n)
- **Space:** O(n) (due to recursion + result list)

Sol 2:Using Stack **(optimized)**

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList();
        Stack<TreeNode> stack = new Stack();
        TreeNode current = root;
        while(current != null || !stack.isEmpty()) {
            while(current != null) {
                stack.push(current);
                current = current.left;
            }
            current = stack.pop();
            list.add(current.val);
            current = current.right;
        }
        return list;
    }
}
```

```
package com.leetcode150;

import java.util.ArrayList;
import java.util.List;
import java.util.Stack;
//Definition for a binary tree node.
class TreeNode {
	int val;
	TreeNode left;
	TreeNode right;
	TreeNode() {
	}
	TreeNode(int val) {
		this.val = val;
	}
	TreeNode(int val, TreeNode left, TreeNode right) {
		this.val = val;
```

```
            this.left = left;
            this.right = right;
        }
}
public class RecursiveInorderTraversal {
        public static void main(String[] args) {
            // Create tree: [1, null, 2, 3]
            TreeNode root = new TreeNode(1);
            root.right = new TreeNode(2);
            root.right.left = new TreeNode(3);
            List<Integer> result = inorderTraversal(root);
            System.out.println(result); // Output should be [1, 3, 2]
        }
        public static List<Integer> inorderTraversal(TreeNode root) {
            List<Integer> list = new ArrayList<>();
            Stack<TreeNode> stack = new Stack<>();
            TreeNode current = root;
            while (current != null || !stack.isEmpty()) {
                while (current != null) {
                    stack.push(current); // 1. Go left, push to stack
                    current = current.left;
                }
                current = stack.pop(); // 2. Pop from stack
                list.add(current.val); // 3. Visit node
                current = current.right; // 4. Move right
            }
            return list;
        }
        }
```

[1, 3, 2]

Step-by-Step Execution:

1. Start from root, go as left as possible and push each node to the stack.
2. When you hit null, pop from the stack.
3. Add node's value to the result list.
4. Move to its right subtree and repeat.

```
1
 \
  2
 /
3
```

Traversal (with stack):

- Push 1 → go right
- Push 2 → go left
- Push 3 → left null
- Pop 3 → add to list → right null
- Pop 2 → add to list → right null
- Pop 1 → add to list

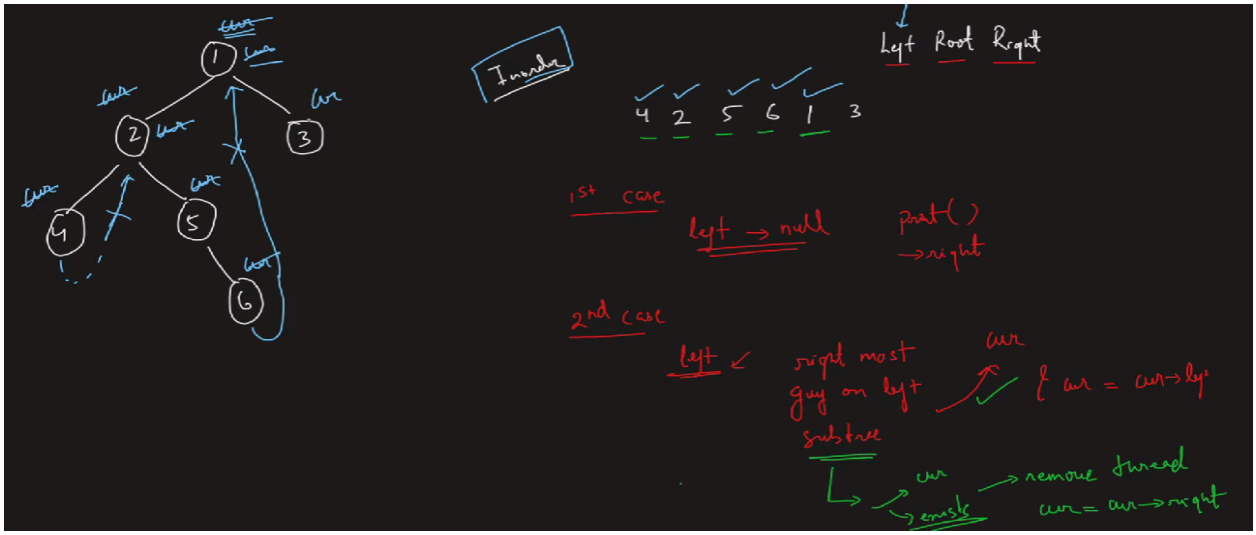
Output: **[1, 3, 2]**

Time & Space Complexity:

- **Time:** O(n)
- **Space:** O(n) (for stack + result list)

Optimized Solution

Morris Traversal is a technique to traverse a binary tree (like Inorder, Preorder) **without using recursion or a stack**, and in **O(1) space**.
It temporarily modifies the tree during traversal using a concept called **threading**, and restores it afterward.



```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        TreeNode current = root;
        while (current != null) {
            if (current.left == null) {
                result.add(current.val);
                current = current.right;
            } else {
                // Find the inorder predecessor of current
                TreeNode predecessor = current.left;
                while (predecessor.right != null && predecessor.right != current) {
                    predecessor = predecessor.right;
                }
                // Make current as the right child of its inorder predecessor
                if (predecessor.right == null) {
                    predecessor.right = current;
                    current = current.left;
                } else {
                    // Revert the changes made (remove the thread)
```

```
                predecessor.right = null;
                result.add(current.val);
                current = current.right;
            }
        }
    }
    return result;
  }
}
```

Two Key Cases in Morris Traversal

✅ **1st Case: left == null**

- If the current node has **no left child**:
  - Visit (print/store) the current node.
  - Move to the **right child**.

Diagram Highlights:

- The note on the right: left -> null then print() → right.

🔁 **2nd Case: left != null**

- If the current node **has a left child**:
  - Find the **rightmost node** in the left subtree (called the **inorder predecessor**).
  - If the predecessor's right is null:
    - Create a **temporary thread** from the predecessor to current.
    - Move current to its left child.
  - Else:
    - Thread already exists:
      - Remove the thread.
        Visit current.
      - Move to the right child.

Time Complexity: O(n)

- Each node is visited at most **twice**.

Space Complexity: O(1)

- No recursion or stack – **in-place** traversal using temporary threading.

Example Tree

Let's consider this binary tree:

This tree in inorder (LNR: Left → Node → Right) should give:
➡️ **[4, 2, 5, 1, 3]**

Morris Traversal Steps

Step 1: Start at root (1)

- Left is not null → find inorder predecessor (rightmost of 2)
  - Predecessor = 5
- Thread 5's right to current (1)
  👉 Move to current = 2

Step 2: At node 2

- Left is not null → find inorder predecessor (rightmost of 4)
  - Predecessor = 4
- Thread 4's right to current (2)
  👉 Move to current = 4

Step 3: At node 4

- Left is null
  ✅ Visit: result = [4]
  👉 Move to current = 2 (because of thread from 4 → 2)

Step 4: Back at node 2

- Predecessor (4)'s right is current (2) → remove thread
  ✅ Visit: result = [4, 2]
  👉 Move to current = 5

Step 5: At node 5

- Left is null
  ✅ Visit: result = [4, 2, 5]
  👉 Move to current = 1 (because of thread from 5 → 1)

Step 6: Back at node 1

- Predecessor (5)'s right is current (1) → remove thread
  ✅ Visit: result = [4, 2, 5, 1]
  👉 Move to current = 3

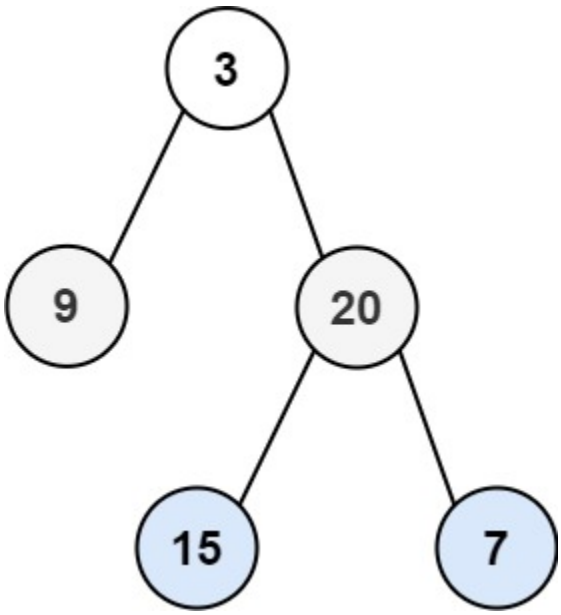
  Step 7: At node 3
- Left is null
  ✅ Visit: result = [4, 2, 5, 1, 3]
  👉 Move to current = null

Final Output:

[4, 2, 5, 1, 3]

37.Binary Tree Level Order Traversal

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).



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

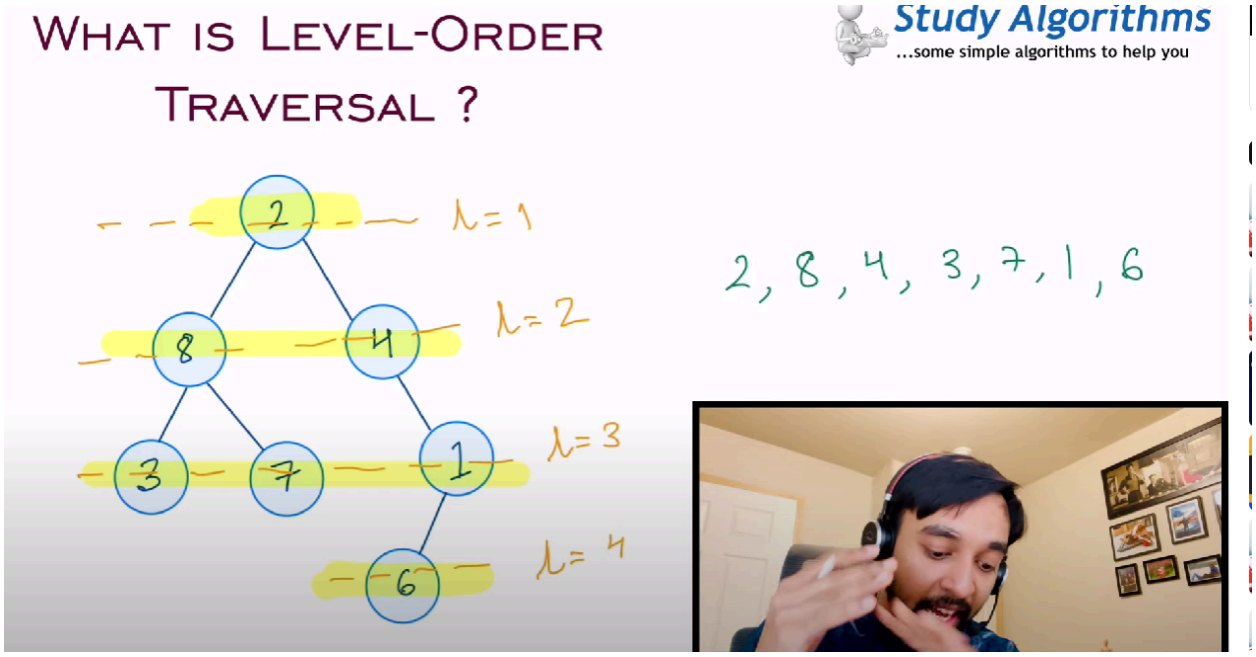
Output: [[3],[9,20],[15,7]]

Example 2:

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

Example 3:

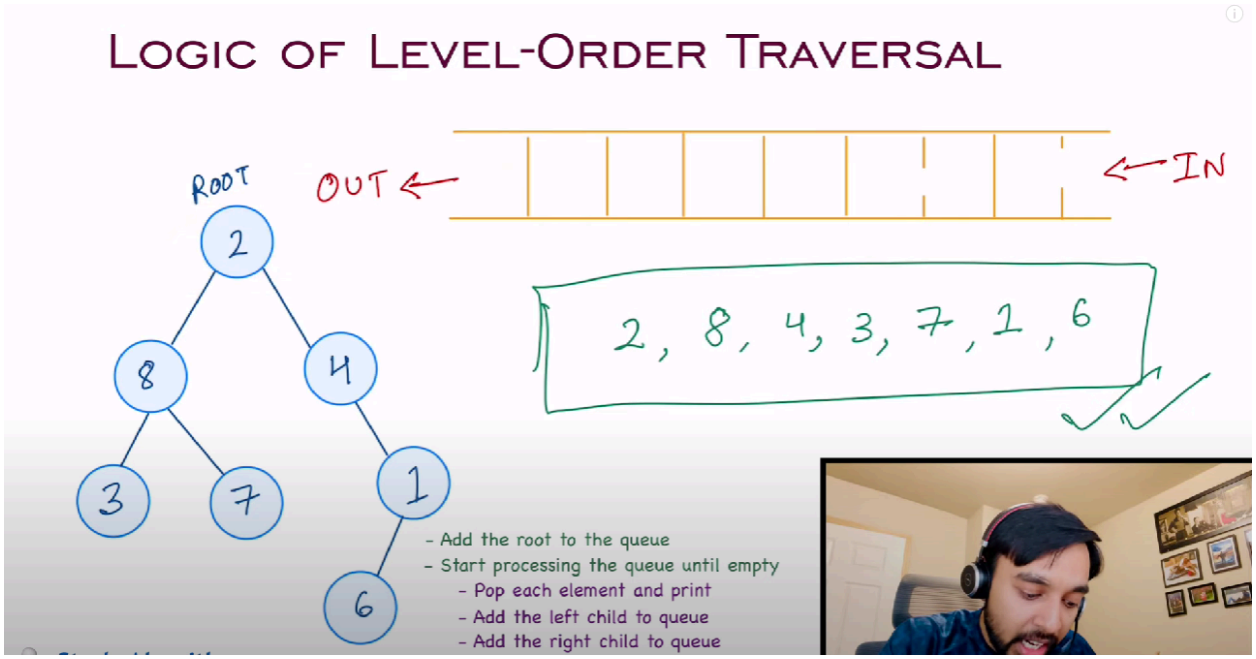
Input: root = []
Output: []

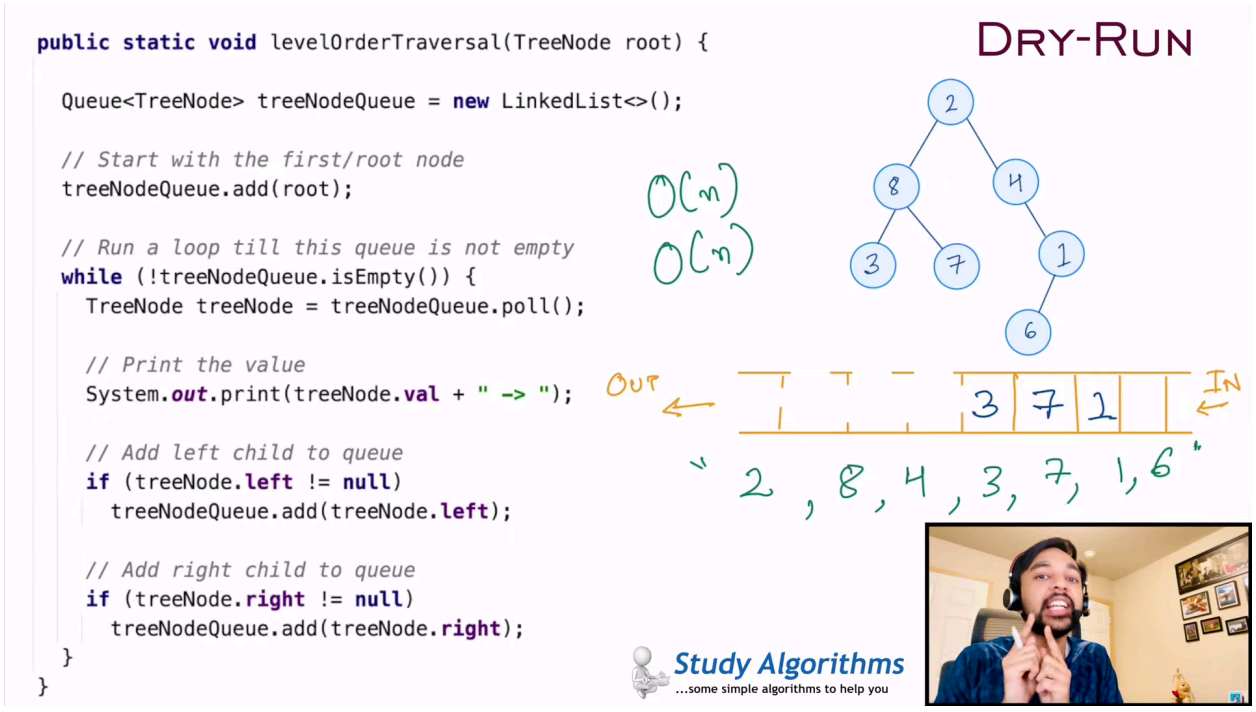


The Nature of Level Order Traversal

Level order traversal means:

- Visiting nodes **level by level**, from **left to right**.
- We must visit a node **before** visiting its children.
- It follows a **First-In-First-Out (FIFO)** process:
    - First, visit the root.
    - Then its children.
    - Then the children's children.
    - And so on...





Steps:

1. **Initialize**:

- Create an empty list result to store each level.
- Create a Queue<TreeNode> and add the root node to it.

2. **Process the tree level by level**:
    - While the queue is not empty:
        - Determine the number of nodes in the current level:
          int levelSize = queue.size();
        - Create a temporary list currentLevel to store values at this level.

3. **Process each node in the current level**:
    - Loop levelSize times:
        - Remove the front node from the queue:
          TreeNode current = queue.poll();
        - Add the node's value to the currentLevel list.
        - If the node has a **left child**, add it to the queue.
        - If the node has a **right child**, add it to the queue.

4. **Save the level's values**:
    - After processing all nodes of the current level, add currentLevel to result.

5. **Return the final result**:
    - After all levels are processed, return the result list.

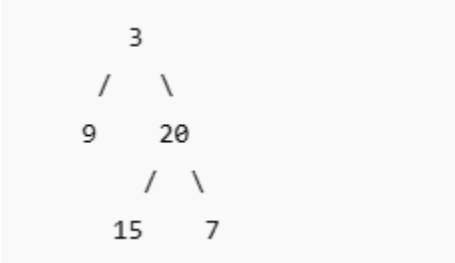
```
package com.leetcode150;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
class TreeNodee {
        int val;
        TreeNodee left;
        TreeNodee right;
        TreeNodee() {
        }
        TreeNodee(int val) {
                this.val = val;
        }
```

```
        TreeNodee(int val, TreeNodee left, TreeNodee right) {
                this.val = val;
                this.left = left;
                this.right = right;
        }
}
public class BinaryTreeLevelOrderTraversal {
        public static void main(String[] args) {
                TreeNodee root = new TreeNodee(3);
                root.left = new TreeNodee(9);
                root.right = new TreeNodee(20);
                root.right.left = new TreeNodee(15);
                root.right.right = new TreeNodee(7);
                List<List<Integer>> result = levelOrder(root);
                System.out.println(result); // Output: [[3], [9, 20], [15, 7]]
        }
        public static List<List<Integer>> levelOrder(TreeNodee root) {
                List<List<Integer>> result = new ArrayList<>();
                if (root == null)
                        return result;
                Queue<TreeNodee> queue = new LinkedList<>();
                queue.add(root);
                while (!queue.isEmpty()) {
                        int levelSize = queue.size(); // Number of nodes at current level
                        List<Integer> currentLevel = new ArrayList<>();
                        for (int i = 0; i < levelSize; i++) {
                                TreeNodee current = queue.poll();
                                currentLevel.add(current.val);
                                if (current.left != null)
                                        queue.add(current.left);
                                if (current.right != null)
                                        queue.add(current.right);
                        }
                        result.add(currentLevel);
                }
                return result;
        }
}
```

Output

[[3], [9, 20], [15, 7]]

Time Complexity: O(n) – each node is visited once.
Space Complexity: O(n) – for the queue and result list.



Step-by-Step Execution of levelOrder(root)

Step 1: Initialization

- result = [] (final answer)
- queue = [3] (start with the root node)

Step 2: First level

- levelSize = queue.size() = 1
- currentLevel = []
- Loop runs once (i = 0)
    - current = queue.poll() → 3
    - Add 3 to currentLevel → [3]
    - current.left = 9 → add to queue
    - current.right = 20 → add to queue
- queue = [9, 20]
- Add [3] to result → result = [[3]]

Step 3: Second level

- levelSize = queue.size() = 2
- currentLevel = []
- Loop runs twice:
    - i = 0
        - current = queue.poll() → 9

        - Add 9 to currentLevel
        - No left/right → nothing added to queue

    - i = 1

        - current = queue.poll() → 20
        - Add 20 to currentLevel
        - current.left = 15 → add to queue
        - current.right = 7 → add to queue

- queue = [15, 7]
- Add [9, 20] to result → result = [[3], [9, 20]]

Step 4: Third level

- levelSize = 2
- currentLevel = []
- Loop runs twice:

    - i = 0
        - current = queue.poll() → 15
        - Add 15 to currentLevel

    - i = 1

        - current = queue.poll() → 7
        - Add 7 to currentLevel

- queue = [] (empty)
- Add [15, 7] to result → result = [[3], [9, 20], [15, 7]

Final Output:

[[3], [9, 20], [15, 7]]

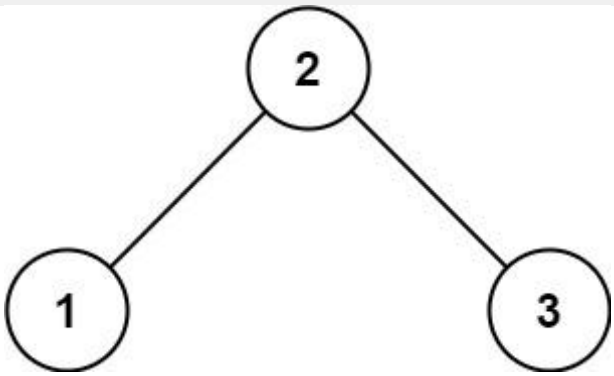
38.Validate Binary Search Tree

Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:



Input: root = [2,1,3]
Output: true

✅ **Example 1:** `root = [2,1,3]`

Tree:

markdown Copy Edit

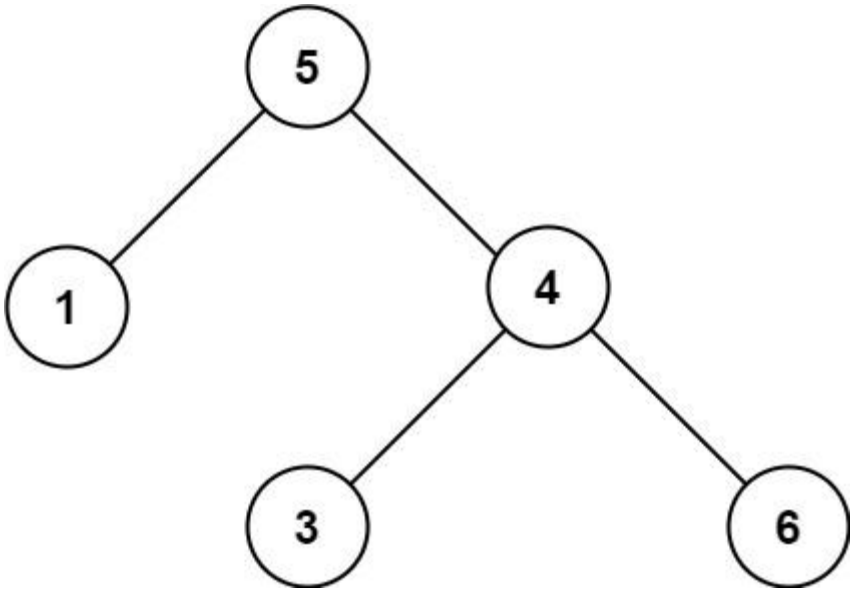
```
   2
  / \
 1   3
```

Validation Steps:

- Node `2`:
  - Left child `1` < `2` ✔
  - Right child `3` > `2` ✔
- Node `1` has no children ✔
- Node `3` has no children ✔

✅ **Result: true**

Example 2:



Input: root = [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right child's value is 4.

❌ **Example 2:** `root = [5,1,4,null,null,3,6]`

Tree:

markdown Copy Edit

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

Validation Steps:

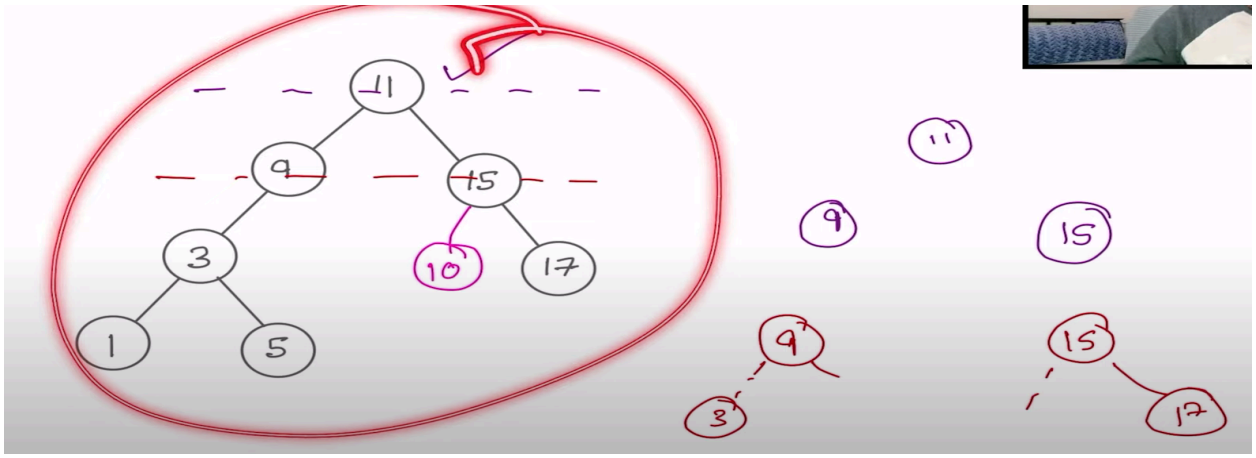
- Node `5`:
  - Left child `1` < `5` ✔
  - Right child `4` > `5` ❌ ← **Violation here**

Even though `4` is a right child of `5`, it's **not greater than 5**, which **violates the BST rule**.

❌ **Result: false**

Feature	Binary Tree	Binary Search Tree (BST)
Max children per node	2	2
Value ordering	No rules	Left < Root < Right
Use case	General structure	Efficient searching/sorting
Duplicate values allowed?	Yes (depending)	Usually not (or one side only)

"We don’t use level order traversal basically, by level wise it will check but overall way it won't be applicable."



Optimized solution

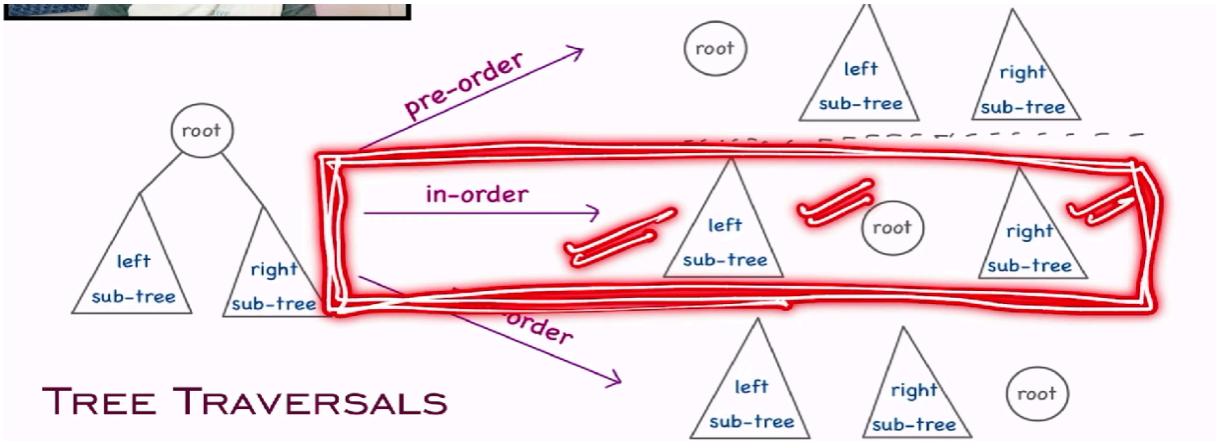
We **can use in-order traversal to check whether a binary tree is a BST (Binary Search Tree)**

In a **Binary Search Tree (BST)**:

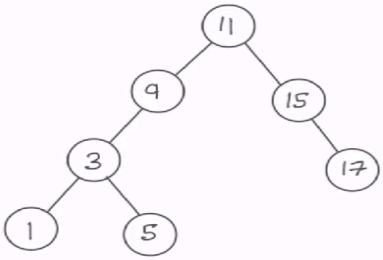
- The **left subtree** has values **less than** the node.
- The **right subtree** has values **greater than** the node.
- So, an **in-order traversal of a BST always produces a sorted array (in ascending order)**.

Approach:

1. **Do an in-order traversal** of the tree.
2. **Keep track of the previously visited node's value**.
3. If at any point the current node's value is **less than or equal to the previous**, it's **not a BST**.

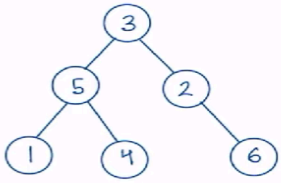


VERIFYING IN-ORDER RESULTS



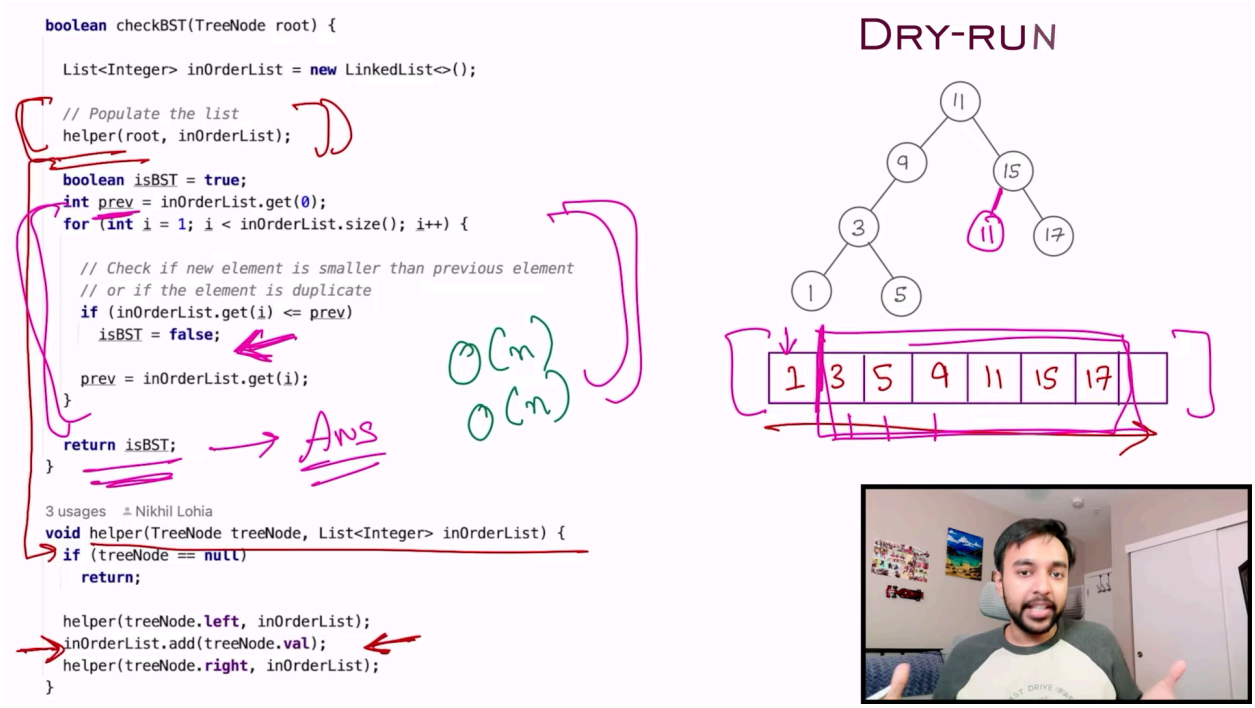
YES

1, 3, 5, 9, 11, 15, 17



NO

1, 5, 4 , 3 , 2, 6



Perform In-order Traversal

- The helper method recursively traverses the tree in the order:
  left → node → right
- It collects values in inOrderList.

Check Sorted Order

After traversal, inOrderList should be strictly increasing.

- A loop checks whether each element is greater than the previous one.
- If not, isBST = false.

```
package com.leetcode150;
import java.util.LinkedList;
import java.util.List;
class TreeNodeee {
        int val;
        TreeNodeee left;
        TreeNodeee right;
        TreeNodeee() {
        }
        TreeNodeee(int val) {
```

```
		this.val = val;
	}
	TreeNodeee(int val, TreeNodeee left, TreeNodeee right) {
		this.val = val;
		this.left = left;
		this.right = right;
	}
}
public class ValidateBinarySearchTree {
	public static void main(String[] args) {
		TreeNodeee root = new TreeNodeee(9);
		root.left = new TreeNodeee(6);
		root.right = new TreeNodeee(20);
		root.right.left = new TreeNodeee(19);
		root.right.right = new TreeNodeee(60);
		System.out.println(checkBST(root));
	}
	public static boolean checkBST(TreeNodeee root) {
		List<Integer> inOrderList = new LinkedList<>();
		// Populate the list with in-order traversal
		helper(root, inOrderList);
		boolean isBST = true;
		int prev = inOrderList.get(0);
		for (int i = 1; i < inOrderList.size(); i++) {
			// Check if new element is smaller than previous element
			// or if the element is duplicate
			if (inOrderList.get(i) <= prev)
				isBST = false;
			prev = inOrderList.get(i);
		}
		return isBST;
	}
	public static void helper(TreeNodeee treeNode, List<Integer> inOrderList) {
		if (treeNode == null)
			return;
		helper(treeNode.left, inOrderList);
		inOrderList.add(treeNode.val);
		helper(treeNode.right, inOrderList);
	}
}
```

Output

true

Total Space Complexity: O(n)

Space (list + stack) : O(n)

Let's see how the recursive helper method builds the inOrderList.

Let's imagine inOrderList = [] at the beginning.

Starting with root node 11

- Go to helper(11.left) → i.e., helper(9)
    - Go to helper(9.left) → i.e., helper(3)
        - Go to helper(3.left) → i.e., helper(1)
            - 1.left is null → return
            - Add 1 → inOrderList = [1]
            - 1.right is null → return
        - Back to node 3, add 3 → inOrderList = [1, 3]
        - Go to helper(3.right) → i.e., helper(5)
            - 5.left is null → return
            - Add 5 → inOrderList = [1, 3, 5]
            - 5.right is null → return
    - Back to node 9, add 9 → inOrderList = [1, 3, 5, 9]

    - Go to helper(9.right) → i.e., helper(10)
        - 10.left is null → return
        - Add 10 → inOrderList = [1, 3, 5, 9, 10]
        - 10.right is null → return

- Back to root 11, add 11 → inOrderList = [1, 3, 5, 9, 10, 11]
- Go to helper(11.right) → i.e., helper(15)
    - 15.left is null → return
    - Add 15 → inOrderList = [1, 3, 5, 9, 10, 11, 15]
    - Go to helper(15.right) → i.e., helper(17)
        - 17.left is null → return
        - Add 17 → inOrderList = [1, 3, 5, 9, 10, 11, 15, 17]
        - 17.right is null → return

inOrderList:

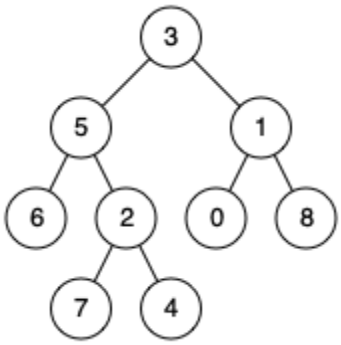
[1, 3, 5, 9, 10, 11, 15, 17]

39.Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

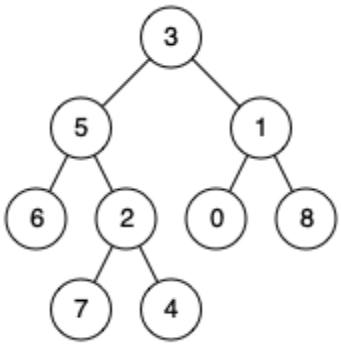
According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**)."

Example 1:



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.

Example 2:



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

Example 3:

Input: root = [1,2], p = 1, q = 2
Output: 1

Sol:

Lowest Common Ancestor (LCA) is the **first node where the paths from both p and q to the root intersect (or meet)** when moving **up** the tree.

For ex: Think of it like two people climbing up a tree from different branches — the **first common point where their paths cross** going **upward** is the **LCA**.

```
        3
       / \
      5   1
     / \ / \
    6  2 0  8
      / \
     7   4
```

Now let's say you want to find the **LCA of nodes 7 and 4**.

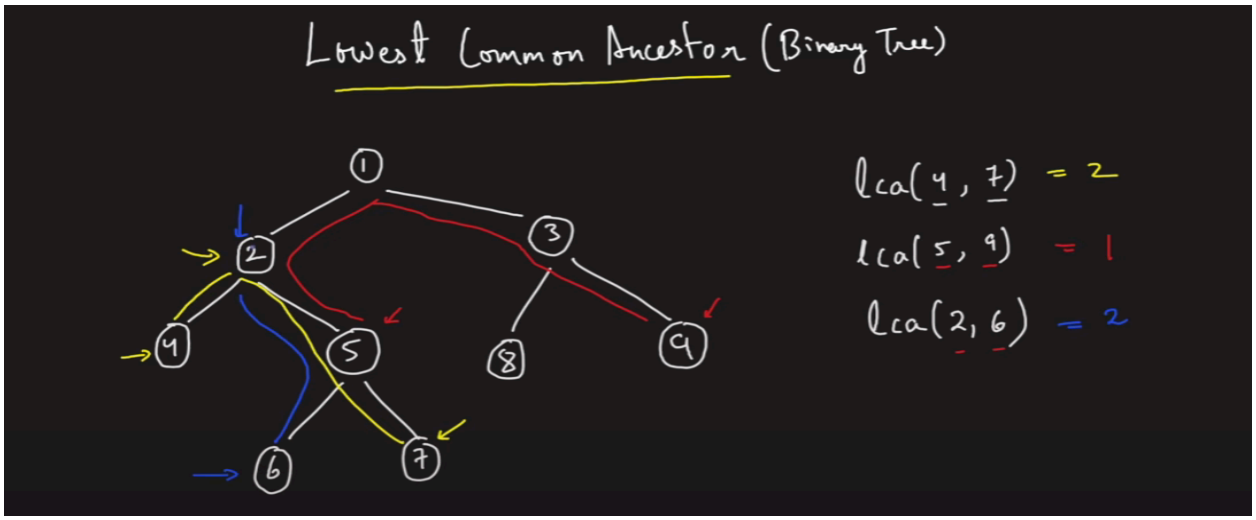
- Go from 7 up: 7 → 2 → 5 → 3
- Go from 4 up: 4 → 2 → 5 → 3

They **first meet at node 2** → So LCA of 7 and 4 is **2**.

LCA of 6 and 4

- 6 path: 6 → 5 → 3
- 4 path: 4 → 2 → 5 → 3

They first meet at **5**, so LCA is 5.



General solution

brute force

For nodes 4 and 7, it:

Builds paths from root to each node:

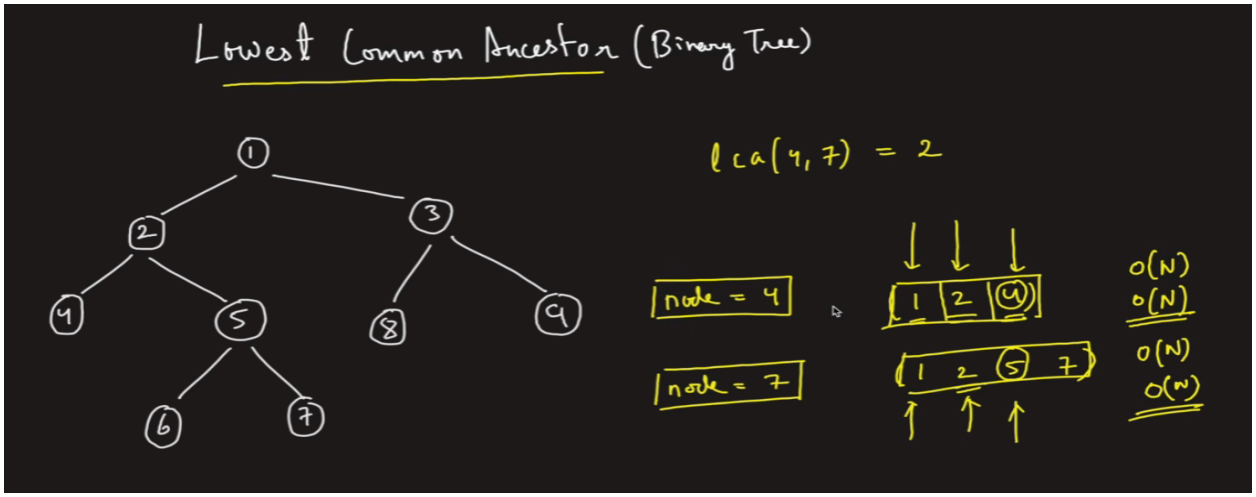
- Path to 4: 1 → 2 → 4
- Path to 7: 1 → 2 → 5 → 7

Then **compare both paths** to find the last common node (2 in this case) → **that's the LCA**.

📈 Time and Space Complexity of This Approach

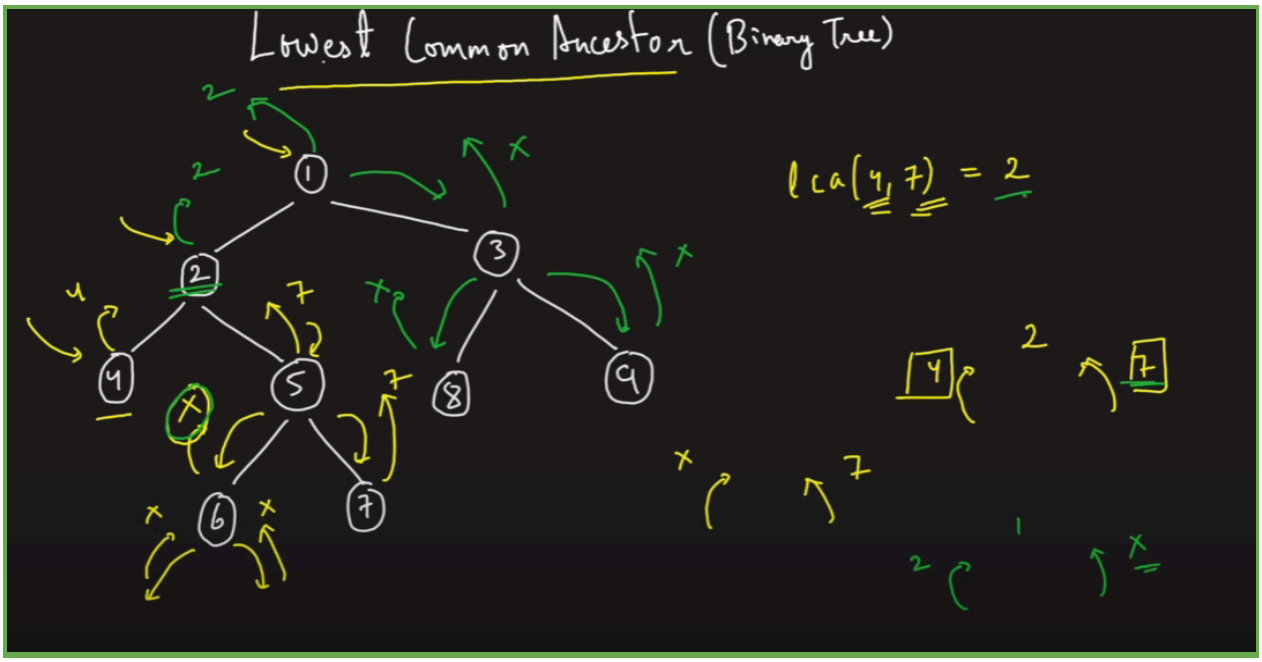
Operation	Complexity
Finding path for node 4	O(N)
Finding path for node 7	O(N)
Comparing paths	O(N)
Total Time	**O(N)**
Space (for paths)	**O(N)**

> So while it's a valid and intuitive solution, it's not optimal in terms of space, especially for large trees.

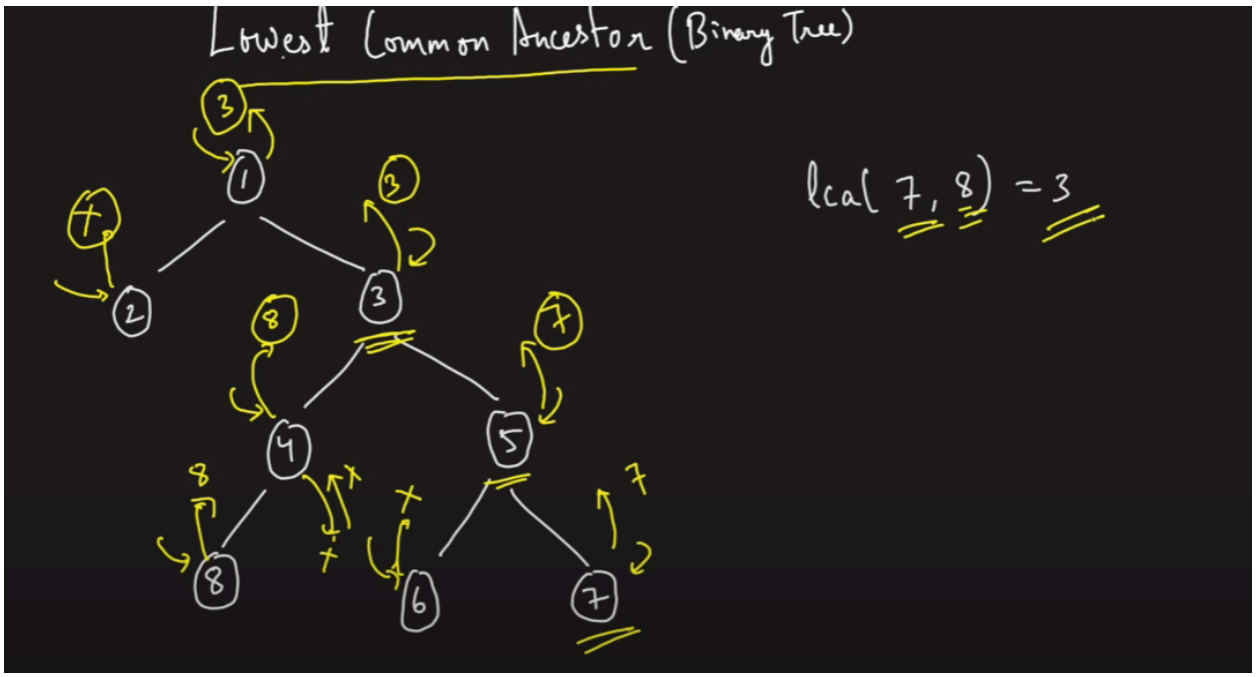


Optimized solution

Optimal DFS-based approach to find the **Lowest Common Ancestor (LCA)** in a binary tree.



Ex2



You're **recursively traversing** the tree using **DFS**.

Each recursive call checks if the current node is either:

- null
- p (node 4)
- q (node 7)

As the recursion **unwinds**, we're:

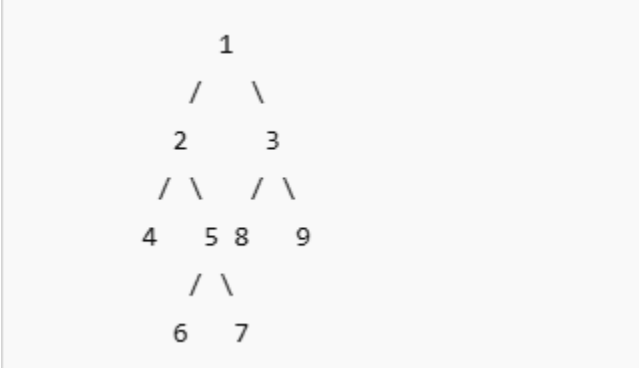
- Returning the node if found (4 or 7),
- Propagating it back up to the parent,
- Identifying the **lowest node** where both left and right return a value — this node is the **LCA**.

```
package com.leetcode150;
class TreeNod {
	int val;
	TreeNod left;
	TreeNod right;
	TreeNod(int x) {
		val = x;
	}
}
public class LowestCommonAncestorofaBinaryTree {
	public static void main(String[] args) {
		// Build the tree from the image
		TreeNod root = new TreeNod(1);
		root.left = new TreeNod(2);
		root.right = new TreeNod(3);
		root.left.left = new TreeNod(4);
		root.left.right = new TreeNod(5);
		root.right.left = new TreeNod(8);
		root.right.right = new TreeNod(9);
		root.left.right.left = new TreeNod(6);
		root.left.right.right = new TreeNod(7);
		TreeNod p = root.left.left; // Node 4
		TreeNod q = root.left.right.right; // Node 7
		TreeNod lca = lowestCommonAncestor(root, p, q);
		System.out.println("LCA of " + p.val + " and " + q.val + " is: " + lca.val);
```

```
    }
    public static TreeNod lowestCommonAncestor(TreeNod root, TreeNod p, TreeNod q) {
        // Base case
        if (root == null || root == p || root == q) {
            return root;
        }
        // Recursive DFS search
        TreeNod left = lowestCommonAncestor(root.left, p, q);
        TreeNod right = lowestCommonAncestor(root.right, p, q);
        // Combine results
        if (left == null) {
            return right;
        } else if (right == null) {
            return left;
        } else {
            // Both left and right are not null, so root is the LCA
            return root;
        }
    }
}
```

Output

LCA of 4 and 7 is: 2



Step-by-Step Code Flow

lowestCommonAncestor(root=1, p=4, q=7)

Step 1: Node = 1

- root != null, root != p, root != q

Recurse left and right:

left = LCA(2, 4, 7)
right = LCA(3, 4, 7)

Step 2: Node = 2

- root = 2 → not null, not 4 or 7

Recurse:
left = LCA(4, 4, 7)
right = LCA(5, 4, 7)

Step 3: Node = 4

- root == p (4) → return 4

Step 4: Node = 5

Not 4 or 7

Recurse:
left = LCA(6, 4, 7)
right = LCA(7, 4, 7)

Step 5: Node = 6

- Not 4 or 7 → recurse into null
- Left and Right are null → return null

Step 6: Node = 7

- root == q (7) → return 7

Step 7: Back to Node = 5

- Left = null, Right = 7 → return 7

Step 8: Back to Node = 2

- Left = 4, Right = 7 → both not null → **return 2 (LCA)**

Step 9: Node = 3

- Recurse into children: 8 and 9 → all return null → return null

Step 10: Back to Node = 1

- Left = 2, Right = null → return **2**

Final Output:

return 2;

Binary Search

40.Median of Two Sorted Arrays

Given two sorted arrays nums1 and nums2 of size m and n respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be O(log (m+n)).

Example 1:

Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2.

Example 2:

Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.

Optimized solution

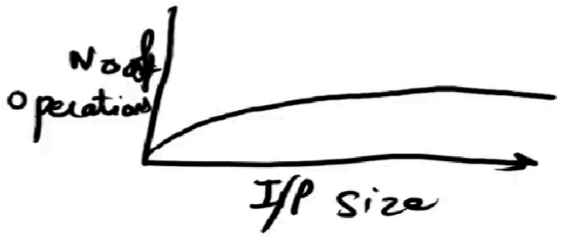
① Median ⇒ {1, 2, (3), 4, 5}

Median = 3

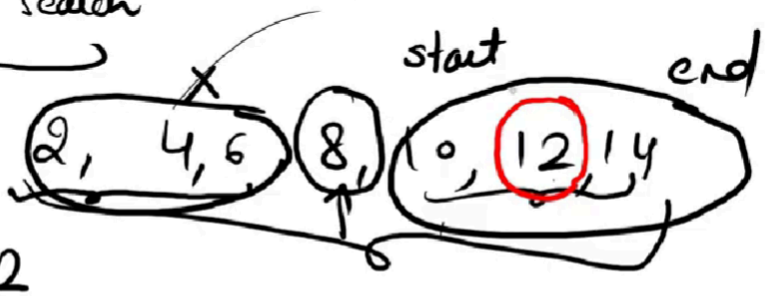
⇒ {1, 2, (3, 4), 5, 6}

$\frac{3+4}{2} = \frac{7}{2} = 3.5$

② $O(\log(m+n))$



Binary search



$\frac{0+2}{2}$ ⇒ 1

search = 12

12 ≠ 8 → false

12 > 8 → Right

12 = 12 → True

nums1 = { 1 | 2, 3 }, nums2 = { 4, 5 | 9 }

m = 3

(1, 4, 5) | (2, 3, 9)

(i) start = 0

(ii) end = 3

(iii) Part1 = $\frac{start + end}{2} = \frac{0+3}{2} \Rightarrow 1$ ✓

(iv) Part2 = $\frac{m+n+1}{2} - Part1 = \frac{3+3+1}{2} - 1$

$\Rightarrow \frac{7}{2} - 1 \Rightarrow 2$ ✓

(v) maxLeftnums1

maxLeftnums2

minRightnums1

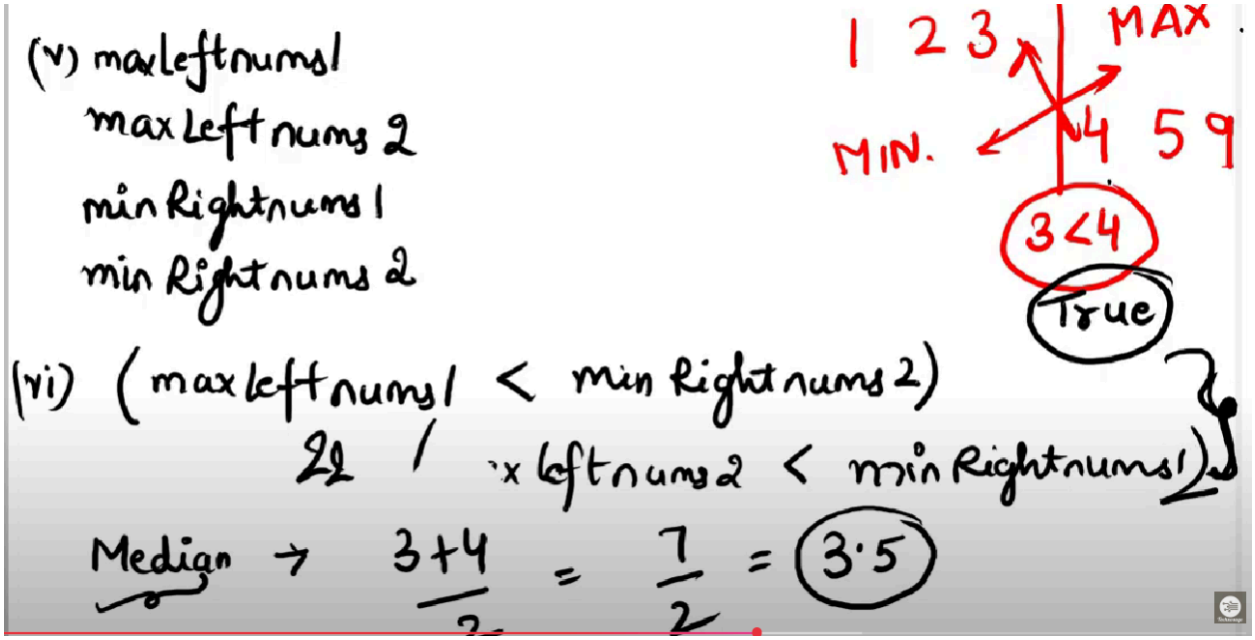
minRightnums2

True → nums2 false

1 < 3 & 5 < 2

(vi) (maxLeftnums1 < minRightnums2) && (maxLeftnums2 < minRightnums1)

Median →




```
package com.leetcode150;
public class MedianOfTwoSortedArrays {
	public double findMedianSortedArrays(int[] nums1, int[] nums2) {
		// Ensure nums1 is the smaller array
		if (nums1.length > nums2.length) {
			return findMedianSortedArrays(nums2, nums1);
		}
		int m = nums1.length;
		int n = nums2.length;
		int low = 0, high = m;
		while (low <= high) {
			int partitionX = (low + high) / 2;
			int partitionY = (m + n + 1) / 2 - partitionX;
			int maxLeftX = (partitionX == 0) ? Integer.MIN_VALUE :
nums1[partitionX - 1];
			int minRightX = (partitionX == m) ? Integer.MAX_VALUE :
nums1[partitionX];
			int maxLeftY = (partitionY == 0) ? Integer.MIN_VALUE :
nums2[partitionY - 1];
			int minRightY = (partitionY == n) ? Integer.MAX_VALUE :
nums2[partitionY];
			if (maxLeftX <= minRightY && maxLeftY <= minRightX) {
				// Correct partition
```

```
                        if ((m + n) % 2 == 0) {
                                return (Math.max(maxLeftX, maxLeftY) +
Math.min(minRightX, minRightY)) / 2.0;
                        } else {
                                return Math.max(maxLeftX, maxLeftY);
                        }
                } else if (maxLeftX > minRightY) {
                        high = partitionX - 1;
                } else {
                        low = partitionX + 1;
                }
        }
        throw new IllegalArgumentException("Input arrays are not sorted properly.");
    }
    public static void main(String[] args) {
        MedianOfTwoSortedArrays solution = new MedianOfTwoSortedArrays();
        int[] nums1 = { 1, 3 };
        int[] nums2 = { 2 };
        System.out.println("Median: " + solution.findMedianSortedArrays(nums1,
nums2)); // 2.0
        nums1 = new int[] { 1, 2 };
        nums2 = new int[] { 3, 4 };
        System.out.println("Median: " + solution.findMedianSortedArrays(nums1,
nums2)); // 2.5
    }
}
```

Median: 2.0
Median: 2.5

New Example

Let's use:

nums1 = [1, 3, 8, 9, 15]
nums2 = [7, 11, 18, 19, 21, 25]

Step 1: Ensure nums1 is smaller

Length of:

- nums1 = 5
- nums2 = 6

No need to swap since nums1 is already smaller.

Step 2: Setup for Binary Search

int m = nums1.length; // 5
int n = nums2.length; // 6
int low = 0, high = m; // low = 0, high = 5

We will now apply binary search on nums1.

Step 3: First Iteration

Calculate partitions

partitionX = (low + high) / 2 = (0 + 5) / 2 = 2
partitionY = (m + n + 1) / 2 - partitionX = (5 + 6 + 1) / 2 - 2 = 6 - 2 = 4

Partitions:

- partitionX = 2 → left of nums1 = [1, 3], right = [8, 9, 15]
- partitionY = 4 → left of nums2 = [7, 11, 18, 19], right = [21, 25]

Get maxLeft and minRight

maxLeftX = nums1[partitionX - 1] = nums1[1] = 3
minRightX = nums1[partitionX] = nums1[2] = 8

maxLeftY = nums2[partitionY - 1] = nums2[3] = 19
minRightY = nums2[partitionY] = nums2[4] = 21

Check partition condition

if (maxLeftX <= minRightY && maxLeftY <= minRightX)
=> if (3 <= 21 && 19 <= 8)
=> true && false ❌

Partition is invalid because maxLeftY > minRightX → too far left in nums1.

3.4: Move right

low = partitionX + 1 = 3

Step 4: Second Iteration

Calculate new partitions

partitionX = (low + high) / 2 = (3 + 5) / 2 = 4
partitionY = (5 + 6 + 1) / 2 - 4 = 6 - 4 = 2

Partitions:

- partitionX = 4 → left of nums1 = [1, 3, 8, 9], right = [15]
- partitionY = 2 → left of nums2 = [7, 11], right = [18, 19, 21, 25]

Get maxLeft and minRight

maxLeftX = nums1[3] = 9
minRightX = nums1[4] = 15

maxLeftY = nums2[1] = 11
minRightY = nums2[2] = 18

Check partition

if (9 <= 18 && 11 <= 15)
=> true && true

Correct partition found!

Step 5: Find the Median

Check total length:

(m + n) % 2 == 0 → (5 + 6) % 2 = 11 % 2 = 1 → Odd

Odd → Return max of left parts:

max(maxLeftX, maxLeftY) = max(9, 11) = 11

Final Answer:

Median = 11.0

41.Search in Rotated Sorted Array

There is an integer array nums sorted in ascending order (with **distinct** values).

Prior to being passed to your function, nums is **possibly rotated** at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (**0-indexed**). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums **after** the possible rotation and an integer target, return *the index of* target *if it is in* nums*, or* -1 *if it is not in* nums.

You must write an algorithm with O(log n) runtime complexity.

Example 1:

Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4

Example 2:

Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1

Example 3:

Input: nums = [1], target = 0
Output: -1

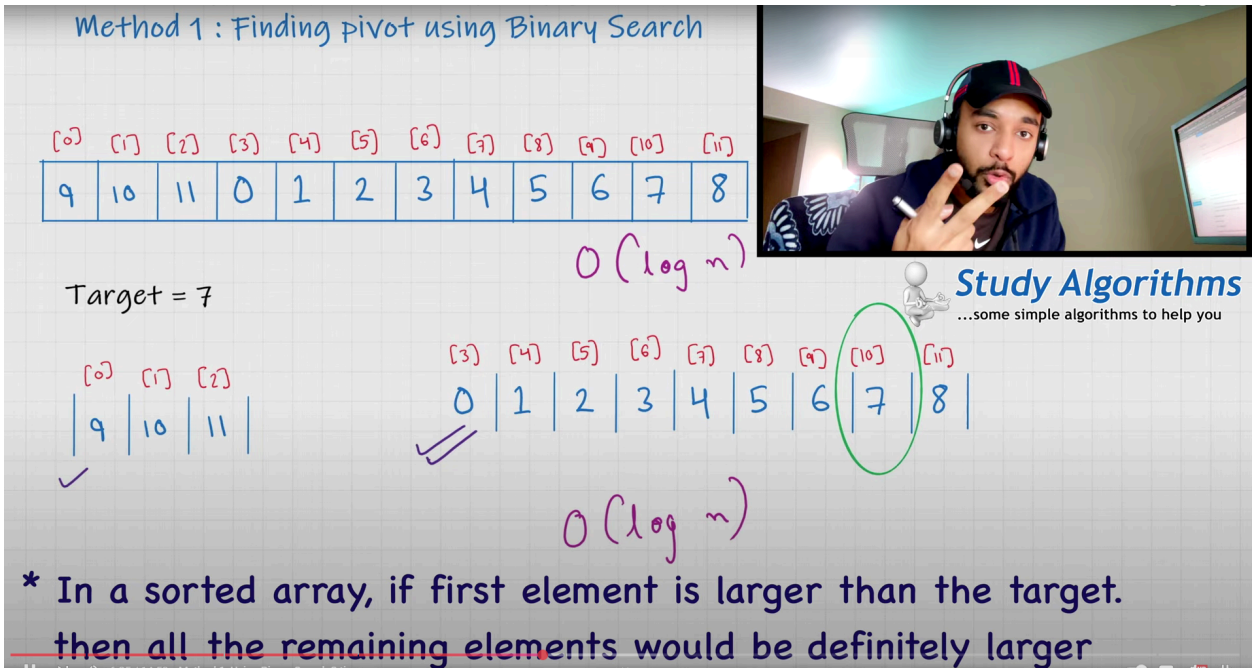
General approach:

Approach 2

In the **Pivot + Binary Search (Approach 2)**:

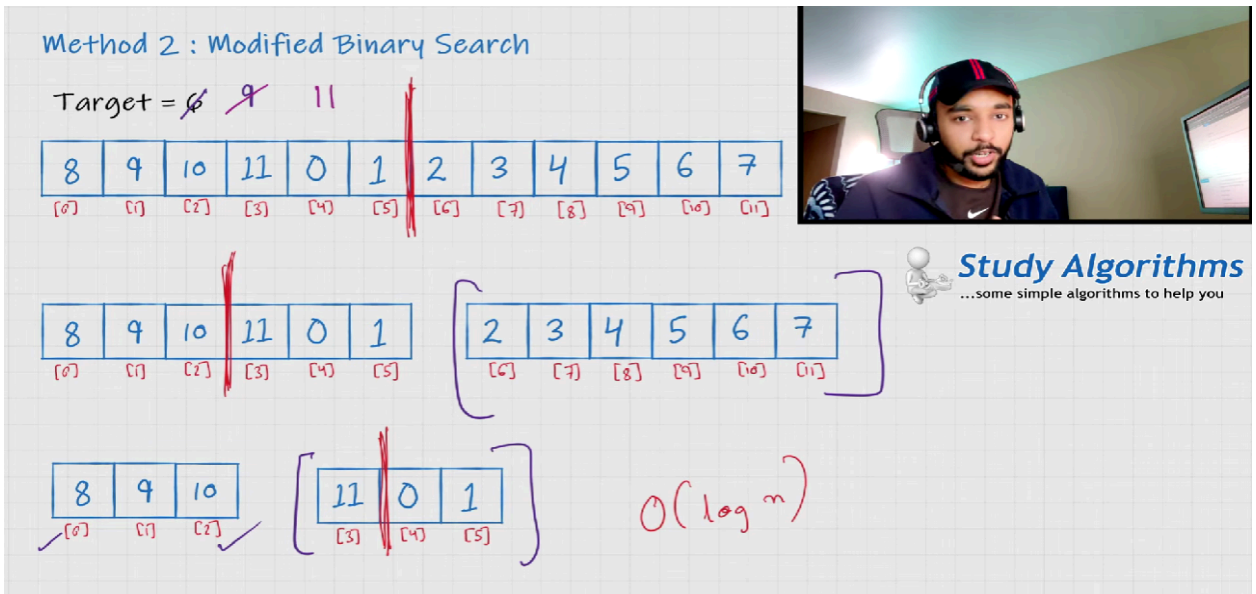
1. **Finding the Pivot** (i.e., the index of the smallest element):
    - Done using binary search.
    - **Time Complexity**: O(log n)
2. **Binary Search in One Half** (either left or right of the pivot):

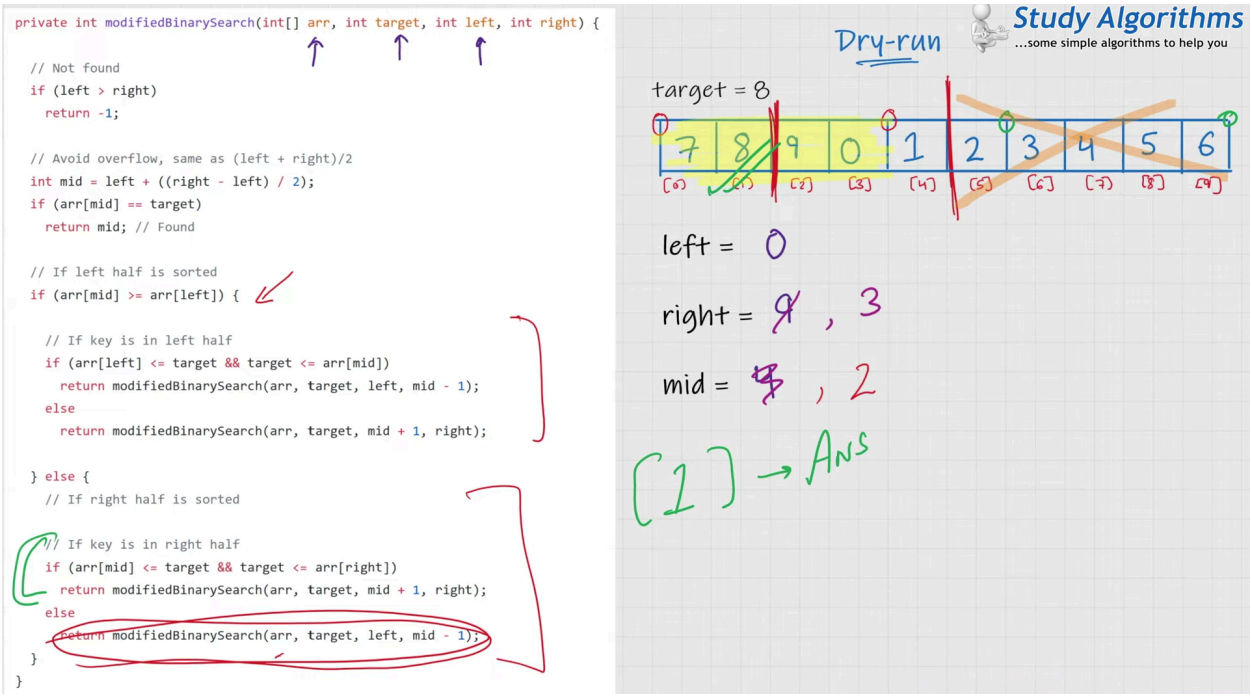
- Again, classic binary search.
- **Time Complexity**: O(log n)



Optimized solution:

Modified Binary Search (Single Pass)





Step-by-Step Algorithm:

1. Base Case:

If left > right, the target isn't present. Return -1.

2.Calculate Mid Safely:

int mid = left + (right - left) / 2;

3.Check if Target Found:

if (arr[mid] == target) return mid;

4.Determine which half is sorted:

- If arr[mid] >= arr[left], then the **left half is sorted**.
    - Now check if the target lies in that range:
      if (arr[left] <= target && target <= arr[mid])
      → search **left**
    - Else → search **right**
- Otherwise, the **right half is sorted**.
    - If arr[mid] <= target && target <= arr[right]
      → search **right**
    - Else → search **left**

Example Dry Run:

Input:
arr = [7, 8, 9, 0, 1, 2, 3, 4, 5, 6]
target = 8

Step 1:

- left = 0, right = 9
- mid = 4 → arr[mid] = 1
- arr[mid] < arr[left] = 7, so right half is not sorted → left is sorted.
- 8 is not between arr[mid]=1 and arr[right]=6
- Go left: left = 0, right = 3

Step 2:

- mid = 1 → arr[mid] = 8
- Found ✅ → return index 1

```
package com.leetcode150;
public class SearchinRotatedSortedArray {
	public static void main(String[] args) {
		int[] arr1 = {7, 8, 9, 0, 1, 2, 3, 4, 5, 6};
		int target1 = 8;
		System.out.println("Index of " + target1 + " is: " + modifiedBinarySearch(arr1,
target1, 0, arr1.length - 1));
	}
	public static int modifiedBinarySearch(int[] arr, int target, int left, int right) {
		// Not found
		if (left > right)
			return -1;
		// Avoid overflow
		int mid = left + ((right - left) / 2);
		if (arr[mid] == target)
			return mid; // Found
		// If left half is sorted
		if (arr[mid] >= arr[left]) {
```

```
                    // If key is in left half
                    if (arr[left] <= target && target <= arr[mid])
                        return modifiedBinarySearch(arr, target, left, mid - 1);
                    else
                        return modifiedBinarySearch(arr, target, mid + 1, right);
                } else {
                    // If key is in right half
                    if (arr[mid] <= target && target <= arr[right])
                        return modifiedBinarySearch(arr, target, mid + 1, right);
                    else
                        return modifiedBinarySearch(arr, target, left, mid - 1);
                }
        }
}
```

Index of 8 is: 1

42.Find Minimum in Rotated Sorted Array

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

- [4,5,6,7,0,1,2] if it was rotated 4 times.
- [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of **unique** elements, return *the minimum element of this array*.

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

Example 1:

Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.

Example 2:

Input: nums = [4,5,6,7,0,1,2]
Output: 0
Explanation: The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.

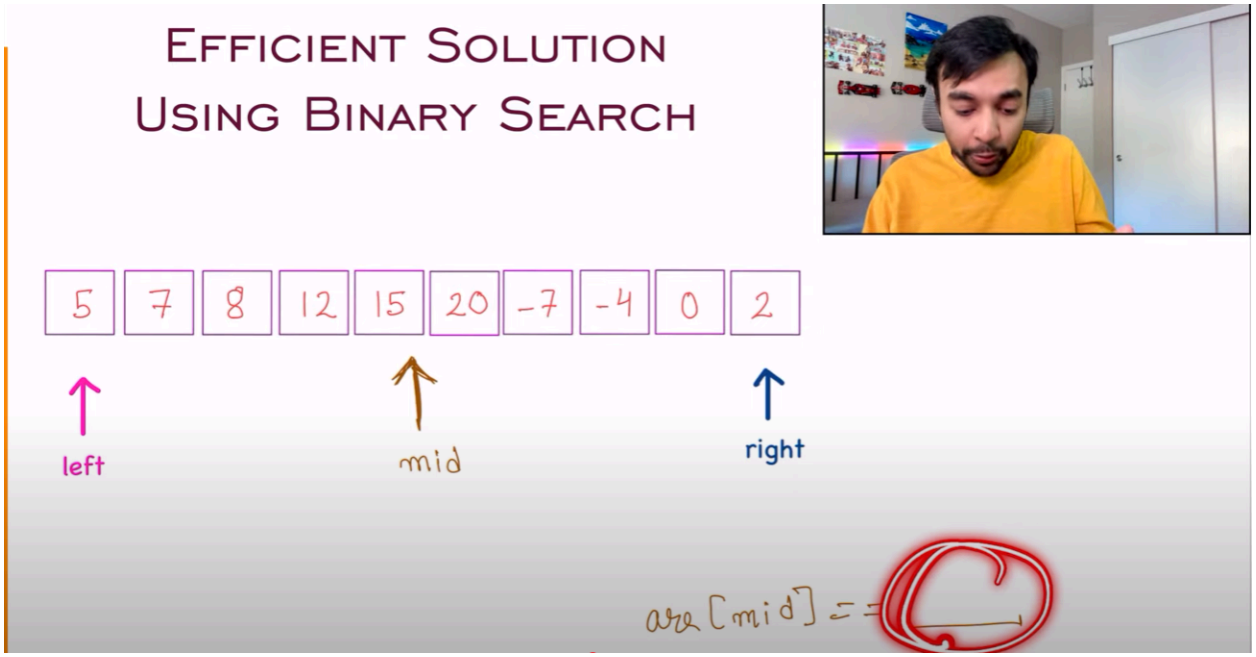
Example 3:

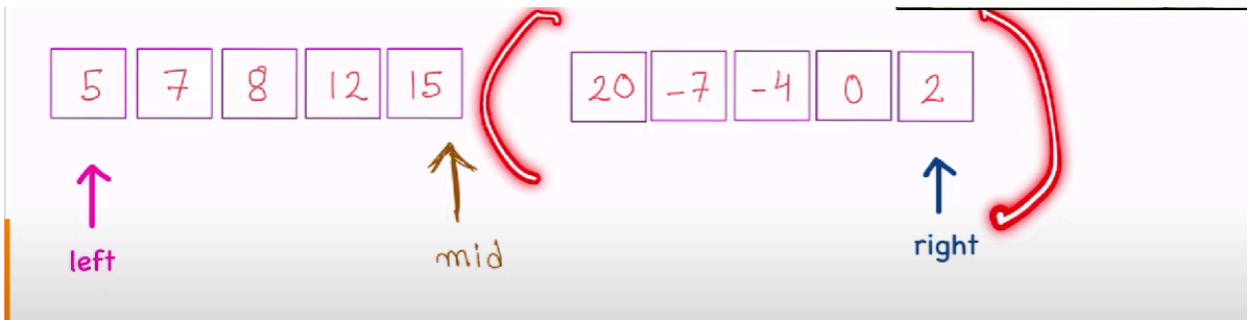
Input: nums = [11,13,15,17]
Output: 11
Explanation: The original array was [11,13,15,17] and it was rotated 4 times.

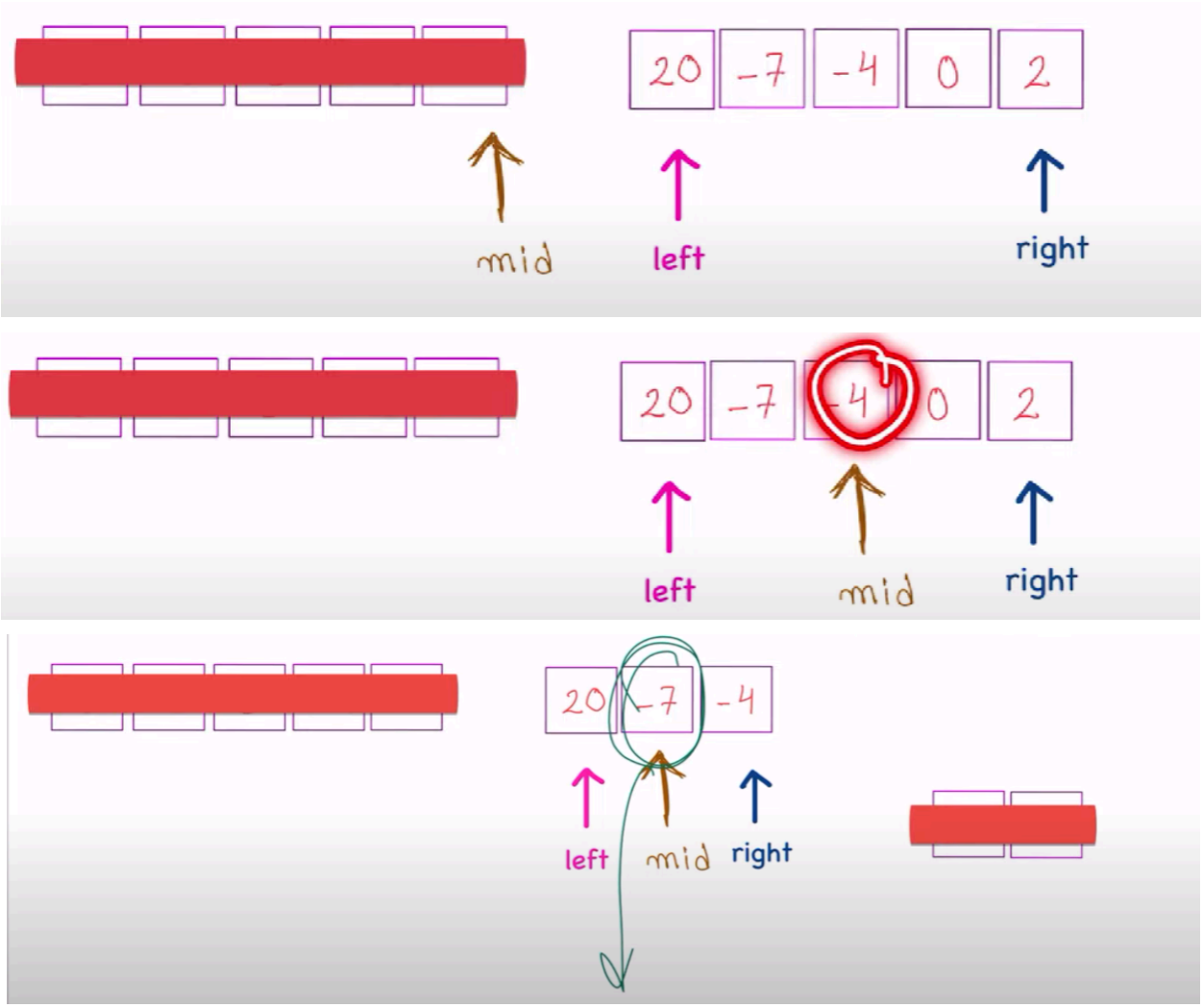
Optimized solutions

Basically if we have target value , we can easily apply binary search and will find it right , but here we do not know the target value what it is

find the minimum in a rotated sorted array using **binary search**







Approach Explained

1. **Initialize two pointers**:
   - left = 0
   - right = nums.length - 1
2. **While** left < right:
   - Find the middle index: mid = (left + right) / 2
   - Compare nums[mid] with nums[right]:
     - If nums[mid] > nums[right]:
       - This means the smallest value is **after** mid, so move left = mid + 1
     - Else:
       - This means the smallest value is at mid or **before**, so move right = mid
3. **Return** nums[left] when the loop ends — that's the minimum.

Step-by-step Example

nums = [4, 5, 6, 7, 0, 1, 2]
0 1 2 3 4 5 6

Step 1:

- left = 0, right = 6
- mid = (0+6)/2 = 3, so nums[mid] = 7, nums[right] = 2
- Since 7 > 2 → Minimum is to the **right** of mid
  👉 So left = mid + 1 = 4

Step 2:

- left = 4, right = 6
- mid = (4+6)/2 = 5, nums[mid] = 1, nums[right] = 2
- Now 1 < 2 → Minimum is in the **left side**, possibly at mid
  👉 So right = mid = 5

Step 3:

- left = 4, right = 5
- mid = (4+5)/2 = 4, nums[mid] = 0, nums[right] = 1
- Now 0 < 1 → Again, minimum is in the **left**, so right = 4

Now left == right == 4 → return nums[4] = 0

Summary of Conditions:

Condition	What it means	Action
nums[mid] > nums[right]	Minimum is **after** mid	left = mid + 1
nums[mid] < nums[right]	Minimum is **at or before** mid	right = mid

```
package com.leetcode150;
public class FindMinimuminRotatedSortedArray {
	public static void main(String[] args) {
```

```
		int num[] = { 3, 4, 5, 1, 2 };
		System.out.println(findMin(num));
	}
	public static int findMin(int[] nums) {
		int left = 0;
		int right = nums.length - 1;
		while (left < right) {
			int mid = left + (right - left) / 2;
			if (nums[mid] > nums[right]) {
				// The minimum is in the right half
				left = mid + 1;
			} else {
				// The minimum is in the left half (including mid)
				right = mid;
			}
		}
		// When loop ends, left == right and points to the minimum
		return nums[left];
	}
}
```

Output

1

43.Contains Duplicate

Given an integer array nums, return true if any value appears **at least twice** in the array, and return false if every element is distinct.

Example 1:

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

Output: true

Explanation:

The element 1 occurs at the indices 0 and 3.

Example 2:

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

Output: false

Explanation:

All elements are distinct.

Example 3:

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

Output: true

```
package com.leetcode150;
import java.util.HashSet;
import java.util.Set;
public class ContainsDuplicate {
	public static void main(String[] args) {
		int[] nums1 = { 1, 2, 3, 1 };
		int[] nums2 = { 1, 2, 3, 4 };
		int[] nums3 = { 1, 1, 1, 3, 3, 4, 3, 2, 4, 2 };
		System.out.println(containsDuplicate(nums1)); // true
		System.out.println(containsDuplicate(nums2)); // false
		System.out.println(containsDuplicate(nums3)); // true
	}
	public static boolean containsDuplicate(int[] nums) {
		Set<Integer> dup = new HashSet<>();
		for (int num : nums) {
			if (dup.contains(num)) {
				return true;
			} else {
				dup.add(num);
			}
		}
		return false;
	}
	// seond way
	/*
```

```
        * Map<Integer, Integer> map = new HashMap<>(); for (int i = 0; i < nums.length;
        * i++) { if (map.containsKey(nums[i])) { return true; } else { map.put(nums[i],
        * i); } } return false; }
        */
}
```

Time and Space Complexity:

- **Time Complexity:** O(n) — single pass through array
- **Space Complexity:** O(n) — in worst case, all elements are stored in the HashSet

Backtracking

44.Permutations

Given an array nums of distinct integers, return all the possible permutations. You can return the answer in **any order**.

Example 1:

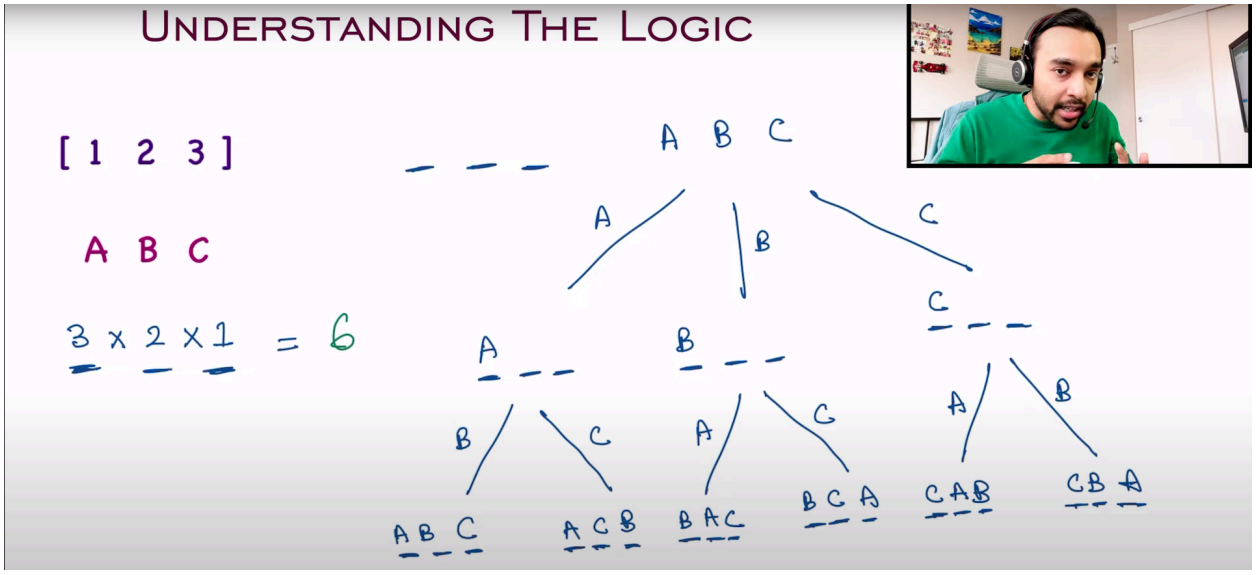
Input: nums = [1,2,3]
Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

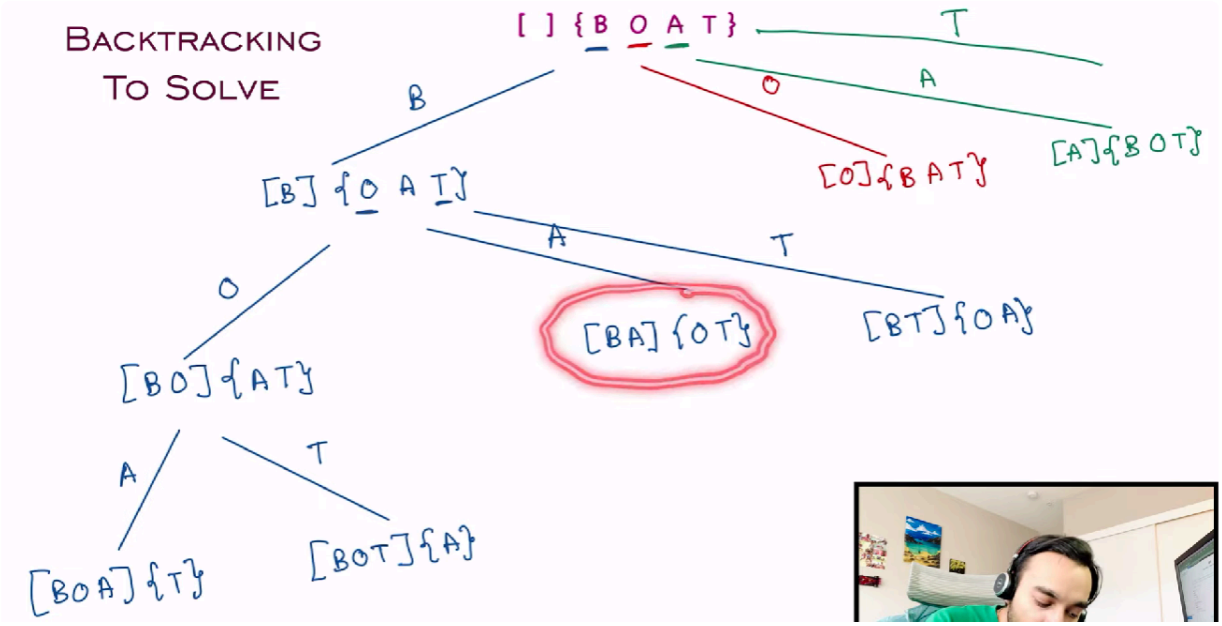
Example 2:

Input: nums = [0,1]
Output: [[0,1],[1,0]]

Example 3:

Input: nums = [1]


UNDERSTANDING THE LOGIC
[1 2 3]
A B C
3 x 2 x 1 = 6
A B C
A
B
C
A
B
C
B
C
A
C
A
B
A
B
C
A C B
B A C
B C A
C A B
C B A


BACKTRACKING
TO SOLVE
[ ] {B O A T}
T
O
A
B
[O] {B A T}
[A] {B O T}
[B] {O A T}
A
T
O
[BA] {O T}
[BT] {O A}
[BO] {A T}
A
T
[BOA] {T}
[BOT] {A}

Step-by-Step Explanation

1. **Recursive Approach**
   We use a helper method backtrack() that tries to build up permutations by **recursively adding one number at a time**.

2. **Avoid Duplicates**
   The line if (tempList.contains(number)) continue; ensures we **don't reuse the same number** in the same permutation.

3. **Build and Remove**
   - Add number to tempList
   - Call backtrack() again
   - Once done, **remove the last number** to backtrack and try another.

4. **Check for Full Permutation**
   When tempList.size() == nums.length, we **have a complete permutation**, so we **copy it into resultList**.

```
	package com.leetcode150;
import java.util.ArrayList;
import java.util.List;
public class Permutations {
	public static void main(String[] args) {
		int[] nums = { 1, 2, 3 };
		List<List<Integer>> result = permute(nums);
		System.out.println(result);
	}
	public static List<List<Integer>> permute(int[] nums) {
		List<List<Integer>> resultList = new ArrayList<>();
		backtrack(resultList, new ArrayList<>(), nums);
		return resultList;
	}
	public static void backtrack(List<List<Integer>> resultList, ArrayList<Integer>
tempList, int[] nums) {
		// If we match the length, it is a permutation
		if (tempList.size() == nums.length) {
			resultList.add(new ArrayList<>(tempList)); //  Copy to avoid reference
issues
			return;
		}
		for (int number : nums) {
			// Skip if we get the same element again
			if (tempList.contains(number)) {
				continue;
			}
			// Add the new element
			tempList.add(number);
			// Go back to try other elements
			backtrack(resultList, tempList, nums);
			// Remove the element (backtrack)
			tempList.remove(tempList.size() - 1);
		}
	}
}
```

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

nums = [1, 2, 3]

Code Call Hierarchy (Dry-Run):

permute([1, 2, 3]) → backtrack([], [1,2,3])

Step-by-Step Flow:

1. **Start with empty tempList**: []
2. Loop over nums: try 1
    - tempList = [1]
    - Recurse: backtrack([1])
        - Try 2
            - tempList = [1, 2]
            - Recurse: backtrack([1,2])
                - Try 3
                    - tempList = [1, 2, 3] ✅ length = 3 → **add to result**
                    - Backtrack: remove 3 → tempList = [1, 2]
            - Try 3 next...
        - Continue exploring...
3. This continues until all 6 permutations are visited:
    - [1,2,3]
    - [1,3,2]
    - [2,1,3]
    - [2,3,1]
    - [3,1,2]
    - [3,2,1]

Each time we **reach a list with 3 elements**, it's a complete permutation → we add a copy to resultList.

Time Complexity

- **O(n!)** – because there are n! permutations of n numbers.
- Each permutation takes O(n) time to copy into the result list → total: **O(n × n!)**

Space Complexity - O(n)

45.Combination Sum

Given an array of **distinct** integers candidates and a target integer target, return *a list of all **unique combinations** of* candidates *where the chosen numbers sum to* target. You may return the combinations in **any order**.

The **same** number may be chosen from candidates an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input.

Example 1:

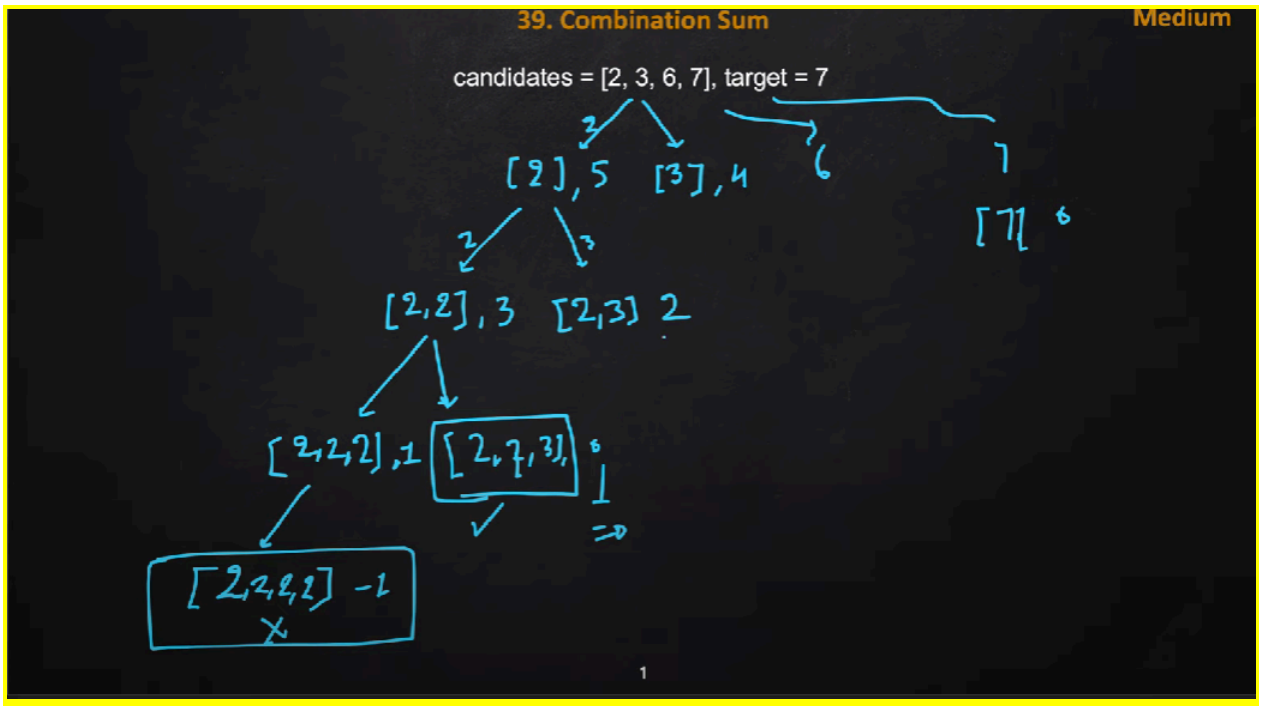
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation:
2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.
7 is a candidate, and 7 = 7.
These are the only two combinations.

Example 2:

Input: candidates = [2,3,5], target = 8
Output: [[2,2,2,2],[2,3,3],[3,5]]

Example 3:

Input: candidates = [2], target = 1
Output: []



Code Flow Example

Input: candidates = [2,3,6,7], target = 7

Step-by-step:

- Start from 0, target = 7.
- Choose 2 → target = 5 → choose 2 again → target = 3 → choose 2 again → target = 1 → choose 2 again → target = -1 (invalid).
- Backtrack → try 3 → now path: [2,2,3] → target = 0 → valid!
- Add [2,2,3] to the result.
- Backtrack again and try 7 → target = 0 → valid!
- Add [7] to the result.

Result = [[2,2,3], [7]]

💡 **How It Works**

- **Recursive Backtracking** builds combinations.
- Starts from a given index to avoid using previous elements again.
- Repeat numbers (no i+1) — allows reuse.
- If the target becomes **0**, a valid combination is found.
- If the target becomes **< 0**, it's invalid — backtrack.
- Uses **backtracking** to explore all paths and revert changes.

```
	package com.leetcode150;
import java.util.ArrayList;
import java.util.List;
public class CombinationSum {
	public static void main(String[] args) {
		int[] candidates = { 2, 3, 6, 7 };
		int target = 7;
		System.out.println(combinationSum(candidates, target));
	}
	public static List<List<Integer>> combinationSum(int[] candidates, int target) {
		List<List<Integer>> result = new ArrayList<>();
		backtrack(candidates, 0, target, new ArrayList<>(), result);
		return result;
	}
	public static void backtrack(int[] cand, int start, int target, List<Integer> list,
List<List<Integer>> result) {
		if (target < 0)
			return; // Invalid, exceeds target
		if (target == 0) { // Found valid combination
			result.add(new ArrayList<>(list));
			return;
		}
		for (int i = start; i < cand.length; i++) {
			list.add(cand[i]); // Choose
			backtrack(cand, i, target - cand[i], list, result); // Not i+1 — reuse same
element
			list.remove(list.size() - 1); // Backtrack
		}
```

```
        }
}
```

Output

[[2, 2, 3], [7]]

46.Subsets

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

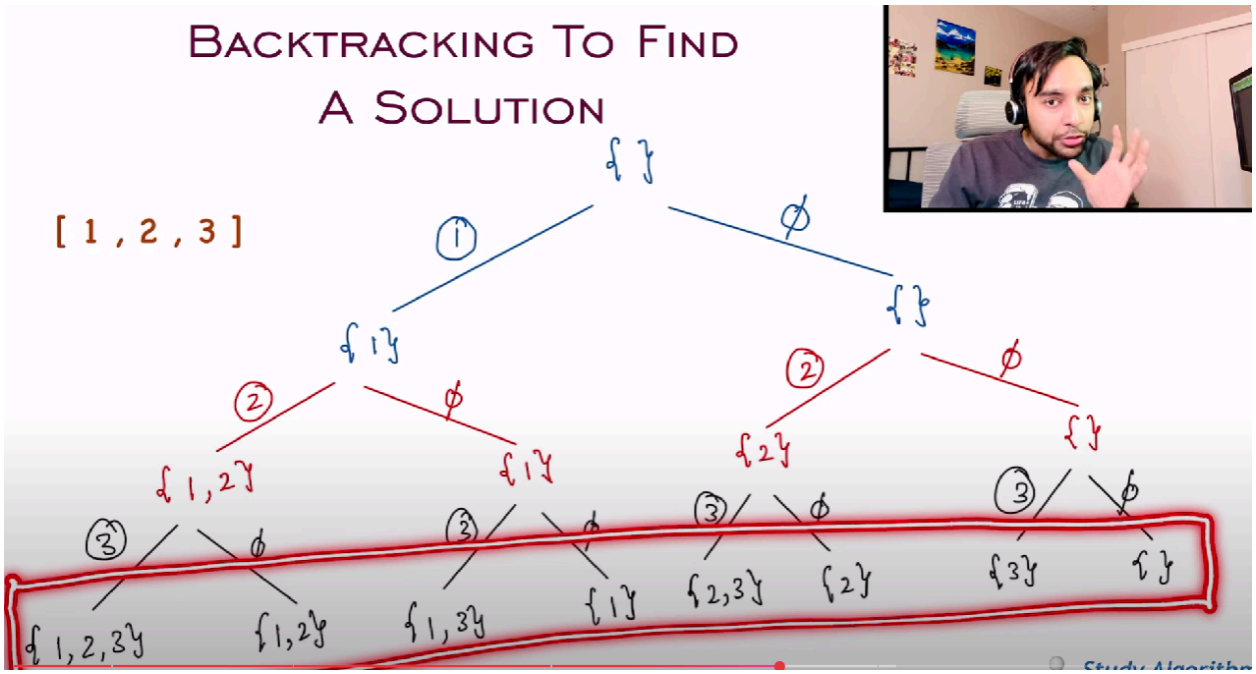
The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

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]]



```java
public List<List<Integer>> subsets(int[] nums) {
  List<List<Integer>> resultList = new ArrayList<>();

  // Start backtracking from the beginning
  backtrack(resultList, new ArrayList<>(), nums, start: 0);
  return resultList;
}

private void backtrack(List<List<Integer>> resultSets,
                       List<Integer> tempSet,
                       int[] nums, int start) {
  // Add the set to result set
  resultSets.add(new ArrayList<>(tempSet));
  for (int i = start; i < nums.length; i++) {

    // Case of including the number
    tempSet.add(nums[i]);

    // Backtrack the new subset
    backtrack(resultSets, tempSet, nums, start: i + 1);

    // Case of not-including the number
    tempSet.remove(index: tempSet.size() - 1);
  }
}
```

DRY-RUN

[1 , 2 , 3]

resultList =

Study Algorithms

For each element, you have **two choices**:

- ✅ Include it
- ❌ Don't include it

So, total combinations: $2 * 2 * 2 * \ldots$ (n times) $= 2^n$

Example:

Input: nums = [1, 2, 3]

- Number of elements: $n = 3$
  Total subsets: $2^3 = 8$

Subsets:

[], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3]

Ex:

If $n = 4$ → You'll get $2^4 = 16$ subsets
If $n = 5$ → You'll get $2^5 = 32$ subsets

Time & Space Complexity:

- **Time Complexity**: $O(2^n \times n)$ — for each of 2^n subsets, copying subset of up to n elements.
- **Space Complexity**: $O(n)$ recursion stack + result list.

EX:
nums = [1, 2]

Step 1: Start in subsets()

```
List<List<Integer>> resultList = new ArrayList<>();
backtrack(resultList, new ArrayList<>(), nums, 0);
```

Step 2: First Call to backtrack(...)

```
backtrack([], [], [1, 2], 0);
```

- tempSet = []
- Add [] to resultList → [[]]
- Start loop from i = 0

🔁 **Step 3: i = 0 → Include nums[0] = 1**

```
tempSet = [1]
backtrack([[]], [1], [1, 2], 1);
```

- Add [1] to resultList → [[], [1]]
- Start loop from i = 1

🔁 **Step 4: i = 1 → Include nums[1] = 2**

```
tempSet = [1, 2]
backtrack([[], [1]], [1, 2], [1, 2], 2);
```

- Add [1, 2] to resultList → [[], [1], [1, 2]]
- start = 2 → End of array → return
- Backtrack → remove last → tempSet = [1]

🔁 **Step 5: Finish loop at i = 1**

- Backtrack again → remove last → tempSet = []

🔁 **Step 6: Back to i = 1 in the original call**

tempSet = [2]
backtrack([[], [1], [1, 2]], [2], [1, 2], 2);

- Add [2] to resultList → [[], [1], [1, 2], [2]]
- start = 2 → End of array → return
- Backtrack → remove 2 → tempSet = []

✅ **Final resultList:**
[[], [1], [1, 2], [2]]

Step	tempSet	Action
Start	`[]`	Add to result
Add 1	`[1]`	Add to result
Add 2	`[1, 2]`	Add to result
Remove 2	`[1]`	Backtrack
Remove 1	`[]`	Backtrack
Add 2	`[2]`	Add to result
Remove 2	`[]`	Backtrack complete

```java
package com.leetcode150;
import java.util.ArrayList;
import java.util.List;
public class Subsets {
	public static void main(String[] args) {
		int nums[] = { 1, 2 };
		System.out.println(subsets(nums));
	}
	public static List<List<Integer>> subsets(int[] nums) {
		List<List<Integer>> resultList = new ArrayList<>();
		backtrack(resultList, new ArrayList<>(), nums, 0);
		return resultList;
	}
	public static void backtrack(List<List<Integer>> resultSets, List<Integer> tempSet, int[]
nums, int start) {
		resultSets.add(new ArrayList<>(tempSet)); // Add current subset
		for (int i = start; i < nums.length; i++) {
			tempSet.add(nums[i]); // Include element
			backtrack(resultSets, tempSet, nums, i + 1); // Move to next index
			tempSet.remove(tempSet.size() - 1); // Backtrack (remove last added)
		}
	}
}
```

Output

[[], [1], [1, 2], [2]]

Heaps

47.Kth Largest Element in an Array

Given an integer array nums and an integer k, return *the* kth *largest element in the array*.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

Can you solve it without sorting?

Example 1:

Input: nums = [3,2,1,5,6,4], k = 2
Output: 5

Example 2:

Input: nums = [3,2,3,1,2,4,5,5,6], k = 4
Output: 4

General Solution

Using a **LinkedList Queue** to solve the problem of finding the kth largest element in an array is not a typical approach, but it's possible to adapt it. Essentially, you'd use the queue to maintain a running list of the largest k elements seen so far. Here's how we can implement this approach using a **LinkedList** as a queue (FIFO structure):

Approach using a LinkedList Queue:

1. **LinkedList as a Queue**: We will use a **LinkedList** to act as a queue to hold the top k largest elements.

2. **Iterate through the array**: As we traverse the array, we add each element to the queue.

3. **Queue size check**: If the size of the queue exceeds k, we remove the smallest element (i.e., the head of the queue) to ensure that we only keep the largest k elements.

4. **Final Element**: The kth largest element will be the first element of the queue once the entire array is processed.

```
package com.leetcode150;
import java.util.*;
public class KthLargestElementinanArray {
	public static int findKthLargest(int[] nums, int k) {
		// Using LinkedList as a queue to maintain the top k elements
		LinkedList<Integer> queue = new LinkedList<>();
		// Iterate through the array
		for (int num : nums) {
			// Add current number to the queue
			queue.add(num);
			// If the size of the queue exceeds k, remove the smallest element
			if (queue.size() > k) {
				Collections.sort(queue); // Sort to find the smallest element
				queue.poll(); // Remove the smallest element (head of the queue)
```

```
                }
            }
            // The head of the queue will be the kth largest element
            Collections.sort(queue); // Sort the final k elements
            return queue.getFirst(); // Return the first element, which is the kth largest
        }
        public static void main(String[] args) {
            int[] nums1 = { 3, 2, 1, 5, 6, 4 };
            int k1 = 2;
            System.out.println(findKthLargest(nums1, k1)); // Output: 5
            int[] nums2 = { 3, 2, 3, 1, 2, 4, 5, 5, 6 };
            int k2 = 4;
            System.out.println(findKthLargest(nums2, k2)); // Output: 4
        }
}
```

Output
5
4

Time Complexity:

- Sorting the queue each time a new element is added: **O(k log k)** for each insertion.
- There are n elements, so the total time complexity will be **O(n * k log k)**, where n is the number of elements in the array and k is the size of the queue (which in this case is constant).

Optimized solution

Using a Min-Heap (Priority Queue)

PriorityQueue **does not store elements in a fully sorted order**, but it **maintains the smallest (or largest, depending on how it's configured) element at the head** of the queue at all times.

A Min-Heap allows you to efficiently keep track of the top k largest elements. By maintaining a heap of size k, the smallest element in the heap will always be the kth largest element.

Default Behavior of PriorityQueue in Java:

- By default, it is a **Min-Heap**.
- So, the **smallest element is always at the head** (peek() returns it).
- Internally, it uses a **heap** structure (not a sorted list), which makes it efficient.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.add(3);

pq.add(1);

pq.add(5);

pq.add(2);

System.out.println(pq);         // Output might look like: [1, 2, 5, 3] (not sorted)

System.out.println(pq.peek());  // Always returns the smallest element: 1
```

So the internal structure isn't fully sorted, but the head is always the smallest element in a Min-Heap.

For Kth Largest:

To find the **kth largest element**, we:

- Keep a **Min-Heap** of size k.
- If the heap grows beyond size k, we remove the smallest element.
- At the end, the root (peek()) gives us the **kth largest**.

```
package com.leetcode150;
import java.util.*;
public class KthLargestElementinanArray {
	public static int findKthLargest(int[] nums, int k) {
		// Min-Heap (PriorityQueue) with a size of k
		PriorityQueue<Integer> minHeap = new PriorityQueue<>();
		// Add elements to the heap
```

```
            for (int num : nums) {
                minHeap.add(num);
                // If the heap size exceeds k, remove the smallest element
                if (minHeap.size() > k) {
                    minHeap.poll();
                }
            }
            // The root of the Min-Heap is the kth largest element
            return minHeap.peek();
        }
        public static void main(String[] args) {
            int[] nums1 = { 3, 2, 1, 5, 6, 4 };
            int k1 = 2;
            System.out.println(findKthLargest(nums1, k1)); // Output: 5
            int[] nums2 = { 3, 2, 3, 1, 2, 4, 5, 5, 6 };
            int k2 = 4;
            System.out.println(findKthLargest(nums2, k2)); // Output: 4
        }
}
```

Output

5
4

Time Complexity:

- **O(n log k)**, where n is the number of elements in the array. This is because we process each element once and each insertion/removal from the heap takes O(log k) time.

Space Complexity :O(k)

We never store more than k elements in the heap.

No other significant data structures are being used.

Kth smallest Element in an Array

For **Kth Largest → Min-Heap of size k**

For **Kth Smallest → Max-Heap of size k**

Using Max-Heap (PriorityQueue with reverse order)

```
package com.leetcode150;

import java.util.*;
public class KthSmallestElement {
	public static int findKthSmallest(int[] nums, int k) {
		// Max-Heap to keep the k smallest elements
		PriorityQueue<Integer> maxHeap = new
PriorityQueue<>(Collections.reverseOrder());
		for (int num : nums) {
			maxHeap.add(num);
			// Keep only k smallest elements in the heap
			if (maxHeap.size() > k) {
				maxHeap.poll(); // remove the largest among them
			}
		}
		// Top of the heap is the Kth smallest element
		return maxHeap.peek();
	}
	public static void main(String[] args) {
		int[] nums1 = { 3, 2, 1, 5, 6, 4 };
		int k1 = 2;
		System.out.println(findKthSmallest(nums1, k1)); // Output: 2
		int[] nums2 = { 7, 10, 4, 3, 20, 15 };
		int k2 = 3;
		System.out.println(findKthSmallest(nums2, k2)); // Output: 7
	}
}
```

Output

```
2
7
```

Time Complexity O(n log k)

Space Complexity O(k)

.Last Substring in Lexicographical Order

Lexicographical order is similar to dictionary order, where words are arranged based on the alphabetical order of their letters.

"a" < "b" < "c" < ... < "z"

- Start comparing from the first character.
- As soon as a difference is found, stop and determine the order.
- If all characters match, compare length (shorter string comes first).

1) "apple" < "banana" → ('a' < 'b')

   Compare first characters: 'a' (from "apple") vs 'b' (from "banana").

   - 'a' < 'b' (Since 'a' comes before 'b' in the alphabet).
   - Stop here! We already know "apple" < "banana".

   No need to check the remaining characters (p, p, l, e) because the first differing character (a vs b) already determined the order.

2) "apple" vs "apricot"

   apple
   apricot

   'a' == 'a' (same, continue).

   'p' == 'p' (same, continue).

   'p' < 'r' (since 'p' comes before 'r' in the alphabet).

   Stop here! "apple" < "apricot".

3) Example Where Length Matters: "apple" vs "apples"

   apple
   apples

   'a' == 'a', 'p' == 'p', 'p' == 'p', 'l' == 'l', 'e' == 'e' (all same).

"apple" ends, but "apples" has an extra 's' → Shorter word comes first.

So "apple" < "apples".

Lexicographical order does NOT compare every character blindly. It follows a step-by-step, left-to-right comparison, stopping as soon as a difference is found.

- If all characters match, compare length (shorter string comes first)

1) "bat" vs "batch"

'b' == 'b' (same, move forward)

'a' == 'a' (same, move forward)

't' == 't' (same, move forward)

"bat" ends, but "batch" has extra characters ('c' and 'h')

- The shorter word comes first in lexicographical order.

Final Answer: "bat" < "batch"

2) apple" vs "apples"

'a' == 'a'

'p' == 'p'

'p' == 'p'

'l' == 'l'

'e' == 'e'

"apple" ends, but "apples" has an extra 's'

Final Answer: "apple" < "apples"

Java compares strings lexicographically using `.compareTo()`.

java Copy Edit

```java
System.out.println("apple".compareTo("banana")); // -1 ("apple" < "banana")
System.out.println("cat".compareTo("bat"));      // 1  ("cat" > "bat")
System.out.println("abc".compareTo("abcd"));     // -1 ("abc" < "abcd")
```

Example: "leetcode"

We want to find the lexicographically largest substring of "leetcode".

Step-by-step walkthrough:

String: l e e t c o d e
Index : 0 1 2 3 4 5 6 7

Step 1: Initialize

- i = 0 → Currently, "leetcode" is the largest substring.
- j = 1 → Next possible candidate is "eetcode".
- n = 8 → Length of "leetcode".

Step 2: Compare i and j

- Compare s[0] (l) with s[1] (e).
- 'l' > 'e', so skip j and move to the next index.
- Now, j = 2.

Step 3: Compare i = 0 with j = 2

- Compare s[0] (l) with s[2] (e).
- 'l' > 'e', so skip j and move to the next index.
- Now, j = 3.

Step 4: Compare i = 0 with j = 3

- Compare s[0] (l) with s[3] (t).
- 'l' < 't', so update i = 3.
- Now, j = 4.

Step 5: Compare i = 3 with j = 4

- Compare s[3] (t) with s[4] (c).
- 't' > 'c', so skip j and move to the next index.
- Now, j = 5.

Step 6: Compare i = 3 with j = 5

- Compare s[3] (t) with s[5] (o).
- 't' > 'o', so skip j and move to the next index.
- Now, j = 6.

Step 7: Compare i = 3 with j = 6

- Compare s[3] (t) with s[6] (d).
- 't' > 'd', so skip j and move to the next index.
- Now, j = 7.

Step 8: Compare i = 3 with j = 7

- Compare s[3] (t) with s[7] (e).
- 't' > 'e', so skip j and move to the next index.
- Now, j = 8.

End of Loop

- Since j = n, we return s.substring(i).
- i = 3, so we return "tcode".

k is used only if s[i] == s[j].

This means the algorithm needs to compare multiple characters.

Initialization

- i = 0 (starting index of the current largest substring)
- j = 1 (candidate index to compare)
- n = 4 (length of "abab")

Comparisons

First Comparison (i = 0, j = 1):

- Compare s[i] = 'a' with s[j] = 'b'
- 'a' < 'b', so update i = 1
- Move j to 2

Second Comparison (i = 1, j = 2):

- Compare s[i] = 'b' with s[j] = 'a'
- 'b' > 'a', so keep i = 1
- Move j to 3

Third Comparison (i = 1, j = 3):

- Compare s[i] = 'b' with s[j] = 'b'
- Since they are equal, we need k to compare further:
    - Compare s[i + 1] = 'a' with s[j + 1] = 'b'
    - 'a' < 'b', so update i = 3 (we found a better starting point)
- Move j to 4, which ends the loop.

Return the Result

After the loop, i = 3, so we return s.substring(i), which is:

arduino

CopyEdit

"bab"

```
class Solution {
    public String lastSubstring(String s) {
        int i = 0, j = 1, n = s.length();

        while (j < n) {
            int k = 0;
            while (j + k < n && s.charAt(i + k) == s.charAt(j + k)) {
                k++;
            }
            if (j + k < n && s.charAt(i + k) < s.charAt(j + k)) {
                i = j;
            }
            j++;
        }

        return s.substring(i);
```

```
    }
}
```