Contents

Data Structures & Algorithms — Complete Course

Every topic below follows the same template: **Definition → Intuition → Real-Life Example → Diagram → Java Code → Dry Run → Brute-Force vs Optimized → Complexity → Common Mistakes → Interview Tips → Practice Questions with Solutions.**

Table of Contents

---

1. How to Use This Course

Each topic is self-contained. Don't skip the "dry run" sections — tracing code by hand on paper, one line at a time, is what converts reading into understanding. Code is in **Java** since it's the most common interview language and forces you to think about types, which matters for correctness.

Study loop for every topic: read definition → understand intuition → trace the dry run yourself before reading it → write the code from memory → solve the practice questions without looking at the solution first.

---

2. Complexity Analysis (Big-O, Big-Ω, Big-Θ)

Definition: Complexity analysis measures how an algorithm's time or memory usage grows as input size `n` grows toward infinity, ignoring constants and hardware-specific factors.

Intuition: You're not measuring "how many milliseconds" — you're measuring "how many more operations do I do if I double the input." That growth *rate* is what Big-O captures. Big-O = upper bound (worst case never exceeds this). Big-Ω = lower bound (best case is never better than this). Big-Θ = tight bound (used when upper and lower match — this is what people mean when they casually say "the complexity").

Real-life example: Looking up a name in a phone book by flipping through every page one at a time (linear, O(n)) versus opening to the middle and halving your search each time because it's alphabetically sorted (logarithmic, O(log n)).

Diagram — growth rates, slowest-growing to fastest-growing:

```
O(1)  <  O(log n)  <  O(n)  <  O(n log n)  <  O(n^2)  <  O(2^n)  <  O(n!)

Operations vs n (illustrative):
n=10      O(log n)=3     O(n)=10      O(n log n)=33     O(n^2)=100
n=1000    O(log n)=10    O(n)=1000    O(n log n)=9966   O(n^2)=1,000,000
```

Java code — measuring growth directly:

```java
public class ComplexityDemo {
    // O(n): one pass
    static int linearSum(int[] arr) {
        int sum = 0;
        for (int x : arr) sum += x;   // n iterations
        return sum;
    }

    // O(n^2): nested loops over the same input
    static int pairCount(int[] arr) {
        int count = 0;
        for (int i = 0; i < arr.length; i++)
```

```
            for (int j = i + 1; j < arr.length; j++)
                count++;              // n*(n-1)/2 iterations -> O(n^2)
        return count;
    }

    // O(log n): halve the problem each step
    static int binarySearch(int[] arr, int target) {
        int lo = 0, hi = arr.length - 1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            if (arr[mid] == target) return mid;
            else if (arr[mid] < target) lo = mid + 1;
            else hi = mid - 1;
        }
        return -1;
    }
}
```

Dry run — `pairCount([3, 7, 2, 9])`, n = 4:

```
i=0: j=1,2,3          -> 3 comparisons
i=1: j=2,3             -> 2 comparisons
i=2: j=3                -> 1 comparison
i=3: (no j left)        -> 0 comparisons
Total = 3+2+1+0 = 6 = n*(n-1)/2 = 4*3/2 = 6  ✓  O(n^2)
```

Brute-force vs optimized: this topic *is* the tool you use to compare brute-force vs optimized approaches everywhere else in this course — every subsequent topic will state a brute-force complexity and an optimized complexity, and this section is what lets you interpret those numbers.

Complexity of complexity analysis itself (rules to derive it): | Rule | Example | |—|—| | Drop constants | O(2n) → O(n) | | Drop lower-order terms | $O(n^2 + n) \rightarrow O(n^2)$ | | Sequential loops add | loop of n then loop of m → O(n+m) | | Nested loops multiply | loop of n inside loop of n → $O(n^2)$ | | Different inputs get different variables | two arrays size n, m → O(n+m), not O(n) |

Common mistakes: - Confusing average case with worst case (quicksort is O(n log n) average but $O(n^2)$ worst case — both facts matter). - Forgetting space complexity — recursion has O(depth) space from the call stack even with no extra data structures. - Treating O(n) and O(2n) as meaningfully different — they aren't, asymptotically. - Assuming a nested loop is always $O(n^2)$ — if the inner loop's range shrinks (like in `pairCount` above), you must actually sum it up; it can still land on $O(n^2)$ but you should verify, not assume.

Interview tips: - Always state complexity out loud before you're asked — it signals you're thinking about it, not bolting it on after. - When asked to optimize, first name *why* the current solution is slow (redundant work? unnecessary full scan?) before jumping to a fix. - Know the Master Theorem exists for recurrences of the form T(n) = aT(n/b) + f(n) — you don't need to derive it live, but recognize O(n log n) as the signature of "divide in half, do O(n) work to combine."

Practice questions with solutions:

Q1: What is the time complexity of the following code?

```
for (int i = 1; i < n; i *= 2) {
    System.out.println(i);
}
```

Solution: `i` doubles each iteration (1, 2, 4, 8, … until ≥ n). Number of iterations = $\log_2(n)$. **Answer: O(log n).**

Q2: What is the time complexity of the following code?

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        System.out.println(i + j);
```

Solution: Outer loop runs n times, inner loop runs m times for each outer iteration → n × m total operations. **Answer: O(n·m)** — not $O(n^2)$, since n and m are independent variables unless stated otherwise.

Q3: Rank these by growth rate (slowest to fastest): O(n log n), $O(2^n)$, O(n), O(log n), $O(n^2)$. **Solution:** $O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$.

---

3. Arrays

Definition: A fixed-size, contiguous block of memory holding elements of the same type, accessed directly by index.

Intuition: Because elements sit next to each other in memory, the address of index `i` is computed by simple arithmetic: `base_address + i * element_size`. No searching, no traversal — this is *why* array access is O(1). Every other array operation's complexity follows from this one fact about how memory is laid out.

Real-life example: A row of numbered parking spots — you don't have to walk past every car to reach spot #47, you drive straight to it because the spot numbers map directly to physical positions.

Diagram:

```
Index:   0    1    2    3    4
Value: [ 10 | 25 | 7  | 42 | 3  ]
Address computation for arr[3]: base + 3*4 bytes (for an int) = direct jump, O(1)
```

Java code — dynamic array resizing (how ArrayList works under the hood):

```
public class DynamicArray {
    private int[] arr;
    private int size = 0;
    private int capacity = 1;

    public DynamicArray() {
        arr = new int[capacity];
    }

    public void append(int value) {
        if (size == capacity) {
            capacity *= 2;                          // double capacity
            int[] newArr = new int[capacity];
            for (int i = 0; i < size; i++) newArr[i] = arr[i];  // O(n) copy
            arr = newArr;
        }
        arr[size++] = value;                        // O(1)
    }

    public int get(int index) {
        if (index < 0 || index >= size) throw new IndexOutOfBoundsException();
        return arr[index];                           // O(1)
```

```
    }
}
```

Dry run — appending 1, 2, 3 into a `DynamicArray` starting at capacity 1:

```
append(1): size=0=capacity(1) -> resize to capacity=2, copy nothing, arr=[1,_], size=1
append(2): size=1<capacity(2) -> arr=[1,2], size=2
append(3): size=2=capacity(2) -> resize to capacity=4, copy [1,2], arr=[1,2,3,_], size=3
```

Across 3 appends, only 2 copy operations happened (1 element, then 2 elements) despite capacity doubling twice — this is why amortized cost per append is O(1), even though individual resizes cost O(n).

Brute-force vs optimized — "two-sum" style example (does a sorted array contain a pair summing to target?):

Brute-force — check every pair:

```java
static boolean hasPairBruteForce(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++)
        for (int j = i + 1; j < arr.length; j++)
            if (arr[i] + arr[j] == target) return true;
    return false;
}
// Time: O(n^2), Space: O(1)
```

Optimized — two pointers (array is sorted):

```java
static boolean hasPairOptimized(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left < right) {
        int sum = arr[left] + arr[right];
        if (sum == target) return true;
        else if (sum < target) left++;
        else right--;
    }
    return false;
}
// Time: O(n), Space: O(1)
```

Dry run of the optimized version — `arr = [2, 7, 11, 15]`, `target = 18`:

```
left=0(2), right=3(15): sum=17 < 18 -> left++
left=1(7), right=3(15): sum=22 > 18 -> right--
left=1(7), right=2(11): sum=18 == 18 -> return true
```

Complexity summary: | Operation | Complexity | Why | |—|—|—| | Access by index | O(1) | Direct address computation | | Search (unsorted) | O(n) | Must check every element | | Search (sorted, binary search) | O(log n) | Halve search space each step | | Insert/delete at end | O(1) amortized | No shifting; occasional resize | | Insert/delete at start/middle | O(n) | Must shift subsequent elements |

Common mistakes: - Off-by-one errors on array bounds (< vs <=) — the single most common source of bugs in this entire course. - Forgetting that Java arrays have fixed size — you cannot "append" to a raw `int[]`; you must create a new array or use `ArrayList`. - Modifying an array while iterating over it with a for-each loop (`ConcurrentModificationException` for `ArrayList`, silent bugs for raw arrays). - Assuming two-pointer technique works on unsorted data — it requires sorted input to guarantee correctness.

Interview tips: - If a problem mentions a *sorted* array, two pointers or binary search should be

your first instinct. - State the brute force and its complexity before optimizing — interviewers want to see you can identify the bottleneck, not just recite a memorized trick. - Mention in-place vs extra-space trade-offs explicitly; many array problems have an O(1)-space solution that's worth calling out even if you first reach for one using extra space.

Practice questions with solutions:

Q1: Find the maximum subarray sum (Kadane's Algorithm). Input: `[-2,1,-3,4,-1,2,1,-5,4]`.

```
static int maxSubArray(int[] nums) {
    int bestHere = nums[0], bestOverall = nums[0];
    for (int i = 1; i < nums.length; i++) {
        bestHere = Math.max(nums[i], bestHere + nums[i]);
        bestOverall = Math.max(bestOverall, bestHere);
    }
    return bestOverall;
}
```

Solution walkthrough: bestHere tracks "best sum ending at this index"; bestOverall tracks the max seen so far. Trace: bestHere sequence = -2,1,-2,4,3,5,6,1,5 → max = **6** (subarray `[4,-1,2,1]`). Time: O(n), Space: O(1).

Q2: Given an array, move all zeros to the end while keeping the relative order of non-zero elements. Input: `[0,1,0,3,12]`.

```
static void moveZeroes(int[] nums) {
    int insertPos = 0;
    for (int num : nums)
        if (num != 0) nums[insertPos++] = num;
    while (insertPos < nums.length)
        nums[insertPos++] = 0;
}
```

Solution walkthrough: Single pass copies non-zero elements to the front (insertPos tracks where), then fill the rest with zeros. Result: `[1,3,12,0,0]`. Time: O(n), Space: O(1).

Q3: Find the missing number in an array containing n distinct numbers from 0 to n. Input: `[3,0,1]` (n=3, missing 2).

```
static int missingNumber(int[] nums) {
    int n = nums.length;
    int expectedSum = n * (n + 1) / 2;
    int actualSum = 0;
    for (int x : nums) actualSum += x;
    return expectedSum - actualSum;
}
```

Solution walkthrough: Sum of 0..n = n(n+1)/2 = 3*4/2 = 6. Actual sum = 3+0+1 = 4. Missing = 6-4 = **2**. Time: O(n), Space: O(1).

---

4. Strings

Definition: Conceptually an array of characters. In Java, `String` is **immutable** — every modification produces a new `String` object rather than changing the original in place.

Intuition: Because strings are immutable in Java, repeated concatenation in a loop silently becomes quadratic — each += copies the entire string built so far. This is one of the highest-value "gotchas" to internalize early: use `StringBuilder` for anything built incrementally.

Real-life example: Rewriting an entire printed letter from scratch every time you add one word, versus writing on a whiteboard where you just add the word at the end — `StringBuilder` is the whiteboard.

Diagram:

```
String s = "cat";  s = s + "s";
"cat" (old object, now garbage) ---> "cats" (brand-new object)
Immutability means the original "cat" is never modified — a new object is allocated.
```

Java code — the concatenation trap and its fix:

```java
// BAD: O(n^2) total — every += copies the whole string so far
static String buildBad(char[] chars) {
    String s = "";
    for (char c : chars) s += c;
    return s;
}

// GOOD: O(n) total — StringBuilder mutates an internal buffer
static String buildGood(char[] chars) {
    StringBuilder sb = new StringBuilder();
    for (char c : chars) sb.append(c);
    return sb.toString();
}
```

Brute-force vs optimized — anagram check. Input: `"listen"`, `"silent"`.

Brute-force — sort both strings and compare, O(n log n):

```java
static boolean isAnagramSort(String a, String b) {
    if (a.length() != b.length()) return false;
    char[] ca = a.toCharArray(), cb = b.toCharArray();
    java.util.Arrays.sort(ca);
    java.util.Arrays.sort(cb);
    return java.util.Arrays.equals(ca, cb);
}
```

Optimized — frequency count with a fixed-size array (assumes lowercase a-z), O(n):

```java
static boolean isAnagramCount(String a, String b) {
    if (a.length() != b.length()) return false;
    int[] freq = new int[26];
    for (int i = 0; i < a.length(); i++) {
        freq[a.charAt(i) - 'a']++;
        freq[b.charAt(i) - 'a']--;
    }
    for (int f : freq) if (f != 0) return false;
    return true;
}
```

Dry run of `isAnagramCount("cat", "act")`:

```
'c'-'a'=2: freq[2]++ -> freq[2]=1   (from a)
'a'-'a'=0: freq[0]++ -> freq[0]=1
't'-'a'=19: freq[19]++ -> freq[19]=1
'a'-'a'=0: freq[0]-- -> freq[0]=0   (from b)
'c'-'a'=2: freq[2]-- -> freq[2]=0
't'-'a'=19: freq[19]-- -> freq[19]=0
All freq entries are 0 -> return true
```

Substring search algorithms (complexity comparison): | Algorithm | Time | Idea | |—|—|—| | Naive sliding compare | O(n·m) | Try every starting position, compare char by char | | KMP | O(n+m) | Precompute a failure function so mismatches never re-scan text | | Rabin-Karp | O(n+m) average | Rolling hash — compare hash values in O(1), not full substrings |

Java code — palindrome check via two pointers:

```
static boolean isPalindrome(String s) {
    int left = 0, right = s.length() - 1;
    while (left < right) {
        if (s.charAt(left) != s.charAt(right)) return false;
        left++; right--;
    }
    return true;
}
```

Complexity summary: | Operation | Complexity | |—|—| | Access char by index | O(1) | | Naive concatenation in a loop (n times) | O(n^2) total | | StringBuilder append in a loop (n times) | O(n) total | | Naive substring search | O(n·m) | | KMP / Rabin-Karp substring search | O(n+m) |

Common mistakes: - Using `+=` in a loop instead of `StringBuilder` — passes small test cases, silently fails performance requirements at scale. - Using `==` instead of `.equals()` to compare String content in Java — `==` compares object references, not content. - Off-by-one on `charAt` indices when doing two-pointer or sliding window string problems. - Forgetting case sensitivity and whitespace when checking palindromes/anagrams — always clarify with the interviewer whether to normalize.

Interview tips: - Always ask about character set (ASCII vs Unicode) — it determines whether a fixed 26/128/256-size frequency array is valid or you need a `HashMap<Character,Integer>`. - Mention `StringBuilder` proactively any time you build a string incrementally — it's a fast signal of Java fluency. - For substring search problems beyond a single pattern match, mention KMP/Rabin-Karp exist even if you implement the naive version under time pressure — naming the optimal approach matters.

Practice questions with solutions:

Q1: Reverse a string in place using two pointers. Input: `"hello"`.

```
static void reverseString(char[] s) {
    int left = 0, right = s.length - 1;
    while (left < right) {
        char temp = s[left];
        s[left] = s[right];
        s[right] = temp;
        left++; right--;
    }
}
```

Solution walkthrough: Swap outer characters inward. `h<->o`, `e<->l` → `"olleh"`. Time: O(n), Space: O(1).

Q2: Find the first non-repeating character in a string. Input: `"leetcode"`.

```
static char firstUniqChar(String s) {
    int[] freq = new int[256];
    for (char c : s.toCharArray()) freq[c]++;
    for (char c : s.toCharArray())
        if (freq[c] == 1) return c;
    return '_'; // none found
}
```

Solution walkthrough: First pass counts frequency of every character; second pass returns the first character with count 1. For `"leetcode"`: l=1,e=3,t=1,c=1,o=1,d=1 → first with count 1 is **'l'**. Time: O(n), Space: O(1) (fixed 256-size array).

Q3: Check if string `b` is a rotation of string `a` (e.g., `"waterbottle"` and `"erbottlewat"`).

```
static boolean isRotation(String a, String b) {
    if (a.length() != b.length()) return false;
    return (a + a).contains(b);
}
```

Solution walkthrough: Any rotation of `a` must appear as a substring of `a` concatenated with itself. `"waterbottle" + "waterbottle"` contains `"erbottlewat"` → **true**. Time: O(n) for the substring check (Java's `contains` uses efficient substring search internally), Space: O(n) for the doubled string. ## 5. Linked Lists

Definition: A sequence of nodes, each holding a value and a pointer to the next node. Memory is *not* contiguous — nodes live wherever the allocator places them, connected only by pointers.

Intuition: You trade array's O(1) random access for O(1) insertion/deletion once you're already at the right node — no shifting elements, just rewiring two pointers. This is the fundamental array-vs-linked-list trade-off, and every comparison between the two structures traces back to this one fact.

Real-life example: A treasure hunt where each clue tells you where to find the next clue — you can't jump straight to clue #7, you must follow clue #1 → #2 → ... → #7. But inserting a new clue between two existing ones just means changing what one clue points to.

Diagram:

```
Singly:  [10|•]-->[20|•]-->[30|•]-->[null]
Doubly:  null<--[10|•|•]<-->[20|•|•]<-->[30|•|•]-->null
                 prev|val|next
```

Java code — singly linked list with insert, reverse:

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }
}

class LinkedList {
    ListNode head;

    void insertAtHead(int val) {
        ListNode node = new ListNode(val);
        node.next = head;
        head = node;                          // O(1)
    }

    ListNode reverse() {
        ListNode prev = null, curr = head;
        while (curr != null) {
            ListNode next = curr.next;        // save next before overwriting
            curr.next = prev;                 // reverse the pointer
            prev = curr;
            curr = next;
        }
        head = prev;
```

```
        return head;
    }
}
```

Dry run — reversing `1 -> 2 -> 3 -> null`:

```
Start: prev=null, curr=1->2->3->null
Step1: next=2->3->null; curr(1).next=null; prev=1->null; curr=2->3->null
Step2: next=3->null;    curr(2).next=1->null; prev=2->1->null; curr=3->null
Step3: next=null;       curr(3).next=2->1->null; prev=3->2->1->null; curr=null
Loop ends. head=prev=3->2->1->null
```

Brute-force vs optimized — detect a cycle:

Brute-force — hash set of visited nodes, O(n) space:

```
static boolean hasCycleHashSet(ListNode head) {
    java.util.Set<ListNode> seen = new java.util.HashSet<>();
    ListNode curr = head;
    while (curr != null) {
        if (seen.contains(curr)) return true;
        seen.add(curr);
        curr = curr.next;
    }
    return false;
}
// Time: O(n), Space: O(n)
```

Optimized — Floyd's fast & slow pointers, O(1) space:

```
static boolean hasCycleFloyd(ListNode head) {
    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) return true;   // they lap each other inside a cycle
    }
    return false;
}
// Time: O(n), Space: O(1)
```

Complexity summary: | Operation | Array | Linked List | |—|—|—| | Access by index | O(1) | O(n) | | Insert/delete at known node | O(n) (shifting) | O(1) (pointer rewire) | | Insert/delete at head | O(n) | O(1) | | Search | O(n) | O(n) |

Common mistakes: - Losing the reference to the rest of the list by overwriting `.next` before saving it (`next = curr.next` must come *before* `curr.next = prev`). - Not handling `null` head or single-node lists as edge cases. - Forgetting to update `head` after reversal (a very common silent bug — the function reverses correctly but returns the wrong node). - Using == correctly here is actually right (comparing node references), unlike Strings — don't second-guess it into `.equals()`.

Interview tips: - Draw the list on paper/whiteboard before coding — pointer bugs are caught visually far faster than mentally. - Always ask: singly or doubly linked? Circular or not? This changes which operations are O(1). - Mention fast/slow pointers immediately when you see "middle of list," "cycle," or "nth from end" — these are near-universal signals for this technique.

Practice questions with solutions:

Q1: Find the middle node of a linked list. Input: 1->2->3->4->5.

```java
static ListNode middleNode(ListNode head) {
    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}
```

Solution walkthrough: fast moves 2x speed of slow; when fast reaches the end, slow is at the midpoint. For 5 nodes: slow ends at node **3**. Time: O(n), Space: O(1).

Q2: Merge two sorted linked lists. Input: 1->2->4 and 1->3->4.

```java
static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode tail = dummy;
    while (l1 != null && l2 != null) {
        if (l1.val <= l2.val) { tail.next = l1; l1 = l1.next; }
        else { tail.next = l2; l2 = l2.next; }
        tail = tail.next;
    }
    tail.next = (l1 != null) ? l1 : l2;
    return dummy.next;
}
```

Solution walkthrough: Use a dummy head to avoid special-casing the first node; always attach the smaller current node, advance that list. Result: `1->1->2->3->4->4`. Time: O(n+m), Space: O(1).

Q3: Remove the nth node from the end of a list. Input: 1->2->3->4->5, n=2.

```java
static ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode fast = dummy, slow = dummy;
    for (int i = 0; i < n; i++) fast = fast.next;   // advance fast n steps first
    while (fast.next != null) { fast = fast.next; slow = slow.next; }
    slow.next = slow.next.next;                      // skip the target node
    return dummy.next;
}
```

Solution walkthrough: Advancing `fast` n steps ahead first means when `fast` hits the end, `slow` sits right before the node to remove. For n=2 on `1->2->3->4->5`: removes node 4 → `1->2->3->5`. Time: O(n), Space: O(1).

---

6. Stacks

Definition: A Last-In-First-Out (LIFO) structure supporting `push` (add to top) and `pop` (remove from top) in O(1).

Intuition: Only the top element is ever accessible — this restriction is the entire point. It naturally models "undo the most recent thing" and "the last unmatched thing" scenarios, and it's literally how function calls work under the hood (the call stack).

Real-life example: A stack of plates in a cafeteria — you always take the top plate, and you always add a clean plate to the top. You cannot pull a plate from the middle without first removing

everything above it.

Diagram:

```
push(1) push(2) push(3):        pop():
   | 3 | <- top                    | 2 | <- top (3 removed)
   | 2 |                           | 1 |
   | 1 |
```

Java code:

```java
import java.util.ArrayDeque;
import java.util.Deque;

Deque<Integer> stack = new ArrayDeque<>();  // Java's recommended Stack implementation
stack.push(1);
stack.push(2);
int top = stack.pop();   // 2, LIFO -- O(1)
```

Brute-force vs optimized — "next greater element" for every element in an array. Input: `[2, 1, 2, 4, 3]`.

Brute-force — for each element, scan rightward, O(n²):

```java
static int[] nextGreaterBrute(int[] arr) {
    int[] result = new int[arr.length];
    for (int i = 0; i < arr.length; i++) {
        result[i] = -1;
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] > arr[i]) { result[i] = arr[j]; break; }
        }
    }
    return result;
}
```

Optimized — monotonic decreasing stack, O(n):

```java
static int[] nextGreaterOptimized(int[] arr) {
    int[] result = new int[arr.length];
    java.util.Arrays.fill(result, -1);
    Deque<Integer> stack = new ArrayDeque<>();  // stores indices
    for (int i = 0; i < arr.length; i++) {
        while (!stack.isEmpty() && arr[stack.peek()] < arr[i]) {
            result[stack.pop()] = arr[i];
        }
        stack.push(i);
    }
    return result;
}
```

Dry run of the optimized version — `arr = [2, 1, 2, 4, 3]`:

```
i=0(2): stack empty -> push 0. stack=[0]
i=1(1): arr[0]=2 not < 1 -> push 1. stack=[0,1]
i=2(2): arr[1]=1 < 2 -> pop 1, result[1]=2. arr[0]=2 not < 2 -> push 2. stack=[0,2]
i=3(4): arr[2]=2 < 4 -> pop 2, result[2]=4. arr[0]=2 < 4 -> pop 0, result[0]=4. push 3. stack=[3]
i=4(3): arr[3]=4 not < 3 -> push 4. stack=[3,4]
End: indices left in stack (3,4) get result=-1 (already initialized)
Result: [4, 2, 4, -1, -1]
```

Complexity summary: | Operation | Complexity | |—|—| | push | O(1) | | pop | O(1) | | peek | O(1) | | Naive “next greater” (brute force) | $O(n^2)$ | | Monotonic stack “next greater” | O(n) |

Common mistakes: - Using Java’s legacy `Stack` class (synchronized, slower) instead of `ArrayDeque` — `ArrayDeque` is the recommended modern choice. - Forgetting to check `stack.isEmpty()` before `peek()`/`pop()` — throws `NoSuchElementException` or `EmptyStackException`. - Confusing what a monotonic stack maintains — decreasing stack for “next greater,” increasing stack for “next smaller.” Mixing these up gives silently wrong answers. - In balanced-parentheses problems, forgetting to check the stack is empty at the very end (unmatched opening brackets left over).

Interview tips: - Say “monotonic stack” explicitly the moment you see “next greater/smaller element” — it’s a strong, recognizable signal of pattern fluency. - Stacks are the standard way to convert recursive solutions into iterative ones (simulate the call stack manually) — mention this if asked to remove recursion. - For expression evaluation problems, mention the two-stack approach (one for operators, one for operands) or postfix conversion.

Practice questions with solutions:

Q1: Check if a string of brackets is balanced. Input: `"{[()]}"`.

```
static boolean isBalanced(String s) {
    Deque<Character> stack = new ArrayDeque<>();
    java.util.Map<Character, Character> pairs = java.util.Map.of(')', '(', ']', '[', '}', '{');
    for (char c : s.toCharArray()) {
        if (c == '(' || c == '[' || c == '{') stack.push(c);
        else if (pairs.containsKey(c)) {
            if (stack.isEmpty() || stack.pop() != pairs.get(c)) return false;
        }
    }
    return stack.isEmpty();
}
```

Solution walkthrough: Push every opening bracket; on a closing bracket, it must match the top of the stack. `"{[()]}"` matches perfectly at every step → **true**. Time: O(n), Space: O(n).

*Q2: Evaluate a postfix expression. Input: `"2 1 + 3 *"` (means (2+1)3).**

```
static int evalPostfix(String expr) {
    Deque<Integer> stack = new ArrayDeque<>();
    for (String token : expr.split(" ")) {
        if (token.matches("-?\\d+")) stack.push(Integer.parseInt(token));
        else {
            int b = stack.pop(), a = stack.pop();
            switch (token) {
                case "+": stack.push(a + b); break;
                case "-": stack.push(a - b); break;
                case "*": stack.push(a * b); break;
                case "/": stack.push(a / b); break;
            }
        }
    }
    return stack.pop();
}
```

Solution walkthrough: Push numbers; on an operator, pop two, apply, push result back. `2,1 -> push,push. + -> pop 1,pop 2 -> push 3. 3 -> push. * -> pop 3,pop 3 -> push 9.` Result: **9**. Time: O(n), Space: O(n).

Q3: Implement a min-stack that supports push, pop, and getMin, all in O(1).

```java
class MinStack {
    Deque<Integer> stack = new ArrayDeque<>();
    Deque<Integer> minStack = new ArrayDeque<>();

    void push(int val) {
        stack.push(val);
        minStack.push(minStack.isEmpty() ? val : Math.min(val, minStack.peek()));
    }
    void pop() { stack.pop(); minStack.pop(); }
    int top() { return stack.peek(); }
    int getMin() { return minStack.peek(); }
}
```

Solution walkthrough: A second stack tracks the running minimum in parallel — each push records what the min would be *at that point*, so popping automatically "restores" the previous min. All operations O(1) time, O(n) space.

---

7. Queues

Definition: A First-In-First-Out (FIFO) structure. `enqueue` adds to the back, `dequeue` removes from the front, both O(1) when implemented correctly (circular buffer or linked list with head/tail pointers).

Intuition: Order of arrival is preserved and respected — the first thing in is the first thing processed. This is why BFS (breadth-first search) fundamentally requires a queue: it must finish exploring everything at the current "distance" before moving further out, and a queue naturally enforces that ordering.

Real-life example: A checkout line at a store — the first person in line is served first; new arrivals join at the back.

Diagram:

```
enqueue(1) enqueue(2) enqueue(3):    dequeue():
front -> [1][2][3] <- back              front -> [2][3] <- back  (1 removed)
```

Java code:

```java
import java.util.ArrayDeque;
import java.util.Queue;

Queue<Integer> queue = new ArrayDeque<>();
queue.offer(1);     // enqueue
queue.offer(2);
int front = queue.poll();  // dequeue -> 1, O(1)
```

Brute-force vs optimized — implementing a queue.

Brute-force — array where dequeue shifts everything left, O(n) per dequeue:

```java
// Conceptual: removing arr[0] and shifting arr[1..n-1] left by one is O(n)
```

Optimized — circular buffer, O(1) per operation:

```java
class CircularQueue {
    int[] data;
    int front = 0, size = 0, capacity;
```

```
    CircularQueue(int capacity) {
        this.capacity = capacity;
        data = new int[capacity];
    }

    boolean enqueue(int val) {
        if (size == capacity) return false;          // full
        int rear = (front + size) % capacity;         // wrap around
        data[rear] = val;
        size++;
        return true;
    }

    int dequeue() {
        if (size == 0) throw new RuntimeException("empty");
        int val = data[front];
        front = (front + 1) % capacity;               // wrap around
        size--;
        return val;
    }
}
```

Dry run — `CircularQueue(3)`: `enqueue(1)`, `enqueue(2)`, `dequeue()`, `enqueue(3)`, `enqueue(4)`:

```
enqueue(1): rear=(0+0)%3=0, data[0]=1, size=1, front=0
enqueue(2): rear=(0+1)%3=1, data[1]=2, size=2, front=0
dequeue():  val=data[0]=1, front=(0+1)%3=1, size=1  -> returns 1
enqueue(3): rear=(1+1)%3=2, data[2]=3, size=2, front=1
enqueue(4): rear=(1+2)%3=0, data[0]=4 (overwrites old 1, which is fine), size=3, front=1
Queue now logically holds: 2, 3, 4 (front=1 -> data[1]=2, data[2]=3, data[0]=4)
```

Complexity summary: | Operation | Naive array (shift on dequeue) | Circular buffer / linked list | |—|—|—| | enqueue | O(1) | O(1) | | dequeue | O(n) | O(1) |

Common mistakes: - Implementing a queue with a plain array and dequeuing from index 0 without a circular buffer — silently O(n) per dequeue, a common performance bug. - Off-by-one errors in the modulo wraparound logic for circular queues. - Confusing `Queue` (FIFO) with `Deque` (double-ended) — a `Deque` can act as either a stack or a queue depending on which end you use. - Forgetting that `Queue.poll()` returns `null` on empty (no exception) while `.remove()` throws — using the wrong one silently masks bugs.

Interview tips: - The moment a problem says “shortest path” or “level order” or “process in order of arrival,” reach for a queue. - Mention `ArrayDeque` over Java’s legacy `LinkedList`-as-queue for performance — `ArrayDeque` has better cache locality and no per-node allocation overhead. - For priority-based processing (not pure FIFO), the right structure is a `PriorityQueue` (a heap), not a plain queue — know the distinction.

Practice questions with solutions:

Q1: Implement a stack using two queues.

```
class StackUsingQueues {
    Queue<Integer> q1 = new ArrayDeque<>(), q2 = new ArrayDeque<>();

    void push(int x) {
        q2.offer(x);
        while (!q1.isEmpty()) q2.offer(q1.poll());
```

```
        Queue<Integer> temp = q1; q1 = q2; q2 = temp;   // swap references
    }
    int pop() { return q1.poll(); }
    int top() { return q1.peek(); }
}
```

Solution walkthrough: Every push puts the new element in q2 first, then drains q1 behind it — so the newest element always ends up at the front of q1, making `pop`/`top` behave like a stack. Push is O(n), pop/top are O(1).

Q2: Perform a level-order traversal of a binary tree using a queue. Input: tree `[3,9,20,null,null,15,7]`.

```
static java.util.List<java.util.List<Integer>> levelOrder(TreeNode root) {
    java.util.List<java.util.List<Integer>> result = new java.util.ArrayList<>();
    if (root == null) return result;
    Queue<TreeNode> q = new ArrayDeque<>();
    q.offer(root);
    while (!q.isEmpty()) {
        int levelSize = q.size();
        java.util.List<Integer> level = new java.util.ArrayList<>();
        for (int i = 0; i < levelSize; i++) {
            TreeNode node = q.poll();
            level.add(node.val);
            if (node.left != null) q.offer(node.left);
            if (node.right != null) q.offer(node.right);
        }
        result.add(level);
    }
    return result;
}
```

Solution walkthrough: Capturing `q.size()` before the inner loop is the key trick — it freezes "how many nodes are in the current level" before you start adding next-level nodes into the same queue. Result: `[[3],[9,20],[15,7]]`. Time: O(n), Space: O(n).

Q3: Design a circular queue with `enqueue`, `dequeue`, `isFull`, `isEmpty` (see CircularQueue code above) — trace `isFull` and `isEmpty` logic. **Solution:** `isEmpty()` returns `size == 0`; `isFull()` returns `size == capacity`. Both O(1) because `size` is tracked explicitly rather than recomputed from front/rear positions (which is ambiguous in a circular buffer when front==rear could mean either empty or full).

---

8. Recursion

Definition: A function that calls itself on a smaller version of the problem, stopping at a base case.

Intuition: Every call is pushed onto the call stack with its own local variables; when the base case is hit, calls return and pop off in reverse order. Recursion *is* a stack — just one the language runtime manages implicitly instead of one you manage explicitly.

Real-life example: Russian nesting dolls — to get to the smallest doll, you open the current one and repeat the same "open the doll" action on what's inside, until you reach the doll that doesn't open (the base case). Then you close them back up in reverse order.

Diagram — call stack for `factorial(4)`:

```
factorial(4) calls factorial(3) calls factorial(2) calls factorial(1) [base case, returns 1]
```

```
                                                    factorial(2) returns 2*1=2
                                          factorial(3) returns 3*2=6
factorial(4) returns 4*6=24

Stack grows downward then unwinds:
| factorial(1) |  <- top, returns first
| factorial(2) |
| factorial(3) |
| factorial(4) |  <- bottom, returns last
```

Java code:

```java
static int factorial(int n) {
    if (n <= 1) return 1;              // base case
    return n * factorial(n - 1);       // recursive case
}
```

Dry run — `factorial(4)`:

```
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1                (base case hit)
Unwinding: factorial(2)=2*1=2, factorial(3)=3*2=6, factorial(4)=4*6=24
```

Brute-force vs optimized — Fibonacci:

Brute-force — naive recursion, recomputes overlapping subproblems, O(2ⁿ):

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

Optimized — memoized recursion, O(n):

```java
static int fibMemo(int n, int[] memo) {
    if (n <= 1) return n;
    if (memo[n] != 0) return memo[n];
    memo[n] = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);
    return memo[n];
}
```

Dry run of `fibNaive(5)` call tree (showing the redundant recomputation that memoization fixes):

```
                fib(5)
              /        \
          fib(4)        fib(3)
         /     \        /     \
     fib(3)  fib(2)  fib(2)  fib(1)
     /   \    /  \    /  \
  fib(2) fib(1) ...  ...
```

```
fib(3) is computed twice, fib(2) is computed three times — this redundancy
compounds exponentially as n grows. Memoization caches each result the first
time it's computed, so each distinct call happens only once.
```

Complexity summary: | Version | Time | Space | |—|—|—| | fibNaive | O(2ⁿ) | O(n) (call stack depth) | | fibMemo | O(n) | O(n) (cache + call stack) |

Common mistakes: - Missing or incorrect base case → infinite recursion → `StackOverflowError`. - Doing redundant work by not caching overlapping subproblems (the exact motivating example for dynamic programming, section 23). - Deep linear recursion on large inputs (e.g., recursing over a 100,000-element list) risking stack overflow — Java does not optimize tail calls, so very deep recursion should be converted to iteration. - Forgetting that each recursive call has its own copy of local variables — mutating a shared/static variable across calls instead can produce confusing bugs.

Interview tips: - State the base case and recursive case explicitly before writing code — it's the fastest way to show structured thinking. - If asked to convert a recursive solution to iterative, mention using an explicit stack to simulate the call stack. - When recursion has overlapping subproblems, name memoization/DP as the next step even before being asked — it shows you see the redundant work.

Practice questions with solutions:

Q1: Compute the sum of digits of a number recursively. Input: 1234.

```
static int sumDigits(int n) {
    if (n == 0) return 0;
    return n % 10 + sumDigits(n / 10);
}
```

Solution walkthrough: `1234 -> 4 + sumDigits(123) -> 4 + (3 + sumDigits(12)) -> ... -> 4+3+2+1 = 10`. Time: O(digits), Space: O(digits).

Q2: Generate all subsets of a set recursively. Input: [1,2,3].

```
static void subsets(int[] nums, int index, java.util.List<Integer> current, java.util.List<java.
    if (index == nums.length) {
        result.add(new java.util.ArrayList<>(current));
        return;
    }
    subsets(nums, index + 1, current, result);            // exclude nums[index]
    current.add(nums[index]);
    subsets(nums, index + 1, current, result);            // include nums[index]
    current.remove(current.size() - 1);                    // backtrack
}
```

Solution walkthrough: At each element, branch into "exclude" and "include." For `[1,2,3]`, produces all $2^3=8$ subsets: `[]`, `[3]`, `[2]`, `[2,3]`, `[1]`, `[1,3]`, `[1,2]`, `[1,2,3]`. Time: $O(2^n)$, Space: O(n) recursion depth + $O(2^n)$ output.

Q3: Implement power function x^n recursively in O(log n) instead of O(n).

```
static double myPow(double x, int n) {
    if (n == 0) return 1;
    if (n < 0) return 1 / myPow(x, -n);
    double half = myPow(x, n / 2);
    return (n % 2 == 0) ? half * half : half * half * x;
}
```

Solution walkthrough: Halving n each call (instead of subtracting 1) gives O(log n) calls instead of O(n). `myPow(2,10) = myPow(2,5)^2 = (myPow(2,2)*2)^2 = ((myPow(2,1))^2 * 2)^2 = ...` → **1024**. Time: O(log n), Space: O(log n). ## 9. Sorting Algorithms

Definition: Rearranging elements of a collection into a defined order (ascending/descending).

Intuition: Comparison-based sorts can never beat O(n log n) in the worst case — a mathematical fact, not an engineering limitation. There are n! possible orderings; each comparison yields at most 1 bit of information; so you need at least $\log_2(n!) \approx n \log n$ comparisons to distinguish

between all possible orderings. This is *why* merge sort and heap sort at O(n log n) are considered asymptotically optimal for comparison-based sorting.

Real-life example: Sorting a hand of playing cards. Insertion sort mirrors how most people physically sort cards — pick up one at a time, insert it into its correct position among the cards already sorted in your hand.

Diagram — merge sort's divide-and-conquer structure:

```
[8, 3, 5, 1]
   /        \
[8, 3]      [5, 1]
 /  \        /  \
[8] [3]    [5]  [1]
 \  /        \  /
[3, 8]      [1, 5]      <- merge sorted halves
    \        /
   [1, 3, 5, 8]          <- final merge
```

Java code — Insertion Sort (good for small/nearly-sorted arrays):

```java
static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];      // shift larger element right
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Dry run — `insertionSort([5, 2, 4, 6, 1])`:

```
i=1, key=2: shift 5 right -> [5,5,4,6,1] -> place 2 -> [2,5,4,6,1]
i=2, key=4: shift 5 right -> [2,5,5,6,1] -> place 4 -> [2,4,5,6,1]
i=3, key=6: 5<6, no shift -> [2,4,5,6,1]
i=4, key=1: shift 6,5,4,2 right -> place 1 -> [1,2,4,5,6]
```

Java code — Merge Sort:

```java
static void mergeSort(int[] arr, int left, int right) {
    if (left >= right) return;
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

static void merge(int[] arr, int left, int mid, int right) {
    int[] temp = new int[right - left + 1];
    int i = left, j = mid + 1, k = 0;
    while (i <= mid && j <= right)
        temp[k++] = (arr[i] <= arr[j]) ? arr[i++] : arr[j++];
    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];
    System.arraycopy(temp, 0, arr, left, temp.length);
}
```

Java code — Quick Sort:

```java
static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            int t = arr[i]; arr[i] = arr[j]; arr[j] = t;
        }
    }
    int t = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = t;
    return i + 1;
}
```

Dry run — `partition([3, 6, 2, 8, 5], low=0, high=4)`, pivot=5:

```
i=-1
j=0(3): 3<=5 -> i=0, swap arr[0],arr[0] -> [3,6,2,8,5]
j=1(6): 6>5 -> skip
j=2(2): 2<=5 -> i=1, swap arr[1],arr[2] -> [3,2,6,8,5]
j=3(8): 8>5 -> skip
Loop ends. swap arr[i+1]=arr[2] with arr[high]=arr[4] -> [3,2,5,8,6]
Return pivotIndex=2. Left partition [3,2] all <=5, right partition [8,6] all >5.
```

Brute-force vs optimized — full comparison table: | Algorithm | Best | Average | Worst | Space | Stable? | |—|—|—|—|—|—| | Bubble Sort | O(n) | $O(n^2)$ | $O(n^2)$ | O(1) | Yes | | Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | O(1) | No | | Insertion Sort | O(n) | $O(n^2)$ | $O(n^2)$ | O(1) | Yes | | Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n) | Yes | | Quick Sort | O(n log n) | O(n log n) | $O(n^2)$ | O(log n) | No | | Heap Sort | O(n log n) | O(n log n) | O(n log n) | O(1) | No | | Counting Sort | O(n+k) | O(n+k) | O(n+k) | O(k) | Yes |

“Stable” = equal elements retain their original relative order — matters when sorting objects by one key while preserving insertion order among ties.

Why quicksort’s worst case is $O(n^2)$: if the pivot is always the smallest or largest remaining element (e.g., already-sorted input with a naive “last element” pivot), each partition only removes one element, giving n recursive levels each doing O(n) work → $O(n^2)$. Randomized pivot selection makes this pathological case exponentially unlikely in practice.

Common mistakes: - Choosing a fixed pivot (like the last element) on data that might already be sorted or reverse-sorted — triggers quicksort’s $O(n^2)$ worst case; use a randomized or median-of-three pivot instead. - Forgetting merge sort needs O(n) auxiliary space — claiming it’s in-place is a factual error interviewers catch immediately. - Off-by-one errors in partition boundaries (`low`, `high`, `mid`) — always trace a 2-3 element array by hand to verify. - Assuming all sorts are stable — selection sort and quicksort are not; this matters when sorting complex objects by a secondary key.

Interview tips: - Know *why* O(n log n) is the comparison-sort floor (the n! argument) — interviewers sometimes ask this directly. - Default recommendation: quicksort for general-purpose (best average case, in-place), merge sort when stability or guaranteed worst-case matters, count-

ing/radix sort when keys are small-range integers. - Mention that Java's `Arrays.sort()` uses dual-pivot quicksort for primitives and Timsort (merge+insertion hybrid) for objects — knowing this shows depth beyond memorized pseudocode.

Practice questions with solutions:

Q1: Sort an array of 0s, 1s, and 2s in one pass (Dutch National Flag problem). Input: `[2,0,2,1,1,0]`.

```
static void sortColors(int[] nums) {
    int low = 0, mid = 0, high = nums.length - 1;
    while (mid <= high) {
        if (nums[mid] == 0) { swap(nums, low++, mid++); }
        else if (nums[mid] == 1) { mid++; }
        else { swap(nums, mid, high--); }
    }
}
static void swap(int[] a, int i, int j) { int t = a[i]; a[i] = a[j]; a[j] = t; }
```

Solution walkthrough: Three pointers partition the array into [0s | 1s | unprocessed | 2s] in a single pass. Result: `[0,0,1,1,2,2]`. Time: O(n), Space: O(1) — beats sorting generically (O(n log n)) by exploiting the constrained value range (just like counting sort).

Q2: Given an almost-sorted array (each element is at most k positions from its sorted position), sort it efficiently. **Solution:** Use a min-heap of size k+1. Insert the first k+1 elements, then repeatedly extract-min and insert the next element. Each extraction/insertion is O(log k), total O(n log k) — much better than O(n log n) generic sorting when k is small. This demonstrates exploiting problem structure (bounded displacement) rather than treating it as a fully general sort.

Q3: Merge two sorted arrays in place, where the first array has enough trailing space. Input: `nums1=[1,2,3,0,0,0], m=3, nums2=[2,5,6], n=3`.

```
static void merge(int[] nums1, int m, int[] nums2, int n) {
    int i = m - 1, j = n - 1, k = m + n - 1;
    while (j >= 0) {
        if (i >= 0 && nums1[i] > nums2[j]) nums1[k--] = nums1[i--];
        else nums1[k--] = nums2[j--];
    }
}
```

Solution walkthrough: Fill from the back to avoid overwriting unprocessed elements in `nums1`. Result: `[1,2,2,3,5,6]`. Time: O(m+n), Space: O(1).

---

10. Searching Algorithms

Definition: Locating a target value within a collection.

Intuition: Linear search has no assumptions about the data and costs O(n). Binary search assumes sorted data and exploits that assumption to eliminate half the remaining candidates with each comparison, giving O(log n) — the exact same halving logic that makes merge sort's height O(log n).

Real-life example: Finding a word in a printed dictionary — you don't start at page 1; you open to the middle, decide if your word is alphabetically before or after, and repeat on the correct half.

Diagram — binary search narrowing on `target=23` in `[2,5,8,12,16,23,38,56,72,91]`:

```
[2,5,8,12,16,23,38,56,72,91]  low=0,high=9,mid=4(16): 16<23 -> low=5
```

```
            [23,38,56,72,91] low=5,high=9,mid=7(56): 56>23 -> high=6
            [23,38]          low=5,high=6,mid=5(23): 23==23 -> FOUND at index 5
```

Java code — Binary Search:

```java
static int binarySearch(int[] arr, int target) {
    int low = 0, high = arr.length - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;   // avoids integer overflow vs (low+high)/2
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

Brute-force vs optimized — find target in a sorted array:

Brute-force — Linear Search, O(n):

```java
static int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++)
        if (arr[i] == target) return i;
    return -1;
}
```

Optimized — Binary Search, O(log n) (code above).

Complexity summary: | Algorithm | Time | Space | Requires | |—|—|—|—| | Linear Search | O(n) | O(1) | Nothing | | Binary Search | O(log n) | O(1) iterative / O(log n) recursive | Sorted data | | Binary search on the answer | O(log(range) × check cost) | Varies | Monotonic condition |

Binary search on the answer — an underused pattern. When a problem asks “find the minimum X such that condition(X) is true” and `condition` is monotonic, binary search over the *answer space* itself, not an array. Example: “minimum eating speed to finish all bananas within h hours” — binary search over possible speeds, checking feasibility of each in O(n).

Common mistakes: - `mid = (low + high) / 2` can overflow for very large indices in some languages; `low + (high - low) / 2` avoids it (matters less in Java’s int range for typical problems, but shows awareness). - Infinite loops from updating `low`/`high` incorrectly (e.g., setting `high = mid` instead of `high = mid - 1` when you meant to exclude `mid`). - Applying binary search to unsorted data — silently wrong answer, no error thrown. - Off-by-one when searching for “first/last occurrence” of a duplicate value — requires continuing the search after finding a match instead of returning immediately.

Interview tips: - Always verify the data is sorted before proposing binary search — state this assumption out loud. - For “find first/last occurrence” or “find insertion point” variants, adjust the standard template rather than treating it as a brand new algorithm — interviewers want to see you recognize it’s the same core idea. - Recognize “binary search on the answer” whenever a problem says “minimum/maximum X such that a condition holds” over a large range — it’s a frequently-missed pattern.

Practice questions with solutions:

Q1: Find the first and last position of a target in a sorted array with duplicates. Input: `[5,7,7,8,8,10]`, target=8.

```java
static int[] searchRange(int[] nums, int target) {
    return new int[]{findBound(nums, target, true), findBound(nums, target, false)};
}
static int findBound(int[] nums, int target, boolean findFirst) {
```

```
    int low = 0, high = nums.length - 1, result = -1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (nums[mid] == target) {
            result = mid;
            if (findFirst) high = mid - 1;   // keep searching left
            else low = mid + 1;               // keep searching right
        } else if (nums[mid] < target) low = mid + 1;
        else high = mid - 1;
    }
    return result;
}
```

Solution walkthrough: Instead of stopping at the first match, keep narrowing in the direction of the first/last occurrence. Result: `[3, 4]` (indices of the two 8s). Time: O(log n), Space: O(1).

Q2: Search in a rotated sorted array. Input: `[4,5,6,7,0,1,2]`, target=0.

```
static int search(int[] nums, int target) {
    int low = 0, high = nums.length - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (nums[mid] == target) return mid;
        if (nums[low] <= nums[mid]) {           // left half is sorted
            if (nums[low] <= target && target < nums[mid]) high = mid - 1;
            else low = mid + 1;
        } else {                                  // right half is sorted
            if (nums[mid] < target && target <= nums[high]) low = mid + 1;
            else high = mid - 1;
        }
    }
    return -1;
}
```

Solution walkthrough: At each step, one half is guaranteed sorted; check if the target lies within that sorted half's range to decide which half to discard. Finds 0 at index **4**. Time: O(log n), Space: O(1).

Q3: Find the peak element in an array (an element greater than its neighbors). Input: `[1,2,3,1]`.

```
static int findPeakElement(int[] nums) {
    int low = 0, high = nums.length - 1;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (nums[mid] > nums[mid + 1]) high = mid;   // peak is at mid or to the left
        else low = mid + 1;                            // peak is to the right
    }
    return low;
}
```

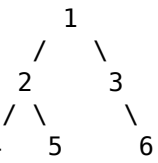
Solution walkthrough: This is "binary search on the answer" — the array isn't sorted, but the condition "am I on the ascending or descending side" is well-defined at every point, so binary search still applies. Returns index **2** (value 3). Time: O(log n), Space: O(1). ## 11. Trees (Binary Trees & Traversals)

Definition: A hierarchical structure of nodes where each node has a value and pointers to child nodes, starting from one root, with no cycles. A binary tree restricts each node to at most 2 children.

Intuition: Trees represent hierarchy and let you eliminate large portions of data at each level of depth — this is what makes balanced trees give O(log n) operations rather than the O(n) of a flat list. Traversal order (preorder/inorder/postorder/level-order) determines *when* you process a node relative to its children — pick the order based on what you need to compute.

Real-life example: A company org chart — the CEO is the root, each manager has employees under them (children), and there's exactly one path from the CEO down to any individual employee.

Diagram:




```
Preorder  (root,left,right): 1,2,4,5,3,6
Inorder   (left,root,right): 4,2,5,1,3,6
Postorder (left,right,root): 4,5,2,6,3,1
Level-order (BFS by depth):  1,2,3,4,5,6
```

Java code:

```
class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int val) { this.val = val; }
}

static void preorder(TreeNode node, java.util.List<Integer> result) {
    if (node == null) return;
    result.add(node.val);
    preorder(node.left, result);
    preorder(node.right, result);
}

static void inorder(TreeNode node, java.util.List<Integer> result) {
    if (node == null) return;
    inorder(node.left, result);
    result.add(node.val);
    inorder(node.right, result);
}

static void postorder(TreeNode node, java.util.List<Integer> result) {
    if (node == null) return;
    postorder(node.left, result);
    postorder(node.right, result);
    result.add(node.val);
}

static java.util.List<Integer> levelOrder(TreeNode root) {
    java.util.List<Integer> result = new java.util.ArrayList<>();
    if (root == null) return result;
    java.util.Queue<TreeNode> q = new java.util.ArrayDeque<>();
    q.offer(root);
    while (!q.isEmpty()) {
```

```
        TreeNode node = q.poll();
        result.add(node.val);
        if (node.left != null) q.offer(node.left);
        if (node.right != null) q.offer(node.right);
    }
    return result;
}
```

Dry run — inorder traversal of the tree in the diagram above, using the call stack:

```
inorder(1): inorder(2) first
  inorder(2): inorder(4) first
    inorder(4): inorder(null)->skip; add 4; inorder(null)->skip. returns.
  add 2
    inorder(5): inorder(null)->skip; add 5; inorder(null)->skip. returns.
add 1
  inorder(3): inorder(null)->skip; add 3
    inorder(6): inorder(null)->skip; add 6; inorder(null)->skip.
Result: [4, 2, 5, 1, 3, 6]
```

Brute-force vs optimized — compute tree height:

Naive/recursive (this is actually already the standard, no brute-force alternative exists that's simpler — but contrast recursive vs iterative approaches):

```
static int heightRecursive(TreeNode node) {
    if (node == null) return -1;
    return 1 + Math.max(heightRecursive(node.left), heightRecursive(node.right));
}
```

Iterative — using level-order BFS to count levels, avoids recursion stack:

```
static int heightIterative(TreeNode root) {
    if (root == null) return -1;
    java.util.Queue<TreeNode> q = new java.util.ArrayDeque<>();
    q.offer(root);
    int height = -1;
    while (!q.isEmpty()) {
        int size = q.size();
        height++;
        for (int i = 0; i < size; i++) {
            TreeNode node = q.poll();
            if (node.left != null) q.offer(node.left);
            if (node.right != null) q.offer(node.right);
        }
    }
    return height;
}
```

Complexity summary: | Operation | Time | Space | |—|—|—| | Any traversal (pre/in/post/level) | O(n) | O(h) recursive stack (or O(w) queue for level-order, w = max width) | | Height calculation | O(n) | O(h) |

Common mistakes: - Confusing preorder/inorder/postorder — remember: the position of "root" in the name tells you when it's visited relative to children (pre=first, in=middle, post=last). - Forgetting the `null` base case, causing a `NullPointerException`. - For level-order, forgetting to capture `q.size()` before the inner loop, which merges levels together instead of separating them. - Assuming inorder traversal gives sorted order for *any* binary tree — this is only true for

a **binary search tree** (section 12).

Interview tips: - State which traversal you're using and *why* — inorder for BST validation/sorted output, postorder when children must be processed before the parent (deletion, computing subtree aggregates), preorder for serialization/copying. - Level-order (BFS) is the right call whenever a problem is phrased in terms of "levels" or "closest nodes." - Practice both recursive and iterative versions of each traversal — iterative versions (using an explicit stack) are a common follow-up question.

Practice questions with solutions:

Q1: Check if two binary trees are identical.

```
static boolean isSameTree(TreeNode p, TreeNode q) {
    if (p == null && q == null) return true;
    if (p == null || q == null || p.val != q.val) return false;
    return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}
```

Solution walkthrough: Recursively compare values and both subtrees; any structural or value mismatch returns false immediately. Time: O(n), Space: O(h).

Q2: Find the diameter of a binary tree (longest path between any two nodes, measured in edges).

```
static int diameter = 0;
static int diameterOfBinaryTree(TreeNode root) {
    height(root);
    return diameter;
}
static int height(TreeNode node) {
    if (node == null) return 0;
    int left = height(node.left);
    int right = height(node.right);
    diameter = Math.max(diameter, left + right);   // path through this node
    return 1 + Math.max(left, right);
}
```

Solution walkthrough: For every node, the longest path *through* it equals left height + right height; track the max across all nodes while computing height in a single postorder pass. Time: O(n), Space: O(h).

Q3: Serialize and deserialize a binary tree (convert to a string and back).

```
static String serialize(TreeNode root) {
    if (root == null) return "null";
    return root.val + "," + serialize(root.left) + "," + serialize(root.right);
}

static int index;
static TreeNode deserialize(String[] data) {
    if (data[index].equals("null")) { index++; return null; }
    TreeNode node = new TreeNode(Integer.parseInt(data[index++]));
    node.left = deserialize(data);
    node.right = deserialize(data);
    return node;
}
```

Solution walkthrough: Preorder traversal naturally encodes enough structure to reconstruct the tree exactly, since it records "root, then everything under left, then everything under right" — deserializing just replays that order. Time: O(n), Space: O(n).

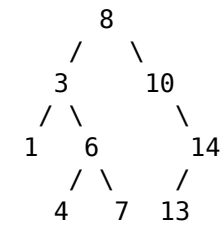
12. Binary Search Trees & Balanced Trees

Definition: A binary tree where, for every node, all values in the left subtree are smaller and all values in the right subtree are larger.

Intuition: This single ordering invariant is what makes search O(log n) *on average* — at each node you can eliminate an entire subtree, exactly like binary search on a sorted array. Inorder traversal of a BST always yields sorted output, directly because of this invariant.

Real-life example: A well-organized filing cabinet where folders are ordered alphabetically at every branch — to find "Miller," you check the current folder's label and go left (earlier alphabetically) or right (later), never having to check folders you've already ruled out.

Diagram:




```
Inorder traversal gives sorted order: 1,3,4,6,7,8,10,13,14
```

Java code — insert, search:

```java
static TreeNode insert(TreeNode node, int val) {
    if (node == null) return new TreeNode(val);
    if (val < node.val) node.left = insert(node.left, val);
    else node.right = insert(node.right, val);
    return node;
}

static boolean search(TreeNode node, int val) {
    if (node == null) return false;
    if (node.val == val) return true;
    return val < node.val ? search(node.left, val) : search(node.right, val);
}
```

Dry run — inserting 5 into the tree above:

```
insert(8,5): 5<8 -> go left to 3
insert(3,5): 5>3 -> go right to 6
insert(6,5): 5<6 -> go left to 4
insert(4,5): 5>4 -> go right, which is null -> attach new node(5) here
Result: 4 now has a right child 5.
```

Java code — delete (the trickiest BST operation, 3 cases):

```java
static TreeNode delete(TreeNode node, int val) {
    if (node == null) return null;
    if (val < node.val) node.left = delete(node.left, val);
    else if (val > node.val) node.right = delete(node.right, val);
    else {
        // Found the node to delete
        if (node.left == null) return node.right;    // 0 or 1 child (right)
        if (node.right == null) return node.left;     // 1 child (left)
```

```
        // 2 children: replace with inorder successor (smallest in right subtree)
        TreeNode successor = node.right;
        while (successor.left != null) successor = successor.left;
        node.val = successor.val;
        node.right = delete(node.right, successor.val);   // remove the successor
    }
    return node;
}
```

Brute-force vs optimized — searching for a value:

Brute-force — treat as unordered, linear scan, O(n): ignores the BST property entirely.

Optimized — exploit the BST invariant, O(log n) average (code above) — but note **O(n) worst case** if the tree degenerates (e.g., inserting already-sorted data into a plain BST makes it a linked list — every node has only a right child).

Complexity summary: | Operation | Average (balanced) | Worst Case (degenerate) | |—|—|—| | Search | O(log n) | O(n) | | Insert | O(log n) | O(n) | | Delete | O(log n) | O(n) |

Self-balancing trees (guarantee O(log n) worst case): - **AVL Tree** — tracks a balance factor (height difference between subtrees) at each node; rotates when it exceeds 1. Strictly balanced, faster reads, more rotation overhead on writes. - **Red-Black Tree** — colors nodes red/black with rules that bound height at 2·log(n+1). Looser balance than AVL → fewer rotations, slightly slower reads. This is what backs Java's `TreeMap`/`TreeSet`.

Common mistakes: - Forgetting the "2 children" delete case requires finding the inorder successor (or predecessor) — deleting incorrectly can violate the BST property. - Assuming BST operations are always O(log n) — they're only O(log n) *average*, and only guaranteed for self-balancing variants. - Confusing BST validation with just checking `node.left.val < node.val < node.right.val` locally — this misses violations further down the tree (a node deep in the left subtree could still exceed the root's value). Correct validation passes down a valid (min, max) range. - Inserting duplicate values without a defined convention (go left or right on equal) — causes inconsistent behavior.

Interview tips: - Mention `TreeMap`/`TreeSet` (Java's Red-Black tree implementations) when a problem needs ordered operations (range queries, floor/ceiling, "next largest") — this signals you know when to reach for a library structure over a hash map. - For BST validation, explicitly state you're passing down a valid range rather than doing a local 3-node check — this is the most common BST interview trap. - Know when to use a hash table instead: if you don't need ordering, a hash table gives O(1) average vs a BST's O(log n) — don't default to BST just because it's more complex-sounding.

Practice questions with solutions:

Q1: Validate whether a binary tree is a valid BST. Input: `[5,1,4,null,null,3,6]` (not valid — 3 is less than 5 but in the right subtree).

```
static boolean isValidBST(TreeNode node, Long min, Long max) {
    if (node == null) return true;
    if ((min != null && node.val <= min) || (max != null && node.val >= max)) return false;
    return isValidBST(node.left, min, (long) node.val) && isValidBST(node.right, (long) node.val
}
```

Solution walkthrough: Each recursive call carries the valid (min, max) range inherited from all ancestors, not just the immediate parent. For the given tree, node 3 is checked against range `(5, null)` inherited from root 5 — since 3 ≤ 5, returns **false**. Time: O(n), Space: O(h).

Q2: Find the Lowest Common Ancestor (LCA) of two nodes in a BST. Input: BST from diagram above, nodes 4 and 7.

```java
static TreeNode lowestCommonAncestor(TreeNode root, int p, int q) {
    if (p < root.val && q < root.val) return lowestCommonAncestor(root.left, p, q);
    if (p > root.val && q > root.val) return lowestCommonAncestor(root.right, p, q);
    return root;   // p and q split here, or one equals root
}
```

Solution walkthrough: Exploit the BST property — if both targets are smaller, LCA is in the left subtree; if both larger, right subtree; otherwise the current node is the split point. For nodes 4 and 7: root 8 (both smaller, go left) → 3 (both larger, go right) → 6 (4<6, 7>6, split here) → **LCA = 6**. Time: O(h), Space: O(h).

Q3: Find the kth smallest element in a BST. Input: BST from diagram above, k=3.

```java
static int kthSmallest(TreeNode root, int k) {
    java.util.Deque<TreeNode> stack = new java.util.ArrayDeque<>();
    TreeNode curr = root;
    while (curr != null || !stack.isEmpty()) {
        while (curr != null) { stack.push(curr); curr = curr.left; }
        curr = stack.pop();
        if (--k == 0) return curr.val;
        curr = curr.right;
    }
    return -1;
}
```

Solution walkthrough: Iterative inorder traversal naturally visits nodes in ascending order — stop at the kth visit instead of building the full list. For k=3 on the diagram's tree (inorder: 1,3,4,6,7,8,10,13,14), 3rd element is **4**. Time: O(h+k), Space: O(h).

---

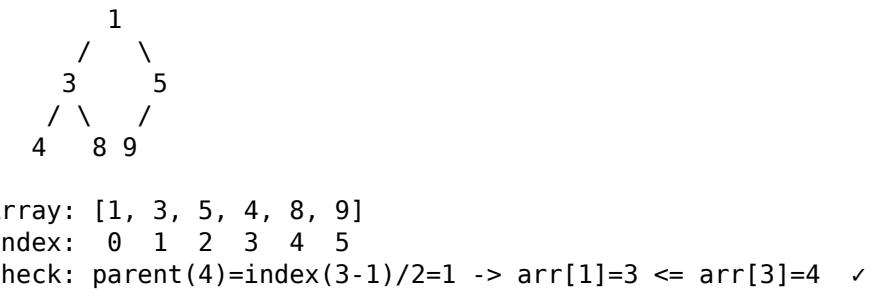
13. Heaps / Priority Queues

Definition: A complete binary tree satisfying the heap property — in a max-heap, every parent ≥ its children (root is the max); in a min-heap, every parent ≤ its children (root is the min).

Intuition: "Complete" means every level is full except possibly the last, filled left to right — this lets the heap be stored in a plain array with no pointers: for a node at index `i`, left child = `2i+1`, right child = `2i+2`, parent = `(i-1)/2`. This array-based representation is what makes heaps memory-efficient.

Real-life example: A hospital emergency room — patients aren't treated in arrival order (that's a queue); the most critical patient (highest priority) is always treated next, which is exactly what a priority queue backed by a heap gives you.

Diagram — min-heap and its array representation:



Java code:

```java
import java.util.PriorityQueue;

PriorityQueue<Integer> minHeap = new PriorityQueue<>();              // min-heap by default
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(java.util.Collections.reverseOrder());

minHeap.offer(5);
minHeap.offer(1);
minHeap.offer(3);
int min = minHeap.poll();   // 1, smallest first, O(log n)
```

Java code — manual min-heap implementation (to understand what `PriorityQueue` does internally):

```java
class MinHeap {
    java.util.List<Integer> heap = new java.util.ArrayList<>();

    void insert(int val) {
        heap.add(val);
        int i = heap.size() - 1;
        while (i > 0 && heap.get((i - 1) / 2) > heap.get(i)) {
            swap(i, (i - 1) / 2);
            i = (i - 1) / 2;             // bubble up
        }
    }

    int extractMin() {
        int min = heap.get(0);
        int last = heap.remove(heap.size() - 1);
        if (!heap.isEmpty()) {
            heap.set(0, last);
            bubbleDown(0);
        }
        return min;
    }

    void bubbleDown(int i) {
        int n = heap.size();
        while (true) {
            int left = 2 * i + 1, right = 2 * i + 2, smallest = i;
            if (left < n && heap.get(left) < heap.get(smallest)) smallest = left;
            if (right < n && heap.get(right) < heap.get(smallest)) smallest = right;
            if (smallest == i) break;
            swap(i, smallest);
            i = smallest;
        }
    }

    void swap(int i, int j) {
        int t = heap.get(i); heap.set(i, heap.get(j)); heap.set(j, t);
    }
}
```

Dry run — `insert(2)` into heap `[1,3,5,4,8,9]`:

```
heap becomes [1,3,5,4,8,9,2], i=6 (last index)
parent((6-1)/2)=heap[2]=5 > heap[6]=2 -> swap -> [1,3,2,4,8,9,5], i=2
```

```
parent((2-1)/2)=heap[0]=1 > heap[2]=2? No (1<2) -> stop
Final: [1,3,2,4,8,9,5]
```

Brute-force vs optimized — find the k largest elements. Input: `[3,1,5,12,2,11]`, k=3.

Brute-force — sort everything, take last k, O(n log n):

```java
static int[] kLargestBrute(int[] arr, int k) {
    int[] sorted = arr.clone();
    java.util.Arrays.sort(sorted);
    return java.util.Arrays.copyOfRange(sorted, sorted.length - k, sorted.length);
}
```

Optimized — min-heap of size k, O(n log k):

```java
static int[] kLargestOptimized(int[] arr, int k) {
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    for (int x : arr) {
        minHeap.offer(x);
        if (minHeap.size() > k) minHeap.poll();   // remove smallest, keep only top k
    }
    return minHeap.stream().mapToInt(Integer::intValue).toArray();
}
```

Why "k largest" uses a MIN-heap, not a max-heap: counterintuitive at first — you use a min-heap of size k so that the *smallest of the current top-k* sits at the root, ready to be evicted the instant a bigger element shows up. A max-heap would put the largest element at the root, which is the wrong thing to be checking against new candidates.

Complexity summary: | Operation | Complexity | Why | |—|—|—| | Peek min/max | O(1) | Always the root | | Insert | O(log n) | Bubble up at most tree height | | Extract min/max | O(log n) | Bubble down at most tree height | | Build heap from array | O(n) | Bottom-up heapify, not n separate inserts | | k largest/smallest (heap approach) | O(n log k) | vs O(n log n) for full sort |

Common mistakes: - Using a max-heap instead of a min-heap (or vice versa) for k-largest/smallest problems — this is the single most common heap mistake. - Forgetting Java's `PriorityQueue` is a min-heap by default — need `Collections.reverseOrder()` or a custom comparator for max-heap behavior. - Assuming heap gives O(1) search for an arbitrary (non-min/max) element — it doesn't; only the root is O(1), everything else is O(n). - Confusing "heap" (the data structure, satisfies parent/child ordering only) with "sorted array" (fully ordered) — a heap is only partially ordered.

Interview tips: - Say "min-heap of size k" out loud the instant you see "k largest/smallest/closest" — it's a strong, recognizable pattern. - Mention that `PriorityQueue` doesn't support O(log n) arbitrary-element removal or decrease-key out of the box in Java — relevant for graph algorithms like Dijkstra if asked about implementation details. - For "median of a stream," mention the two-heap technique (max-heap for lower half, min-heap for upper half) — a very common follow-up.

Practice questions with solutions:

Q1: Find the kth largest element in an unsorted array. Input: `[3,2,1,5,6,4]`, k=2.

```java
static int findKthLargest(int[] nums, int k) {
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    for (int x : nums) {
        minHeap.offer(x);
        if (minHeap.size() > k) minHeap.poll();
    }
    return minHeap.peek();
}
```

Solution walkthrough: Same min-heap-of-size-k pattern; after processing all elements, the root is the kth largest. Result: **5**. Time: O(n log k), Space: O(k).

Q2: Merge k sorted linked lists.

```
static ListNode mergeKLists(ListNode[] lists) {
    PriorityQueue<ListNode> pq = new PriorityQueue<>((a, b) -> a.val - b.val);
    for (ListNode node : lists) if (node != null) pq.offer(node);
    ListNode dummy = new ListNode(0), tail = dummy;
    while (!pq.isEmpty()) {
        ListNode smallest = pq.poll();
        tail.next = smallest;
        tail = tail.next;
        if (smallest.next != null) pq.offer(smallest.next);
    }
    return dummy.next;
}
```

Solution walkthrough: A min-heap always gives the globally smallest current head across all k lists in O(log k). Time: O(n log k) where n = total nodes, Space: O(k).

Q3: Find the median of a running data stream.

```
class MedianFinder {
    PriorityQueue<Integer> lowerHalf = new PriorityQueue<>(java.util.Collections.reverseOrder());
    PriorityQueue<Integer> upperHalf = new PriorityQueue<>();

    void addNum(int num) {
        lowerHalf.offer(num);
        upperHalf.offer(lowerHalf.poll());          // rebalance
        if (upperHalf.size() > lowerHalf.size())
            lowerHalf.offer(upperHalf.poll());
    }

    double findMedian() {
        if (lowerHalf.size() > upperHalf.size()) return lowerHalf.peek();
        return (lowerHalf.peek() + upperHalf.peek()) / 2.0;
    }
}
```

Solution walkthrough: Two heaps split the stream into a “lower half” (max-heap, so its max is accessible) and “upper half” (min-heap, so its min is accessible); kept balanced in size so the median is always at one or both roots. Insert: O(log n), findMedian: O(1). ## 14. Hashing

Definition: A technique for O(1) average-case lookup by mapping keys to array indices via a hash function, instead of searching.

Intuition: A hash function converts a key into an integer, reduced modulo table size to get an index — this is what turns “search through data” into “compute where the data should be.” Two different keys mapping to the same index is a **collision**, and how you resolve collisions determines both performance and correctness.

Real-life example: A library that shelves books by the first letter of the author’s last name — you don’t scan every shelf; you compute which shelf to check directly. If two authors share the same first letter (a “collision”), that shelf holds multiple books and you do a short scan within just that shelf.

Diagram — chaining collision resolution:

```
Hash table (size 5), hash(key) = key % 5

Index 0: -> 10 -> 15   (10%5=0, 15%5=0, collision resolved via linked list)
Index 1: -> 6
Index 2: (empty)
Index 3: -> 3 -> 18    (3%5=3, 18%5=3, collision)
Index 4: -> 9
```

Java code — using `HashMap`:

```java
import java.util.HashMap;
import java.util.Map;

Map<String, Integer> counts = new HashMap<>();
for (String word : new String[]{"a", "b", "a", "c", "a"}) {
    counts.put(word, counts.getOrDefault(word, 0) + 1);  // O(1) average
}
// counts = {a=3, b=1, c=1}
```

Java code — implementing a simple hash table with chaining (to see what `HashMap` does internally):

```java
class HashTable {
    private java.util.LinkedList<int[]>[] buckets;   // each bucket: list of [key, value] pairs
    private int capacity = 16;

    @SuppressWarnings("unchecked")
    HashTable() {
        buckets = new java.util.LinkedList[capacity];
        for (int i = 0; i < capacity; i++) buckets[i] = new java.util.LinkedList<>();
    }

    int hash(int key) { return Math.abs(key) % capacity; }

    void put(int key, int value) {
        int idx = hash(key);
        for (int[] pair : buckets[idx]) {
            if (pair[0] == key) { pair[1] = value; return; }   // update existing
        }
        buckets[idx].add(new int[]{key, value});
    }

    Integer get(int key) {
        int idx = hash(key);
        for (int[] pair : buckets[idx]) {
            if (pair[0] == key) return pair[1];
        }
        return null;
    }
}
```

Dry run — `put(10, "A")`, `put(15, "B")` into a table of capacity 5, `hash(k) = k % 5`:

```
put(10,"A"): idx=10%5=0, bucket[0]=[] -> add (10,"A"). bucket[0]=[(10,"A")]
put(15,"B"): idx=15%5=0, bucket[0]=[(10,"A")], no key match -> add (15,"B")
             bucket[0]=[(10,"A"),(15,"B")]   <- collision resolved by chaining
get(15): idx=0, scan bucket[0]: (10,"A") no match, (15,"B") match -> return "B"
```

Brute-force vs optimized — Two Sum. Input: `[2,7,11,15]`, target=9.

Brute-force — check every pair, O(n²):

```
static int[] twoSumBrute(int[] nums, int target) {
    for (int i = 0; i < nums.length; i++)
        for (int j = i + 1; j < nums.length; j++)
            if (nums[i] + nums[j] == target) return new int[]{i, j};
    return new int[]{-1, -1};
}
```

Optimized — hash map, O(n):

```
static int[] twoSumOptimized(int[] nums, int target) {
    Map<Integer, Integer> seen = new HashMap<>();   // value -> index
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (seen.containsKey(complement)) return new int[]{seen.get(complement), i};
        seen.put(nums[i], i);
    }
    return new int[]{-1, -1};
}
```

Dry run of optimized Two Sum — `[2,7,11,15]`, target=9:

```
i=0(2): complement=7, seen={} -> not found -> seen={2:0}
i=1(7): complement=2, seen={2:0} -> FOUND! return [0,1]
```

Complexity summary: | Operation | Average | Worst Case | Why worst case happens | |—|—|—|—| | Insert | O(1) | O(n) | All keys collide into one bucket (bad hash function) | | Search | O(1) | O(n) | Same | | Delete | O(1) | O(n) | Same |

Collision resolution strategies: | Strategy | How it works | Trade-off | |—|—|—| | Chaining | Each bucket holds a list of entries | Simple, degrades gracefully, extra memory per bucket | | Open addressing (linear probing) | On collision, check next slot (i+1, i+2, ...) | No pointer overhead, but clustering can hurt performance | | Open addressing (double hashing) | Use a second hash function for step size | Reduces clustering vs linear probing |

Common mistakes: - Using a mutable object as a `HashMap` key and then modifying it after insertion — breaks the key's hash-to-bucket mapping, making it unfindable even though it's still in the map. - Not overriding both `equals()` and `hashCode()` together for custom object keys — Java requires both to be consistent, or lookups silently fail. - Assuming O(1) is guaranteed rather than average-case — a poorly distributed hash function degrades to O(n). - Forgetting `getOrDefault()` exists and writing verbose null-checking boilerplate instead.

Interview tips: - The instant you're doing repeated linear scans to check "have I seen this before," say "I can trade space for time with a hash map" — this is the single highest-leverage optimization pattern in interviews. - For custom objects as keys, mention overriding `equals()`/`hashCode()` — shows Java-specific depth. - Know the difference between `HashMap` (unordered), `LinkedHashMap` (insertion order), and `TreeMap` (sorted order, O(log n) not O(1)) — picking the right one for stated requirements is a good signal.

Practice questions with solutions:

Q1: Group anagrams together. Input: `["eat","tea","tan","ate","nat","bat"]`.

```
static java.util.List<java.util.List<String>> groupAnagrams(String[] strs) {
    Map<String, java.util.List<String>> groups = new HashMap<>();
    for (String s : strs) {
        char[] chars = s.toCharArray();
        java.util.Arrays.sort(chars);
```

```
        String key = new String(chars);          // sorted string as the grouping key
        groups.computeIfAbsent(key, k -> new java.util.ArrayList<>()).add(s);
    }
    return new java.util.ArrayList<>(groups.values());
}
```

Solution walkthrough: All anagrams share the same sorted-character key ("eat"→"aet", "tea"→"aet", "ate"→"aet" all map together). Result: `[["eat","tea","ate"],["tan","nat"],["bat"]]`. Time: O(n·k log k) where k = max string length, Space: O(n·k).

Q2: Find the longest consecutive sequence in an unsorted array. Input: `[100,4,200,1,3,2]`.

```
static int longestConsecutive(int[] nums) {
    java.util.Set<Integer> set = new java.util.HashSet<>();
    for (int x : nums) set.add(x);
    int longest = 0;
    for (int x : set) {
        if (!set.contains(x - 1)) {              // only start counting from sequence beginnings
            int length = 1;
            while (set.contains(x + length)) length++;
            longest = Math.max(longest, length);
        }
    }
    return longest;
}
```

Solution walkthrough: Only start a count from numbers that are the start of a sequence (no `x-1` present) — this guarantees each element is visited a bounded number of times total, keeping it O(n) despite the nested-looking loop. Sequence `1,2,3,4` gives length **4**. Time: O(n), Space: O(n).

Q3: Design a data structure supporting `insert`, `remove`, and `getRandom` all in O(1) average.

```
class RandomizedSet {
    java.util.List<Integer> list = new java.util.ArrayList<>();
    Map<Integer, Integer> indexMap = new HashMap<>();    // value -> index in list

    boolean insert(int val) {
        if (indexMap.containsKey(val)) return false;
        indexMap.put(val, list.size());
        list.add(val);
        return true;
    }

    boolean remove(int val) {
        if (!indexMap.containsKey(val)) return false;
        int idx = indexMap.get(val);
        int lastVal = list.get(list.size() - 1);
        list.set(idx, lastVal);                  // move last element into the removed slot
        indexMap.put(lastVal, idx);
        list.remove(list.size() - 1);
        indexMap.remove(val);
        return true;
    }

    int getRandom() {
        return list.get(new java.util.Random().nextInt(list.size()));
```

```
    }
}
```

Solution walkthrough: The hash map gives O(1) lookup for `remove`, while swapping the target with the last list element (instead of shifting) keeps removal O(1) instead of O(n). Combining both structures is what achieves all three operations at O(1) — neither alone would suffice.

---

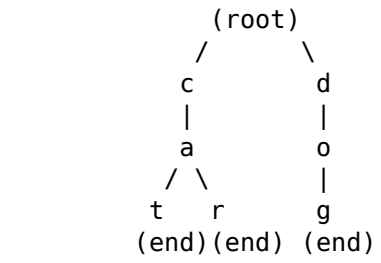
15. Tries

Definition: A tree specialized for storing strings, where each edge represents a character and each root-to-node path represents a prefix. Also called a prefix tree.

Intuition: A hash table gives O(1) exact-match lookup but can't efficiently answer "what words start with 'pre'?" A trie answers prefix queries in O(m), m = prefix length, *independent of how many words are stored* — you're just walking down the tree following characters, not scanning anything.

Real-life example: A phone's autocomplete/predictive text — as you type each letter, it narrows suggestions to only words sharing that prefix, without re-scanning the entire dictionary on every keystroke.

Diagram — trie storing "cat", "car", "dog":



Path root->c->a->t spells "cat"; root->c->a->r spells "car" (shares prefix "ca").

Java code:

```java
class TrieNode {
    Map<Character, TrieNode> children = new HashMap<>();
    boolean isEnd = false;
}

class Trie {
    TrieNode root = new TrieNode();

    void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            node = node.children.computeIfAbsent(c, k -> new TrieNode());
        }
        node.isEnd = true;
    }

    boolean search(String word) {
        TrieNode node = find(word);
        return node != null && node.isEnd;
    }
```

```
    boolean startsWith(String prefix) {
        return find(prefix) != null;
    }

    private TrieNode find(String s) {
        TrieNode node = root;
        for (char c : s.toCharArray()) {
            node = node.children.get(c);
            if (node == null) return null;
        }
        return node;
    }
}
```

Dry run — `insert("cat")` then `search("car")`:

```
insert("cat"): root -> create 'c' -> create 'a' -> create 't', mark isEnd=true
search("car"): root -> 'c' exists -> 'a' exists -> 'r' NOT in children of 'a' -
> return null -> false
```

Brute-force vs optimized — find all words with a given prefix, among n stored words.

Brute-force — check every stored word's prefix, O(n·m):

```
static java.util.List<String> prefixSearchBrute(java.util.List<String> words, String prefix) {
    java.util.List<String> result = new java.util.ArrayList<>();
    for (String w : words) if (w.startsWith(prefix)) result.add(w);
    return result;
}
```

Optimized — trie walk to the prefix node, then DFS to collect completions, O(m + k) where k = total characters in matching words:

```
static java.util.List<String> prefixSearchTrie(Trie trie, String prefix) {
    java.util.List<String> result = new java.util.ArrayList<>();
    TrieNode node = trie.root;
    for (char c : prefix.toCharArray()) {
        node = node.children.get(c);
        if (node == null) return result;   // no words with this prefix
    }
    collectWords(node, new StringBuilder(prefix), result);
    return result;
}
static void collectWords(TrieNode node, StringBuilder path, java.util.List<String> result) {
    if (node.isEnd) result.add(path.toString());
    for (Map.Entry<Character, TrieNode> entry : node.children.entrySet()) {
        path.append(entry.getKey());
        collectWords(entry.getValue(), path, result);
        path.deleteCharAt(path.length() - 1);
    }
}
```

Complexity summary: | Operation | Complexity | |—|—| | Insert | O(m), m = word length | | Search (exact) | O(m) | | Prefix search (existence) | O(p), p = prefix length | | Space | O(alphabet size × total nodes) — shared prefixes stored once |

Common mistakes: - Forgetting to mark `isEnd = true` on insert — makes `search` unable to distinguish a complete word from a mere prefix of a longer word. - Using a fixed-size array (e.g.,

`children[26]`) when the character set is unknown or includes more than lowercase letters — a `HashMap<Character, TrieNode>` is more flexible at a small performance cost. - Underestimating memory usage — tries can use more memory than a hash set for the same words, since every node needs a children structure even for single-branch paths. - Not handling the empty string or empty prefix as edge cases.

Interview tips: - Reach for a trie immediately when a problem mentions "prefix," "autocomplete," or "many patterns against one text/dictionary." - Contrast explicitly with a hash set when asked "why not just use a HashSet" — the answer is always prefix queries, which a hash set cannot do efficiently. - Mention space/time trade-off honestly — tries aren't automatically better than hash sets; they're better specifically for prefix operations.

Practice questions with solutions:

Q1: Implement `Trie` with `insert`, `search`, `startsWith` (see code above) — dry run `insert("app")`, `insert("apple")`, `search("app")`, `startsWith("app")`. **Solution:** `search("app")` → walks to node after 'a','p','p' → `isEnd=true` (marked during `insert("app")`) → **true**. `startsWith("app")` → same walk, node exists regardless of `isEnd` → **true**. Both O(3) = O(word length).

Q2: Find the longest common prefix among an array of strings using a trie. Input: `["flower","flow","flight"]`.

```
static String longestCommonPrefix(String[] words) {
    Trie trie = new Trie();
    for (String w : words) trie.insert(w);
    StringBuilder prefix = new StringBuilder();
    TrieNode node = trie.root;
    while (node.children.size() == 1 && !node.isEnd) {
        char c = node.children.keySet().iterator().next();
        prefix.append(c);
        node = node.children.get(c);
    }
    return prefix.toString();
}
```

Solution walkthrough: Walk down the trie while there's exactly one branch (no divergence yet). Divergence happens after "fl" (into "flo..." vs "fli..."). Result: **"fl"**. Time: O(sum of all word lengths), Space: O(same).

Q3: Implement a word search (word break) checker: given a dictionary trie, can a string be segmented into dictionary words? Input: dictionary `["leet","code"]`, string `"leetcode"`.

```
static boolean wordBreak(String s, Trie dict) {
    boolean[] dp = new boolean[s.length() + 1];
    dp[0] = true;
    for (int i = 1; i <= s.length(); i++) {
        for (int j = 0; j < i; j++) {
            if (dp[j] && dict.search(s.substring(j, i))) { dp[i] = true; break; }
        }
    }
    return dp[s.length()];
}
```

Solution walkthrough: This combines a trie (fast dictionary lookup) with dynamic programming (section 23) — `dp[i]` means "can the first i characters be segmented." `dp[4]=true` ("leet"), then `dp[8]=true` via `dp[4] && "code" in dict`. Result: **true**. Time: O(n^2) substring checks (each O(m) via trie), Space: O(n). ## 16. Graphs & Traversal (BFS/DFS)

Definition: A set of vertices (nodes) connected by edges. The most general structure in this course — trees and linked lists are special cases of graphs.

Intuition: Graphs model relationships, not just hierarchy — any two nodes can connect, cycles are allowed, and there's no single "root." The two fundamental ways to explore a graph, BFS and DFS, differ in *order*: BFS explores level by level (nearest first), DFS commits to one path as deep as possible before backtracking. This ordering difference is why BFS guarantees shortest path (by edge count) on unweighted graphs and DFS doesn't.

Real-life example: Mapping out a social network. BFS answers "who are my friends, then friends-of-friends, then friends-of-friends-of-friends" (degrees of separation, level by level). DFS answers "follow one chain of friendships as deep as it goes before trying a different chain."

Diagram — adjacency list vs adjacency matrix for the same graph:

```
Graph:  A -- B
        |    |
        C -- D

Adjacency List:            Adjacency Matrix:
A: [B, C]                      A  B  C  D
B: [A, D]                   A [0, 1, 1, 0]
C: [A, D]                   B [1, 0, 0, 1]
D: [B, C]                   C [1, 0, 0, 1]
                             D [0, 1, 1, 0]
```

Java code — representations:

```java
// Adjacency list — preferred for sparse graphs, O(V+E) space
Map<Character, java.util.List<Character>> graph = new HashMap<>();
graph.put('A', java.util.Arrays.asList('B', 'C'));
graph.put('B', java.util.Arrays.asList('A', 'D'));
graph.put('C', java.util.Arrays.asList('A', 'D'));
graph.put('D', java.util.Arrays.asList('B', 'C'));

// Adjacency matrix — O(V^2) space, O(1) edge-existence check
int[][] matrix = {
    {0,1,1,0},
    {1,0,0,1},
    {1,0,0,1},
    {0,1,1,0}
};
```

Java code — BFS:

```java
static java.util.List<Character> bfs(Map<Character, java.util.List<Character>> graph, char start
    java.util.List<Character> order = new java.util.ArrayList<>();
    java.util.Set<Character> visited = new java.util.HashSet<>();
    java.util.Queue<Character> queue = new java.util.ArrayDeque<>();
    visited.add(start);
    queue.offer(start);
    while (!queue.isEmpty()) {
        char node = queue.poll();
        order.add(node);
        for (char neighbor : graph.getOrDefault(node, java.util.List.of())) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                queue.offer(neighbor);
            }
        }
    }
```

```
    return order;
}
```

Java code — DFS (recursive):

```
static java.util.List<Character> dfs(Map<Character, java.util.List<Character>> graph, char start
    java.util.List<Character> order = new java.util.ArrayList<>();
    java.util.Set<Character> visited = new java.util.HashSet<>();
    dfsHelper(graph, start, visited, order);
    return order;
}
static void dfsHelper(Map<Character, java.util.List<Character>> graph, char node,
                      java.util.Set<Character> visited, java.util.List<Character> order) {
    visited.add(node);
    order.add(node);
    for (char neighbor : graph.getOrDefault(node, java.util.List.of())) {
        if (!visited.contains(neighbor)) dfsHelper(graph, neighbor, visited, order);
    }
}
```

Dry run — BFS from 'A' on the diagram's graph:

```
visited={A}, queue=[A]
pop A -> order=[A]; neighbors B,C unvisited -> visited={A,B,C}, queue=[B,C]
pop B -> order=[A,B]; neighbors A(visited),D(unvisited) -> visited={A,B,C,D}, queue=[C,D]
pop C -> order=[A,B,C]; neighbors A(visited),D(visited) -> queue=[D]
pop D -> order=[A,B,C,D]; neighbors B(visited),C(visited) -> queue=[]
Final order: A, B, C, D
```

Brute-force vs optimized — is there a path between two nodes?

Brute-force — try all possible paths (exponential without visited tracking, risks infinite loops on cycles): not viable without marking visited nodes.

Optimized — BFS or DFS with a visited set, O(V+E): (code above) — this "optimized" version is really the *only* correct version; the point here is that visited-tracking is what makes graph traversal correct and efficient at once, not an afterthought.

Complexity summary: | Operation | Time | Space | |—|—|—| | BFS | O(V+E) | O(V) (queue + visited set) | | DFS | O(V+E) | O(V) (recursion stack + visited set) — worst case O(V) depth on a skewed graph | | Adjacency list build | O(V+E) | O(V+E) | | Adjacency matrix build | $O(V^2)$ | $O(V^2)$ |

Common mistakes: - Forgetting the visited set entirely — causes infinite loops on any graph with a cycle. - Marking a node visited *after* popping it from the queue in BFS instead of *when adding it* — can enqueue the same node multiple times, wasting work (though not incorrect, since a visited-check on pop still terminates it — but it's a common efficiency slip). - Using DFS when the problem asks for *shortest* path on an unweighted graph — DFS finds *a* path, not necessarily the shortest one. - Not handling disconnected graphs — a single BFS/DFS call from one start node won't reach nodes in other components; need to loop over all nodes and start a new traversal from any unvisited one.

Interview tips: - State your representation choice (adjacency list vs matrix) and why — list for sparse graphs (most real problems), matrix only when dense or O(1) edge checks are needed. - BFS = shortest path (unweighted) and "level" problems. DFS = exhaustive exploration, cycle detection, topological sort, connected components. - Always ask: directed or undirected? Weighted or not? Can it be disconnected? These change the correct algorithm.

Practice questions with solutions:

Q1: Count the number of islands in a grid (connected components of '1's). Input:

```
11000
11000
00100
00011
```

```
static int numIslands(char[][] grid) {
    int count = 0;
    for (int r = 0; r < grid.length; r++)
        for (int c = 0; c < grid[0].length; c++)
            if (grid[r][c] == '1') { count++; sinkIsland(grid, r, c); }
    return count;
}
static void sinkIsland(char[][] grid, int r, int c) {
    if (r < 0 || c < 0 || r >= grid.length || c >= grid[0].length || grid[r][c] != '1') return;
    grid[r][c] = '0';   // mark visited by "sinking"
    sinkIsland(grid, r+1, c); sinkIsland(grid, r-1, c);
    sinkIsland(grid, r, c+1); sinkIsland(grid, r, c-1);
}
```

Solution walkthrough: DFS from every unvisited '1', flipping connected land to '0' as you go so it's never recounted. Grid above has **3** islands. Time: O(rows×cols), Space: O(rows×cols) worst case recursion depth.

Q2: Detect a cycle in a directed graph.

```
static boolean hasCycle(Map<Integer, java.util.List<Integer>> graph, int nodes) {
    int[] state = new int[nodes];   // 0=unvisited, 1=in-progress, 2=done
    for (int i = 0; i < nodes; i++)
        if (state[i] == 0 && dfsCycle(graph, i, state)) return true;
    return false;
}
static boolean dfsCycle(Map<Integer, java.util.List<Integer>> graph, int node, int[] state) {
    state[node] = 1;   // mark in-progress (on current DFS path)
    for (int neighbor : graph.getOrDefault(node, java.util.List.of())) {
        if (state[neighbor] == 1) return true;               // back edge -> cycle
        if (state[neighbor] == 0 && dfsCycle(graph, neighbor, state)) return true;
    }
    state[node] = 2;   // mark fully done
    return false;
}
```

Solution walkthrough: Tracking three states (not just visited/unvisited) distinguishes "still on the current path" (state 1) from "fully explored, safe" (state 2) — reaching a state-1 node means you've looped back onto your own path. Time: O(V+E), Space: O(V).

Q3: Perform a bipartite check on a graph (can nodes be 2-colored so no edge connects same-colored nodes?).

```
static boolean isBipartite(int[][] graph) {
    int n = graph.length;
    int[] color = new int[n];   // 0=uncolored, 1 or -1 = two colors
    for (int start = 0; start < n; start++) {
        if (color[start] != 0) continue;
        color[start] = 1;
        java.util.Queue<Integer> queue = new java.util.ArrayDeque<>();
        queue.offer(start);
        while (!queue.isEmpty()) {
```

```
            int node = queue.poll();
            for (int neighbor : graph[node]) {
                if (color[neighbor] == 0) {
                    color[neighbor] = -color[node];
                    queue.offer(neighbor);
                } else if (color[neighbor] == color[node]) {
                    return false;   // same color on both ends of an edge
                }
            }
        }
    }
    return true;
}
```

Solution walkthrough: BFS while alternating colors between neighbors; a conflict (neighbor already has the same color as the current node) means no valid 2-coloring exists. Time: O(V+E), Space: O(V).

---

17. Shortest Path & Minimum Spanning Tree

Definition: Shortest path algorithms find the minimum-cost route between vertices in a weighted graph. A Minimum Spanning Tree (MST) connects all vertices in an undirected weighted graph with the minimum total edge weight and no cycles.

Intuition: Dijkstra's algorithm greedily expands the closest unvisited vertex first, using a min-heap to always know which vertex that is — this greedy choice is provably safe *only* because edge weights are non-negative (a shorter path can't later appear from an already-finalized vertex). Bellman-Ford relaxes this restriction by checking every edge repeatedly, trading speed for the ability to handle negative weights.

Real-life example: Dijkstra's is exactly what GPS navigation does to find the shortest driving route, when all road "costs" (distances/times) are positive. MST is what a telecom company solves when laying cable to connect every city with the minimum total cable length.

Diagram — Dijkstra expanding outward from source A (numbers are edge weights):

```
      A --4-- B
      |       |
      1       2
      |       |
      C --1-- D


dist: A=0, then relax neighbors: C=1, B=4
expand C(1): relax D via C -> D=1+1=2
expand D(2): relax B via D -> B=min(4, 2+2)=4 (no improvement)
expand B(4): done
Final shortest distances from A: A=0, C=1, D=2, B=4
```

Java code — Dijkstra's algorithm:

```
static Map<Character, Integer> dijkstra(Map<Character, java.util.List<int[]>> graph, char start)
    // graph: node -> list of [neighborAsCharCode, weight]
    Map<Character, Integer> dist = new HashMap<>();
    for (char node : graph.keySet()) dist.put(node, Integer.MAX_VALUE);
    dist.put(start, 0);
    PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[1] - b[1]);  // [node, distance]
```

```
    pq.offer(new int[]{start, 0});
    while (!pq.isEmpty()) {
        int[] curr = pq.poll();
        char node = (char) curr[0];
        int d = curr[1];
        if (d > dist.get(node)) continue;    // stale entry, skip
        for (int[] edge : graph.getOrDefault(node, java.util.List.of())) {
            char neighbor = (char) edge[0];
            int weight = edge[1];
            int newDist = d + weight;
            if (newDist < dist.get(neighbor)) {
                dist.put(neighbor, newDist);
                pq.offer(new int[]{neighbor, newDist});
            }
        }
    }
    return dist;
}
```

Java code — Kruskal's MST (uses Union-Find, section 18):

```
static int kruskalMST(int n, int[][] edges) {
    // edges: [u, v, weight]
    java.util.Arrays.sort(edges, (a, b) -> a[2] - b[2]);   // sort by weight ascending
    UnionFind uf = new UnionFind(n);
    int totalWeight = 0, edgesUsed = 0;
    for (int[] edge : edges) {
        if (uf.union(edge[0], edge[1])) {   // true if this edge connects two different componen
            totalWeight += edge[2];
            edgesUsed++;
            if (edgesUsed == n - 1) break;   // MST complete
        }
    }
    return totalWeight;
}
```

Dry run — Kruskal's on edges `[(0,1,4),(0,2,1),(1,2,2),(1,3,5),(2,3,8)]`, n=4:

```
Sorted by weight: (0,2,1), (1,2,2), (0,1,4), (1,3,5), (2,3,8)
(0,2,1): union(0,2) -> different components -> add, weight=1, edgesUsed=1
(1,2,2): union(1,2) -> different components -> add, weight=1+2=3, edgesUsed=2
(0,1,4): union(0,1) -> 0 and 1 already connected via 2 -> SKIP (would create cycle)
(1,3,5): union(1,3) -> different components -> add, weight=3+5=8, edgesUsed=3=n-1 -
> DONE
MST total weight = 8
```

Complexity summary: | Algorithm | Time | Handles negative weights? | Use case | |—|—|—|—| | Dijkstra (binary heap) | O((V+E) log V) | No | Single-source shortest path, non-negative weights | | Bellman-Ford | O(V·E) | Yes (detects negative cycles too) | Single-source, possibly negative weights | | Floyd-Warshall | O(V^3) | Yes (not negative cycles) | All-pairs shortest path, small V | | Kruskal's MST | O(E log E) | N/A | MST via edge sorting + Union-Find | | Prim's MST | O(E log V) | N/A | MST via vertex growth + min-heap |

Common mistakes: - Using Dijkstra on a graph with negative edge weights — it will silently produce a wrong (too-short) answer without any error, because it assumes finalized distances never improve. - Forgetting to check for stale priority queue entries in Dijkstra (`if (d > dist.get(node)) continue;`) — without this, you may process outdated distances. - Choosing

Prim's when the graph is sparse (Kruskal's with Union-Find is typically simpler and just as fast) or Kruskal's when the graph is dense without considering Prim's better constant factors — not wrong, but worth knowing the trade-off. - Confusing "shortest path" (Dijkstra/Bellman-Ford, works on any connected graph, cares about a specific source) with "MST" (Kruskal/Prim, only defined for undirected graphs, connects everything, doesn't optimize for any particular source).

Interview tips: - Always ask about edge weights (can they be negative?) before choosing Dijkstra — this single question demonstrates you understand its core limitation. - Mention Union-Find as the reason Kruskal's cycle detection is efficient — connects two topics and shows you see how techniques compose. - For "connect all points with minimum cost" problems, immediately recognize this as MST, not shortest path — a common categorization mistake.

Practice questions with solutions:

Q1: Given `times[i] = (u, v, w)` representing a signal from u to v taking time w, find how long it takes for a signal starting at node k to reach all n nodes (Network Delay Time). This is Dijkstra applied directly. **Solution:** Run Dijkstra from k, then return `max(dist.values())` — if any node is still `Integer.MAX_VALUE` (unreachable), return -1. Time: O((V+E) log V).

Q2: Find the cheapest flights within k stops, given weighted directed edges. Why doesn't plain Dijkstra work here? **Solution:** Plain Dijkstra optimizes purely for lowest cost, ignoring the stop-count constraint — it might find the cheapest path but with too many stops. Instead use Bellman-Ford-style relaxation limited to exactly k+1 rounds (each round represents one more allowed edge/stop), or BFS/DFS with state `(node, stopsUsed)`. This illustrates that adding a constraint can invalidate a greedy algorithm's core assumption.

Q3: Given a graph, determine if it's connected (find if an MST spans all n vertices) using Kruskal's approach. **Solution:** Run Kruskal's; if `edgesUsed < n - 1` after processing all edges, the graph is disconnected (no spanning tree exists that connects everyone). This reuses the exact code above with the completion check as the answer, rather than the summed weight.

---

18. Union-Find (Disjoint Set Union)

Definition: A structure tracking a partition of elements into disjoint sets, optimized for `find` (which set does x belong to?) and `union` (merge two sets).

Intuition: Repeatedly asking "are these two elements connected?" while the graph is being built incrementally is expensive to answer from scratch each time with BFS/DFS. Union-Find answers both operations in *near*-O(1) by keeping a forest of trees where each tree's root is the set's "representative," and using two optimizations (path compression, union by rank) to keep those trees almost perfectly flat.

Real-life example: Tracking friend groups as friendships form one at a time at a party — instead of recomputing all friend groups from scratch every time two people become friends, you just merge their existing groups, and "are these two people in the same group" is a fast lookup.

Diagram — union by rank with path compression:

```
Initial: each element is its own parent: 0 1 2 3 4 (5 separate trees)

union(0,1): 1's root becomes 0 (or vice versa, by rank)
    0        2   3   4
   /
  1

union(2,3): similarly
    0        2
   /        /
```

```
  1       3

union(0,2): attach smaller tree under larger
      0
    / | \
   1  2  4? (4 still separate)
      |
      3

find(3): path compression makes 3 point directly to 0 next time, flattening the tree
```

Java code:

```java
class UnionFind {
    int[] parent, rank;

    UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) parent[i] = i;   // each node starts as its own root
    }

    int find(int x) {
        if (parent[x] != x) parent[x] = find(parent[x]);   // path compression
        return parent[x];
    }

    boolean union(int x, int y) {
        int rootX = find(x), rootY = find(y);
        if (rootX == rootY) return false;    // already connected -> would form a cycle
        if (rank[rootX] < rank[rootY]) { int t = rootX; rootX = rootY; rootY = t; }
        parent[rootY] = rootX;               // attach smaller-rank tree under larger
        if (rank[rootX] == rank[rootY]) rank[rootX]++;
        return true;
    }
}
```

Dry run — `union(0,1)`, `union(2,3)`, `union(1,2)`, then `find(3)`:

```
Initial: parent=[0,1,2,3], rank=[0,0,0,0]
union(0,1): find(0)=0, find(1)=1, ranks equal -> parent[1]=0, rank[0]=1
            parent=[0,0,2,3]
union(2,3): find(2)=2, find(3)=3, ranks equal -> parent[3]=2, rank[2]=1
            parent=[0,0,2,2]
union(1,2): find(1)=0 (path compress: parent[1] already 0), find(2)=2
            rank[0]=1, rank[2]=1, equal -> parent[2]=0, rank[0]=2
            parent=[0,0,0,2]
find(3): parent[3]=2, parent[2]=0 -> path compression sets parent[3]=0 directly
         returns 0. parent=[0,0,0,0]
```

Brute-force vs optimized — check graph connectivity as edges are added one at a time.

Brute-force — rerun BFS/DFS from scratch after every new edge, O(V+E) per query:

```
// Rerun full BFS/DFS each time — correct but wasteful if there are many queries
```

Optimized — Union-Find, near-O(1) amortized per operation (code above).

Complexity summary: | Operation | Without optimizations | With path compression + union

by rank | |—|—|—| | find | O(n) worst case (tall tree) | O(α(n)) amortized ≈ O(1) | | union | O(n) worst case | O(α(n)) amortized ≈ O(1) |

α(n) is the inverse Ackermann function — grows so slowly it's less than 5 for any n you'll ever encounter, effectively constant in practice.

Common mistakes: - Implementing `find` without path compression — still correct, but degrades to O(n) per call on skewed unions, defeating the purpose. - Implementing `union` without union by rank/size — trees can become tall chains, also degrading performance. - Forgetting `union` should return whether the merge actually happened (i.e., whether the two elements were previously in different sets) — this return value is exactly what Kruskal's algorithm needs to detect cycles. - Using Union-Find for problems requiring *disconnection* (removing an edge/element from a set) — Union-Find only supports merging, not splitting; it's the wrong tool for dynamic disconnection queries.

Interview tips: - Recognize Union-Find whenever a problem involves connectivity queries as edges are added incrementally, or cycle detection in an undirected graph (Kruskal's MST). - Mention both optimizations (path compression, union by rank) by name — implementing Union-Find without at least one is a common and noticeable gap. - For "number of connected components" style problems, Union-Find is often simpler to write correctly under time pressure than BFS/DFS over all components.

Practice questions with solutions:

Q1: Count the number of connected components in an undirected graph. Input: n=5, edges=[[0,1],[1,2],[3,4]].

```
static int countComponents(int n, int[][] edges) {
    UnionFind uf = new UnionFind(n);
    int components = n;
    for (int[] edge : edges) if (uf.union(edge[0], edge[1])) components--;
    return components;
}
```

Solution walkthrough: Start assuming n separate components; every successful union (connecting two previously-separate sets) reduces the count by 1. Result: components {0,1,2} and {3,4} → **2**. Time: O(E·α(n)), Space: O(n).

Q2: Detect a cycle in an undirected graph using Union-Find. Input: edges=[[0,1],[1,2],[2,0]]. **Solution:** For each edge, call `union(u, v)`; if it returns `false`, u and v were already connected before this edge — adding it creates a cycle. Edge `(2,0)` returns false since 0,1,2 are already unified → **cycle detected**. Time: O(E·α(n)), Space: O(n).

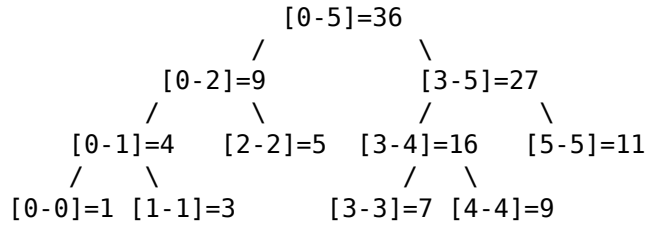
Q3: Given a list of accounts with emails, merge accounts that share any email (Accounts Merge problem). **Solution:** Union-Find over email strings (map each unique email to an index). For each account, union all its emails together. Then group emails by their root parent — each group is a merged account. This is a real-world application of Union-Find beyond graph edges: any "these things belong together" grouping problem fits the pattern. Time: O(N·α(N)) where N = total emails, Space: O(N). ## 19. Segment Trees & Fenwick Trees

Definition: Structures answering range queries (sum/min/max over [l,r]) while also supporting point/range updates efficiently — something a plain prefix-sum array can't do without an O(n) update cost.

Intuition: A segment tree is a binary tree where each node represents a range of the array; leaves are individual elements, and each internal node stores the aggregate of its children's ranges. Because the tree has O(log n) height, both queries and updates only touch O(log n) nodes. A Fenwick Tree (Binary Indexed Tree) achieves the same O(log n) bounds for prefix-sum-style queries using a cleverer, more compact array-based encoding based on the binary representation of indices.

Real-life example: A company tracking regional sales in a hierarchy — country total is the sum of state totals, which are sums of city totals. Updating one city's number only requires updating the O(log n) ancestors above it (city → state → country), not recomputing every total from scratch.

Diagram — segment tree over [1, 3, 5, 7, 9, 11]:




```
Query sum(1,4): combines [1-1]=3, [2-2]=5, [3-4]=16 -> 3+5+16=24
```

Java code — Segment Tree (sum queries + point updates):

```java
class SegmentTree {
    int[] tree;
    int n;

    SegmentTree(int[] arr) {
        n = arr.length;
        tree = new int[4 * n];
        build(arr, 0, 0, n - 1);
    }

    void build(int[] arr, int node, int start, int end) {
        if (start == end) { tree[node] = arr[start]; return; }
        int mid = (start + end) / 2;
        build(arr, 2*node+1, start, mid);
        build(arr, 2*node+2, mid+1, end);
        tree[node] = tree[2*node+1] + tree[2*node+2];
    }

    void update(int idx, int val) { update(0, 0, n - 1, idx, val); }
    void update(int node, int start, int end, int idx, int val) {
        if (start == end) { tree[node] = val; return; }
        int mid = (start + end) / 2;
        if (idx <= mid) update(2*node+1, start, mid, idx, val);
        else update(2*node+2, mid+1, end, idx, val);
        tree[node] = tree[2*node+1] + tree[2*node+2];
    }

    int query(int l, int r) { return query(0, 0, n - 1, l, r); }
    int query(int node, int start, int end, int l, int r) {
        if (r < start || end < l) return 0;                // no overlap
        if (l <= start && end <= r) return tree[node];     // full overlap
        int mid = (start + end) / 2;                        // partial overlap
        return query(2*node+1, start, mid, l, r) + query(2*node+2, mid+1, end, l, r);
    }
}
```

Java code — Fenwick Tree (Binary Indexed Tree):

```java
class FenwickTree {
    int[] tree;
    int n;

    FenwickTree(int n) {
        this.n = n;
        tree = new int[n + 1];
    }

    void update(int i, int delta) {          // i is 0-indexed
        for (i++; i <= n; i += i & (-i)) tree[i] += delta;
    }

    int prefixSum(int i) {                    // sum of arr[0..i]
        int sum = 0;
        for (i++; i > 0; i -= i & (-i)) sum += tree[i];
        return sum;
    }

    int rangeSum(int l, int r) {
        return prefixSum(r) - (l > 0 ? prefixSum(l - 1) : 0);
    }
}
```

Dry run — Fenwick Tree `update(2, 5)` on `n=6` (adding 5 to index 2, 0-indexed):

```
i = 2+1 = 3 (1-indexed)
tree[3] += 5
i += i & (-i) = 3 + (3 & -3) = 3 + 1 = 4 -> tree[4] += 5
i += i & (-i) = 4 + (4 & -4) = 4 + 4 = 8 > n=6 -> stop
Both tree[3] and tree[4] updated -- these are exactly the ancestors that
"cover" index 2 in the Fenwick binary-indexed scheme.
```

Brute-force vs optimized — range sum with updates. Input: array of size n, q queries mixing updates and range-sum lookups.

Brute-force — plain array, O(n) per range sum, O(1) per update:

```
// sum(l,r): loop l..r and add -> O(n) per query
// update(i,val): arr[i]=val -> O(1)
// Total for q queries: O(q*n) worst case
```

Optimized — Segment Tree or Fenwick Tree, O(log n) per operation:

```
// sum(l,r): O(log n)
// update(i,val): O(log n)
// Total for q queries: O(q log n)
```

Complexity summary: | Structure | Build | Range Query | Point Update | Range Update (with lazy propagation) | |—|—|—|—| | Prefix sum array | O(n) | O(1) | O(n) | O(n) | | Segment Tree | O(n) | O(log n) | O(log n) | O(log n) with lazy propagation | | Fenwick Tree | O(n log n) naive / O(n) optimized | O(log n) | O(log n) | O(log n) with a second Fenwick tree trick |

Lazy propagation (segment tree range updates): instead of updating every leaf in a range immediately (O(n)), mark internal nodes with a pending update and only push it down to children when that subrange is actually queried further. This is what makes *range* updates O(log n) instead of O(n).

Common mistakes: - Under-allocating the segment tree array — `4*n` is the standard safe size;

smaller allocations can cause `ArrayIndexOutOfBoundsException` on certain tree shapes. - Forgetting to propagate updates back up to parent nodes after a leaf update — leaves stale aggregate values in ancestors. - Choosing a Fenwick Tree for min/max range queries — Fenwick trees are naturally suited to prefix-aggregable operations like sum; min/max require a segment tree (or a specially adapted Fenwick variant) since they aren't invertible the way subtraction undoes addition. - Reaching for these structures when updates never occur — a plain prefix sum array is simpler and sufficient without the update requirement.

Interview tips: - These are less common in typical software interviews and more common in competitive programming — but recognizing the phrase "range query" + "updates" together should trigger this section regardless of context. - If asked to implement one under time pressure, Fenwick Tree is faster to write correctly (fewer lines, no recursion) for sum-based problems. - Mention that without updates, a simple prefix-sum array in O(n) build / O(1) query is strictly better — don't over-engineer.

Practice questions with solutions:

Q1: Given an array, support range-sum queries and point updates efficiently. Input: `[1,3,5,7,9,11]`, then `update(1,10)`, then `query(1,3)`. **Solution:** Build a Fenwick Tree; `update(1,10)` adds `10-3=7` delta at index 1; `query(1,3)` computes `prefixSum(3)-prefixSum(0)`. After update, array is conceptually `[1,10,5,7,9,11]`, so `query(1,3) = 10+5+7 = 22`. Time: O(log n) per operation.

Q2: Count the number of smaller elements to the right of each element in an array, using a Fenwick Tree. Input: `[5,2,6,1]`. **Solution:** Process the array right to left, using a Fenwick Tree indexed by value (after coordinate compression) to track counts seen so far; for each element, query the prefix sum up to (value-1) to count how many smaller elements have been seen. Result: `[2,1,1,0]`. Time: O(n log n).

Q3: Given an array, find the minimum in any range [l,r] with point updates allowed. Why is a Fenwick Tree a poor fit here? **Solution:** Fenwick Trees rely on operations being invertible (you can "subtract out" a previous contribution, as sum allows via delta). Min has no inverse — you can't undo a min contribution when a value is updated. A Segment Tree (storing min instead of sum at each node) is the correct structure here, giving O(log n) query and update.

---

20. Divide and Conquer

Definition: Split a problem into independent subproblems of the same type, solve each recursively, then combine the results. Three steps: **divide**, **conquer** (recurse), **combine**.

Intuition: This pattern converts many $O(n^2)$ brute-force approaches into O(n log n), because the combine step is usually cheap (often linear) while recursive halving contributes the log n factor. You've already used this pattern in merge sort (section 9) and binary search (section 10) — this section names the general technique and shows two more applications.

Real-life example: Organizing a company-wide election by having each department count its own votes independently (divide), then summing department totals at the top (combine) — instead of one person counting every single vote from the entire company sequentially.

Diagram — closest pair of points, high level:

```
Sort points by x-coordinate.
Split into left half and right half.
Recursively find closest pair in each half.
Combine: check pairs straddling the midline within the current best distance.
This "combine" step, done carefully, is only O(n), keeping the total at O(n log n)
instead of the brute-force O(n^2) of checking every pair.
```

Java code — Quick Select (find the kth smallest element without fully sorting):

```
static int quickSelect(int[] arr, int k) {
    return quickSelectHelper(arr, 0, arr.length - 1, k - 1);   // k-1 for 0-indexed
}
static int quickSelectHelper(int[] arr, int low, int high, int k) {
    int pivotIndex = partition(arr, low, high);
    if (pivotIndex == k) return arr[pivotIndex];
    else if (pivotIndex < k) return quickSelectHelper(arr, pivotIndex + 1, high, k);
    else return quickSelectHelper(arr, low, pivotIndex - 1, k);
}
// partition() is the same as in Quick Sort, section 9
```

Dry run — `quickSelect([3,2,1,5,6,4], k=2)` (find the 2nd smallest = index 1):

```
Partition around pivot=4 (last element): elements <=4 move left
After partition: [3,2,1,4,6,5] roughly, pivotIndex lands at index 3 (value 4)
pivotIndex(3) > k(1) -> recurse left: quickSelectHelper(arr, 0, 2, 1)
Partition [3,2,1] around pivot=1: after partition, pivotIndex=0
pivotIndex(0) < k(1) -> recurse right: quickSelectHelper(arr, 1, 2, 1)
Partition [3,2] (conceptually) around pivot=2: pivotIndex=1=k -> return arr[1]=2
Result: 2nd smallest is 2
```

Brute-force vs optimized — find the kth smallest element. Input: `[7,10,4,3,20,15]`, k=3.

Brute-force — full sort, O(n log n):

```
static int kthSmallestBrute(int[] arr, int k) {
    int[] sorted = arr.clone();
    java.util.Arrays.sort(sorted);
    return sorted[k - 1];
}
```

Optimized — Quick Select, O(n) average (only recurses into the needed side, unlike quicksort which recurses into both):

```
// quickSelect() above, O(n) average, O(n^2) worst case (rare with randomized pivot)
```

Complexity summary: | Algorithm | Time | Space | Notes | |—|—|—|—| | Merge Sort | O(n log n) | O(n) | Always splits in half, combine is O(n) merge | | Quick Select | O(n) average, O(n^2) worst | O(log n) | Only recurses into one side, unlike quicksort | | Closest Pair of Points | O(n log n) | O(n) | vs O(n^2) brute force checking all pairs | | Karatsuba Multiplication | O(n^1.585) | O(n) | vs O(n^2) grade-school multiplication | | Binary Search | O(log n) | O(1) | Degenerate case: no “combine” step needed |

Common mistakes: - Confusing divide-and-conquer with dynamic programming — D&C subproblems are independent (no overlap); if you notice the same subproblem being solved repeatedly, that’s the signal to switch to DP with memoization instead. - Forgetting Quick Select’s worst case is still O(n^2) without randomized pivot selection — same underlying issue as quicksort. - Applying D&C to a problem without genuine optimal substructure — splitting doesn’t help if subproblems can’t be solved independently and combined validly. - Underestimating the “combine” step’s cost — if combining two solved halves costs more than O(n), you may not get the expected O(n log n) overall.

Interview tips: - When a problem has a natural “split in half” structure, name divide-and-conquer explicitly and identify the three steps (divide/conquer/combine) before coding. - Quick Select is the standard answer for “kth largest/smallest without fully sorting” — mention its O(n) average complexity as the reason to prefer it over sorting. - If two D&C subproblems overlap, say so and pivot to DP — recognizing this distinction is a strong signal in interviews.

Practice questions with solutions:

Q1: Find the maximum subarray sum using divide and conquer (contrast with Kadane's O(n) from section 3). **Solution:** Split the array in half; the max subarray either lies entirely in the left half, entirely in the right half, or crosses the midpoint (compute this case by extending outward from the middle in both directions, O(n) for the combine step). Recurrence: T(n) = 2T(n/2) + O(n) → O(n log n) — worse than Kadane's O(n), illustrating that D&C isn't always the fastest option even when it applies.

Q2: Multiply two large integers represented as strings, better than $O(n^2)$. **Solution:** Karatsuba's algorithm splits each number into two halves and reduces 4 recursive multiplications (naive divide-and-conquer) to 3, through the algebraic identity `(a+b)(c+d) = ac + bd + ((a+b)(c+d) - ac - bd)`. Recurrence: T(n) = 3T(n/2) + O(n) → O(n^1.585) by the Master Theorem.

Q3: Count inversions in an array (pairs where i < j but arr[i] > arr[j]) using divide and conquer. Input: `[8,4,2,1]`. **Solution:** Modify merge sort — every time an element from the right half is placed before remaining elements in the left half during the merge step, all remaining left-half elements form an inversion with it. For `[8,4,2,1]`: total inversions = **6** (all pairs are inverted since it's fully descending). Time: O(n log n), vs $O(n^2)$ brute force checking every pair.

---

21. Greedy Algorithms

Definition: At each step, make the locally optimal choice without reconsidering past choices, aiming for a globally optimal solution.

Intuition: Greedy only produces the correct global answer when the problem has the **greedy-choice property** (a locally optimal choice leads to a globally optimal solution) *and* **optimal substructure**. This must be proven or verified per problem — greedy can produce a confident, plausible-looking *wrong* answer with no error thrown. This is the most important thing to internalize about greedy algorithms: they are not a universal optimization tool, they are correct only for a specific class of problems.

Real-life example: Making change with the fewest coins using US currency (quarters, dimes, nickels, pennies) — always taking the largest coin that fits happens to give the optimal answer for this specific coin system. But with an arbitrary coin system (e.g., coins of 1, 3, 4 for a target of 6), greedy taking the largest first (4, then 1, then 1 = 3 coins) beats the intended optimal (3+3 = 2 coins) — wait, actually here greedy is suboptimal (3 coins vs the true optimum of 2 coins: 3+3). This is the canonical demonstration that greedy fails without provable structure — general coin change requires dynamic programming (section 23).

Diagram — activity selection (pick max non-overlapping intervals):

```
Activities (start,end): (1,4) (3,5) (0,6) (5,7) (3,9) (5,9) (6,10) (8,11) (8,12) (2,14) (12,16)

Greedy: sort by END time, always pick the next activity whose start >= last picked end.
Sorted by end: (1,4)(3,5)(0,6)(5,7)(3,9)(5,9)(6,10)(8,11)(8,12)(2,14)(12,16)
Pick (1,4) [end=4] -> next start>=4: (3,5) has start 3 <4 skip, (0,6) skip, (5,7) start=5>=4 PICK [end
-> next start>=7: (6,10) skip(start 6<7), (8,11) start=8>=7 PICK [end=11]
-> next start>=11: (8,12) skip, (2,14) skip, (12,16) start=12>=11 PICK
Result: (1,4),(5,7),(8,11),(12,16) -- 4 non-overlapping activities
```

Java code — Activity Selection:

```java
static int maxActivities(int[][] activities) {
    java.util.Arrays.sort(activities, (a, b) -> a[1] - b[1]);   // sort by end time
    int count = 1;
    int lastEnd = activities[0][1];
    for (int i = 1; i < activities.length; i++) {
```

```
        if (activities[i][0] >= lastEnd) {
            count++;
            lastEnd = activities[i][1];
        }
    }
    return count;
}
```

Brute-force vs optimized — activity selection:

Brute-force — try all subsets of activities, check which are non-overlapping, O(2ⁿ): combinatorially infeasible beyond small n.

Optimized — greedy, sort by end time, O(n log n) (code above) — provably optimal because always keeping the earliest-finishing option leaves maximum room for future activities.

Complexity summary: | Problem | Greedy Strategy | Time | Provably Optimal? | |—|—|—|—| | Activity Selection | Pick earliest finish time | O(n log n) | Yes | | Huffman Coding | Merge two lowest-frequency nodes | O(n log n) | Yes | | Fractional Knapsack | Highest value/weight ratio first | O(n log n) | Yes | | 0/1 Knapsack | Highest value/weight ratio first | O(n log n) | **No — requires DP** | | Kruskal's / Prim's MST | Cheapest valid edge first | O(E log E) / O(E log V) | Yes | | Dijkstra's shortest path | Closest unvisited vertex first | O((V+E) log V) | Yes, if weights non-negative |

How to verify greedy is safe for a new problem — the exchange argument: assume an optimal solution differs from the greedy choice at some step; show you can modify that optimal solution to match the greedy choice without making it worse. If this is always possible, greedy is provably safe.

Common mistakes: - Applying greedy to 0/1 knapsack (items can't be split) — value/weight ratio greedy fails here because you might waste capacity on a high-ratio item when a different combination fits better overall. This needs DP. - Not sorting by the correct key — activity selection requires sorting by *end* time, not start time or duration; sorting by the wrong key gives a plausible-looking but wrong answer. - Assuming greedy is always faster/simpler and skipping the correctness check — greedy's speed is only a benefit if it's actually correct for the problem. - Forgetting greedy's correctness must be proven per-problem, not assumed by analogy to a similar-looking problem that happened to work.

Interview tips: - If you propose a greedy solution, briefly justify *why* it's correct (exchange argument or citing the specific greedy-choice property) — this distinguishes genuine understanding from a lucky guess. - Know the canonical pairing: fractional knapsack = greedy works; 0/1 knapsack = greedy fails, needs DP. This is the standard interview illustration of greedy's limits. - If unsure whether greedy applies, say so explicitly and propose testing it against a brute-force solution on small examples — this is a legitimate and valued approach under time pressure.

Practice questions with solutions:

Q1: Given coin denominations and an amount, find the minimum number of coins using greedy — and identify when it fails. Input: coins=[1,5,10,25] (US), amount=41 vs coins=[1,3,4], amount=6. **Solution:** For US coins: greedy picks 25,10,5,1 = 4 coins, which is optimal for this canonical coin system. For `[1,3,4]`, amount=6: greedy picks 4, then 1, then 1 = 3 coins, but optimal is 3+3 = **2 coins**. This demonstrates greedy is denomination-system-dependent; general coin change requires DP (section 23).

Q2: Job sequencing to maximize profit, where each job has a deadline and profit, and only one job can run per time slot. **Solution:** Sort jobs by profit descending (greedy: always try to schedule the most profitable job first); for each job, place it in the latest available slot before its deadline (using a simple array or Union-Find for efficiency). This greedy choice is provably optimal —

proof follows an exchange argument similar to activity selection. Time: O(n log n) for sorting + O(n·deadline) or O(n α(n)) with Union-Find for slot assignment.

Q3: Gas station problem — determine if there's a starting gas station from which you can complete a circular route, given gas and cost arrays.

```
static int canCompleteCircuit(int[] gas, int[] cost) {
    int total = 0, tank = 0, start = 0;
    for (int i = 0; i < gas.length; i++) {
        int diff = gas[i] - cost[i];
        total += diff;
        tank += diff;
        if (tank < 0) { start = i + 1; tank = 0; }   // can't reach i from current start; reset
    }
    return total >= 0 ? start : -1;
}
```

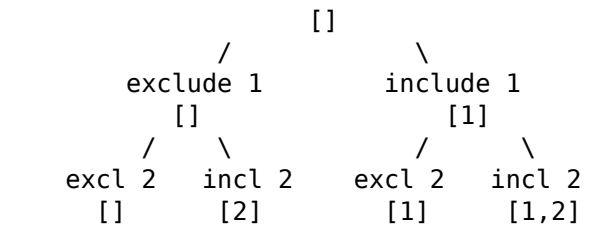
Solution walkthrough: Greedy insight — if the tank goes negative at station i, no station between the current start and i could have been a valid start either (they'd all fail for the same or worse reason), so jump the candidate start past i. If total gas ≥ total cost, a valid start is guaranteed to exist. Time: O(n), Space: O(1). ## 22. Backtracking

Definition: Build a solution incrementally; the moment a partial solution can't possibly lead to a valid complete solution, abandon it (backtrack) rather than continuing to explore that branch. It's exhaustive search with early pruning.

Intuition: Pruning invalid branches early avoids exploring huge portions of a search space that couldn't possibly work — often the practical difference between finishing in milliseconds and not finishing at all, even though worst-case complexity remains exponential. Backtracking is best understood as "trying a choice, recursing, then undoing that choice to try the next option" — the undo step is what makes it backtracking rather than plain recursive brute force.

Real-life example: Solving a maze by walking forward, and the instant you hit a dead end, walking back to the last junction and trying a different direction — instead of giving up entirely or re-exploring from the start each time.

Diagram — decision tree for generating subsets of `[1,2]` (include/exclude each element):



Leaves are the 2^2 = 4 subsets: [], [2], [1], [1,2]

Java code — general backtracking template:

```
static void backtrack(java.util.List<Integer> path, /* other state */ Object... context) {
    if (isComplete(path)) {
        recordSolution(path);
        return;
    }
    for (int choice : getChoices(/* context */)) {
        if (isValid(path, choice)) {
            path.add(choice);              // make choice
            backtrack(path, context);      // recurse
```

```
            path.remove(path.size() - 1);  // undo choice -- this IS backtracking
        }
    }
}
```

Java code — N-Queens:

```
static java.util.List<java.util.List<Integer>> solveNQueens(int n) {
    java.util.List<java.util.List<Integer>> solutions = new java.util.ArrayList<>();
    java.util.Set<Integer> cols = new java.util.HashSet<>();
    java.util.Set<Integer> diag1 = new java.util.HashSet<>();   // row - col
    java.util.Set<Integer> diag2 = new java.util.HashSet<>();   // row + col
    backtrackQueens(0, n, new java.util.ArrayList<>(), cols, diag1, diag2, solutions);
    return solutions;
}

static void backtrackQueens(int row, int n, java.util.List<Integer> placement,
        java.util.Set<Integer> cols, java.util.Set<Integer> diag1, java.util.Set<Integer> diag2,
        java.util.List<java.util.List<Integer>> solutions) {
    if (row == n) { solutions.add(new java.util.ArrayList<>(placement)); return; }
    for (int col = 0; col < n; col++) {
        if (cols.contains(col) || diag1.contains(row - col) || diag2.contains(row + col)) contin
        cols.add(col); diag1.add(row - col); diag2.add(row + col);
        placement.add(col);
        backtrackQueens(row + 1, n, placement, cols, diag1, diag2, solutions);
        cols.remove(col); diag1.remove(row - col); diag2.remove(row + col);   // undo
        placement.remove(placement.size() - 1);                               // undo
    }
}
```

Dry run — placing queens for n=4, showing the pruning:

```
row=0: try col=0 -> place. cols={0}
  row=1: try col=0 -> blocked(same col). try col=1 -> blocked(diag1=0). try col=2 -
> place. cols={0,2}
   row=2: try col=0 -> blocked(col). col=1 -> blocked(diag2: row-col relationships). col=2 -
> blocked(col).
           col=3 -> blocked(diag) -- ALL blocked, backtrack to row=1
  row=1: try col=3 -> place. cols={0,3}
    row=2: try col=1 -> place. cols={0,3,1}
      row=3: all columns blocked -- backtrack all the way
row=0: try col=1 -> place. (continues exploring...)
Eventually finds 2 valid solutions for n=4:
[1,3,0,2] and [2,0,3,1]  (column index for each row)
```

Brute-force vs optimized — N-Queens:

Brute-force — generate all n^n placements (one queen per row, any column), check validity after the fact: O(n^n) generation plus O(n^2) validation each — wildly infeasible even for n=10.

Optimized — backtracking with pruning (checking column/diagonal conflicts as you place each queen, not after): still exponential worst case, but prunes invalid branches immediately, making n up to ~15-20 tractable in practice (code above).

Complexity summary: | Problem | Complexity | Notes | |—|—|—| | N-Queens | O(n!) worst case, heavily pruned in practice | Pruning via column/diagonal sets is what makes it tractable | | Subsets | O(2^n) | Exactly 2^n subsets exist; no pruning possible, this is optimal | | Permutations | O(n!) | Exactly n! permutations exist | | Sudoku Solver | O(9^(empty cells)) worst case, heavily

pruned | Constraint checking prunes almost all branches in practice |

Common mistakes: - Forgetting the “undo” step (backtracking) after the recursive call — without it, state leaks between branches and produces wrong results. - Checking validity only at the end instead of pruning early — turns backtracking into plain brute force, losing all the performance benefit. - Not copying the path/list when recording a solution (`new ArrayList<>(path)`) — storing the same mutable reference means all recorded “solutions” end up reflecting only the final state. - Using backtracking when a greedy or DP solution exists and is more efficient — backtracking should be reached for once you’ve confirmed the problem genuinely requires exploring multiple possibilities, not before.

Interview tips: - Immediately name backtracking when you see “generate all,” “find all valid arrangements,” or a puzzle-constraint problem (N-Queens, Sudoku, word search). - Emphasize the pruning step in your explanation — interviewers want to see you understand *why* backtracking beats brute force, not just that you can write recursive code. - Mention time complexity honestly as exponential worst-case, while noting that pruning drastically reduces the actually-explored nodes in practice — this nuance shows depth.

Practice questions with solutions:

Q1: Generate all permutations of an array. Input: [1,2,3].

```
static java.util.List<java.util.List<Integer>> permute(int[] nums) {
    java.util.List<java.util.List<Integer>> result = new java.util.ArrayList<>();
    backtrackPermute(nums, new java.util.ArrayList<>(), new boolean[nums.length], result);
    return result;
}
static void backtrackPermute(int[] nums, java.util.List<Integer> path, boolean[] used,
        java.util.List<java.util.List<Integer>> result) {
    if (path.size() == nums.length) { result.add(new java.util.ArrayList<>(path)); return; }
    for (int i = 0; i < nums.length; i++) {
        if (used[i]) continue;
        used[i] = true;
        path.add(nums[i]);
        backtrackPermute(nums, path, used, result);
        used[i] = false;
        path.remove(path.size() - 1);
    }
}
```

Solution walkthrough: A `used[]` array prevents reusing the same index twice within one permutation. Produces all 3!=6 permutations of `[1,2,3]`. Time: O(n·n!), Space: O(n) recursion depth.

Q2: Solve a word search in a grid — does the word exist as a path of adjacent cells? Input: grid with “ABCE”,“SFCS”,“ADEE”, word=“ABCCED”.

```
static boolean exist(char[][] board, String word) {
    for (int r = 0; r < board.length; r++)
        for (int c = 0; c < board[0].length; c++)
            if (backtrackSearch(board, word, r, c, 0)) return true;
    return false;
}
static boolean backtrackSearch(char[][] board, String word, int r, int c, int idx) {
    if (idx == word.length()) return true;
    if (r < 0 || c < 0 || r >= board.length || c >= board[0].length || board[r][c] != word.charA
    char temp = board[r][c];
    board[r][c] = '#';    // mark visited (avoid revisiting within this path)
    boolean found = backtrackSearch(board, word, r+1, c, idx+1) ||
```

```
                    backtrackSearch(board, word, r-1, c, idx+1) ||
                    backtrackSearch(board, word, r, c+1, idx+1) ||
                    backtrackSearch(board, word, r, c-1, idx+1);
    board[r][c] = temp;    // undo the mark -- backtrack
    return found;
}
```

Solution walkthrough: Marking and unmarking `board[r][c]` is the backtracking step — it prevents reusing a cell within the current path while allowing it to be reused in a different path attempt. Result: **true**. Time: O(rows·cols·4^(word length)) worst case, heavily pruned in practice.

Q3: Solve a Sudoku puzzle (9x9 grid, fill empty cells so each row/column/3x3 box contains 1-9 exactly once). **Solution:** Backtrack cell by cell; for each empty cell, try digits 1-9, checking row/column/box validity before placing (early pruning); recurse to the next empty cell; if all subsequent cells fail, undo and try the next digit. Time: O(9^(empty cells)) worst case, but constraint checking prunes the vast majority of branches, making real puzzles solve quickly in practice.

---

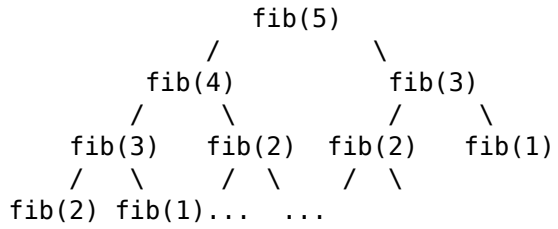
23. Dynamic Programming

Definition: Applies when a problem has **overlapping subproblems** (the same subproblem is solved repeatedly under plain recursion) and **optimal substructure** (the optimal solution to the whole is built from optimal solutions to subproblems). DP's core trick: solve each subproblem once, cache the result, reuse it — trading memory for eliminating redundant computation.

Intuition: Naive recursive Fibonacci is O(2^n) because `fib(5)` calls `fib(4)` and `fib(3)`, and `fib(4)` *also* calls `fib(3)` — the same subproblem is recomputed from scratch, compounding exponentially. Caching each result the first time it's computed collapses this to O(n). Every DP problem is this same idea applied to a different recurrence.

Real-life example: A shipping company precomputing the shortest route between every pair of major cities once, storing the results, instead of recalculating the route from scratch every single time a customer ships between the same two cities.

Diagram — Fibonacci call tree, showing redundant recomputation:




```
fib(3) computed 2 times, fib(2) computed 3 times -- redundancy compounds
exponentially with depth. Memoization caches each result on first computation.
```

Java code — Top-down (memoization) vs Bottom-up (tabulation):

```
// Top-down: recursion + cache
static int fibMemo(int n, int[] memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    memo[n] = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);
    return memo[n];
}
```

```java
// Bottom-up: iterative, build from base cases
static int fibTab(int n) {
    if (n <= 1) return n;
    int[] dp = new int[n + 1];
    dp[1] = 1;
    for (int i = 2; i <= n; i++) dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
}

// Space-optimized bottom-up: O(1) space, since only the last 2 values matter
static int fibOptimized(int n) {
    if (n <= 1) return n;
    int prev2 = 0, prev1 = 1;
    for (int i = 2; i <= n; i++) {
        int curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

How to approach any DP problem — the 5-step method: 1. Define the state — what does `dp[i]` (or `dp[i][j]`) actually represent? The hardest and most important step. 2. Find the recurrence — how does `dp[i]` relate to smaller states? 3. Identify base cases. 4. Decide computation order (bottom-up) or add memoization (top-down). 5. Optimize space once correctness is established (often O(n)→O(1) by keeping only the last few states, as in `fibOptimized` above).

Java code — 0/1 Knapsack (space-optimized to 1D):

```java
static int knapsack(int[] weights, int[] values, int capacity) {
    int n = weights.length;
    int[] dp = new int[capacity + 1];
    for (int i = 0; i < n; i++) {
        for (int w = capacity; w >= weights[i]; w--) {   // reverse order avoids reusing item i
            dp[w] = Math.max(dp[w], dp[w - weights[i]] + values[i]);
        }
    }
    return dp[capacity];
}
```

Dry run — `knapsack(weights=[1,3,4], values=[15,20,30], capacity=4):`

```
dp = [0,0,0,0,0] initially (indices 0..4)

item0 (w=1,v=15): for w=4..1:
 dp[4]=max(0, dp[3]+15)=15; dp[3]=max(0,dp[2]+15)=15; dp[2]=max(0,dp[1]+15)=15; dp[1]=max(0,dp[0]+
  dp = [0,15,15,15,15]

item1 (w=3,v=20): for w=4..3:
  dp[4]=max(15, dp[1]+20)=max(15,35)=35; dp[3]=max(15, dp[0]+20)=max(15,20)=20
  dp = [0,15,15,20,35]

item2 (w=4,v=30): for w=4..4:
  dp[4]=max(35, dp[0]+30)=max(35,30)=35
  dp = [0,15,15,20,35]
```

```
Final answer: dp[4] = 35  (take item0 + item1: weight 1+3=4, value 15+20=35)
```

Brute-force vs optimized — 0/1 Knapsack:

Brute-force — try all 2^n subsets of items, check which fit and maximize value, $O(2^n)$:

```java
// Recursive: for each item, branch into "take it" or "skip it" -- no caching
static int knapsackBrute(int[] w, int[] v, int cap, int i) {
    if (i == w.length || cap == 0) return 0;
    int skip = knapsackBrute(w, v, cap, i + 1);
    int take = (w[i] <= cap) ? v[i] + knapsackBrute(w, v, cap - w[i], i + 1) : 0;
    return Math.max(skip, take);
}
```

Optimized — DP, O(n × capacity) (code above) — the state `(item index, remaining capacity)` overlaps massively across the brute-force recursion tree; DP caches each distinct `(i, capacity)` pair once instead of recomputing it.

Complexity summary — classic DP problems: | Problem | State Definition | Time | Space (optimized) | |—|—|—|—| | Fibonacci | dp[i] = ith Fibonacci number | O(n) | O(1) | | 0/1 Knapsack | dp[w] = max value with capacity w | O(n·W) | O(W) | | Longest Common Subsequence | dp[i][j] = LCS length of first i, j chars | O(n·m) | O(min(n,m)) | | Longest Increasing Subsequence | dp[i] = LIS length ending at i | $O(n^2)$, or O(n log n) with binary search | O(n) | | Edit Distance | dp[i][j] = min ops to transform first i chars to first j chars | O(n·m) | O(min(n,m)) | | Coin Change (min coins) | dp[amount] = min coins for that amount | O(amount × coins) | O(amount) |

Java code — Longest Common Subsequence:

```java
static int lcs(String a, String b) {
    int n = a.length(), m = b.length();
    int[][] dp = new int[n + 1][m + 1];
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (a.charAt(i-1) == b.charAt(j-1)) dp[i][j] = dp[i-1][j-1] + 1;
            else dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[n][m];
}
```

Common mistakes: - Not correctly identifying the state — the single most common DP failure mode; if `dp[i]` doesn't capture enough information to compute `dp[i+1]` correctly, the whole solution is wrong regardless of how carefully the rest is coded. - Iterating in the wrong order for space-optimized knapsack (must go from high capacity to low, so each item is only used once per pass — see the knapsack code's reverse loop). - Confusing DP with divide-and-conquer — if subproblems don't overlap, plain D&C is simpler and doesn't need a cache. - Assuming a recurrence is correct without testing it on a small example — always trace `dp[1]`, `dp[2]`, `dp[3]` by hand before trusting the pattern.

Interview tips: - Always state the recurrence relation and the meaning of your DP state out loud before coding — this is what interviewers are actually evaluating, not syntax. - Start with the brute-force recursive solution, identify the overlapping subproblems explicitly, then add memoization — this narrative demonstrates understanding rather than pattern-matching a memorized solution. - Mention space optimization (2D→1D, or O(n)→O(1)) as a follow-up improvement even if you implement the full 2D version first — shows you understand *why* it's possible (only recent rows/values are needed).

Practice questions with solutions:

Q1: Coin Change — minimum number of coins to make an amount. Input: coins=[1,2,5],

amount=11.

```
static int coinChange(int[] coins, int amount) {
    int[] dp = new int[amount + 1];
    java.util.Arrays.fill(dp, amount + 1);   // "infinity" sentinel
    dp[0] = 0;
    for (int a = 1; a <= amount; a++) {
        for (int coin : coins) {
            if (coin <= a) dp[a] = Math.min(dp[a], dp[a - coin] + 1);
        }
    }
    return dp[amount] > amount ? -1 : dp[amount];
}
```

Solution walkthrough: `dp[a]` = fewest coins to make amount a, built from smaller amounts. For amount=11: 5+5+1 = **3 coins**. Time: O(amount × coins), Space: O(amount).

Q2: Longest Increasing Subsequence. Input: `[10,9,2,5,3,7,101,18]`.

```
static int lengthOfLIS(int[] nums) {
    int[] dp = new int[nums.length];
    java.util.Arrays.fill(dp, 1);
    int maxLen = 1;
    for (int i = 1; i < nums.length; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[j] < nums[i]) dp[i] = Math.max(dp[i], dp[j] + 1);
        }
        maxLen = Math.max(maxLen, dp[i]);
    }
    return maxLen;
}
```

Solution walkthrough: `dp[i]` = length of the longest increasing subsequence ending exactly at index i. For the input, the LIS is `[2,3,7,101]` or `[2,3,7,18]`, length **4**. Time: O(n^2) (can be improved to O(n log n) with binary search + patience sorting), Space: O(n).

Q3: Edit Distance — minimum operations (insert/delete/replace) to convert one string to another. Input: "horse" → "ros".

```
static int minDistance(String a, String b) {
    int n = a.length(), m = b.length();
    int[][] dp = new int[n + 1][m + 1];
    for (int i = 0; i <= n; i++) dp[i][0] = i;
    for (int j = 0; j <= m; j++) dp[0][j] = j;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (a.charAt(i-1) == b.charAt(j-1)) dp[i][j] = dp[i-1][j-1];
            else dp[i][j] = 1 + Math.min(dp[i-1][j-1], Math.min(dp[i-1][j], dp[i][j-1]));
        }
    }
    return dp[n][m];
}
```

Solution walkthrough: Base cases handle converting to/from an empty string (pure insertions/deletions). For "horse"→"ros": **3 operations** (replace 'h'→'r', remove 'r', remove 'e'). Time: O(n·m), Space: O(n·m), reducible to O(min(n,m)).

---

24. Bit Manipulation

Definition: Operating directly on the binary representation of integers using bitwise operators — AND (&), OR (|), XOR (^), NOT (~), left shift (<<), right shift (>>).

Intuition: Bitwise operations run in true O(1) at the hardware level and let you pack boolean state, sets, or flags into a single integer. XOR's special property — `x ^ x = 0` and `x ^ 0 = x` — is the basis for several elegant tricks, most notably finding a single non-duplicate number in O(n) time and O(1) space.

Real-life example: A light switch panel where each switch is a single bit — checking if switch 3 is on (&), turning it on (|), or flipping it (^) are all direct, instant operations, unlike scanning a list of switch states one by one.

Diagram — how XOR cancels duplicates:

```
Array: [4, 1, 2, 1, 2]
Binary: 4=100, 1=001, 2=010, 1=001, 2=010

XOR all together:
100 ^ 001 = 101
101 ^ 010 = 111
111 ^ 001 = 110
110 ^ 010 = 100  = 4

Every number that appears twice cancels itself out (x^x=0);
only the single non-duplicate (4) survives.
```

Java code — core bit tricks:

```java
// Check power of 2: a power of 2 has exactly one bit set
static boolean isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}

// Count set bits (Brian Kernighan's algorithm) -- O(number of set bits), not O(32)
static int countBits(int n) {
    int count = 0;
    while (n != 0) {
        n &= (n - 1);   // clears the lowest set bit
        count++;
    }
    return count;
}

// Find the single number where every other number appears twice
static int singleNumber(int[] nums) {
    int result = 0;
    for (int x : nums) result ^= x;
    return result;
}

// Get, set, clear, toggle bit i
static int getBit(int n, int i)   { return (n >> i) & 1; }
static int setBit(int n, int i)   { return n | (1 << i); }
static int clearBit(int n, int i) { return n & ~(1 << i); }
static int toggleBit(int n, int i){ return n ^ (1 << i); }
```

Dry run — `countBits(13)`, binary `1101`:

```
n=1101 (13)
n & (n-1) = 1101 & 1100 = 1100 (12) -> count=1
n & (n-1) = 1100 & 1011 = 1000 (8)  -> count=2
n & (n-1) = 1000 & 0111 = 0000 (0)  -> count=3
n=0, loop ends. Total set bits = 3
```

Brute-force vs optimized — count set bits in a number.

Brute-force — check all 32 bits individually, O(32) = O(1) but wasteful for sparse numbers:

```java
static int countBitsBrute(int n) {
    int count = 0;
    for (int i = 0; i < 32; i++) if (((n >> i) & 1) == 1) count++;
    return count;
}
```

Optimized — Brian Kernighan's, O(number of set bits) instead of always O(32): (code above) — skips directly from one set bit to the next instead of checking every position.

Complexity summary: | Operation | Complexity | |—|—| | Any single bitwise op (&, |, ^, ~, «, ») | O(1) | | Check power of 2 | O(1) | | Count set bits (naive, check all 32 bits) | O(32) = O(1) but constant-heavy | | Count set bits (Brian Kernighan's) | O(number of set bits) | | Find single non-duplicate via XOR | O(n) | | Bitmask DP over subsets of n items | O(2^n × n) typically |

Bitmask DP: represent a subset of n items as an n-bit integer, where bit i = 1 means item i is included. This turns "iterate over all subsets" problems (like Traveling Salesman for small n) into DP over 2^n states — feasible for n up to roughly 20. This is the standard technique for TSP-style problems in competitive programming, and worth naming even if a full implementation isn't expected.

Common mistakes: - Confusing ^ (XOR) with ** (exponentiation, which doesn't exist as an operator in Java) — a common mistake for people coming from Python. - Off-by-one on shift amounts, or forgetting operator precedence — `n >> i & 1` may not parenthesize the way you expect; always parenthesize bitwise expressions explicitly. - Using bit tricks where plain boolean logic would be equally fast and far more readable — bit tricks trade readability for speed; that trade should be justified, not reflexive. - Forgetting Java's >> is arithmetic shift (preserves sign) while >>> is logical shift (fills with 0) — using the wrong one on negative numbers gives unexpected results.

Interview tips: - Reach for bit manipulation explicitly when a problem is phrased in terms of "single number," "count bits," "subsets," or explicitly mentions binary representation. - Mention bitmask DP by name for small-n subset/permutation optimization problems (like TSP) even if you don't implement the full solution — it signals awareness of the technique. - For everyday code outside of explicitly bit-related problems, prefer readable boolean logic — don't force bit tricks into contexts where they reduce clarity without a real performance need.

Practice questions with solutions:

Q1: Find the two numbers that appear once in an array where every other number appears exactly twice. Input: [1,2,1,3,2,5].

```java
static int[] singleNumberII(int[] nums) {
    int xorAll = 0;
    for (int x : nums) xorAll ^= x;               // xorAll = a ^ b (the two unique numbers)
    int diffBit = xorAll & (-xorAll);              // isolate the lowest set bit (where a and b
    int a = 0, b = 0;
    for (int x : nums) {
        if ((x & diffBit) != 0) a ^= x; else b ^= x;   // partition into two groups by that bit
    }
```

```
    return new int[]{a, b};
}
```

Solution walkthrough: XORing everything cancels all duplicates, leaving `a^b`. The lowest set bit of `a^b` marks a position where a and b differ, letting you split the array into two groups (each containing one unique number plus its own duplicates, which cancel out). Result: **[3, 5]**. Time: O(n), Space: O(1).

Q2: Reverse the bits of a 32-bit unsigned integer.

```
static int reverseBits(int n) {
    int result = 0;
    for (int i = 0; i < 32; i++) {
        result <<= 1;
        result |= (n & 1);
        n >>= 1;
    }
    return result;
}
```

Solution walkthrough: Shift `result` left to make room, then OR in the lowest bit of `n`, then shift `n` right to expose its next bit — repeated 32 times reverses the full bit order. Time: O(32) = O(1), Space: O(1).

Q3: Given an integer, determine if it's a power of 4 (not just power of 2).

```
static boolean isPowerOfFour(int n) {
    return n > 0 && (n & (n - 1)) == 0 && (n & 0x55555555) != 0;
}
```

Solution walkthrough: First check power of 2 (`n & (n-1) == 0`). Then check the single set bit lands on an even position — `0x55555555` has 1s at every even bit position (bit 0, 2, 4, ...), which are exactly the valid positions for powers of 4 (1, 4, 16, 64, ...). Time: O(1), Space: O(1). ## 25. Interview Playbook — Pattern Recognition

The real skill interviews test is recognizing which pattern a problem belongs to, not memorizing solutions. Use this as a lookup table when stuck.

Signal in the problem	Likely pattern	Section
Sorted array, need a pair/triplet with a condition	Two pointers	3
Subarray/substring with a "contiguous" constraint	Sliding window	3
"Find minimum X such that..." over a monotonic condition	Binary search on the answer	10
Tree/graph, need shortest path (unweighted)	BFS	16
Tree/graph, need to explore all paths / detect cycle	DFS	16
Weighted graph, shortest path, non-negative weights	Dijkstra	17
"All subsets / permutations / combinations"	Backtracking	22
Overlapping subproblems, optimization ("min/max/count ways")	Dynamic programming	23

Signal in the problem	Likely pattern	Section
Interval scheduling, “earliest/cheapest first” provably works	Greedy	21
“k-th largest/smallest,” “top k,” “median of stream”	Heap	13
Range sum/min/max query + frequent updates	Segment tree / Fenwick tree	19
Prefix matching, autocomplete, many-pattern search	Trie	15
Connectivity queries as edges are added incrementally	Union-Find	18
Fast lookup or frequency counting, no ordering needed	Hash map/set	14
LIFO — nested structures, undo, next-greater-element	Stack (often monotonic)	6
FIFO — level order, task scheduling	Queue	7
Need sorted order maintained dynamically with inserts	Balanced BST / heap	12, 13
“Connect all points/cities with minimum total cost”	MST (Kruskal’s/Prim’s)	17
Binary representation, “single number,” subset masks	Bit manipulation	24
Divide problem into independent, non-overlapping halves	Divide and conquer	20

General debugging approach for any DSA problem: 1. Clarify constraints first — size of n, value ranges, duplicates allowed, sorted or not. These determine which complexity is acceptable and rule out approaches immediately. 2. State the brute-force solution and its complexity before optimizing — this anchors you and often reveals the exact bottleneck to attack. 3. Ask what’s making it slow: usually redundant work (→ DP/memoization, or hashing) or an unnecessary full scan (→ two pointers/sliding window/binary search). 4. Trace through a small example by hand before coding — catches logic errors cheaply, before they cost debugging time. 5. Check edge cases explicitly: empty input, single element, all duplicates, already sorted/reverse sorted, negative numbers if applicable.

How interviewers actually grade (beyond “does it compile”): - Can you state and justify time/space complexity unprompted? - Do you communicate your approach before diving into code, so a wrong direction can be caught early? - Do you test your own code with a trace-through, not just submit and hope? - Do you know the follow-up optimization even if you don’t have time to implement it (e.g., “this is $O(n^2)$ but could be O(n log n) with a heap”)?

---

26. Master Complexity Cheat Sheet

Data structures:

Structure	Access	Search	Insert	Delete	Spac
Array	O(1)	O(n)	O(n)	O(n)	O(n)
Dynamic Array (ArrayList)	O(1)	O(n)	O(1) amortized	O(n)	O(n)
Linked List	O(n)	O(n)	O(1)*	O(1)*	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)
Hash Table	—	O(1) avg / O(n) worst	O(1) avg	O(1) avg	O(n)
BST (balanced)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
BST (unbalanced)	O(n) worst	O(n) worst	O(n) worst	O(n) worst	O(n)
Heap	O(1) min/max only	O(n)	O(log n)	O(log n)	O(n)
Trie	—	O(m)	O(m)	O(m)	O(al
Segment Tree	—	O(log n) query	O(log n)	O(log n)	O(n)
Union-Find (optimized)	—	O(α(n)) find	O(α(n))	N/A	O(n)

*Linked list insert/delete is O(1) only with a pointer already at the position; O(n) if you must search for it first.

Algorithms:

Algorithm Family	Typical Complexity
Comparison sorting (optimal)	O(n log n)
Non-comparison sorting (counting/radix)	O(n+k)
Binary search	O(log n)
BFS/DFS on graphs	O(V+E)
Dijkstra (binary heap)	O((V+E) log V)
Bellman-Ford	O(V·E)
Floyd-Warshall (all-pairs)	$O(V^3)$
MST (Kruskal's/Prim's)	O(E log V)
Union-Find operations (optimized)	O(α(n)) ≈ O(1) amortized
Dynamic programming	Problem-specific, typically O(states × transitions)
Backtracking	Exponential worst case, pruning-dependent in practice

Growth rate reference (fastest to slowest):

```
O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)
```

---

27. Practice Roadmap

Reading explains the "why." Only solving problems builds the pattern recognition reading alone can't provide. Suggested order, roughly 15-25 problems per stage before moving on:

1. **Arrays & Strings** — two pointers, sliding window, prefix sums. Most later topics build on comfort with array manipulation.
2. **Hashing** — frequency counting, two-sum family. Highest-leverage early topic; turns brute force into single-pass solutions constantly.
3. **Linked Lists, Stacks, Queues** — reversal, cycle detection, monotonic stack problems, BFS setup.
4. **Recursion & Sorting/Searching** — build recurrence intuition before trees make recursion mandatory.
5. **Trees & BSTs** — traversals first, then BST-specific problems (validate BST, kth smallest, LCA).
6. **Heaps** — top-k problems, merge k sorted lists, median maintenance.

7. **Graphs** — BFS/DFS first, then shortest path and MST once traversal is automatic.
8. **Backtracking** — subsets, permutations, N-Queens, Sudoku, word search.
9. **Dynamic Programming** — start with 1D DP (Fibonacci-style, climbing stairs), then 2D (knapsack, LCS, edit distance), then DP on trees/graphs.
10. **Advanced structures** (Trie, Segment Tree, Union-Find, Bit Manipulation) — pull these in as specific problems demand them rather than studying in isolation; they're tools you reach for once you recognize the pattern, not a checklist to memorize up front.

An honest note on difficulty: DP and graph algorithms are where most learners plateau, because success depends on recognizing structure in a novel problem rather than applying a memorized template. The only real fix is volume — enough problems that the underlying patterns (overlapping subproblems, optimal substructure, when BFS beats DFS, when greedy actually holds) become recognizable on sight rather than derived from scratch every time.

Core insight: every structure in this course exists because of a single trade-off — you're always paying in one resource (time, memory, or code complexity) to save in another. Understanding *why* a structure has its complexity is what transfers to problems you've never seen; memorized code does not.

Key principles: - State brute-force and complexity before optimizing — it anchors your reasoning and reveals the actual bottleneck. - Recognize the pattern (this course's section 25 table) before reaching for code. - Prove greedy correctness (exchange argument) rather than assuming it — greedy fails silently more often than any other technique here. - DP is recursion plus caching for overlapping subproblems — if subproblems don't overlap, it's plain divide-and-conquer instead.

Practical action items: - Work through the practice roadmap in order; don't skip to DP/graphs before arrays/hashing are automatic. - For every practice question in this course, re-derive the solution from memory before checking it against what's written here. - Time yourself on medium-difficulty problems (25-35 minutes) once fundamentals are solid — recognizing patterns quickly under time pressure is a distinct skill from solving them untimed.

Next question worth asking yourself: which of the 27 topics above would you fail to reproduce from memory right now, with no notes? Start your practice there — that gap is the highest-leverage thing to close next.