1.Question to Code

Given an array of integers, nums, and an integer, target, **find the 0-based indices of the two numbers in the array that add up exactly to the target value.**

Constraints & Assumptions:

1. Assume there is **exactly one** valid solution pair in the input array.
2. The same element **cannot** be used twice to form the sum.
3. The array is **not** guaranteed to be sorted.

Function Signature (Example Java/C++)

The solution should be implemented within a function/method that accepts the array and the target, and returns an array or list of two integers (the indices).

Java
public int[] twoSum(int[] nums, int target)

🔢 Sample Input & Output

Category	Description	Example
Input: nums	The array of integers.	[2, 7, 11, 15]
Input: target	The required sum.	9
Output	The 0-based indices of the two numbers that sum to target.	[0, 1]

Explanation of Sample:

- The number at index $\mathbf{0}$ is **2**.
- The number at index $\mathbf{1}$ is **7**.
- Since $2 + 7 = 9$, the indices $\mathbf{[0, 1]}$ are returned.

🟦 Level 1 — Core Variations

1. Two Sum (Classic – Unsorted Array, Return Indices)

- Input array unsorted
- Return **indices**
- O(N) using HashMap

2. Two Sum – Sorted Array

- Array sorted
- Use **two-pointer approach**
- O(N), constant space

3. Two Sum – Return Boolean

- Only check if a pair exists
- No need to return indices

4. Two Sum – Return the Pair of Values (Not Indices)

- Return the actual numbers
- Sorted / Unsorted variants

🟩 Level 2 — Applied Variants

5. Two Sum – All Unique Pairs

- Return **all pairs** that match the target
- Handle duplicates carefully

- Usually solved with sorting + 2-pointer

6. Two Sum – Count of All Pairs

- Return **number of valid pairs**
- Not the pairs themselves

7. Two Sum – No Repeating Pair Variation

- If (1, 3) is counted, (3, 1) should not repeat

8. Two Sum – Using Only Constant Space

- Generally sorted → two pointers
- No HashMap allowed

🟨 Level 3 — Data-Structure Variants

9. Two Sum Using HashSet

- Return True/False
- Do not use a map
- Simplified version

10. Two Sum Using Binary Search

- For each element, search complement in sorted array
- O(N log N)

11. Two Sum in a Linked List

- Need two pointers? Not possible directly

- Use HashSet
- Or convert to array

12. Two Sum in a BST (Binary Search Tree)

- Inorder traversal → sorted list → two pointers
- Or use HashSet during traversal

13. Two Sum Using Streams (Java 8)

- More functional-programming style
- Not interviewer favorite, but trendy

🟧 Level 4 — Constraint-Driven Variants

14. Two Sum with Limited Memory

- Memory limit forces two-pointer or external sort

15. Two Sum with Huge Input (Streaming Data)

- You cannot store whole array
- Use sliding windows or limited hash memory

16. Two Sum where Indices Must Be Minimum

- Return pair with smallest index difference

🟥 Level 5 — Extended Family (CS Royalty Tier)

These are direct evolutions of Two Sum used in mid-to-hard interviews.

17. 3 Sum

- Find triplets that sum to target (usually 0)
- $O(N^2)$ with sorting + two pointers

18. 3 Sum Closest

- Find a triplet whose sum is closest to the target

19. 3 Sum With Multiplicity

- Count the number of valid triplets (heavy combinatorics)

20. 4 Sum

- Quadruplets
- $O(N^3)$

21. K Sum (Generalized Version)

- Multiple nested layers
- Recursion + two pointers
- Template-based questions

🟪 Level 6 — Trick Variants (FAANG Loves These)

22. Two Sum – Subarray Version

- Not "two numbers anywhere"

- Find a **subarray** with sum = target (Prefix sum problem)

23. Two Sum – Multiplicative Version

- Find pair where a * b = target
- Use factor checking

24. Two Sum – Modulo Version

- (a + b) % k == 0

25. Two Sum – Absolute Difference Variant

- |nums[i] – nums[j]| = target

26. Two Sum Under a Constraint

Examples:

- Only adjacent indices
- Must pick from different halves
- Each number can be used only once globally

📝 Maximum Subarray Sum

Problem Definition

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

A **subarray** is a contiguous non-empty sequence of elements within an array.

Example Cases

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 entire array [5, 4, -1, 7, 8] has the largest sum = 23.

Constraints

- $1 \le nums.\text{length} \le 10^5$
- $-10^4 \le nums[i] \le 10^4$

Function Signature

You are expected to implement the following function:

Java
```
class Solution {
    /**
     * Finds the maximum contiguous subarray sum using Kadane's Algorithm.
     * * @param nums The input array of integers.
     * @return The maximum sum of any contiguous subarray.
```

```
     */
    public int maxSubArray(int[] nums) {
        // Your implementation goes here
    }
}
```

🟦 LEVEL 1 — Core Variations

(Standard Kadane Family)

1. Maximum Subarray Sum (Classic)

Return the **max sum only**.
→ Standard Kadane.

2. Maximum Subarray Sum — Return Subarray

Return:

- max sum
- start index
- end index
  or the actual subarray list.

3. Maximum Subarray Sum (Allow Empty Subarray)

If all numbers are negative, result = 0.

4. Maximum Subarray Sum for Circular Array (Circular Kadane)

Subarray can wrap around the end of the array.
Use:

- normal Kadane
- min-subarray trick
- total sum logic.

🟩 LEVEL 2 — Variants Based on Subarray Constraints

5. Maximum Subarray of Size K

Fixed window size.
→ Sliding window.

6. Maximum Subarray with Length Between L and R

Variable window with boundaries.
→ Prefix sum + deque optimization.

7. Maximum Subarray with At Most K Negatives

Constraint-based sliding window.

8. Maximum Subarray with Exactly K Elements

Cool variation for windowing.

9. Maximum Sum of Two Non-Overlapping Subarrays

Two windows, dynamic programming.

10. Maximum Sum of K Non-Overlapping Subarrays

Segment DP, LC hard-style.

🟨 LEVEL 3 — Data Structure Variations

11. Maximum Subarray in 2D Matrix

Find submatrix with max sum.
→ 2D Kadane ($O(N^3)$).

12. Maximum Subarray in a Circular Linked List

Convert to array or use advanced logic.

13. Maximum Subarray in K-Concat Array

Example: array repeated k times.
Use:

- Kadane on one copy
- Kadane on double copy
- handle k ≥ 3 via max-prefix + max-suffix logic.

🟧 LEVEL 4 — Prefix Sum Variants

14. Maximum Subarray With Prefix Method

Classic improvement:
max subarray sum = max(current prefix - minimum prefix so far).

15. Maximum Subarray With Modulo Constraints

Find max sum % m.
→ Uses TreeSet, tricky AF.

16. Maximum Subarray With Limited Memory

Streaming input → must drop elements.

🟥 LEVEL 5 — Advanced / Interview Special Variants

17. Maximum Product Subarray

Kadane's evil twin.
Track both max & min product.

18. Maximum Average Subarray

Fixed/variable lengths.
Binary search on answer + prefix sums.

19. Maximum XOR Subarray

Find subarray with max XOR.
→ Use trie of prefix XORs.

20. Maximum Subarray After One Deletion

Can delete one element to maximize sum.
→ DP using two states:

- no deletion used
- one deletion used

21. Maximum Subarray With One Flip (invert sign)

You may flip exactly one element's sign.

22. Maximum Gain from Stock Prices (Same Logic)

Profit = max subarray sum of differences.

🟪 LEVEL 6 — Trick and Company-Favorite Variants

23. Maximum Sum Increasing Subarray

Elements must be strictly increasing.

24. Maximum Sum Mountain Subarray

Peak in the middle (rise then fall).

25. Maximum Bitonic Subarray Sum

Increasing then decreasing.

26. Maximum Sum with No Adjacent Elements (House Robber Variant)

Not contiguous → DP variation.

27. Max Subarray in Infinite Array

Array is infinite repetition of given nums.

28. Maximum Subarray After Reversal

Reverse one subarray to maximize overall sum.

🟫 LEVEL 7 — Contest / Research Flavor

29. Maximum Subarray Online (Dynamic Input Stream)

Numbers come 1 by 1.
Must update max in real-time.

30. Maximum Subarray with Queries (Segment Tree)

Multiple queries:

- find max subarray in any range
  → Use segment tree with special node structure.

31. Maximum Subarray Under Budget k

Sum ≤ k but as large as possible.
→ Prefix + TreeSet.

32. Maximum Weighted Subarray

Each element has weight; maximize weighted sum.

Longest Substring Without Repeating Characters

🎯 Problem Definition

You are given a string **s**.
Your task is to determine the **length of the longest contiguous substring** in **s** that contains **no repeated characters**.

In simple terms:
Find the maximum stretch in the string where **all characters are unique**.

This is a classic **windowing / sliding-window** problem widely asked in coding interviews (FAANG-standard).

🔢 Input Format

- A single string **s**, containing lowercase/uppercase English letters, digits, symbols, or spaces.

🔙 Output Format

- An integer representing the **maximum length** of a substring without duplicate characters.

📌 Constraints

- 1≤∣s∣≤1051 \leq |s| \leq 10^51≤∣s∣≤105
- String may have any printable ASCII characters.
- Substring must be **contiguous**.
- Characters cannot repeat inside the chosen substring.

🧪 Sample Test Cases

Example 1

Input:

```
s = "abcabcbb"
```

Output:

```
3
```

Explanation:
The substring `"abc"` has all unique characters, and its length is 3.
Although `"abc"` appears twice, the longest unique window length is still **3**.

Coin Change (Minimum Number of Coins)

(LeetCode 322)

📘 Problem: Minimum Coins to Make Amount

🎯 Problem Definition

You are given:

- A list of **coin denominations**:
  C=[c1,c2,c3,...]C = [c_1, c_2, c_3, ...]C=[c1,c2,c3,...]
- A **target amount** AAA

Your task is to compute the **minimum number of coins** required to make the amount **A**.

You can use **unlimited coins** of each denomination.

If it is **impossible** to form the amount using the available coins, return:

`-1`

💡 Core Concept & Optimal Approach: Dynamic Programming (Bottom-Up)

This is a classic **unbounded knapsack problem**.

Key Idea

Build a **DP array**:

dp[x]=minimum coins to make amount xdp[x] = \text{minimum coins to make amount }
xdp[x]=minimum coins to make amount x

Transition:

dp[x]=min(dp[x], 1+dp[x−coin])dp[x] = \min (dp[x],\ 1 + dp[x - coin])dp[x]=min(dp[x],
1+dp[x−coin])

The solution is at:

dp[A]dp[A]dp[A]

Complexity

- **Time:** O(N×A)O(N \times A)O(N×A)
  (N = number of coin types)

- **Space:** O(A)O(A)O(A)

Guaranteed optimal using DP.

🔢 Input Format

- `coins`: integer array containing denominations

- `amount`: target integer

🔙 Output Format

- An integer representing the **minimum number of coins**
- Or `-1` if the amount cannot be formed

🧪 Sample Test Cases

Example 1

Input:

```
coins = [1, 2, 5]
amount = 11
```

Output:

```
3
```

Explanation:
11 can be formed as:

```
5 + 5 + 1 → 3 coins
```

This is the minimum possible.

Merge Intervals (LeetCode 56)

📘 Problem: Merge Overlapping Intervals

🎯 Problem Definition

You are given an array of intervals, where each interval is represented as:

[starti, endi][start_i,\ end_i][starti, endi]

Your task is to **merge all intervals that overlap** and return a **minimal set of non-overlapping intervals** that fully cover the original ranges.

Two intervals **overlap** if:

startnew≤endpreviousstart_{new} \leq end_{previous}startnew≤endprevious

Your final output must contain only **merged**, **sorted**, and **non-overlapping** intervals.

💡 Core Concept & Optimal Strategy

This problem is solved using a **sorting + greedy** approach:

1. **Sort intervals by start time**
   This ensures you always process intervals in increasing order.
   Sorting sets the lower bound of the algorithm at:
   O(NlogN)O(N \log N)O(NlogN)
2. **Linear Scan + Merge**
    - Compare current interval with the previously stored interval
    - If they overlap → extend the previous interval
    - If not → push the previous one and move ahead

This gives an overall complexity of:

O(NlogN)(due to sorting)O(N \log N) \quad (\text{due to sorting})O(NlogN)(due to sorting)

🔢 Input Format

A 2D integer array of size `N × 2`, where each row is an interval:

```
[start_i, end_i]
```

-

🔙 Output Format

- A 2D integer array of merged, non-overlapping intervals.

🧪 Sample Test Cases

Example 1

Input:

```
intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
```

Output:

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

Explanation:

- `[1,3]` and `[2,6]` overlap → merged to `[1,6]`
- `[8,10]` and `[15,18]` do not overlap → kept as-is

Reverse a Singly Linked List

🎯 Problem Definition

You are given the **head** of a singly linked list.
Your task is to **reverse the list in-place** and return the **new head** of the reversed list.

A singly linked list is a chain of nodes where each node contains:

- an integer value
- a pointer/reference to the next node

The goal is to flip the direction of the list so that the last node becomes the first.

💡 Core Concept & Optimal Approach: *In-Place Pointer Manipulation*

The optimal solution uses **three pointers** and runs in **O(N)** time with **O(1)** extra space:

Pointer	Role
prev	Points to the part of the list already reversed (starts as `null`)
current	Points to the node currently being processed

nextTemp	Temporarily stores `current.next` so the list is not lost

✔ Key Idea

Reverse the link direction one node at a time:

```
nextTemp = current.next
current.next = prev
prev = current
current = nextTemp
```

Repeat until the list is fully reversed.

⏳ Time Complexity:

- **O(N)** — each node visited once
- **O(1)** — no extra memory used

🔢 Input Format

- A singly linked list represented by its **head node**

🔙 Output Format

- The **head node** of the reversed linked list

🧪 Sample Test Cases

Example 1

Input:

```
1 → 2 → 3 → 4 → 5 → null
```

Output:

```
5 → 4 → 3 → 2 → 1 → null
```

Explanation:
All pointers are reversed, and the previous tail (5) becomes the new head.

📘 Problem: Binary Tree Level Order Traversal

🎯 Problem Definition

You are given the **root** of a binary tree.
Your task is to perform a **Level Order Traversal** — also known as **Breadth-First Search (BFS)** — and return the values of nodes **level by level**, from **left to right**.

The output must be organized as a **list of levels**, where each level contains all nodes that appear at that depth in the tree.

🧠 Core Concept: BFS Using a Queue

The optimal approach uses a **queue**, because BFS explores the tree **horizontally**, level by level:

1. Push the root into the queue
2. For each level:
    - Process all nodes currently in the queue
    - Add their children into the queue
3. Continue until the queue is empty

This ensures left → right traversal at every level.

📈 Optimal Complexity

- **Time:** O(N)O(N)O(N) — each node visited once
- **Space:** O(N)O(N)O(N) — queue stores up to one full level of the tree

---

🔢 Input Format

- A binary tree represented by its **root node**

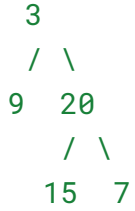
🔙 Output Format

- A list of lists, where each inner list contains the node values of a level (from left to right)

🧪 Sample Test Cases

Example 1

Input Tree:



Output:

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

Explanation:

- Level 0 → [3]
- Level 1 → [9, 20]
- Level 2 → [15, 7]

Lowest Common Ancestor (LCA) in a Binary Search Tree (BST)

(LeetCode 235)

📘 Problem: Lowest Common Ancestor in a BST

🎯 Problem Definition

You are given the **root** of a **Binary Search Tree (BST)** and two **distinct nodes** p and q, both guaranteed to exist in the tree.
Your task is to find the **Lowest Common Ancestor (LCA)** of these two nodes.

✨ What is the LCA?

The **Lowest Common Ancestor** is the **deepest** node in the tree that has **both p and q as descendants** (a node can be a descendant of itself).

💡 Core Concept: BST Property

A **BST** has a unique structural advantage:

For any node `x`:

- All values in the **left subtree** < `x.val`
- All values in the **right subtree** > `x.val`

This allows a greedy, directional search:

✔ LCA Logic in BST

- If **both p and q < root**, LCA must be in the **left subtree**
- If **both p and q > root**, LCA must be in the **right subtree**
- Otherwise → the current node is the **LCA**

This gives an optimal time complexity of:

- **Time:** O(H)O(H)O(H) where H = height of BST (best: logN, worst: N)
- **Space:** O(1)O(1)O(1)

🔢 Input Format

- `root` → the root node of the BST
- `p` → a node in the BST
- `q` → another node in the BST

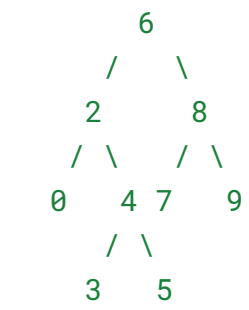
🔙 Output Format

- Returns the **node** representing the LCA of `p` and `q`

🧪 Sample Test Cases

Example 1

Input Tree:



Input:

```
p = 2, q = 8
```

Output:

```
6
```

Explanation:

- 2 is in left subtree
- 8 is in right subtree
- The split happens at 6 → LCA = **6**

Kth Largest Element in a Stream

(LeetCode 703)

📘 Problem: Kth Largest Element in an Array/Stream

🎯 Problem Definition

You need to design a class that supports a **dynamic stream of integers**.
The class is initialized with:

- an integer **k**
- an initial array of integers **nums**

As new values arrive through the method:

```
add(val)
```

the method must:

✔ Insert the value into the stream
✔ Return the **k-th largest** element **seen so far**

The data structure must efficiently maintain the **top k largest elements** at all times.

💡 Core Concept & Optimal Approach: Min-Heap of Size K

To keep retrieving the **k-th largest**, maintain a **min-heap** with exactly **k elements**:

- The **root of the min-heap** always stores the **k-th largest element**
- When a new value comes in:
    - Add it to the heap
    - If heap size > k → remove the smallest
- This ensures the heap always tracks the top k largest elements

Optimal Complexity

- Insert: **O(log k)**
- Space: **O(k)**

This is the gold-standard approach for streaming data.

🔢 Input Format

The system will call:

```
KthLargest(k, nums)
```

Then for each operation:

```
add(val)
```

Each call returns the **k-th largest** element so far.

🔙 Output Format

Each call to `add(val)` returns an **integer** → the **k-th largest** item at that moment.

🧪 Sample Test Case

Example

Input:

```
k = 3
nums = [4, 5, 8, 2]

add(3)
add(5)
add(10)
add(9)
add(4)
```

Output:

```
4
5
5
8
8
```

Explanation:

Initialize with `[4,5,8,2]` → The top 3 largest are `[5,8,4]` → 3rd largest is **4**

📘 Problem: Detect Cycle in an Undirected Graph

🎯 Problem Definition

You are given an **undirected graph**, typically represented using an **adjacency list**.
Your task is to **determine whether the graph contains any cycle**.

A **cycle** exists if you can start from some node and return back to it by following edges, **without reusing the same edge**.

💡 Core Concept: DFS With Parent Tracking

Cycle detection in an undirected graph relies on **Depth-First Search (DFS)**.

During DFS from a node **u**, when exploring its neighbors:

- If a neighbor **v** is **not visited**, continue DFS
- If neighbor **v** *is visited* **AND** `v != parent of u`
  → This is a **back-edge** → **cycle detected**

The critical part is passing the **parent node** in DFS so we can differentiate between:

- A legitimate step back to the parent
- A real cycle

📈 Optimal Complexity

- **Time:** O(V+E)O(V + E)O(V+E)
- **Space:** O(V)O(V)O(V) for visited array + recursion stack
  (V = number of vertices, E = number of edges)

🔢 Input Format

- `V`: number of vertices
- `adj`: adjacency list (array/list of lists)
  Example: `adj.get(u)` → list of neighbors of `u`

🔙 Output Format

- Return:
  - `true` → if the graph contains a cycle
  - `false` → otherwise

🧪 Sample Test Cases

Example 1

Input:

```
V = 5
Edges:
0 - 1
1 - 2
2 - 0
3 - 4
```

Graph Visualization:

Output:

```
true
```

Explanation:
Nodes 0 → 1 → 2 → 0 form a cycle.