# Mastering Must-Know Coding Problems for Interview Prep

## Array

### Maximum Subarray
Given an integer array, arr, return the sum of the subarray with the largest sum.

**Example:**

| Input | Output |
|---|---|
| -3, -1, 2, 5, 7, -4 | 14 |

**Naive approach**
Find all possible subarrays, which are O(n^2) in number/total. Calculate the sum of each subarray while maintaining the maximum sum.

**Complexity**

| Time | $O(n^3)$ | Space | O(1) |
|---|---|---|---|

**Optimal approach**
Initialize two variables, curr and max, with the value of the first element in the array. Iterate through the array, updating curr by adding each element. If curr becomes negative, reset it to zero. Update max whenever a new maximum is found. After traversing the whole array, return max.

**Complexity**

| Time | O(n) | Space | O(1) |
|---|---|---|---|

### Product of Array Except Self
Given an integer array, arr, return an array, prod, so that prod[i] is equal to the product of all the elements of arr except arr[i].

**Example:**

| Input | Output |
|---|---|
| 1, 2, 3, 4 | 24, 12, 8, 6 |

**Naive approach**
For each element, calculate the product of all other elements except the current one.

**Complexity**

| Time | $O(n^2)$ | Space | O(1) |
|---|---|---|---|

**Optimal approach**
Populate the resultant array, res, where res[i] contains the product of all the numbers to the left of i with res[0]=1. Initialize a variable, R=1, to keep track of the running product of elements to the right of each index. For each index i from the end of the array, update res as res[i]=res[i]*R and R as R=R*arr[i].

**Complexity**

| Time | O(n) | Space | O(1) |
|---|---|---|---|

## Hash Map

### Two Sum
Given an array of integers arr and an integer t, return the indexes of the two numbers that add up to t.

**Example:**

| Input | Output |
|---|---|
| 8, 13, 6, 4<br>10 | 2, 3 |

**Naive approach**
Iterate through every pair of elements in the array and check if their sum equals the target.

**Complexity**

| Time | $O(n^2)$ | Space | O(1) |
|---|---|---|---|

**Optimal approach**
Calculate complement of each element by subtracting it from the target sum. Return the indices of the two numbers if the complement is already in the hash map. Otherwise, insert the current element and its index in the hash map.

**Complexity**

| Time | O(n) | Space | O(n) |
|---|---|---|---|

### First Unique Character in a String
Given a string, return the index of the first non-repeating character in it. If it does not exist, return -1.

**Example:**

| Input | Output |
|---|---|
| "popcorn" | 3 |

**Naive approach**
For each character in the string, count its occurences. The first character with a count of 1 is considered the first unique character.

**Complexity**

| Time | $O(n^2)$ | Space | O(1) |
|---|---|---|---|

**Optimal approach**
Store the frequency of each character in a hash map. Then, iterate through the string again checking the frequency of each character in the hash map, and return the index of the first character with a frequency of 1.

**Complexity**

| Time | O(n) | Space | O(n) |
|---|---|---|---|

educative

# Stack

### Valid Parentheses

Given a string containing the characters '(', ')', '{', '}', '[', and ']', check if the input string is valid. An input string is valid if the open brackets are closed in the correct order by the same type of brackets.

**Example:**

| Input | Output |
|---|---|
| `"{([])}"` | `true` |

**Naive approach**

Check all combinations of parentheses for balance.

**Complexity**

| Time | $O(n^2)$ | Space | O(1) |
|---|---|---|---|

**Optimal approach**

For each character, push it to a stack if it's an opening parenthesis and pop the stack if it's a closing parenthesis. If the popped (opening) parenthesis doesn't match the closing parenthesis, return false. After traversing the whole string, return true if the stack is empty. Otherwise, return false.

**Complexity**

| Time | O(n) | Space | O(n) |
|---|---|---|---|

### Next Greater Element

Given an array of integers arr, find the next greater element for each element in the array. The next greater element for an element x is the first greater element to its right. If there is no greater element to its right, the answer for that element is -1.

**Example:**

| Input | Output |
|---|---|
| 1 2 3 4 5 | 2 3 4 5 -1 |

**Naive approach**

For each element, iterate through the elements to its right to find the next greater element.

**Complexity**

| Time | $O(n^2)$ | Space | O(1) |
|---|---|---|---|

**Optimal approach**

Initialize an empty stack. Traverse the array from right to left, and for each element E and stack top T:
- If the stack is empty, assign the Next Greater Element (NGE) of E as -1.
- If E is less than T, assign the NGE of E as T.
- While E is greater than T, pop elements from the stack.
- Push E onto the stack.

**Complexity**

| Time | O(n) | Space | O(n) |
|---|---|---|---|

# Queue

### Generate Binary Numbers from 1 to N

Given a positive integer N, generate binary representations of all numbers from 1 to N in the form of strings.

**Example:**

| Input | Output |
|---|---|
| 3 | `["1", "10", "11"]` |

**Naive approach**

For each decimal number from 1 to N, convert it into its binary representation.

**Complexity**

| Time | O(n log n) | Space | O(1) |
|---|---|---|---|

**Optimal approach**

Initialize a queue with "1". Dequeue and add to the result array. Generate next two binary numbers by appending "0" and "1" to the dequeued number, and enqueue them. Repeat this process for N numbers.

**Complexity**

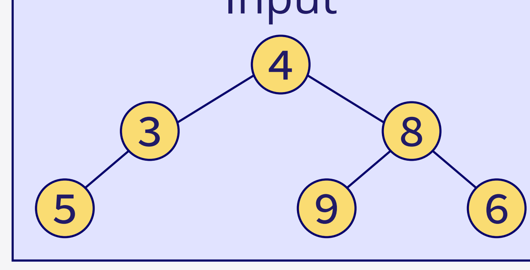
| Time | O(n) | Space | O(n) |
|---|---|---|---|

### Binary Tree Level Order Traversal

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

**Example:**

Input




Output

`[4][3,8][5,9,6]`

**Naive approach**

Calculate the height, H, of the tree. Then, for each level from 1 to H, run a recursive function (on the left and right child respectively) from the root node and pass the current level along. During recursion, print the node whenever the current level matches the node's level.

**Complexity**

| Time | $O(n^2)$ | Space | O(n) |
|---|---|---|---|

**Optimal approach**

Initialize a queue with the root node. Dequeue a node, visit it, and enqueue its children if they exist. Repeat until the queue is empty.

**Complexity**

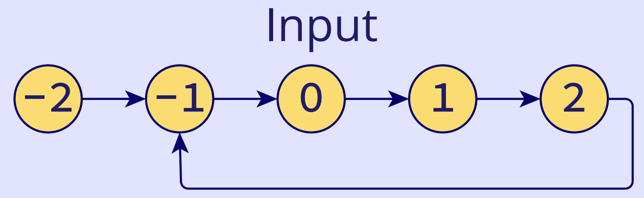
| Time | O(n) | Space | O(n) |
|---|---|---|---|

# Linked List

### Linked List Cycle
Given the head of a linked list, determine if the linked list contains a cycle.

**Example:**

Input




Output

`true`

**Naive approach**
Use a hash map to keep track of the visited node. Traverse the linked list and check whether the current node has been visited before.

**Complexity**

| | | | |
|---|---|---|---|
| Time | O(n) | Space | O(n) |

**Optimal approach**
Initialize two pointers, slow and fast, at the head node. Move slow and fast pointers, one and two steps, respectively. If there is a cycle, both pointers will enter the cycle and keep iterating within it until they meet, indicating the presence of a cycle.Otherwise, the fast pointer will reach the end of the list.

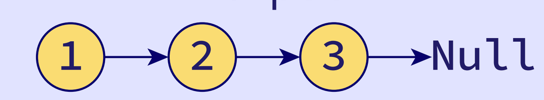
**Complexity**

| | | | |
|---|---|---|---|
| Time | O(n) | Space | O(1) |

### Palindrome Linked List
Given a singly linked list, return true if it is a palindrome or false if it's not.

**Example:**

Input




Output

`false`

**Naive approach**
Create a reversed copy of the linked list and compare it with the original.

**Complexity**

| | | | |
|---|---|---|---|
| Time | O(n) | Space | O(n) |

**Optimal approach**
Use slow and fast pointers to find the middle of the list and reverse the second half in-place. Compare each node in the first half with the corresponding node in the reversed second half.

**Complexity**

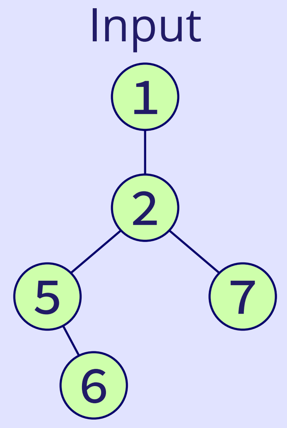
| | | | |
|---|---|---|---|
| Time | O(n) | Space | O(1) |

# Binary Tree

### Balanced Binary Tree
Given a binary tree, check if it is height-balanced.

**Example:**

Input




Output

`False`

**Naive approach**
For each node, calculate the height of the left and right subtrees. If the difference between their heights is greater than 1, return false. If the whole tree has been traversed, return true.

**Complexity**

| | | | |
|---|---|---|---|
| Time | O(n log n) | Space | O(1) |

**Optimal approach**
Calculate the height of each subtree using the post-order traversal, given that:
- The height of a leaf node is 0
- The height of a subtree is the maximum of left and right subtrees' heights plus 1.

Use the heights as follows:
- If the difference between heights of left and right subtrees of any node is greater than 1, return false.
- If the whole tree has been traversed, return true.

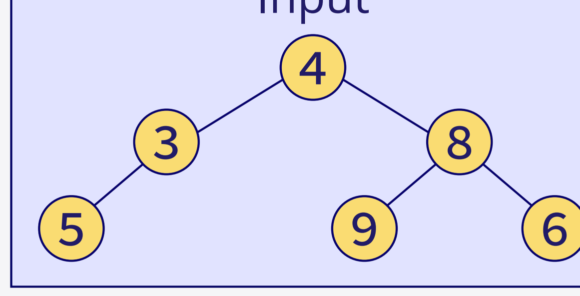
**Complexity**

| | | | |
|---|---|---|---|
| Time | O(n) | Space | O(h) |

### Diameter of Binary Tree
Given a binary tree, return the length of its diameter, where the diameter of a binary tree is the length of the longest path between any two nodes.

**Example:**

Input




Output

4

**Naive approach**
For each node, find the distance to every other node, and keep track of the maximum value.

**Complexity**

| | | | |
|---|---|---|---|
| Time | $O(n^2)$ | Space | O(1) |

**Optimal approach**
Find the diameter of a binary tree by checking the heights of each node's left and right subtrees during a bottom-up traversal. Keep track of the maximum diameter globally, and update it whenever a new maximum diameter is found.

**Complexity**

| | | | |
|---|---|---|---|
| Time | O(n) | Space | O(n) |

educative

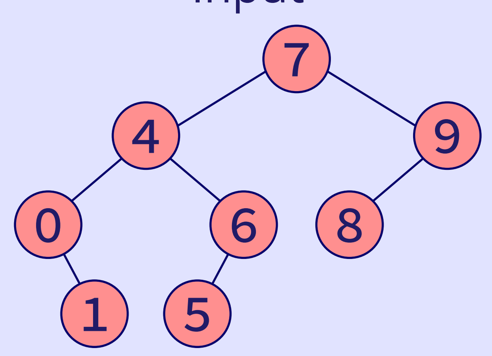
# Binary Search Tree

### Second Minimum Node in a Binary Search Tree

Given a binary search tree, return the value of the second minimum node. If there is no second minimum node, return -1.

**Example:**

Input




Output

1

**Naive approach**

Traverse the tree, store all unique node values in a set, and find the second minimum value.

**Complexity**

| Time | O(n) | Space | O(n) |
|---|---|---|---|

**Optimal approach**

Start the in-order traversal of the BST as the in-order traversal visits nodes in ascending order. Initialize a counter with 0, and increment it as soon as you visit a node. The node on which the counter becomes 2 is the second minimum node.

**Complexity**

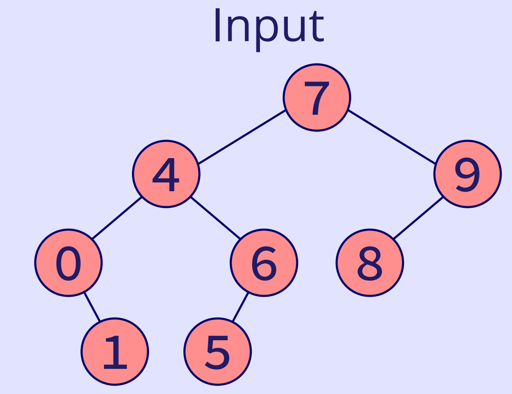
| Time | O(n) | Space | O(h) |
|---|---|---|---|

### Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree, find the lowest common ancestor (LCA) node of two given nodes.
The LCA of two nodes is the lowest node in the tree that has both nodes as descendants, where a node can be a descendant of itself.

**Example:**

Input




| 1 | 6 |
|---|---|

Output

4

**Naive approach**

Find and store the paths to both nodes and compare these paths to find the last common node.

**Complexity**

| Time | O(log n) | Space | O(n) |
|---|---|---|---|

**Optimal approach**

Traverse the BST while comparing the given nodes with the current node:
- If both values are smaller, move to the left subtree.
- If both values are greater, move to the right subtree.
- If one value is smaller and the other one is greater, it indicates that both nodes lie in different subtrees. Return the current node as the Lowest Common Ancestor (LCA).

**Complexity**

| Time | O(n) | Space | O(h) |
|---|---|---|---|

# A Comprehensive Look at Time Complexities of Essential Data Structures

| # | Data Structure | Operation | Time Complexity | |
|---|---|---|---|---|
| | | | Average | Worst Case |
| 1 | **Array** | Search | θ(n) | O(n) |
| | | Insert | | |
| | | Delete | | |
| 2 | **Hash Map** | Search | θ(1) | O(n) |
| | | Insert | | |
| | | Delete | | |
| 3 | **Stack** | Search | θ(n) | O(n) |
| | | Insert | θ(1) | O(1) |
| | | Delete | | |
| 4 | **Queue** | Search | θ(n) | O(n) |
| | | Insert | θ(1) | O(1) |
| | | Delete | | |

| # | Data Structure | Operation | Time Complexity | |
|---|---|---|---|---|
| | | | Average | Worst Case |
| 5 | **Linked List** | Search | θ(n) | O(n) |
| | | Insert | θ(1) | O(1) |
| | | Delete | θ(n) | O(n) |
| 6 | **Binary Tree** | Search | θ(n) | O(1) |
| | | Insert | | |
| | | Delete | | |
| 7 | **Binary Search Tree** | Search | θ(log n) | O(n) |
| | | Insert | | |
| | | Delete | | |