Algorithm and Problem Solving Cheatsheet in C++

Data Structures, Algorithms and Coding Interview Problem Patterns in C++

Rui F. David, 2025

rfdavid.com

MOTIVATION

Motivation

The tech industry hiring standard is based on algorithm and data structure.

There are plenty of free resources available around algorithms and data structures. The purpose of this project is to be a quick guide where you can learn and review learned algorithms and data structures.

Some of the intended **key features:**

- Non-verbose, short-structured, and easy to follow descriptions
- Slide-based, practical for reviewing
- Free and open-source

⭐ **If you like, please add a star at *github.com/rfdavid/cpp-algo-cheatsheet***

Useful links

Tech Interview Handbook
https://www.techinterviewhandbook.org
A very well-structured resource for interview preparation

TUF
https://takeuforward.org/interviews/blind-75-leetcode-problems-detailed-video-solutions
Contains explanation and some videos for the problems from blind 75 list

Blind 75 Leetcode Questions
https://leetcode.com/discuss/general-discussion/460599/blind-75-leetcode-questions

Blind 75

- Blind 75 is a popular list of algorithm problems that intends to cover the main data structures and patterns.
- It is a curated list of 75 popular coding questions created by an ex-Meta Staff Engineer

Array

✓ Two Sum

✓ Best Time to Buy and Sell Stock

Contains Duplicate

Product of Array Except Self

Maximum Subarray

Maximum Product Subarray

Find Minimum in Rotated Sorted Array

Search in Rotated Sorted Array

3 Sum

Container With Most Water

Binary

Sum of Two Integers

Number of 1 Bits

Counting Bits

Missing Number

Reverse Bits

Dynamic Programming

✓ Climbing Stairs

Coin Change

Longest Increasing Subsequence

Longest Common Subsequence

Word Break Problem

Combination Sum

House Robber

House Robber II

Decode Ways

Unique Paths

Jump Game

Matrix

Set Matrix Zeroes

Spiral Matrix

Rotate Image

Word Search

Blind 75

Tree

- ✓ Maximum Depth of Binary Tree
- Same Tree
- Invert/Flip Binary Tree
- Binary Tree Maximum Path Sum
- Binary Tree Level Order Traversal
- Serialize and Deserialize Binary Tree
- Subtree of Another Tree
- Construct Binary Tree from Preorder and Inorder Traversal
- Validate Binary Search Tree
- Kth Smallest Element in a BST
- Lowest Common Ancestor of BST
- Implement Trie (Prefix Tree)
- Add and Search Word
- Word Search II

Heap

- Merge K Sorted Lists
- Top K Frequent Elements
- Find Median from Data Stream

String

- ✓ Longest Substring Without Repeating Characters
- Longest Repeating Character Replacement
- Minimum Window Substring
- Valid Anagram
- Group Anagrams
- ✓ Valid Parentheses
- Valid Palindrome
- Longest Palindromic Substring
- Palindromic Substrings
- Encode and Decode Strings ⭐

Linked List

- Reverse a Linked List
- Detect Cycle in a Linked List
- Merge Two Sorted Lists
- Merge K Sorted Lists
- Remove Nth Node From End Of List
- Reorder List

Graph

- ✓ Clone Graph
- Course Schedule
- Pacific Atlantic Water Flow
- Number of Islands
- Longest Consecutive Sequence
- Alien Dictionary ⭐
- Graph Valid Tree ⭐
- Number of Connected Components in an Undirected Graph ⭐

Interval

- Insert Interval
- Merge Intervals
- Non-overlapping Intervals
- Meeting Rooms ⭐
- Meeting Rooms II ⭐

⭐ = leetcode premium

Other problems

Tree

- ✓ Maximum Level Sum of a Binary Tree
- ✓ Minimum Number of Increments on Subarrays to Form a Target Array

⭐ = leetcode premium

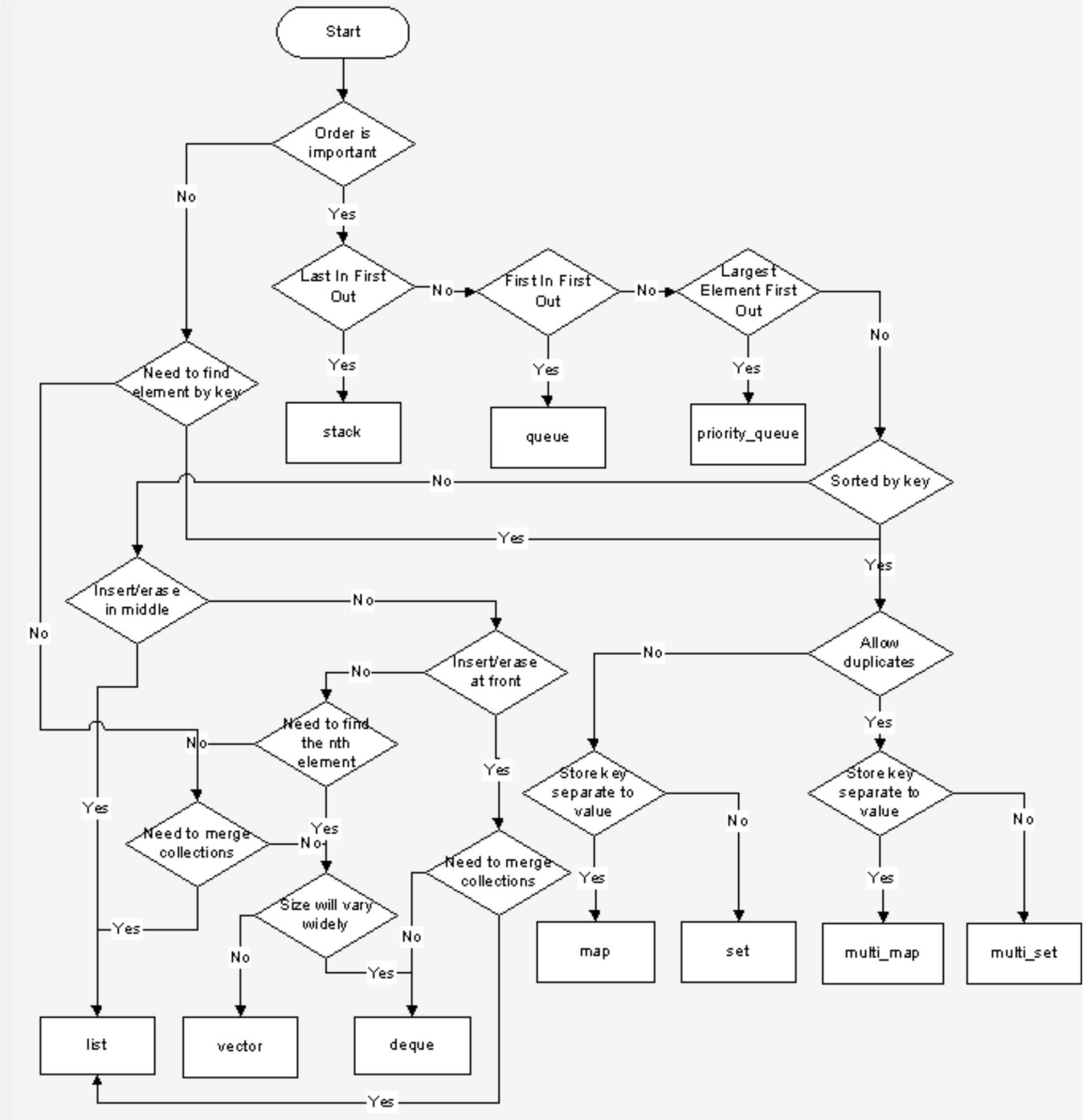
DATA STRUCTURES

Vectors

- std::vector is a sequence container that encapsulates dynamic sized arrays*

*https://en.cppreference.com/w/cpp/container/vector

Data Structure Decision Diagram



Note: I don't have the source of this diagram. If you know it, please drop me a msg so I can add it here.

- The following diagram gives you the direction to which data structure to use in C++ according to the problem you are trying to solve

ARRAY

Arrays

Characteristics

- **Memory layout**: hold values in a **contiguous** block of memory.
- **Fixed Size**: the size of an array is defined when it is created and cannot be changed. However, high-level languages have different implementations, making it dynamic.
- **Homogeneous elements:** all elements are of the same data type (int, float, char…)
- **Efficiency**: accessing elements by index is very efficient *O(1)*, since each index maps directly to a memory location. Also, range scans benefit from CPU cache lines since arrays are stored in contiguous blocks of memory.

Arrays – Kadane's algorithm

Problem – Two Sum

Easy

LeetCode leetcode.com/problems/two-sum

Problem Statement

- Given an **`array`** of numbers and a **`target`**, example: **`array [2,7,11,15]`** and **`target 9`**
- Return indices of two numbers where they add up to **`target`**
- **Output**: [0,1]

  `array[0] + array[1] = 2 + 7 = 9`

Solution – Two Sum

Easy

LeetCode leetcode.com/problems/two-sum

Solution

- Iterative over each number in the array
- Calculate the difference between target and each number, example:

  `array[0] = 2, target 9, then 9 - 2 = 7`
- Now we know we need the number **7** to sum up to **9**
- Check in a *hashmap* if we have 7 in some part of the array

  `hash[7] exists?`
- If yes, return the current index and the index of 7
- If not, store the index of the current number in the hashmap for future evaluation

  `hash[2] = 0`

Code – Two Sum

Easy

LeetCode leetcode.com/problems/two-sum

Code O(n)

```
vector<int> twoSum(vector<int>& nums, int target) {
    std::unordered_map<int, int> numMap;
    // O(n)
    // n being the size of nums
    for (int i = 0; i < nums.size(); i++) {
        // current number of the array
        int number = nums[i];
        int diff = target - number;

        // check if the difference is in some part of the array
        // by using a hashmap
        if (numMap.find(diff) != numMap.end()) {
            return { numMap[diff], i};
        }

        // register the current number index
        numMap[number] = i;
    }
    // no matches
    return {};
}
```

Problem - Best Time to Buy and Sell Stock

Easy

https://leetcode.com/problems/best-time-to-buy-and-sell-stock

You are given an array **prices** where **prices[i]** is the price of a given stock on the i^{th} 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.

Solution - Best Time to Buy and Sell Stock

Easy

https://leetcode.com/problems/best-time-to-buy-and-sell-stock

```
int maxProfit(vector<int>& prices) {
    int profit = 0;
    int buy = prices[0];
    for (auto i = 1; i < prices.size(); i++) {
        if (prices[i] < buy) {
            buy = prices[i];
        } else if (prices[i] - buy > profit) {
              profit = prices[i] - buy;
        }
    }
    return profit;
}
```

STRING

Problem – Valid Parentheses

Easy

LeetCode leetcode.com/problems/valid-parentheses

Problem Statement

- Given a string …

Solution – Valid Parentheses

Easy

LeetCode leetcode.com/problems/valid-parentheses

Solution

- Explain…

Code – Valid Parentheses

Easy

LeetCode leetcode.com/problems/valid-parentheses

Code

```
bool isValid(string s) {
    // stack (LIFO)
    std::stack<char> brackets;
    // O(n)
    for (int i = 0; i < s.size(); ++i) {
        char bracket = s[i];
        if (bracket == '(' || bracket == '[' || bracket == '{') {
            brackets.push(bracket);
        } else {
            if (brackets.size() == 0) return false;
            char lastBracket = brackets.top();
            if (bracket == ')' && lastBracket != '(') return false;
            if (bracket == '}' && lastBracket != '{') return false;
            if (bracket == ']' && lastBracket != '[') return false;
            brackets.pop();
        }
    }
    // all brackets must be closed
    return brackets.size() == 0;
}
```

Problem – Minimum Number of Increments on Subarrays

Hard

LeetCode leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array

Problem Statement

Given an array of integers initialized with zeros (example [0,0,0,0]), the goal is to reach some target (example [1, 2, 2, 3]). The valid operations is to take a subarray and increment by one. The output is the total number of operations. In this case:

[1,1,1,1] → increment the subarray starting from 0 to total size
[1,2,2,2] → increment the subarray starting from 1 to total size
[1,2,2,3] → increment the subarray starting and ending from the last element

Output: 3 (total number of operations)

Solution – Minimum Number of Increments on Subarrays

Hard

LeetCode leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array

Solution

- Explain…

Code – Minimum Number of Increments on Subarrays

Hard

LeetCode leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array

Code

```
int minNumberOperations(vector<int>& target) {
    int totalOp = target[0];
    for (int i = 1; i < target.size(); ++i) {
        // can't reuse
        if (target[i - 1] < target[i]) {
            totalOp += target[i] - target[i - 1];
        }
    }
    return totalOp;
}
```

Code (2) – Minimum Number of Increments on Subarrays

Hard

LeetCode leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array

Code

```
int minNumberOperations(vector<int>& target) {
    return target[0] +
        inner_product(target.begin() + 1, target.end(),
                target.begin(), 0,
                plus<int>(),
                [](int curr, int prev) { return max(curr - prev, 0); });
}
```

BINARY

Bit Manipulation in C

- **Operators**

  & AND **|** OR **^** XOR **~** NOT **<<** LEFT SHIFT **>>** RIGHT SHIFT

- **Common Operations**

  set bit: `num |= (1 << pos)`

  clear bit: `num &= (1 << pos)`

  toggle bit: `num ^= (1 << pos)`

  check bit: `(num & (1 << pos)) != 0`

  extract bit: `(num >> pos) & 1`

  extract a range of bits: `(num >> pos) & ((1 << length) - 1)`

- **Example**

```
void copyBit(int *dst, int src, int srcPos, int dstPos) {
      int bit = (src >> srcPos) & 1;  // extract bit
      *dst &= ~(1 << dstPos);  // clear destination bit
      *dst |= (bit << dstPos);  // set destination bit
}
```

Binary

- In C++, **`std::bitset`** represents a fixed-size sequence of N bits
- Example:

```
std::bitset<8> bitmask;
bitmask.reset(1)
bitmask.set(1)
if (bitmask.test(1)) { // true
…
```

- **reset** : set bit to false
- **set** : set a specific bit
- **test** : check a specific bit
- **count** : return the number of bits set to true
- **flip** : toggle the value of the bits (if true, set to false and vice-versa)

Problem 3 – Longest Substring Without Repeating Characters

Medium

LeetCode leetcode.com/problems/longest-substring-without-repeating-characters

Given a string, the goal is to find the longest substring without repeating characters.

For example, given a string “abcdb”, it should return 4:

abcd since “b” is repeated

Solution 3 – Longest Substring Without Repeating Characters

Medium

- Using bitset: create a bitmask with 128 bits where each bit represent a character
- Use sliding window represented by **left** and **right**
- Set the bit to true for each character found
- Check if it is repeated and reset the bit

```
int lengthOfLongestSubstring(string s) {
    std::bitset<128> bitmask;
    uint32_t left = 0;
    uint32_t maxLength = 0;

    for (uint32_t right = 0; right < s.length(); ++right) {
        uint32_t bitIndex = s[right];
        // if char is already in the bitmask, move left until we reset the bits
        while (bitmask.test(bitIndex)) {
            bitmask.reset(s[left]);
            ++left;
        }

        bitmask.set(bitIndex);
        maxLength = std::max(maxLength, right - left + 1);
    }
    return maxLength;
}
```

Negabinary

- Non-standard positional numeral system that uses base of -2
- Allow representing negative numbers in binary
- Example:

  1101_{-2}

  $(-2)^3 + (-2)^2 + 0 + (-2)^0 = -8 + 4 + 0 + 1 = -3$

Summing Negabinary

- Add as a regular binary number, but with **negative carry**

  $0 + 0 = 0$

  $1 + 0 = 1$

  $1 + 1 = 0$ with a negative carry 1

  $\mathbf{1} + 1 = 0$ (subtract)

  $\mathbf{1} + 0 = 1$ with a positive carry 1

Negabinary

Example 1

red 1 = negative carry

green 1 = regular carry

$$
\begin{array}{r}
11\ \ 1\\
1011\\
+\,1110\\
\hline
=\boxed{110001}
\end{array}
$$

1 + 0 = 1

1 + 1 = 0 with negative carry 1

1 + 1 = 0

1 + 1 = 0 with negative carry 1

1 + 0 = 1 with positive carry 1

1 + 0 = 1

Example 2

$$
\begin{array}{r}
1111\\
101010\\
+\,101100\\
\hline
=11110110
\end{array}
$$

Reference

https://math.stackexchange.com/questions/3251605/how-to-add-negabinary-numbers

Problem 1073 – Adding Two Negabinary Numbers

Medium

https://leetcode.com/problems/adding-two-negabinary-numbers

Given two numbers **arr1** and **arr2** in base -2, return the result of adding them together.

Each number is given in *array format*: as an array of 0s and 1s, from most significant bit to least significant bit. For example, **arr = [1,1,0,1]** represents the number **(-2)^3 + (-2)^2 + (-2)^0 = -3.** A number **arr** in array, format is also guaranteed to have no leading zeros: either **arr == [0]** or **arr[0] == 1**.

Return the result of adding **arr1** and **arr2** in the same format: as an array of 0s and 1s with no leading zeros.

Example 1

Input: arr1 = [1,1,1,1,1], arr2 = [1,0,1]

Output: [1,0,0,0,0]

Explanation: arr1 represents 11, arr2 represents 5, the output represents 16.

Example 2

Input: arr1 = [0], arr2 = [0]

Output: [0]

Example 3

Input: arr1 = [0], arr2 = [1]

Output: [1]

Solution 1073 – Adding Two Negabinary Numbers

Medium

https://leetcode.com/problems/adding-two-negabinary-numbers

GRAPH (DFS)

Problem - Keys and Rooms

Medium

https://leetcode.com/problems/keys-and-rooms

```
int maxProfit(vector<int>& prices) {
    int profit = 0;
    int buy = prices[0];
    for (auto i = 1; i < prices.size(); i++) {
        if (prices[i] < buy) {
            buy = prices[i];
        } else if (prices[i] - buy > profit) {
              profit = prices[i] - buy;
        }
    }
    return profit;
}
```

Problem – Clone Graph

Medium

LeetCode https://leetcode.com/problems/clone-graph

Problem Statement

- Given a node reference, create a deep copy of the graph
- The class node has two variables: val and neighbours

```
class Node {
  public int val;
  public List<Node> neighbors;
}
```

- **Output** is the node reference of the copy

Solution – Clone Graph

Medium

LeetCode https://leetcode.com/problems/clone-graph

Solution

- First check the edge cases (is the node null?)
- Create a hash map to store the nodes that is already created

```
unordered<int, Node*> graph;
```

- Check if the current node already exists in the graph
- If not, create a new Node object and store in the hashmap
- Visit all the neighbors and add the neighbors to this current node

Code – Clone Graph

Medium

LeetCode https://leetcode.com/problems/clone-graph

```cpp
std::unordered_map<int, Node*> graph;

Node* cloneGraph(Node* node) {
    if (node == NULL) {
        return NULL;
    }
    // does this node object exists?
    if (graph.find(node->val) == graph.end()) {
        // node wasn't visited yet, store in the hashmap
        graph[node->val] = new Node(node->val);
        // visit all neighbours
        for (const auto& n : node->neighbors) {
            graph[node->val]->neighbors.push_back(cloneGraph(n));
        }
    }
    return graph[node->val];
}
```

GRAPH (BFS)

Problem – Maximum Level Sum of a Binary Tree

Medium

LeetCode https://leetcode.com/problems/maximum-level-sum-of-a-binary-tree

Problem Statement

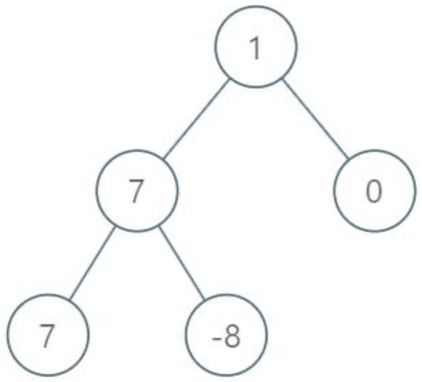
- Given the root of a binary tree, find the smallest level with the maximum sum
- For example, the tree below has the follow sums for each level:

  level 1 (root) = 1

  level 2 = 7 + 0 = 7

  level 3 = 7 – 8 = -1

- Therefore, **level 2** has the maximum sum



Solution – Maximum Level Sum of a Binary Tree

Medium

LeetCode https://leetcode.com/problems/maximum-level-sum-of-a-binary-tree

Solution

- Have a queue with the nodes for the current level
- Sum the values from that level by taking the nodes from the queue
- Example, we know that level 1 has one node. Hence, pop the first node from the queue

  If level 2 has 2 nodes, pop two nodes, sum the values
- In addition, add left and right to the end of the queue to process the next level

Code – Maximum Level Sum of a Binary Tree

Medium

LeetCode https://leetcode.com/problems/maximum-level-sum-of-a-binary-tree

```
int maxLevelSum(TreeNode* root) {
    std::queue<TreeNode*> nodes;
    int currentLevel = 0;
    int maxLevel = 1;
    int maxSum = INT_MIN;

    nodes.push(root);

    // traverse the graph
    while(!nodes.empty()) {
        int levelSum = 0;
        int levelSize = nodes.size();
        currentLevel++;

        // sum the values in current level
        for (int i = 0; i < levelSize; ++i) {
            TreeNode* node = nodes.front();
            levelSum += node->val;
            nodes.pop();

            if (node->left) nodes.push(node->left);
            if (node->right) nodes.push(node->right);
        }

        if (levelSum > maxSum) {
            maxLevel = currentLevel;
            maxSum = levelSum;
        }
    }

    return maxLevel;
}
```

SHORTEST PATH

Shortest Path Algorithms

Algorithms

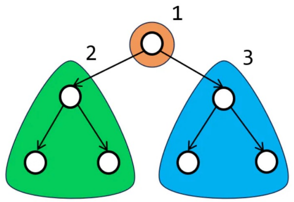
- BFS
- Dijkstra
- Bellman-Ford
- Floyd-Warshall
- A* search
- Johnson's
- SPFA (Shortest Path Faster)
- Bidirectional Search

TREE

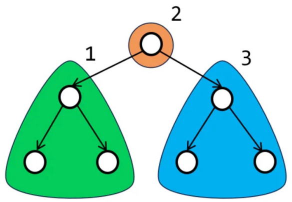
Tree Traversals

Depth-First Traversals

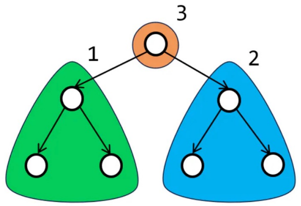
- **Pre-order**: Root – Left – Right



- **In-order**: Left – Root – Right



- **Post-order**: Left – Right – Root



Breadth-First Traversal (Level Order Traversal)

Visit every node on a level before moving to a lower level.

Image credits:
https://deriveit.org/coding/tree-dfs-113

Tree Traversals

Depth-First Traversals

Use a recursive algorithm to traverse according to the order

- **Pre-order**: Root – Left – Right

```
if (!root) return;
doSomething();
visit(node.left);
visit(node.right);
```

- **In-order**: Left – Root – Right

```
if (!root) return;
visit(node.left);
doSomething();
visit(node.right);
```

- **Post-order**: Left – Right – Root

```
if (!root) return;
visit(node.left);
visit(node.right);
doSomething();
```

Tree Traversals

Example of pre-order and in-order

```
struct TreeNode {
    int val;
    TreeNode *left, *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// Pre-order traversal
void preorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->val << " ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}

// In-order traversal
void inorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    inorderTraversal(root->left);
    cout << root->val << " ";
    inorderTraversal(root->right);
}
```

Tree Traversals

Example of post-order and level-order

```cpp
struct TreeNode {
    int val;
    TreeNode *left, *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// Post-order traversal
void postorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    cout << root->val << " ";
}

// Level-order traversal using a queue
void levelOrderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();
        cout << current->val << " ";
        if (current->left != nullptr) q.push(current->left);
        if (current->right != nullptr) q.push(current->right);
    }
}
```

Problem – Maximum Depth of Binary Tree

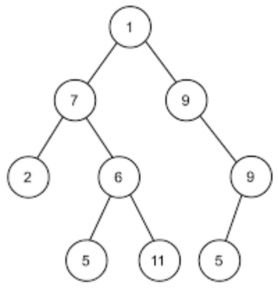
Easy

LeetCode https://leetcode.com/problems/maximum-depth-of-binary-tree

Problem Statement

- Given the root of a binary tree, find the *maximum depth*
- **Example:**

  root = [1,7,9,2,6,null,9,null,null,5,11,5,null]
- **Output**: 4



Solution – Maximum Depth of Binary Tree

Easy

LeetCode https://leetcode.com/problems/maximum-depth-of-binary-tree

Solution

- Perform **post-order** traversal: left - right - root
- Recursively go left and right to find each value
- Return the max of each one

Code – Maximum Depth of Binary Tree

Easy

LeetCode https://leetcode.com/problems/maximum-depth-of-binary-tree

```
int maxDepth(TreeNode* root) {
    if (!root) return 0;
    // find max left
    int maxLeft = maxDepth(root->left);
    // find max right
    int maxRight = maxDepth(root->right);
    // return max +1 (account for root)
    return std::max(maxLeft, maxRight) + 1;
}
```

Problem – Path Sum

Easy

LeetCode https://leetcode.com/problems/path-sum

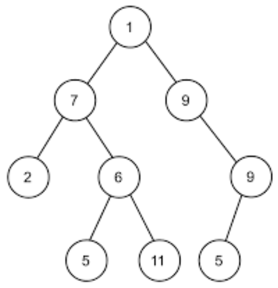
Problem Statement

- It is given the **root** of a binary tree and an integer **target sum**
- **Example:**

  root = [1,7,9,2,6,null,9,null,null,5,11,5,null]

  target sum = 10
- Return true if there is a path from root to leaf that adds up to 10
- **Output**: true

  Node 1 + Node 7 + Node 2 = 10



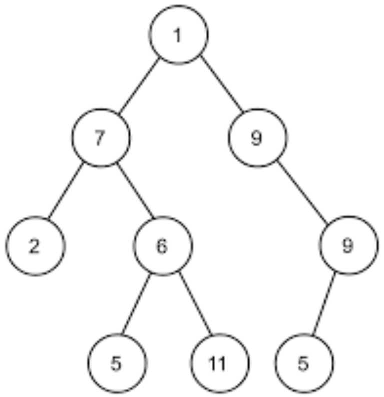
Solution – Path Sum

Easy

LeetCode https://leetcode.com/problems/path-sum

Solution

- Start from root node (1)
- Subtract from target number (example 10 – 1 = 9)
- Continue going down the tree, until the target is 0, return true
- After visiting all nodes, if the target is not zero, return false



Code – Path Sum

Easy

LeetCode https://leetcode.com/problems/path-sum

```
bool hasPathSum(TreeNode* root, int targetSum) {
    if (!root) {
        return false;
    }
    // we want targetSum to be zero
    targetSum -= root->val;
    // if there is no left, no right, we've reached the end of the path
    // so if the targetSum is zero, then the nodes summed up to the targetSum
    if (!root->left && !root->right && targetSum == 0) {
        return true;
    }
    // propagate to left and right
    return hasPathSum(root->left, targetSum) || hasPathSum(root->right, targetSum);
}
```

Also, a small performance tweak can be made by avoiding writing *targetSum:* `targetSum -= root->val`

This will avoid a memory write access, making the calculation directly in the CPU, but also at a cost of readability

```
    if (!root->left && !root->right && targetSum - root->val == 0) {
     …
    return hasPathSum(root->left, targetSum - root->val) || hasPathSum(root->right, targetSum - root->val);
```

Problem – Kth Smallest Element in a BST

Medium

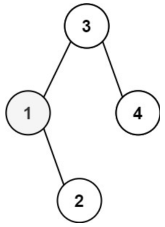
https://leetcode.com/problems/kth-smallest-element-in-a-bst

Given the **root** of a binary search tree, and an integer **k**, return the **k**th smallest value (1-indexed) of all the values of the nodes in the tree.

Example 1

Input: root = [3,1,4,null,2], k = 1

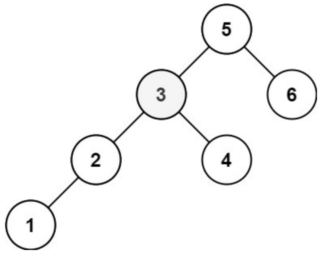
Output: 1



Example 2

Input: root = [5,3,6,2,4,null,null,1], k = 3

Output: 3



Solution – Kth Smallest Element in a BST

Medium

https://leetcode.com/problems/maximum-depth-of-binary-tree

```
int kthSmallest(TreeNode* root, int k) {
    int count = 0;
    int output;
    traverse(root, count, output, k);
    return output;
}

// perform in-order traversal:  left, node, right
void traverse(TreeNode* node, int& count, int &output, int k) {
    if (!node) return;
    traverse(node->left, count, output, k);
    count++;
    if (count == k) {
        output = node->val;
        return;
    }
    traverse(node->right, count, output, k);
}
```

LINKED LIST

Problem – Swap Nodes in Pair

Medium

https://leetcode.com/problems/swap-nodes-in-pairs

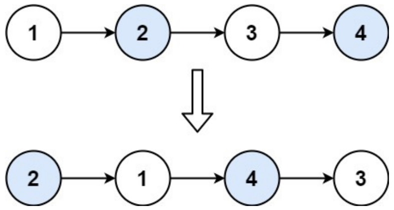
Problem

Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

Example 1

Input: head = [1,2,3,4]

Output: [2,1,4,3]



Example 2

Input: head = []

Output: []

Example 3:

Example 3

Input: head = [1]

Output: [1]

Solution – Swap Nodes in Pair

Medium

https://leetcode.com/problems/swap-nodes-in-pairs

```
ListNode* swapPairs(ListNode* head) {
    if (head == NULL || head->next == NULL) {
        return head;
    }
    ListNode *node = head;
    ListNode *prev = NULL;
    head = head->next;

    while (node && node->next) {
        ListNode *second = node->next;
        ListNode *next_pair = second->next;
        second->next = node;
        node->next = next_pair;
        if (prev) {
            prev->next = second;
        }
        prev = node;
        node = next_pair;
    }
    return head;
}
```

Solution (recursive) – Swap Nodes in Pair

Medium

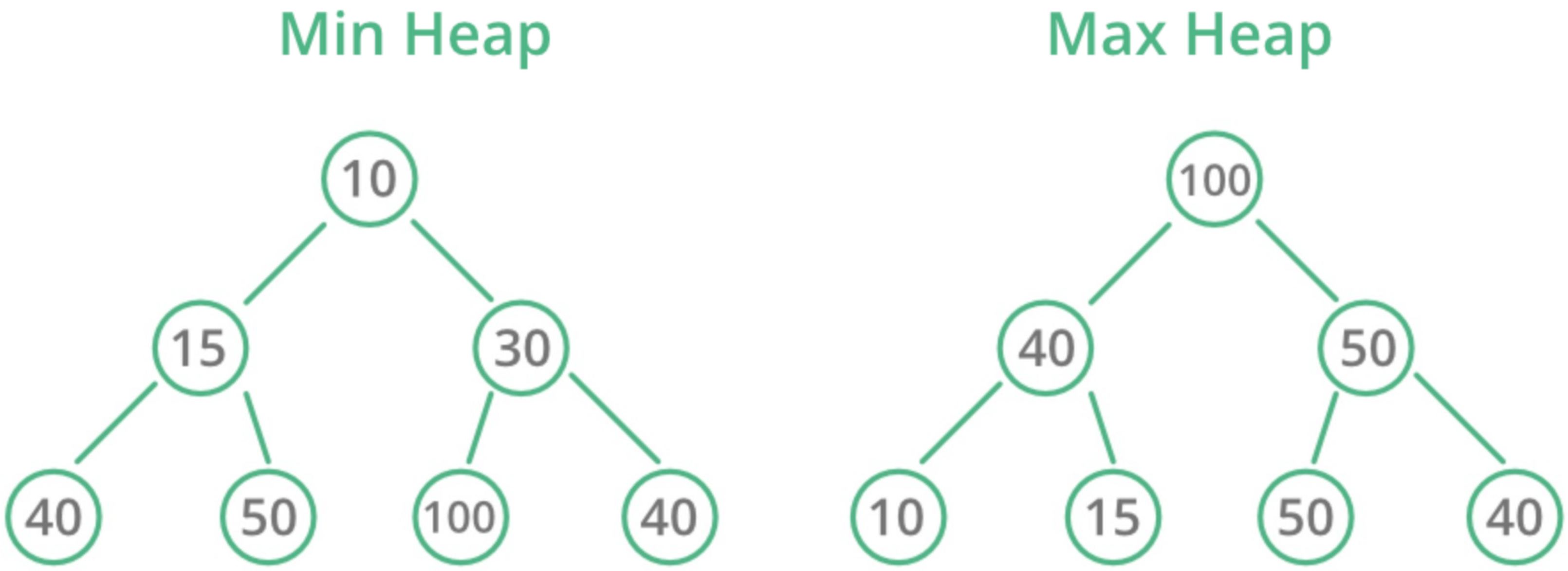
https://leetcode.com/problems/swap-nodes-in-pairs

```
ListNode* swapPairs(ListNode* head) {
    if(!head || !head->next)
        return head;
    ListNode* newHead = head->next;
    head->next = swapPairs(head->next->next);
    newHead->next = head;
    return newHead;
}
```

HEAP / PRIORITY QUEUE

Heap

- **Heap** is a complete binary tree that satisfy the heap property (max or min)
- **Min heap**: root node contains the minimum value
- **Max heap**: root node contains the maximum value



Heap in C++

Two main ways to implement:

1. Using **std::make_heap** from **<algorithm>**

```
std::make_heap(RandomIt first, RandomIt last)
std::push_heap(RandomIt first, RandomIt last)
std::pop_heap(RandomIt first, RandomIt last)
std::sort_heap(RandomIt first, RandomIt last)
```

2. Using **std::priority_queue** from **<queue>** **(recommended)**

```
std::priority_queue<T, Container, Compare>
```

Heap in C++ – std::priority_queue example

Min heap

`std::priority_queue<int, std::vector<int>, std::greater<int>>`

Max heap

`std::priority_queue<int>` or

`std::priority_queue<int, std::vector<int> std::less<int>>`

```
// Min heap
std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;

minHeap.push(3);
minHeap.push(6);
minHeap.push(4);
// remove top element (3)
minHeap.pop();
// root node (top) is now 4
std::cout << minHeap.top();
```

Problem – Kth Largest Element in an Array

Medium

https://leetcode.com/problems/kth-largest-element-in-an-array

Problem

Given an integer array **nums** and an integer **k**, return the **k**th largest element in the array.

Note that it is the **k**th largest element in the sorted order, not the **k**th distinct element.

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

Although this problem is classified as “medium”, in my opinion it should be classified as “easy”

Solution 1 – Kth Largest Element in an Array

Medium

https://leetcode.com/problems/kth-largest-element-in-an-array

```
// SOLUTION 1

int findKthLargest(vector<int>& nums, int k) {
    std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;
    for (const auto& num : nums) {
        if (minHeap.size() < k) {
            minHeap.push(num);
        } else if (num > minHeap.top()) {
            minHeap.pop();
            minHeap.push(num);
        }
    }
    return minHeap.top();
}
```

Solution 2 – Kth Largest Element in an Array

Medium

https://leetcode.com/problems/kth-largest-element-in-an-array

```
// SOLUTION 2 - Simpler approach

int findKthLargest(vector<int>& nums, int k) {
    // min heap: minimum values will be always at the top
    std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;
    for (const auto& num : nums) {
        // push each num to the heap
        minHeap.push(num);
        // we need the kth largest element only, so once after pushing more than k
        // elements, remove the smallest one (the top)
        if (minHeap.size() > k) {
            minHeap.pop();
        }
    }
    return minHeap.top();
}
```

DYNAMIC PROGRAMMING

Dynamic Programming

Dynamic Programming (DP) is an algorithm technique used to solve problems that can be broken down into **simpler, overlapping subproblems.**

Key Concepts of Dynamic Programming

- **Overlapping subproblems**: a problem has overlapping subproblems if it can be broken down into subproblems.
- **Memoization (Top-Down Approach)**: store the results in a cache (typically a dictionary or array) to avoid recalculation – recursion and caching approach.
- **Tabulation (Bottom-Up Approach)**: first solve all possible subproblems iteratively, and store them in a table.

Dynamic Programming – Example – Fibonacci Sequence

Naive Recursive Approach $O(2^n)$

```
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

Memoization (Top-Down DP) $O(n)$

```
std::unordered_map<int, int> memo;

int fib(int n) {
    if (n <= 1) {
        return n;
    }
    if (memo.find(n) != memo.end()) {
        return memo[n];
    }
    memo[n] = fib(n - 1) + fib(n - 2);
    return memo[n];
}
```

Tabulation (Bottom-up DP) $O(n)$

```
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    int dp[n + 1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
}
```

Problem – Climbing Stairs

Easy

LeetCode leetcode.com/problems/climbing-stairs

Problem Statement

You need to climb a staircase with n steps to get to the top. Each time you can choose to climb either **1 step** or **2 steps** at a time. Find out how many different ways you can climb to the top of the staircase.

Example 1

Input: n = 2

Output: 2

Explanation: There are two ways to get to the top

1. Climb 1 step at a time, twice
2. Climb 2 steps in one go

Example 2:

Input: n = 3

Output: 3

Explanation: There are three ways to get to the top:

1. Climb 1 step at a time, three times
2. Climb 1 step, then 2 steps
3. Climb 2 steps, then 1 ste.

Solution – Climbing Stairs

Blind 75

Easy

LeetCode leetcode.com/problems/climbing-stairs

```
std::unordered_map<int, int> memo;

int climbStairs(int n) {
    // Identify the sequence, when:
    // n = 0 (0 way), there is no way to get up
    // n = 1 (1 way): only one way : 1-step
    // n = 2 (2 ways): 1s + 1s | 2s
    // n = 3 (3 ways): 1s + 1s + 1s |  1s + 2s | 2s + 1s
    // n = 4 (5 ways): 1s + 1s + 1s + 1s | 1s + 1s + 2s | 1s + 2s + 1s | 2s + 1s + 1s | 2s + 2s |

    if (n <= 2) {
        return n;
    }

    if (memo.find(n) != memo.end()) {
        return memo[n];
    }

    memo[n] = climbStairs(n - 1) + climbStairs(n - 2);
    return memo[n];
}
```

EOF