Purwadhika
Digital Technology School

Full Stack AI Software Development

Algorithm and Data Structure

Job Connector Program

Outline

Algorithm
Fundamental algorithm techniques, complexity analysis, and TypeScript implementations.

Data Structure
Organizing, storing, and managing data in a computer so that it can be accessed and modified efficiently.

What is Algorithm?

A finite set of well-defined instructions to solve a problem or perform a computation.

It's a step-by-step recipe for solving a problem.

Analogy: A cookbook recipe.

- Problem: Bake a cake.
- Inputs: Flour, eggs, sugar (Data).
- Algorithm: Mix ingredients, preheat oven, bake for 30 min (Instructions).
- Output: A cake (Result).

Why Study Algorithms?

Problem Solving

Provides a toolbox of established patterns for common problems (searching, sorting, graphing, etc.).

Efficiency & Performance

It's not just about getting an answer, but getting it fast and with minimal resources (memory).

Scalability

How does your code perform as input size (*n*) grows?. Will it work for 10 users? 10 million users?

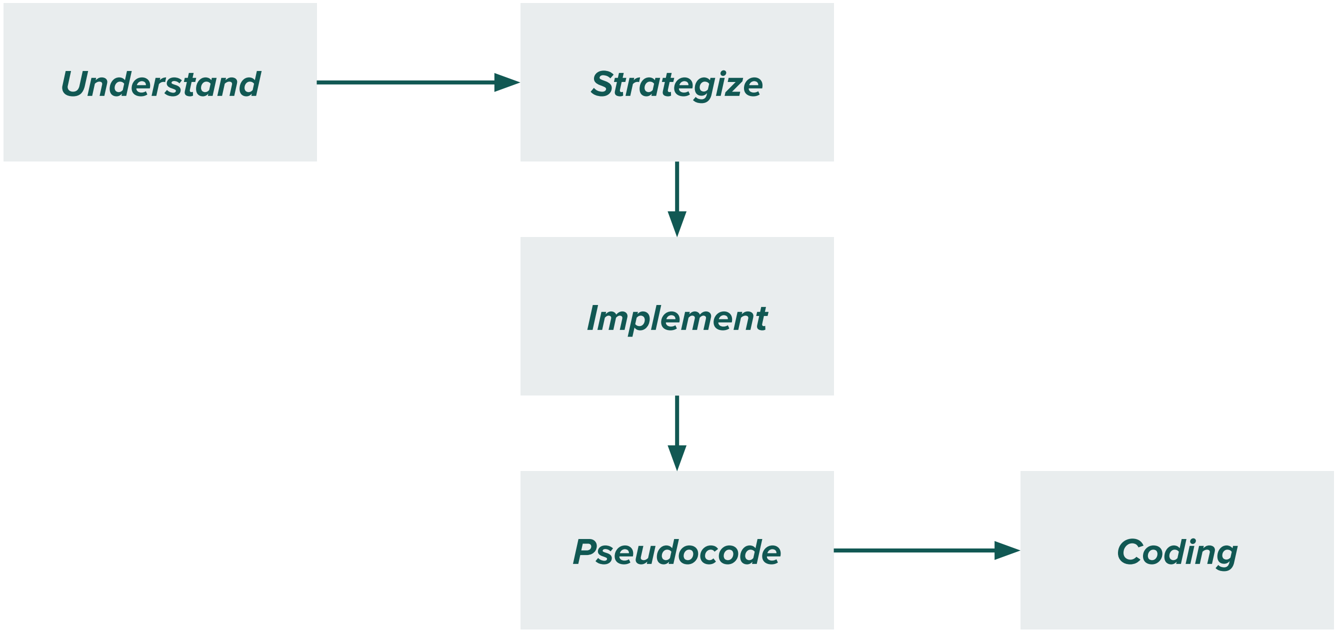
Technical Interviews

A universal way to measure a candidate's problem-solving and foundational knowledge.

Better Code

Writing algorithmic code forces you to think clearly, handle edge cases, and write clean, reusable logic.

How to Use Algorithms to Solve Problems?



Understand the Problem

Before designing an algorithm, you must clearly **understand what the problem is asking**.

Key points:

- Read the problem statement carefully.
- Identify the input (what data is given).
- Identify the output (what result you need).
- Understand any constraints (such as time limits, data ranges, or edge cases).
- Think about examples: What happens in simple or extreme cases?

Example:

- Problem: Find the largest number in an array of integers.
- Input: [5, 2, 9, 1, 7]
- Output: 9

Strategize (Plan Your Approach)

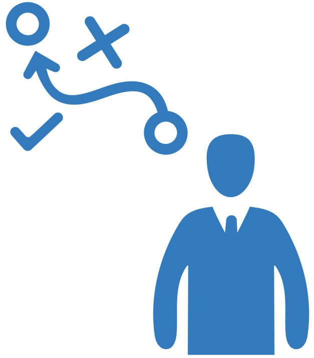
Now that you understand the problem, **decide on a strategy to solve it**.

Think about:

- What algorithmic techniques might apply? (e.g., iteration, recursion, sorting, searching)
- How can you break down the problem into smaller steps?
- What is the simplest and most efficient way to get the correct answer?

Example Strategy:

- Loop through all numbers in the array and keep track of the largest number found so far.

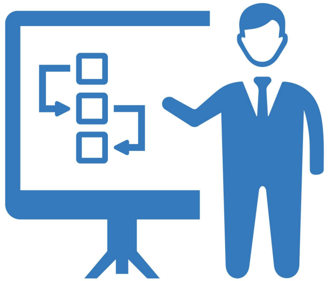


Implement the Logic (Conceptually)

Before writing pseudocode, visualize or write down your step-by-step logic in plain language. This helps you clarify your thought process.

Example:

- Start with the first number as the largest.
- Compare each number in the list with the current largest.
- If a number is greater, update the largest value.
- After checking all numbers, return the largest one.



Write Pseudocode

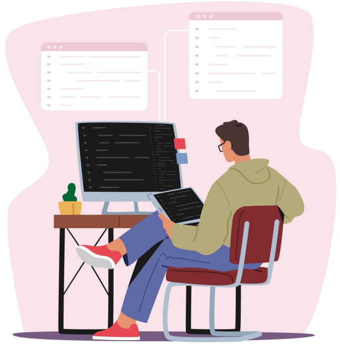
Pseudocode bridges the gap between the logical steps and actual code. It’s a way to express the algorithm using structured plain English, not tied to any programming language.

```
FUNCTION findLargest(numbers):
    SET largest = numbers[0]
    FOR each number in numbers:
        IF number > largest:
            largest = number
    RETURN largest
```

Translate to Code

Now you can confidently write your algorithm in TypeScript (or any language), since your logic is clear.

```typescript
function findLargest(numbers: number[]): number {
  let largest = numbers[0];
  for (let i = 1; i < numbers.length; i++) {
    if (numbers[i] > largest) {
      largest = numbers[i];
    }
  }
  return largest;
}

// Example usage
console.log(findLargest([5, 2, 9, 1, 7])); // Output: 9
```

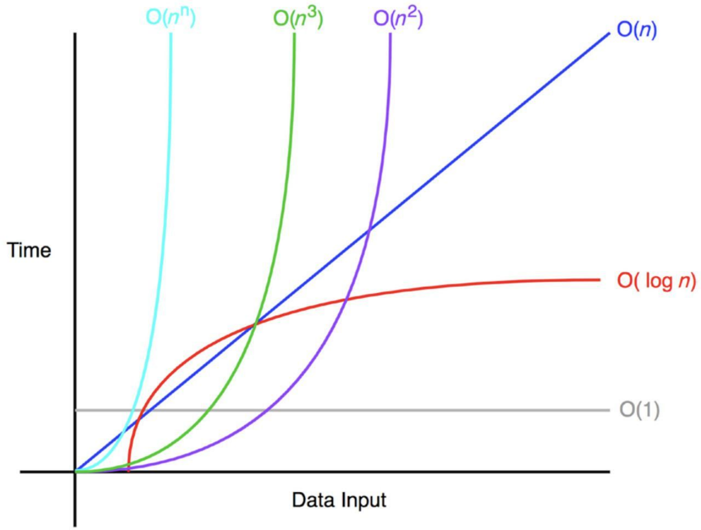


Analyzing Algorithms: Big O Notation

- Big O notation describes the scalability and efficiency of an algorithm. It tells us how the runtime (or space) requirements grow as the input size (*n*) increases. **It's not about seconds, it's about the rate of growth.**
- There are two parts to measuring efficiency:
  - **Time complexity** is a measure of how long the function takes to run in terms of its computational steps.
  - **Space complexity** has to do with the amount of memory used by the function.

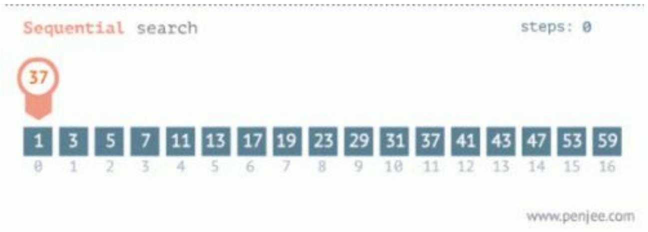
Common Complexities

- ***O(1)* - Constant Time:**
  - ***Excellent***. Same time regardless of input size. (e.g., accessing an array element by index: arr[5]).
- ***O(log n)* - Logarithmic Time:**
  - ***Very good***. Runtime grows very slowly. (e.g., Binary Search).
- ***O(n)* - Linear Time:**
  - ***Good***. Runtime scales directly with input size. (e.g., Linear Search).
- **$O(\,^{n}log_{n}\,)$ - Log-Linear Time:**
  - ***Great***. The gold standard for sorting. (e.g., Merge Sort, Quick Sort).
- **$O(n^2)$ - Quadratic Time:**
  - ***Poor.*** Okay for small inputs, but scales poorly. (e.g., Bubble Sort, nested loops).
- **$O(2^n)$ - Exponential Time:**
  - **Very bad.** Becomes unusable very quickly. (e.g., recursive Fibonacci without memoization).



Example: Linear Search

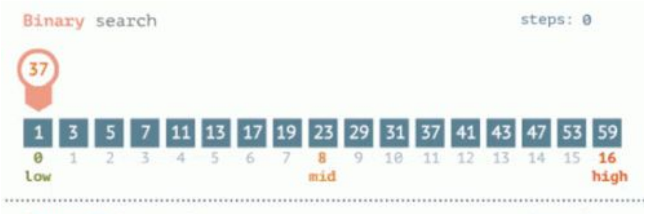
- Goal: Find an element in an array.
- Method: Check every element one by one, from start to finish.
- Analysis:
    - Best Case: *O(1)* (Element is the first item).
    - Worst Case: *O(n)* (Element is the last item, or not present).
    - Average Case: *O(n)*




```javascript
function search(arr, x) {
    let i;
    for (i = 0; i < arr.length; i++) {
        if (arr[i] == x) {
            return i;
        }
    }
    return -1;
}

console.log(search([2, 20, 10, 3], 20));
```

Example: Binary Search

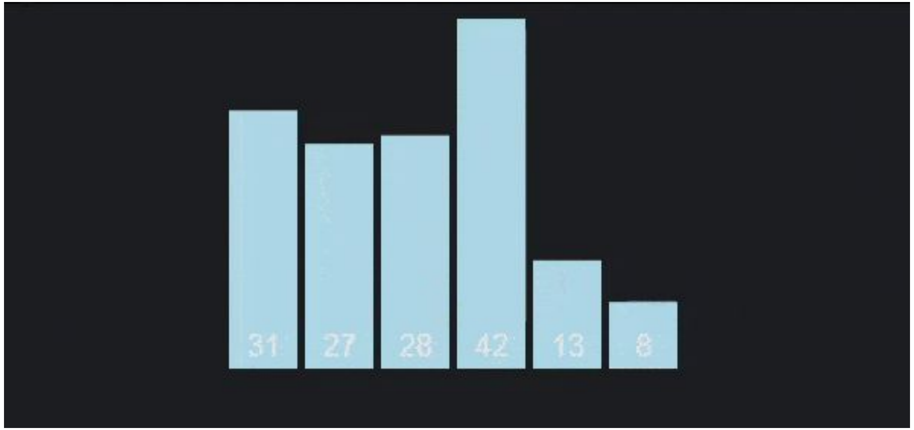
- Goal: Find an element in a sorted array.
- Method: A **"Divide and Conquer"** strategy.
  - Look at the middle element.
  - If it's the target, you're done.
  - If the target is smaller, repeat the search on the left half.
  - If the target is larger, repeat the search on the right half.
- Analysis:
  - Cuts the search area in half with each step.
  - Runtime: *O(log n)* - Extremely fast and scalable.




```javascript
function binarySearch(arr, l, r, x) {
    if (r >= l) {
        let mid = l + Math.floor((r - l) / 2);

        // If the element is found at the middle position
        if (arr[mid] == x) return mid;

        // If the element is smaller than mid, search in the left subarray
        if (arr[mid] > x) {
            return binarySearch(arr, l, mid - 1, x);
        }

        // Otherwise, search in the right subarray
        return binarySearch(arr, mid + 1, r, x);
    }
    return -1; // Element not found
}

let arr = [2, 3, 4, 10, 40];
let x = 10;

console.log(binarySearch(arr, 0, arr.length - 1, x)); // Output: 3
```

Example: Bubble Sort

- Goal: Sort an array in place.
- Method: A simple, but inefficient, "brute force" approach.
    - Repeatedly step through the list.
    - Compare each adjacent pair.
    - Swap them if they are in the wrong order.
    - Repeat until no swaps are needed.
- Analysis:
    - Runtime: *O(n^2)* - Two nested loops.
    - Very slow for large datasets. Good for educational purposes.

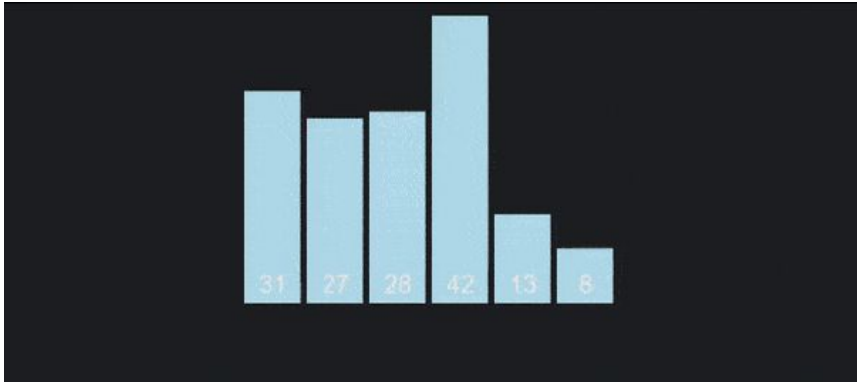


Example: Bubble Sort

```
function bubbleSort(arr) {
    for (let i = 0; i < arr.length; i++) {
        for (let j = 0; j < arr.length - i - 1; j++) {
            if (arr[j + 1] < arr[j]) {
                [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
            }
        }
    }
    return arr;
}

console.log(bubbleSort([5, 3, 8, 4, 6]));
```

Example: Selection Sort

- Goal: Sort an array in place by repeatedly "selecting" the smallest element.
- Method:
  - Divide the array into two parts: a sorted part (at the beginning) and an unsorted part (the rest).
  - Find the absolute minimum element in the unsorted part.
  - Swap that minimum element with the first element of the unsorted part.
  - Move the boundary of the sorted part one element to the right.
  - Repeat until the entire array is sorted.
- Analysis:
  - Runtime: *O(n^2)* (Best, Worst, and Average). It always performs a full scan to find the minimum, regardless of whether the array is already sorted.
  - Space: *O(1)* - (In-place).



Example: Selection Sort

```javascript
function selectionSort(arr) {
    let min;
    for (let i = 0; i < arr.length; i++) {
        min = i;

        for (let j = i + 1; j < arr.length; j++) {
            if (arr[j] < arr[min]) min = j;
        }

        if (min !== i) [arr[i], arr[min]] = [arr[min], arr[i]];
    }
    return arr;
}

console.log(selectionSort([29, 72, 98, 13, 87, 66, 52, 51, 36]));
```

Example: Insertion Sort

- Goal: Sort an array in place by building up a sorted sublist.
- Method:
  - Analogy: How you sort a hand of playing cards.
  - Start at the second element (index 1), assuming the first element (index 0) is a "sorted list" of size one.
  - Pick the current element (the "key").
  - Compare the "key" to the elements in the sorted list to its left.
  - Shift all elements in the sorted list that are greater than the key one position to the right (to make space).
  - "Insert" the key into its correct position.
- Analysis:
  - Best Case: *O(n)* (If the array is already sorted, it just does one check per element).
  - Worst Case: *O(n^2)* (If the array is reverse-sorted).
  - Average Case: *O(n^2)*
  - Space: *O(1)* - (In-place).
  - Note: Very efficient for small or mostly sorted datasets.

Example: Insertion Sort

```javascript
function insertionSort(arr) {
  const n = arr.length;

  for (let i = 1; i < n; i++) {
    let key = arr[i];
    let j = i - 1;

    while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j = j - 1;
    }

    arr[j + 1] = key;
  }
}

console.log(insertionSort([12, 11, 13, 5, 6]));
```

Exercise

- Create a function to convert Excel sheet column title to its corresponding column number.
- **Example :**

```
A -> 1
B -> 2
C -> 3
...
Z -> 26
AA -> 27
AB -> 28
...
```

- **Example :**
  - Input : AB
  - Output : 28

Exercise

- Given a non-empty array of integers nums, every element appears twice except for one. Find that single one.
- **Example 1:**
    - Input: nums = [2,2,1]
    - Output: 1
- **Example 2:**
    - Input: nums = [4,1,2,1,2]
    - Output: 4
- **Example 3:**
    - Input: nums = [1]
    - Output: 1

Exercise

- Given two strings **s** and **t**, return true *if* t *is an anagram of* s, *and* false *otherwise*.
- An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.
- **Example 1:**
  - Input: s = "anagram", t = "nagaram"
  - Output: true
- **Example 2:**
  - Input: s = "rat", t = "car"
  - Output: false

Exercise

- 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

What is Data Structure?

A data structure is a way to organize, manage, and store data efficiently for access and modification. Algorithms operate on Data Structures. ***You can't have one without the other.***

In TypeScript: Data structures benefit from strong typing, generics, and OOP concepts.

```
// Example: defining a type-safe array
const numbers: number[] = [1, 2, 3, 4];
```

Why Do Data Structures Matter?

Efficiency & Performance

- Does your app need to find data quickly? (e.g., finding a user by ID).
- Does it need to add/remove data quickly? (e.g., a "to-do" list).
- Choosing the wrong DS can make an operation take minutes instead of milliseconds.

Problem Modeling

- Data structures allow us to model real-world problems in code.
- Social Network: A Graph
- Browser "Back" Button: A Stack
- A Print Queue: A Queue
- A File System: A Tree
- A Dictionary: A Hash Map

Scalability

- A data structure that works for 100 items might fail completely at 10 million.
- We use Big O Notation to measure this scalability.

TypeScript for Data Structures

TypeScript makes building and using data structures safer, clearer, and more robust.

- **Generics (<T>)**
  - This is the #1 reason. We can create reusable structures that work for any type of data.
  - Instead of a **StringStack** and a **NumberStack**, we create one **Stack<T>**.
  - `const numStack = new Stack<number>();`
  - `const nameStack = new Stack<string>();`
- **Strong Typing & Type Safety**
  - We can explicitly define the shape of our data.
  - `class TreeNode<T> { ... public left: TreeNode<T> | null; }`
  - The compiler catches bugs for us, like trying to access a node that might be null.
- **Interfaces**
  - We can define the contract (the required operations) for a data structure separately from its implementation.
  - `interface IQueue<T> { enqueue(item: T): void; dequeue(): T | undefined; }`
  - This forces our code to be clean and consistent.
- **Classes**
  - Classes are a natural fit for data structures, bundling state (the data) and methods (the operations) together.

TypeScript Generics

To write reusable code, you must use **generics**.

- **A generic (<T>)** is a placeholder for a type.
- It allows a function or class to work with various types, decided at the time it's called.

Example: A generic identity function

```typescript
// This function takes a value of *some* type 'T'
// and returns a value of that same type 'T'.
function identity<T>(arg: T): T {
  return arg;
}

// TypeScript infers the type 'T' from the argument
let output1 = identity("hello"); // Type is 'string'
let output2 = identity(123);     // Type is 'number'

// We can also be explicit
let output3 = identity<boolean>(true); // Type is 'boolean'
```

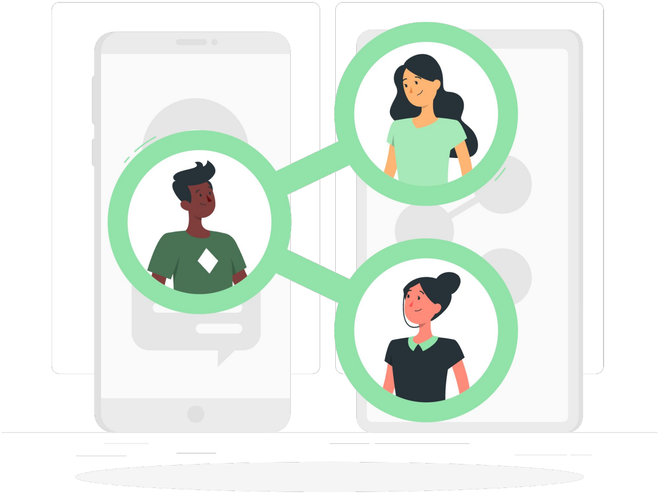
Array

- A contiguous block of memory where elements are stored one after another.
- Core Feature: Accessing elements by a numeric index.
- Big O Profile:
  - Access (by index): O(1)
  - Search (by value): O(n)
  - Insertion (at end): O(1) (Amortized)
  - Insertion (at beginning): O(n) (Must shift all other elements)
  - Deletion (at beginning): O(n) (Same reason)
- Use When: You need fast access by index and have a collection of fixed or rarely-changing size.

```typescript
const numbers: number[] = [10, 20, 30, 40];
const names: Array<string> = ["Alice", "Bob"];

// O(1) Access
console.log(numbers[2]); // 30
```

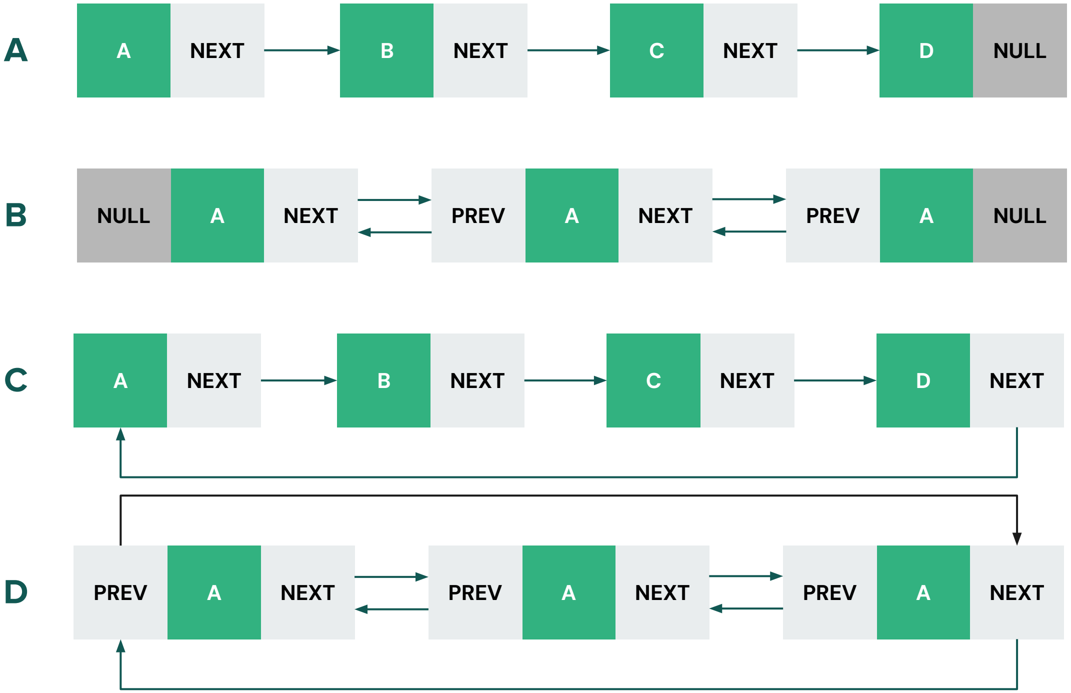
Linked List (The Chain)

- A sequence of Nodes. Each node contains a value and a pointer (reference) to the next node in the chain. The last node points to null.
- Why? Solves the array's ***O(n)*** insertion/deletion problem.
- Big O Profile:
  - Access (by index): ***O(n)*** (Must traverse from the head)
  - Search (by value): ***O(n)***
  - Insertion (at beginning): ***O(1)***
  - Deletion (at beginning): ***O(1)***
  - Insertion (at end): ***O(n)*** (Must traverse... unless you add a tail pointer!)
- Use When: You have lots of insertions/deletions, especially at the beginning or end.



Linked List (The Chain)

A. Single linked lists
B. Doubly linked lists
C. Circular linked lists
D. Circular doubly linked lists

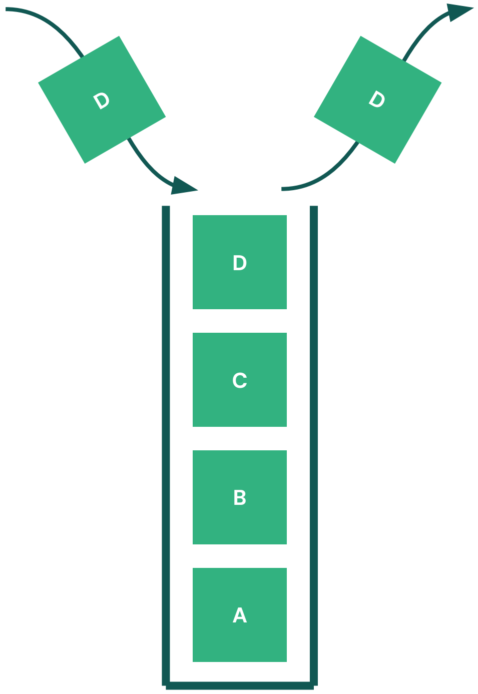


Linked List (The Chain)

```
class ListNode<T> {
  public next: ListNode<T> | null = null;

  constructor(public value: T) {}
}

class LinkedList<T> {
  public head: ListNode<T> | null = null;
  // ... methods to add, remove, find
}
```

Stack (LIFO)

- A "Last-In, First-Out" (LIFO) structure.
- Analogy: A stack of plates. You push a new plate onto the top, and you pop a plate off the top.
- Core Operations:
  - push(value): Add an item to the top.
  - pop(): Remove and return the top item.
  - peek(): Look at the top item without removing it.
- Use Cases: Function call stack, "Undo" functionality, Browser "Back" button.
- Big O Profile: All primary operations (push, pop, peek) are ***O(1)***.



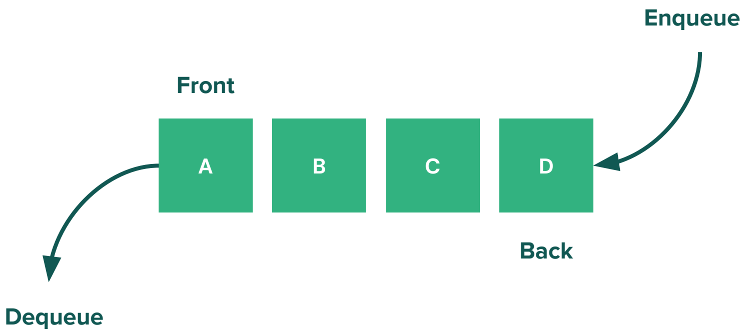
Stack (LIFO)

```typescript
class Stack<T> {
  private storage: T[] = [];

  // O(1)
  push(item: T): void {
    this.storage.push(item);
  }

  // O(1)
  pop(): T | undefined {
    return this.storage.pop();
  }

  // O(1)
  peek(): T | undefined {
    return this.storage[this.storage.length - 1];
  }

  isEmpty(): boolean {
    return this.storage.length === 0;
  }
}
```

Queue (FIFO)

- A "First-In, First-Out" (FIFO) structure.
- Analogy: A checkout line. The first person in line is the first person to be served.
- Core Operations:
  - enqueue(value): Add an item to the back of the line.
  - dequeue(): Remove and return the item from the front.
  - peek(): Look at the front item without removing it.
- Use Cases: Print queues, message processing, Breadth-First Search (BFS) algorithm.

Note: A naive array-based queue is inefficient! A proper ***O(1)*** queue is built using a Linked List (by adding to the tail and removing from the head) or a more complex object-based ring buffer.

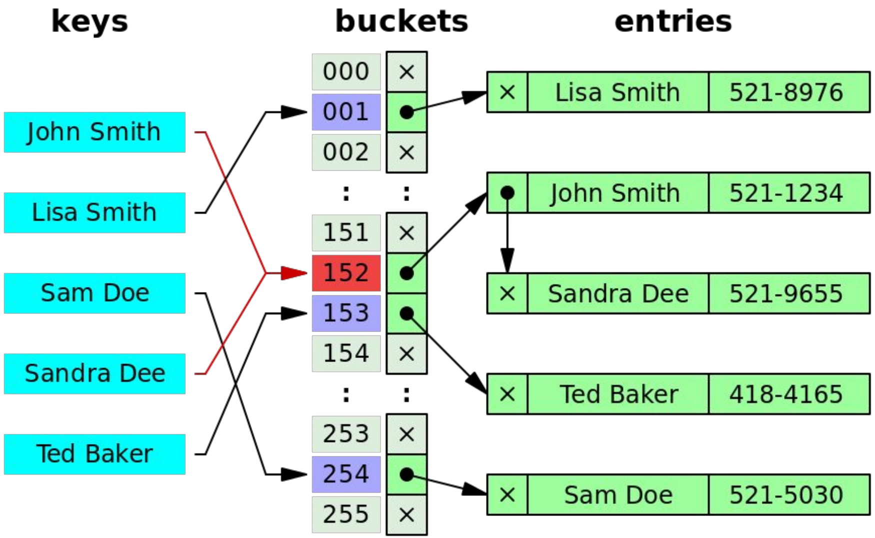


Queue (FIFO)

```typescript
class SimpleQueue<T> {
  private storage: T[] = [];

  // O(1)
  enqueue(item: T): void {
    this.storage.push(item);
  }

  // O(n) - BAD!
  dequeue(): T | undefined {
    // .shift() re-indexes the entire array!
    return this.storage.shift();
  }
}
```

Hash Map (or Hash Table)

- The most useful data structure. It stores data as Key-Value pairs.
- Analogy: A dictionary. The Key is the word (e.g., "algorithm"), and the Value is the definition.
- How it Works (The Magic):
  - A Key (e.g., "userId-123") is passed to a Hash Function.
  - The function computes a unique Hash (e.g., 5).
  - This hash is used as an index in an array, where the Value is stored.
- Collisions: If two keys hash to the same index, we store them in a Linked List at that spot (called "chaining").
- Big O Profile:
  - Insertion (set): ***O(1)*** (Average), ***O(n)*** (Worst Case)
  - Deletion (delete): ***O(1)*** (Average), ***O(n)*** (Worst Case)
  - Search (get): ***O(1)*** (Average), ***O(n)*** (Worst Case)
- Use When: You need fast lookup, insertion, and deletion of data by a unique key.



Hash Map (or Hash Table)

```
// Map<KeyType, ValueType>
const userRoles = new Map<number, string>();

// set() - O(1) on average
userRoles.set(101, "Admin");
userRoles.set(102, "User");
userRoles.set(103, "Guest");

// get() - O(1) on average
const adminRole = userRoles.get(101); // "Admin"

// has() - O(1) on average
console.log(userRoles.has(102)); // true

// delete() - O(1) on average
userRoles.delete(103);

// Also useful: .size, .clear(), .keys(), .values()
```

Exercise

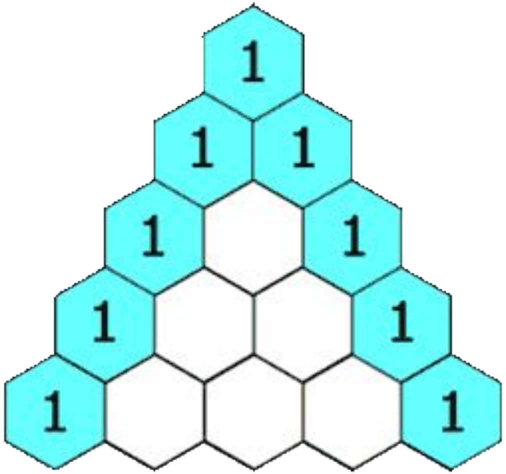
- Given an array nums of size n, return *the majority element*. The majority element is the element that appears more than $\lfloor n / 2 \rfloor$ times. You may assume that the majority element always exists in the array.
- **Example 1:**
  - **Input:** nums = [3,2,3]
  - **Output:** 3
- **Example 2:**
  - **Input:** nums = [2,2,1,1,1,2,2]
  - **Output:** 2

Exercise

- Create a function to convert roman numeral to integer.
- **Example 1:**
  - Input: s = "III”
  - Output: 3
  - Explanation: III = 3.
- **Example 2:**
  - Input: s = "LVIII"
  - Output: 58
  - Explanation: L = 50, V= 5, III = 3.
- **Example 3:**
  - Input: s = "MCMXCIV"
  - Output: 1994
  - Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

Exercise

- Given an integer numRows, return the first numRows of **Pascal's triangle**.
- In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown ➔
- **Example 1:**
  - **Input:** numRows = 5
  - **Output:** [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]
- **Example 2:**
  - **Input:** numRows = 1
  - **Output:** [[1]]



Exercise

- 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.

Thank you