Data Structures & Algorithms

Table of Contents

Arrays

Arrays

1. About Arrays

An array is a linear data structure that stores a collection of elements of the same data type in contiguous memory locations. Each element is identified by a unique, non-negative integer called an index, which typically starts at 0. Because of this indexed structure, arrays offer direct access to any of their elements. ■

2. Syntax (Java)

In Java, arrays are objects. They need to be declared and then instantiated.

Initialization: Declares, instantiates, and initializes the array with values in a single statement. java dataType[] arrayName = {value1, value2, ...}; Example: ```java // Declares and instantiates an integer array of size 5 int[] numbers = new int[5];

// Initializes a string array with three elements String[] fruits = {"Apple", "Banana", "Cherry"}; ```

3. Operations with Syntax

4. Time Complexity of Operations

The efficiency of array operations is measured using Big O notation.

5. Common Patterns

Arrays are fundamental to many algorithmic patterns.

6. Common Questions:

7. Advantages and Disadvantages

■ Advantages

■ Disadvantages

Hash Maps

A HashMap is a data structure that stores data as key-value pairs. It uses a technique called hashing to compute an index where an element is stored, allowing for very fast retrieval, insertion, and deletion of data. Keys in a HashMap must be unique. It's a part of the Java Collections Framework and does not maintain the order of insertion.

Syntax (Java)

To use a HashMap, you first need to import it from the java.util package.

import java.util.HashMap; // Declaration and Initialization HashMap mapName = new HashMap<>(); // Example with String keys and Integer values HashMap studentScores = new HashMap<>();

Operations with Syntax

Here are the most common operations performed on a HashMap:

Add/Update an element: Associates the specified value with the specified key. If the key already exists, the old value is replaced.

java studentScores.put("Alice", 95); // Adds a new entry studentScores.put("Alice", 98); // Updates the value for "Alice"

Access an element: Retrieves the value to which the specified key is mapped. Returns null if the key is not found.

java Integer score = studentScores.get("Alice"); // returns 98

Remove an element: Removes the mapping for a key.

java studentScores.remove("Bob");

Check for a key: Returns true if the map contains a mapping for the specified key.

java boolean hasAlice = studentScores.containsKey("Alice"); // returns true

Get the size: Returns the number of key-value mappings.

java int size = studentScores.size();

Iterate over the map:

```java // Iterate over keys for (String name : studentScores.keySet()) { System.out.println("Key: " + name); }

// Iterate over values for (Integer value : studentScores.values()) { System.out.println("Value: " + value); }

// Iterate over key-value pairs (entries) for (Map.Entry entry : studentScores.entrySet()) { System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue()); } ```

## Time Complexity of the Operations

The performance of a HashMap is heavily dependent on its hash function, which should distribute elements uniformly across buckets.

Note: The worst case occurs when all keys hash to the same bucket (a hash collision), and the elements in that bucket are stored in a list.

■ Note: Since Java 8, the worst-case time complexity has been improved from O(n) to O(\log n) for buckets that grow too large by replacing the linked list with a balanced binary search tree.

## Patterns

HashMaps are incredibly versatile and are a go-to solution for many common coding problems. ■■

## Common Questions:

## Advantages and Disadvantages

### ■ Advantages
```

■ Disadvantages

Hash Sets

HashSet: DSA Short Notes

A HashSet is a collection that stores unique elements. It's an implementation of the Set interface, backed by a hash table (specifically, a HashMap instance). It makes no guarantees concerning the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. It permits the null element.

Syntax (Java)

To use HashSet, you need to import java.util.HashSet.

import java.util.HashSet; import java.util.Set; // General Declaration and Initialization Set setName = new HashSet<>(); // Example with Strings Set fruits = new HashSet<>();

Common Operations

Here are the fundamental operations for a HashSet.

Time Complexity

The performance of a HashSet is heavily dependent on the hash function of its elements. Assuming a good hash function that distributes elements uniformly, the complexity is:

Common Patterns

HashSet is ideal for problems where uniqueness is key and order is not. ■

Common Questions

Advantages and Disadvantages

Advantages ■

Disadvantages ■

Two Pointers

The Two Pointer technique is an algorithmic pattern, not a data structure. It uses two pointers to iterate through a data structure, like an array, until they meet or satisfy a condition. It's primarily used to optimize problems involving sorted arrays or linked lists, often reducing the time complexity from quadratic to linear. ■

1. About the Two Pointer Pattern

The Two Pointer pattern is an efficient technique for problems that involve finding pairs, triplets, or subarrays in a sorted array or linked list. Instead of using nested loops which result in O(N^2) time complexity, this approach uses two pointers that traverse the data structure in a single pass (O(N)). The pointers can move in opposite directions (from start and end) or in the same direction (fast and slow pointers).

2. General Syntax (Java)

This pattern doesn't have a fixed syntax but is typically implemented inside a function using a while loop. Here's a common template for pointers moving towards each other in an array.

```
public void twoPointerTemplate(int[] arr) { // Initialize pointers at opposite ends int left = 0; int right =
arr.length - 1; // Loop until pointers meet or cross while (left < right) { // --- Condition check and logic
--- // Example: Check if the sum of elements at pointers meets a target int sum = arr[left] + arr[right];
if (sum == target) { // Found a pair, handle it // Potentially move both pointers left++; right--; } else if
(sum < target) { // Sum is too small, need a larger value // Move the left pointer to the right left++; }
else { // sum > target // Sum is too large, need a smaller value // Move the right pointer to the left
right--; } } }
```

3. Core Operations

The "operations" are the logical movements of the pointers within the loop based on some condition.

4. Time Complexity

5. Patterns and Sub-patterns

This technique is applicable to a variety of problems:

6. Common Questions

Valid Palindrome This is a classic use case. You place one pointer at the start of the string and another at the end. You move them inwards, comparing characters until they meet in the middle.

Two Sum II - Input Array Is Sorted Since the array is sorted, you can use two pointers. One starts at the beginning (left) and one at the end (right). If their sum is too small, you move left forward. If it's too big, you move right backward.

3Sum The most common solution involves sorting the array first. Then, you iterate through the array with a primary index i and use two pointers (left and right) on the rest of the array to find two numbers that sum up to -nums[i].

Container With Most Water You start with pointers at the two ends of the height array. You calculate the area and then move the pointer that corresponds to the shorter line inward, as moving the taller line's pointer can't possibly create a larger area.

Trapping Rain Water This is a more advanced but very clever application of the two-pointer technique. Pointers are placed at both ends. At each step, you advance the pointer that has the smaller maximum height seen so far, calculating the trapped water based on that maximum.

7. Advantages and Disadvantages

Advantages ■

Disadvantages ■

Sliding Window

Sliding Window

1. About the Algorithm ■■■

The Sliding Window is an algorithmic technique used for solving problems that involve finding a subarray or substring in an array or string that satisfies a certain condition. The core idea is to maintain a "window" (a contiguous sub-part of the data) and "slide" it over the data structure, one element at a time. This avoids re-computation by extending the window at one end and shrinking it from the other, making it highly efficient.

2. Syntax (Java)

Here is a general template for a variable-sized sliding window in Java.

```
// Generic template for a variable-sized sliding window public int slidingWindowTemplate(int[] arr, int target) { int windowStart = 0; int currentSum = 0; // Or other metric like character count int minLength = Integer.MAX_VALUE; for (int windowEnd = 0; windowEnd < arr.length; windowEnd++) { // Add the next element to the window currentSum += arr[windowEnd]; // Shrink the window from the left as long as the condition is met while (currentSum >= target) { // Update the result (e.g., find minimum length) minLength = Math.min(minLength, windowEnd - windowStart + 1); // Remove the element from the left of the window currentSum -= arr[windowStart]; windowStart++; // Slide the window forward } } return minLength == Integer.MAX_VALUE ? 0 : minLength; }
```

3. Operations with Syntax

The primary operations involve manipulating the window's boundaries.

Expand Window: The window is expanded by moving its right boundary (windowEnd). This is typically done inside a for loop.

```
java // windowEnd is incremented by the for loop for (int windowEnd = 0; windowEnd < arr.length; windowEnd++) { // ... }
```

Process Window: Calculations are performed on the current window to check if it meets the problem's criteria.

```
java // Example: Update a running sum currentSum += arr[windowEnd];
```

Shrink Window: The window is contracted by moving its left boundary (windowStart) forward. This is usually done in a while loop when a certain condition is met.

```
java // When a condition is met, shrink the window while (condition) { // Remove the leftmost element's contribution currentSum -= arr[windowStart]; // Move the left pointer to the right windowStart++; }
```

4. Time Complexity of the Operations

The key benefit of the sliding window pattern is its efficiency.

Time Complexity: O(N) Each element in the array is visited at most twice: once by the windowEnd pointer (when expanding) and once by the windowStart pointer (when shrinking). This results in a linear time complexity.

Space Complexity: O(1) or O(K) The space complexity is typically constant, O(1), if no auxiliary data structures are used. If a hash map or frequency array is used to store the contents of the window, the space complexity becomes O(K), where K is the number of distinct elements in the window.

5. Patterns

The sliding window pattern is ideal for problems involving contiguous subarrays or substrings. Common variations include:

6. Common Questions

Best Time to Buy and Sell Stock This is a simple application where the "window" is the time between a potential buy day and a sell day. The left pointer tracks the lowest price found so far, and the right pointer explores future days to find the maximum profit.

Longest Substring Without Repeating Characters A dynamic-sized window is used. The right pointer expands the window, and a hash set tracks characters inside it. If a duplicate character is found, the left pointer shrinks the window until the substring is valid again.

Longest Repeating Character Replacement The window expands as long as the number of characters that need to be replaced (k) is sufficient. The condition to check is window length - count of most frequent character <= k. If the condition is violated, the window shrinks from the left.

Permutation in String This uses a fixed-size sliding window (the size of the string s1). The window slides across s2, and for each position, you check if the character frequencies within the window match the frequencies in s1.

Minimum Window Substring This is a classic and more complex sliding window problem. The window expands until it contains all the required characters from string t. Then, it shrinks from the left as much as possible while still being a valid window, aiming to find the smallest one.

Sliding Window Maximum As the name suggests, this is a sliding window problem. The most efficient solution uses a deque (a double-ended queue) to maintain the indices of potential maximums within the fixed-size window, allowing you to find the max of any window in O(1) time.

7. Advantages and Disadvantages

Linked Lists

LinkedList

A LinkedList is a linear data structure where elements are stored in nodes, and each node points to the next node in the sequence. Unlike arrays, nodes are not stored in contiguous memory locations.

1. About the Data Structure ■■

A LinkedList is a collection of nodes ordered in a linear sequence. Each node contains two parts:

There are three main types:

2. Syntax (Java)

In Java, you can use the LinkedList class from the java.util package. It is implemented as a Doubly LinkedList.

import java.util.LinkedList; // Create a new LinkedList of Strings LinkedList linkedList = new LinkedList<>();

3. Operations with Syntax

Adding Elements

add(element): Appends to the end java linkedList.add("Apple");

add(index, element): Inserts at a specific position java linkedList.add(1, "Banana");

addFirst(element): Adds to the beginning java linkedList.addFirst("Mango");

addLast(element): Adds to the end java linkedList.addLast("Orange");

Accessing Elements

get(index): Retrieves an element by index java String fruit = linkedList.get(0);

getFirst(): Retrieves the first element java String firstFruit = linkedList.getFirst();

getLast(): Retrieves the last element java String lastFruit = linkedList.getLast();

Removing Elements

remove(index): Removes by index java linkedList.remove(1);

remove(object): Removes the first occurrence of an element java linkedList.remove("Apple");

removeFirst(): Removes the first element java linkedList.removeFirst();

removeLast(): Removes the last element java linkedList.removeLast();

Other Operations

size(): Returns the number of elements java int size = linkedList.size();

contains(object): Checks if the list contains an element java boolean hasApple = linkedList.contains("Apple");

4. Time Complexity of Operations

The performance depends on the operation. Here, 'n' is the number of elements in the list.

5. Common Patterns

LinkedLists are fundamental for solving specific types of problems:

6. Common Questions

Direct LinkedList Manipulations

These problems primarily involve iterating through lists and re-wiring next pointers.

LinkedLists Combined with Other Structures

These problems require a linked list as a core component but combine it with another data structure for an optimal solution.

Conceptual LinkedList Application

This problem is unique because it uses the concept of a linked list to solve a problem on a different data structure.

7. Advantages and Disadvantages

Advantages ■

Disadvantages ■

Stacks

Stack

A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle. Think of it like a stack of plates; you can only add a new plate to the top or remove the topmost plate.

1. About the Stack Data Structure ■

A stack is an abstract data type that serves as a collection of elements, with two primary operations:

This behavior is known as LIFO (Last-In, First-Out). The element placed last will be the first one to be removed.

2. Syntax (Java)

In Java, you can use the Stack class from the java.util package.

import java.util.Stack; // To create a stack of Integers Stack numberStack = new Stack<>(); // To create a stack of Strings Stack stringStack = new Stack<>();

3. Operations with Syntax

Here are the fundamental operations for a stack in Java.

4. Time Complexity of Operations ■■

All standard stack operations are highly efficient.

The primary operations only involve manipulating the top element, hence the constant time complexity.

5. Common Patterns & Problems ■

Stacks are useful for solving problems that involve reversing order or managing nested structures.

6. Common Questions

Valid Parentheses This is the most fundamental stack problem. You push opening brackets (, {, [onto the stack. When you encounter a closing bracket, you check if the top of the stack is the matching opening bracket and pop it.

Min Stack This problem requires you to implement a stack. The challenge is solved by using an auxiliary stack (or storing pairs in one stack) to keep track of the current minimum value at every level of the main stack.

Evaluate Reverse Polish Notation When you encounter a number, you push it onto the stack. When you see an operator (+, -, *, /), you pop two numbers, perform the operation, and push the result back onto the stack.

Generate Parentheses This is typically solved with recursion/backtracking. The function call stack itself acts as the "stack" that manages the state of the parentheses string being built.

Daily Temperatures A monotonic decreasing stack is used here. You store indices of days in the stack. When you find a day with a warmer temperature, you can pop all the previous days from the stack that are cooler than the current day and calculate the waiting period.

Car Fleet After calculating the arrival times for each car, you can use a stack. By processing cars from closest to the destination to farthest, you push their arrival times onto a stack. A car forms a new fleet only if its arrival time is later than the fleet in front of it (the time at the top of the stack).

Largest Rectangle In Histogram This is a famous monotonic increasing stack problem. The stack stores the indices of histogram bars. By maintaining an increasing height order in the stack, you can calculate the maximum area for each bar when it's popped.

7. Advantages and Disadvantages

Priority Queue / Heaps

Heap/Priority Queue

A heap is a specialized tree-based data structure that satisfies the heap property. A Priority Queue is an abstract data type that operates like a regular queue but assigns a priority to each element. Heaps are a common and efficient way to implement Priority Queues.

By default, Java's PriorityQueue is a min-heap.

Syntax (Java)

You can use the PriorityQueue class from the java.util package.

```
import java.util.PriorityQueue; import java.util.Collections; // Min-Heap (default behavior)
PriorityQueue minHeap = new PriorityQueue<>(); // Max-Heap (using a custom comparator or
Collections.reverseOrder()) PriorityQueue maxHeap = new
PriorityQueue<>(Collections.reverseOrder());
```

Operations with Syntax

The primary operations involve adding elements, removing the element with the highest priority (the root), and inspecting the root.

```
// Example Usage PriorityQueue minHeap = new PriorityQueue<>(); minHeap.add(10);
minHeap.add(5); minHeap.add(20); System.out.println(minHeap.peek()); // Outputs 5
System.out.println(minHeap.poll()); // Outputs 5 System.out.println(minHeap.peek()); // Outputs 10
```

Time Complexity of the Operations

Let N be the number of elements in the heap.

Building a heap from an array of N elements can be done in O(N) time (this process is called heapify).

Patterns

Heaps are excellent for problems that require efficiently finding the smallest or largest element among a changing set of data. ■■

Common Question

Kth Largest Element in a Stream

This is a classic use case for a min-heap of size k. The heap stores the k largest elements seen so far, and its root always represents the kth largest element.

Last Stone Weight

A max-heap is ideal here. You can add all stones to it and then efficiently extract the two heaviest stones in each step until one or zero remain.

K Closest Points to Origin

This problem is solved by maintaining a max-heap of size k. The heap stores the k points with the smallest distances found so far. You iterate through the points, and if a point is closer than the "farthest" point in your heap, you replace it.

Kth Largest Element in an Array

Similar to the stream version, this can be solved by pushing all elements into a min-heap of size k. After iterating through the array, the root of the heap is your answer.

Task Scheduler

A max-heap is used to implement a greedy strategy. The heap stores the frequencies of available tasks, allowing the CPU to always pick the most frequent task that is not on cooldown.

Design Twitter

To generate a user's news feed, you need to merge the sorted tweet lists from everyone they follow. A max-heap is the perfect tool to solve this "merge k sorted lists" sub-problem and efficiently find the 10 most recent tweets.

Find Median from Data Stream

This is a famous problem solved with a clever two-heap structure. A max-heap stores the smaller half of the numbers, and a min-heap stores the larger half. By keeping the heaps balanced, the median can be calculated in O(1) from their top elements.

Advantages and Disadvantages

Advantages ■

Disadvantages ■

Trees

Tree

A tree is a non-linear, hierarchical data structure consisting of a root node and potentially many levels of additional nodes that form a parent-child relationship. ■

1. About the Data Structure

A tree is a collection of entities called nodes linked together to simulate a hierarchy. It's a non-linear structure because, unlike arrays or linked lists, data is not stored sequentially. The top-most node is called the root. Each node can have zero or more child nodes. A node with no children is called a leaf. Trees are fundamental for representing data with a natural hierarchical relationship, like file systems or family trees. A common type is the Binary Tree, where each node has at most two children.

2. Syntax (Java)

In Java, a tree is typically represented by creating a Node class that holds the data and references (pointers) to its children. Here is the syntax for a basic Binary Tree Node.

```
class TreeNode { int data; // The data value of the node TreeNode left; // Reference to the left child
TreeNode right; // Reference to the right child // Constructor to create a new node public
TreeNode(int data) { this.data = data; this.left = null; this.right = null; } }
```

3. Operations with Syntax

Operations are usually performed starting from the root of the tree. The most common operations are traversals, searching, insertion, and deletion, especially in a Binary Search Tree (BST), a special type of binary tree where the left child's value is less than the parent's, and the right child's value is greater.

4. Time Complexity of Operations

The efficiency of tree operations heavily depends on the height (h) of the tree. For a balanced tree, h \approx \log_2 n. For a skewed (unbalanced) tree, h \approx n.

(Here, 'n' is the number of nodes in the tree.)

5. Patterns

Certain algorithmic patterns are frequently used to solve tree-based problems:

6. Common Question

General Tree Problems (solved with DFS/BFS)

These problems concern generic binary trees and are typically solved using a Depth-First Search (DFS) or Breadth-First Search (BFS) traversal. DFS is often more concise for these.

7. Advantages and Disadvantages

Advantages ■

Disadvantages ■

Binary Search Trees

Binary Search Tree (BST)

A Binary Search Tree is a special type of binary tree data structure where nodes are organized in a specific order. This structure makes searching, insertion, and deletion operations very efficient on average. ■

The key property of a BST is that for any given node:

Syntax (Java)

A BST is typically implemented using a Node class and a main tree class that holds a reference to the root node.

```
// Node class representing each element in the tree class Node { int key; Node left; Node right;
public Node(int item) { key = item; left = right = null; } } // Main BST class class BinarySearchTree { //
Root of the BST Node root; public BinarySearchTree() { root = null; } // Methods for operations like
insert, search, delete go here... }
```

Operations with Syntax

The core operations rely on the BST property, usually implemented recursively.

Time Complexity

The efficiency of a BST depends heavily on its height (h).

Common Question

These problems specifically leverage the properties of a Binary Search Tree (e.g., left child < parent < right child) for an efficient solution.

Common Patterns

BSTs are fundamental to solving many problems efficiently.

Advantages and Disadvantages

Tree Map

TreeMap

A TreeMap in Java is a sorted map implementation that stores key-value pairs. It's based on a Red-Black Tree, which is a self-balancing binary search tree. This structure ensures that the keys are always kept in a sorted order, either natural ordering or a custom order defined by a Comparator.

1. About the Data Structure

A TreeMap is a part of the Java Collections Framework and implements the SortedMap interface. Unlike a HashMap, which is unordered, a TreeMap maintains its entries in ascending key order. It does not allow null keys (though null values are permitted). Because it's based on a tree structure, it provides efficient retrieval of elements in a sorted manner.

2. Syntax (Java)

You can declare and initialize a TreeMap in a couple of ways:

Using natural ordering of keys:

\`\`\`java import java.util.TreeMap; import java.util.Map;

// The keys must implement the Comparable interface (e.g., Integer, String). Map treeMap = new TreeMap<>(); \`\`\`

Using a custom Comparator:

\`\`\`java import java.util.TreeMap; import java.util.Map; import java.util.Comparator;

// Provide a Comparator to define a custom sorting logic for the keys. Map treeMapWithComparator = new TreeMap<>(Comparator.reverseOrder()); \`\`\`

3. Operations with Syntax

Here are the most common operations performed on a TreeMap:

4. Time Complexity

The performance of TreeMap operations is directly related to the height of the underlying Red-Black Tree.

Add, Remove, Get, Contains Key (put, remove, get, containsKey): O(\log n)

Get First/Last Key (firstKey, lastKey): O(\log n)

Contains Value (containsValue): O(n)

Space Complexity: O(n)

Here, 'n' is the number of key-value pairs in the TreeMap.

5. Patterns

TreeMap is particularly useful in scenarios where you need to maintain sorted keys or perform operations based on the order. ■■

6. Advantages and Disadvantages

Advantages ■

Disadvantages ■

Tree Set

TreeSet

A TreeSet is a class in the Java Collections Framework that implements the Set and NavigableSet interfaces. It stores unique elements in a sorted order (either natural ordering or by a custom Comparator). It is backed by a TreeMap, which is a Red-Black Tree implementation. ■

1. About the TreeSet

A TreeSet is a sorted collection that does not allow duplicate elements. It uses a self-balancing binary search tree (specifically, a Red-Black Tree in Java) to store elements. This structure ensures that the elements are always in a sorted order, which allows for efficient retrieval operations based on their order.

2. Syntax (Java)

You can create a TreeSet in two primary ways:

Using natural ordering: The elements must implement the Comparable interface.

```java import java.util.TreeSet; import java.util.Set;

// Creates an empty TreeSet sorted according to the natural ordering of its elements. Set numbers = new TreeSet<>(); ```

Using a custom Comparator: Provide a Comparator at creation time to define a custom sorting logic.

```java import java.util.Comparator; import java.util.TreeSet; import java.util.Set;

// Creates a TreeSet sorted in descending order. Set descendingNumbers = new TreeSet<>(Comparator.reverseOrder()); ```

## 3. Operations with Syntax

Here are the fundamental operations for a TreeSet:

## 4. Time Complexity

The performance of TreeSet is logarithmic due to its underlying tree structure.

Here, n is the number of elements in the TreeSet.

## 5. Patterns

TreeSet is particularly useful in scenarios that require both uniqueness and order.

## 6. Advantages and Disadvantages

## Advantages ■

## Disadvantages ■

# Tries

## 1. About Tries

A Trie (pronounced "try"), also known as a prefix tree, is a tree-based data structure used for storing and retrieving a dynamic set of strings. In a Trie, nodes do not store keys themselves. Instead, a node's position in the tree defines the key it's associated with. Each path from the root to a node represents a common prefix, and nodes have a special flag to indicate the end of a complete word. ■

## 2. Syntax (Java)

A Trie is typically implemented with two classes: a TrieNode to represent each character and a Trie class to manage the overall structure.

// Represents a single node in the Trie class TrieNode { TrieNode[] children; boolean isEndOfWord; public TrieNode() { // Assuming only lowercase 'a'-'z' characters children = new TrieNode[26]; isEndOfWord = false; } } // The main Trie class class Trie { private final TrieNode root; public Trie() { root = new TrieNode(); } // Operations will be defined here... }

## 3. Operations with Syntax

The primary operations are inserting a word, searching for a complete word, and checking if any word starts with a given prefix.

Insert: Adds a word to the Trie.

java public void insert(String word) { TrieNode current = root; for (char c : word.toCharArray()) { int index = c - 'a'; if (current.children[index] == null) { current.children[index] = new TrieNode(); } current = current.children[index]; } current.isEndOfWord = true; }

Search: Checks if a complete word exists in the Trie.

java public boolean search(String word) { TrieNode current = root; for (char c : word.toCharArray()) { int index = c - 'a'; if (current.children[index] == null) { return false; } current = current.children[index]; } return current.isEndOfWord; }

Starts With: Checks if there is any word in the Trie that starts with the given prefix.

```
java public boolean startsWith(String prefix) { TrieNode current = root; for (char c :
prefix.toCharArray()) { int index = c - 'a'; if (current.children[index] == null) { return false; } current =
current.children[index]; } return true; }
```

4. Time Complexity

The efficiency of Trie operations depends on the length of the key (word or prefix), not the number of keys in the Trie.

Let L be the length of the string being processed.

5. Patterns

Tries are ideal for problems involving prefixes or sets of strings.

6. Common Question

Implement Trie (Prefix Tree)

This is the most fundamental problem. You are asked to build the Trie data structure from scratch, implementing its core functionalities: insert, search (for a whole word), and startsWith (for a prefix).

Design Add and Search Words Data Structure

This is a variation of the standard Trie. You build a Trie to store words, but the search functionality is enhanced to handle the . wildcard character. This requires a modified search algorithm that can explore multiple branches of the Trie simultaneously.

Word Search II

This is an advanced problem that combines a Trie with a Depth-First Search (DFS). The most efficient solution involves first inserting all the target words into a Trie. Then, you perform a DFS on the grid, traversing the Trie at the same time. The Trie allows you to instantly prune any search path that doesn't match a valid prefix, making the solution much faster than searching for each word individually.

7. Advantages and Disadvantages

Graphs

DSA Notes: Graphs

1. About Data Structure

A graph is a non-linear data structure consisting of vertices (nodes) and edges that connect these vertices[1][2]. Graphs are used to represent relationships between objects and are fundamental in modeling networks, dependencies, and complex relationships[1][3].

Unlike linear data structures such as arrays or linked lists, graphs allow multiple paths between elements, making them versatile for representing real-world scenarios like social networks, transportation systems, computer networks, and dependency graphs[2][4][5].

Key Components: - Vertices (Nodes): The fundamental units representing objects or entities[2][3] - Edges (Arcs): Connections between vertices representing relationships[2][3]

Types of Graphs: - Directed vs Undirected: Edges have direction (directed) or no direction (undirected)[1][2] - Weighted vs Unweighted: Edges may have associated weights/costs or all have equal weight[4] - Connected vs Disconnected: All vertices reachable from any vertex (connected) or isolated components exist[2] - Cyclic vs Acyclic: Contains cycles or no cycles[4]

2. Syntax (Java)

Basic Graph Class Structure

```
import java.util.*; // Graph using Adjacency List class Graph { private Map> adjList; public Graph() {
adjList = new HashMap<>(); } // Add vertex public void addVertex(int vertex) {
adjList.putIfAbsent(vertex, new ArrayList<>()); } // Add edge (undirected) public void addEdge(int
src, int dest) { adjList.get(src).add(dest); adjList.get(dest).add(src); neighbors public List
getNeighbors(int vertex) { return adjList.getOrDefault(vertex, new ArrayList<>()); } // Print graph
public void printGraph() { for (int vertex : adjList.keySet()) { System.out.print("Vertex " + vertex + ":
"); for (int neighbor : adjList.get(vertex)) { System.out.print(neighbor + " "); } System.out.println(); } } }
```

Generic Graph Implementation

```
import java.util.*; class Graph { private Map> adjList = new HashMap<>(); public void addVertex(T
vertex) { adjList.put(vertex, new LinkedList<>()); } public void addEdge(T source, T destination,
boolean bidirectional) { if (!adjList.containsKey(source)) addVertex(source); if
(!adjList.containsKey(destination)) addVertex(destination); adjList.get(source).add(destination); if
(bidirectional) { adjList.get(destination).add(source); } } public boolean hasVertex(T vertex) { return
adjList.containsKey(vertex); } public boolean hasEdge(T source, T destination) { return
adjList.get(source).contains(destination); } }
```

Adjacency Matrix Implementation

```
class GraphMatrix { private int[][] adjMatrix; private int numVertices; public GraphMatrix(int
numVertices) { this.numVertices = numVertices; adjMatrix = new int[numVertices][numVertices]; }
public void addEdge(int i, int j) { adjMatrix[i][j] = 1; adjMatrix[j][i] = 1; // For undirected graph } public
void removeEdge(int i, int j) { adjMatrix[i][j] = 0; adjMatrix[j][i] = 0; } public boolean hasEdge(int i, int
j) { return adjMatrix[i][j] == 1; } }
```

3. Operations with Syntax

Basic Operations

Depth-First Search (DFS)

```
public void dfs(int vertex, Set visited) { visited.add(vertex); System.out.print(vertex + " "); for (int
neighbor : adjList.get(vertex)) { if (!visited.contains(neighbor)) { dfs(neighbor, visited); } } } // Iterative
DFS public void dfsIterative(int start) { Set visited = new HashSet<>(); Stack stack = new Stack<>();
stack.push(start); while (!stack.isEmpty()) { int vertex = stack.pop(); if (!visited.contains(vertex)) {
visited.add(vertex); System.out.print(vertex + " "); for (int neighbor : adjList.get(vertex)) { if
(!visited.contains(neighbor)) { stack.push(neighbor); } } } } }
```

Breadth-First Search (BFS)

```
public void bfs(int start) { Set visited = new HashSet<>(); Queue queue = new LinkedList<>();
visited.add(start); queue.offer(start); while (!queue.isEmpty()) { int vertex = queue.poll();
System.out.print(vertex + " "); for (int neighbor : adjList.get(vertex)) { if (!visited.contains(neighbor)) {
visited.add(neighbor); queue.offer(neighbor); } } } }
```

Cycle Detection

```
// Cycle detection in undirected graph public boolean hasCycleUndirected() { Set visited = new
HashSet<>(); for (int vertex : adjList.keySet()) { if (!visited.contains(vertex)) { if (dfsHasCycle(vertex,
-1, visited)) { return true; } } } return false; } private boolean dfsHasCycle(int vertex, int parent, Set
visited) { visited.add(vertex); for (int neighbor : adjList.get(vertex)) { if (!visited.contains(neighbor)) {
if (dfsHasCycle(neighbor, vertex, visited)) { return true; } } else if (neighbor != parent) { return true; //
Cycle found } } return false; }
```

Topological Sort

```
public List topologicalSort() { Map inDegree = new HashMap<>(); Queue queue = new
LinkedList<>(); List result = new ArrayList<>(); // Calculate in-degrees for (int vertex :
adjList.keySet()) { inDegree.put(vertex, 0); } for (int vertex : adjList.keySet()) { for (int neighbor :
adjList.get(vertex)) { inDegree.put(neighbor, inDegree.get(neighbor) + 1); } } // Add vertices with 0
in-degree for (int vertex : inDegree.keySet()) { if (inDegree.get(vertex) == 0) { queue.offer(vertex); }
} while (!queue.isEmpty()) { int vertex = queue.poll(); result.add(vertex); for (int neighbor :
adjList.get(vertex)) { inDegree.put(neighbor, inDegree.get(neighbor) - 1); if (inDegree.get(neighbor)
== 0) { queue.offer(neighbor); } } } return result.size() == adjList.size() ? result : null; // Null if cycle
exists }
```

4. Time Complexity of Operations

Graph Representation Complexities

Traversal Algorithms

Explanation of O(V+E) complexity: - V: Each vertex is visited exactly once[6][7] - E: Each edge is examined once during traversal[6][7] - The algorithm processes all vertices and all edges, hence O(V+E)[7][9]

5. Patterns

Common Graph Patterns in Coding Interviews

1. Graph Traversal Pattern - DFS/BFS for connectivity: Finding connected components, path existence - Applications: Island counting, friend circles, maze solving

2. Cycle Detection Pattern - Undirected graphs: Use DFS with parent tracking - Directed graphs: Use DFS with recursion stack or topological sort - Applications: Dependency resolution, deadlock detection

3. Topological Sorting Pattern - Kahn's Algorithm: BFS-based using in-degree - DFS-based: Post-order traversal - Applications: Course scheduling, build systems, task dependencies

4. Shortest Path Pattern - Unweighted graphs: BFS - Weighted graphs: Dijkstra's algorithm - Negative weights: Bellman-Ford algorithm - All pairs: Floyd-Warshall algorithm

5. Union Find (Disjoint Set) Pattern - Dynamic connectivity: Check if nodes are connected - Applications: Minimum spanning tree, cycle detection, network connectivity

6. Bipartite Graph Pattern - Two-coloring: Check if graph can be colored with two colors - Applications: Matching problems, scheduling conflicts

7. Minimum Spanning Tree Pattern - Kruskal's Algorithm: Union-find based - Prim's Algorithm: Priority queue based - Applications: Network design, clustering

8. Strongly Connected Components Pattern - Kosaraju's Algorithm: Two-pass DFS - Tarjan's Algorithm: Single-pass DFS - Applications: Web crawling, social network analysis

Problem Recognition Patterns

6. Advantages and Disadvantages

Advantages

1. Flexibility and Expressiveness - Can model complex relationships and dependencies[10] - Suitable for representing real-world networks and systems[4][5] - Supports both directed and undirected relationships[4]

2. Efficient Algorithms - Well-established algorithms for common problems[10] - Linear-time complexity for many operations (O(V+E))[6][7] - Reusable implementations across different applications[10]

3. Scalability - Can handle large datasets efficiently with proper representation[10] - Adjacency list representation is memory-efficient for sparse graphs[11][12]

4. Real-world Applications - Social networks, transportation systems, computer networks[5] - Dependency management, scheduling, pathfinding[8] - Recommendation systems, web crawling[4]

Disadvantages

1. Complexity - Can be complex to understand and implement[13][10] - Requires careful consideration of representation choice[11] - Some algorithms have steep learning curves[10]

2. Memory and Performance Issues - Adjacency matrix uses $O(V^2)$ space even for sparse graphs[13][12] - Large graphs can become difficult to manage[13][14] - Performance degrades with increasing vertices and edges[13]

3. Representation Limitations - Graphs only represent relationships, not object properties[13] - Choice between adjacency list vs matrix affects performance[11][12] - Dense graphs may require different optimization strategies[11]

4. Visualization and Debugging Challenges - Large graphs are difficult to visualize and understand[13][14] - Debugging graph algorithms can be challenging[13] - Maintenance of complex graph structures requires additional effort[10]

5. Specific Algorithmic Limitations - Some algorithms only work with specific graph types (e.g., DAGs)[8] - Negative cycles can cause issues in shortest path algorithms - Strongly connected component algorithms are complex to implement[8]

When to Use Different Representations

Adjacency List: - Best for: Sparse graphs ($E << V^2$)[11][12] - Advantages: Memory efficient, fast neighbor iteration[11] - Use cases: Social networks, web graphs, dependency graphs[11]

Adjacency Matrix: - Best for: Dense graphs ($E \approx V^2$)[11][12] - Advantages: Constant-time edge lookup, simple implementation[11] - Use cases: Complete graphs, small graphs with frequent edge queries[11]

This comprehensive overview provides the foundation for understanding graphs as a fundamental data structure in computer science, with practical Java implementations and real-world applications that align with your software development background.

[1] https://www.simplilearn.com/tutorials/data-structure-tutorial/graphs-in-data-structure [2] https://www.geeksforgeeks.org/dsa/introduction-to-graphs-data-structure-and-algorithm-tutorials/ [3] https://en.wikipedia.org/wiki/Graph_(abstract_data_type) [4] https://dzone.com/articles/introduction-to-the-graph-data-structure [5] https://www.w3schools.com/dsa/dsa_theory_graphs.php [6] https://www.geeksforgeeks.org/dsa/time-and-space-complexity-of-dfs-and-bfs-algorithm/ [7] https://www.geeksforgeeks.org/dsa/why-is-the-complexity-of-both-bfs-and-dfs-ove/ [8] https://www.geeksforgeeks.org/dsa/graph-based-algorithms-for-gate-exam/ [9] https://stackoverflow.com/questions/11468621/why-is-the-time-complexity-of-both-dfs-and-bfs-o-v-e [10] https://askfilo.com/user-question-answers-smart-solutions/advantages-disadvantages-and-applications-of-dsa-algorithm-3335313934313335 [11] https://imkarthikeyans.hashnode.dev/understanding-graphs-in-javascript-adjacency-matrix-vs-adjacency-list [12] https://www.baeldung.com/cs/graphs [13] https://www.geeksforgeeks.org/dsa/applications-advantages-and-disadvantages-of-graph/ [14] https://www.tutorialspoint.com/applications-advantages-and-disadvantages-of-graph [15] https://dev.to/hellocodeclub/graph-implementation-example-in-java-1mhh [16] https://dev.to/vijayr00/understanding-dsa-patterns-p9b [17] https://www.softwaretestinghelp.com/java-graph-tutorial/ [18] https://www.scribd.com/document/843938448/DSA-Techniques-and-Patterns [19] https://www.programiz.com/java-programming/examples/graph-implementation [20] https://blog.algomaster.io/p/master-graph-algorithms-for-coding [21] https://herovired.com/learning-hub/blogs/graphs-in-data-structure/ [22] https://www.geeksforgeeks.org/java/implementing-generic-graph-in-java/ [23] https://www.inf.unibz.it/~nutt/Teaching/DSA1617/DSASlides/chapter07.pdf [24] https://www.geeksforgeeks.org/dsa/what-is-graph-data-structure/ [25] https://docs.vultr.com/java/examples/implement-the-graph-data-structure [26] https://javachallengers.com/graph-data-structure-with-java/ [27] https://hackernoon.com/5-graph-patterns-to-ace-coding-interviews [28] https://byjus.com/gate/graph-and-its-applications/ [29] https://www.baeldung.com/java-graphs [30] https://www.geeksforgeeks.org/dsa/time-complexity-and-space-complexity/ [31] https://stackoverflow.com/questions/77568628/adjacency-matrix-and-adjacency-list-time-complexity [32] https://www.w3schools.com/dsa/dsa_timecomplexity_theory.php [33] https://www.youtube.com/watch?v=0QnFZlorhqc [34] https://www.geeksforgeeks.org/dsa/graph-data-structure-and-algorithms/ [35] https://www.educative.io/answers/what-is-an-adjacency-list [36] https://www.wscubetech.com/resources/dsa/graph-algorithms [37] https://www.youtube.com/watch?v=ZpOy0-QBVPM [38] https://dev.to/akashdeep/big-o-complexity-for-graphs-adjacency-matrix-vs-adjacency-list-3akf [39] https://www.baeldung.com/java-collections-complexity [40] https://www.geeksforgeeks.org/time-and-space-complexity-of-depth-first-search-dfs/ [41] https://www.cs.cmu.edu/~eugene/teach/algs03b/works/s10.pdf [42] https://www.educative.io/cours

es/data-structures-coding-interviews-java/complexities-of-graph-operations [43]
https://www.slideshare.net/slideshow/bfs-and-dfs-244040667/244040667 [44]
https://stackoverflow.com/questions/44983431/time-complexity-of-adjacency-list-representation
[45] https://dev.to/_hm/data-structure-operations-time-complexity-guide-19pi [46]
https://visualgo.net/en/dfsbfs [47]
https://www.cs.cornell.edu/courses/cs2112/2012sp/lectures/lec24/lec24-12sp.html [48] https://open
dsa.cs.vt.edu/ODSA/Books/vt/cs5040/summer-ii-2023/Summer2023/html/GraphTraversal.html [49]
https://www.ida.liu.se/opendsa/Books/TDDE22F24/html/GraphTraversal.html [50] https://www.stud
ocu.com/en-us/document/oklahoma-state-university/data-structures-and-algorithms-ii/graph-repres
entation-and-operations-adjacency-matrix-vs-list-6092/125925831 [51]
https://www.ida.liu.se/opendsa/Books/TDDI16F20/html/GraphTraversal.html [52] https://www.slides
hare.net/CHIRANTANMONDAL2/brief-reletionship-between-adjacency-matrix-and-listpptx [53]
https://www.tutorialspoint.com/applications-advantages-and-disadvantages-of-directed-graph [54] h
ttps://opendsa-server.cs.vt.edu/ODSA/Books/pubbook/cs3/fall-2024/CS3_Public_F24/html/GraphTr
aversal.html [55] https://www.geeksforgeeks.org/dsa/graph-and-its-representations/ [56] https://ww
w.designgurus.io/answers/detail/what-are-common-graph-traversal-questions-in-coding-interviews
[57] https://www.geeksforgeeks.org/comparison-between-adjacency-list-and-adjacency-matrix-repr
esentation-of-graph/

Binary Search

Binary Search

Binary Search is a highly efficient searching algorithm that works on sorted arrays. It follows a divide-and-conquer approach by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, it narrows the interval to the lower half; otherwise, it narrows it to the upper half. This process continues until the value is found or the interval is empty.

Syntax (Java)

Here are the standard iterative and recursive implementations for binary search in Java.

Iterative Implementation

```
int binarySearch(int[] arr, int target) { int left = 0; int right = arr.length - 1; while (left <= right) { // To
prevent overflow for large left/right values int mid = left + (right - left) / 2; if (arr[mid] == target) {
return mid; // Target found, return its index } if (arr[mid] < target) { left = mid + 1; // Search in the
right half } else { right = mid - 1; // Search in the left half } } return -1; // Target not found }
```

Recursive Implementation

```
int recursiveBinarySearch(int[] arr, int left, int right, int target) { if (left <= right) { int mid = left + (right
- left) / 2; if (arr[mid] == target) { return mid; // Target found } if (arr[mid] < target) { // Recur on the
right half return recursiveBinarySearch(arr, mid + 1, right, target); } else { // Recur on the left half
return recursiveBinarySearch(arr, left, mid - 1, target); } } return -1; // Target not found }
```

Time & Space Complexity

The efficiency of binary search is one of its key advantages.

Common Patterns ■

Binary search is not just for finding an element. It's a powerful technique for a variety of problems on sorted or partially sorted data.

Common Question

The key insight for these problems is to find a monotonic property (something that is sorted or consistently increasing/decreasing) that allows you to repeatedly cut the search space in half.

Direct Binary Search

These problems apply binary search directly to a sorted data structure.

Binary Search on a Modified Sorted Array

These problems involve arrays that are "almost sorted," requiring a modified binary search to navigate the rotation or pivot point.

Binary Search on the "Answer"

For these problems, you aren't searching through the input array itself, but rather searching for the correct answer within a possible range of answers.

Advantages and Disadvantages

■ Advantages

■ Disadvantages

Greedy Algorithms

A greedy algorithm is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. It makes the locally optimal choice at each stage with the hope of finding a global optimum. greedy algorithms don't always produce the optimal solution, but they are very effective for certain problems.

Syntax (Java General Template)

Greedy algorithms are a design approach, not a specific data structure. The implementation varies by problem. Here's a general structure.

```
// A generic structure for a greedy algorithm in Java public class GreedySolution { // Represents an
item or choice in the problem static class Item { // Attributes like weight, value, start time, end time,
etc. // ... public Item(/*...*/) { /*...*/ } } // The core greedy logic public Result solve(List items) { // 1.
Pre-processing: Often involves sorting the items based on a greedy criterion. // For example, sort by
value/weight ratio, end time, etc. Collections.sort(items, (a, b) -> { // Custom comparator logic based
on the greedy choice return /* comparison result */; }); Result solution = new Result(); // 2. Iteration:
Loop through the sorted items. for (Item currentItem : items) { // 3. Greedy Choice: Decide if the
current item can be added to the solution. if (isFeasible(currentItem, solution)) { // 4. Update
Solution: Add the item and update the state. addToSolution(currentItem, solution); } } return
solution; } // Helper methods specific to the problem private boolean isFeasible(Item item, Result
currentSolution) { /*...*/ return true; } private void addToSolution(Item item, Result currentSolution) {
/*...*/ } // Placeholder for the result structure static class Result { /*...*/ } }
```

Operations with Syntax

Let's use the Activity Selection Problem as an example. The goal is to select the maximum number of non-overlapping activities from a set of activities, each with a start and end time.

The greedy choice is to always pick the next activity that finishes earliest.

```
import java.util.*; class Activity { int start; int end; Activity(int start, int end) { this.start = start; this.end
= end; } } public class ActivitySelection { public static List selectActivities(List activities) { // 1. Sort
activities based on their finish times in ascending order. activities.sort(Comparator.comparingInt(a
-> a.end)); List result = new ArrayList<>(); if (activities.isEmpty()) { return result; } // 2. Select the
first activity. Activity lastSelected = activities.get(0); result.add(lastSelected); // 3. Iterate through the
remaining activities. for (int i = 1; i < activities.size(); i++) { Activity currentActivity = activities.get(i); //
4. If the current activity starts after or at the same time the last one finishes, select it. if
(currentActivity.start >= lastSelected.end) { result.add(currentActivity); lastSelected =
currentActivity; } } return result; } }
```

Time Complexity of the Operations

The time complexity of a greedy algorithm is typically determined by two main parts: the sorting step and the iteration step.

Therefore, the overall time complexity is O(N \log N). If the input is already sorted, the complexity reduces to O(N).

Patterns

Greedy algorithms are effective for problems that exhibit two key properties:

Common problems that can be solved using a greedy approach include:

Common Questions

Maximum Subarray

This is solved with Kadane's algorithm. The greedy choice at each step is to either extend the previous subarray by adding the current number or to start a new subarray beginning with the current number, whichever is larger.

Jump Game

The greedy strategy is to always maximize your reach. As you iterate through the array, you keep track of the farthest index you can possibly get to. At each position, you make the jump that extends this maximum reach.

Jump Game II

This is also greedy. For each jump, you explore all the positions you can currently reach and find the one that allows you to jump the farthest in the next move. This ensures you cover the maximum distance with each jump, minimizing the total number of jumps.

Gas Station

The greedy approach involves a single pass. If your total gas in the tank becomes negative at any point, you know that no starting position up to that point could have worked. So, you greedily try the next station as a new potential starting point.

Hand of Straights

A greedy solution works well here. First, count the card frequencies. Then, repeatedly find the smallest card number available and greedily form a consecutive group starting with that card. If you can't form a group at any point, it's impossible.

Merge Triplets to Form Target Triplet

The greedy choice is to first discard any triplet that has a value larger than the target in any position. Then, you iterate through the remaining "good" triplets and greedily merge them by taking the maximum value at each position to build towards the target.

Partition Labels

This uses a greedy strategy where you extend the current partition to include the last occurrence of every character within that partition. Once your pointer reaches the end of the current partition's boundary, you cut it off and start a new one.

Valid Parenthesis String

A greedy approach tracks the possible range of open parentheses [min, max]. At each step, you make the choices for the wildcard * that are most likely to keep the string valid (i.e., using it as (for the max count and) for the min count).

Advantages and Disadvantages

■ Advantages

■ Disadvantages

Interval Patterns

Merge Intervals Pattern

1. About the Algorithm ■

The Merge Intervals pattern is a popular algorithmic technique used to solve problems involving intervals. An interval is typically represented by a start and end point, like [start, end]. This pattern is used to deal with overlapping intervals in an efficient way. The core idea is almost always to sort the intervals by their start times and then iterate through the sorted list, merging or processing them based on whether they overlap.

2. Syntax (Java) ■

In Java, you can represent intervals using a 2D array (int[][]) or a custom Interval class for better readability.

```
// Using a 2D array int[][] intervals = {{1, 3}, {2, 6}, {8, 10}, {15, 18}}; // Using a custom class
(recommended) class Interval { int start; int end; Interval(int start, int end) { this.start = start; this.end
= end; } }
```

3. Operations with Syntax ■■

The fundamental operation is merging. After sorting the intervals by their start time, you iterate through them and merge any that overlap.

Example: Merging Overlapping Intervals

```
import java.util.ArrayList; import java.util.Arrays; import java.util.List; class Solution { public int[][]
merge(int[][] intervals) { if (intervals.length <= 1) { return intervals; } // 1. Sort intervals based on the
start time Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0])); List merged = new
ArrayList<>(); int[] currentInterval = intervals[0]; merged.add(currentInterval); // 2. Iterate and merge
for (int[] interval : intervals) { int currentEnd = currentInterval[1]; int nextStart = interval[0]; int
nextEnd = interval[1]; if (currentEnd >= nextStart) { // Overlap detected currentInterval[1] =
Math.max(currentEnd, nextEnd); } else { // No overlap currentInterval = interval;
merged.add(currentInterval); } } return merged.toArray(new int[merged.size()][]); } }
```

4. Time Complexity of the Operations ■■

5. Patterns ■

This approach is the foundation for solving many related problems, including:

6. Advantages and Disadvantages ■■