DSA Interview Questions for Infosys Specialist Programmer

. Array

1. Find the largest and smallest element in an array

Problem
Given an integer array, find the maximum and minimum elements.

Approach (detailed)

- Walk the array once and keep two variables: currentMin and currentMax.
- Initialize both to the first element (or use Integer.MAX_VALUE / MIN_VALUE carefully).
- For each element, compare and update currentMin/currentMax.
- This is a single-pass O(n) solution and is optimal in time. There is a slight variation (pairwise comparison) that reduces the number of comparisons from ~2n to ~1.5n on average — useful for micro-optimization.

Implementation notes

- Handle empty arrays (throw IllegalArgumentException or return sentinel).
- Use long if values might overflow int operations.

Input / Output
Input: [3, 1, 7, 0, -2, 7]
Output: min = -2, max = 7

Java code

```java
import java.util.*;

public class MinMax {
	public static int[] findMinMax(int[] a) {
		if (a == null || a.length == 0) throw new IllegalArgumentException("Array must be non-empty");
		int min = a[0], max = a[0];
		for (int i = 1; i < a.length; i++) {
			if (a[i] < min) min = a[i];
			if (a[i] > max) max = a[i];
		}
		return new int[]{min, max};
	}

	// simple driver
	public static void main(String[] args) {
		int[] a = {3,1,7,0,-2,7};
```

```
                int[] res = findMinMax(a);
                System.out.println("min = " + res[0] + ", max = " + res[1]);
        }
}
```

Interviewer discussion

- Explain initialization choice for empty array handling.
- Discuss pairwise comparison method (process elements in pairs to reduce comparisons).
- Discuss numeric types if array contains long or BigInteger.

Optimized solution (pairwise comparisons)

- Iterate in pairs: for each pair (x,y) compare x and y first, then compare the smaller with min and larger with max — reduces comparisons.

Complexity

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

---

2. Reverse an array

Problem
Reverse the elements of an array in-place.

Approach (detailed)

- Swap elements symmetric around the center: i from 0..(n/2 -1), swap a[i] with a[n-1-i].
- This uses two-pointer technique and is in-place.

Implementation notes

- For immutable arrays (or language constraints), produce a new array with reversed order.

Input / Output
Input: [1,2,3,4,5]
Output: [5,4,3,2,1]

Java code

```java
import java.util.*;

public class ReverseArray {
```

```
	public static void reverse(int[] a) {
		if (a == null) return;
		int i = 0, j = a.length - 1;
		while (i < j) {
			int t = a[i]; a[i] = a[j]; a[j] = t;
			i++; j--;
		}
	}
	public static void main(String[] args) {
		int[] a = {1,2,3,4,5};
		reverse(a);
		System.out.println(Arrays.toString(a));
	}
}
```

Interviewer discussion

- Talk about in-place vs returning new array tradeoffs.
- Discuss reversing subarray, or reversing words in a sentence (two-level reversal trick).

Complexity

- Time: O(n)
- Space: O(1) (in-place)

---

3. Rotate an array by k steps

Problem
Rotate the array to the right by k steps (or left, clarify with interviewer). Example: rotate right by 2: [1,2,3,4,5] -> [4,5,1,2,3].

Approach (detailed)

- Several approaches:
  1. Naive: perform k single-step rotations (O(k*n) worst-case).
  2. Using extra array: copy elements to new positions in O(n) time and O(n) space.
  3. In-place using reversal trick (recommended):
  - Normalize k = k % n.
  - Reverse whole array.
  - Reverse first k elements.
  - Reverse remaining n-k elements.

Implementation notes

- Handle k >= n, negative k (left rotation) by normalization.

Input / Output
Input: a=[1,2,3,4,5], k=2 (right rotate)
Output: [4,5,1,2,3]

Java code (reversal trick)

```java
import java.util.*;

public class RotateArray {
	public static void rotate(int[] a, int k) {
		if (a == null || a.length == 0) return;
		int n = a.length;
		k = ((k % n) + n) % n; // normalize
		reverse(a, 0, n-1);
		reverse(a, 0, k-1);
		reverse(a, k, n-1);
	}
	private static void reverse(int[] a, int l, int r) {
		while (l < r) {
			int t = a[l]; a[l] = a[r]; a[r] = t;
			l++; r--;
		}
	}
	public static void main(String[] args) {
		int[] a = {1,2,3,4,5};
		rotate(a, 2);
		System.out.println(Arrays.toString(a));
	}
}
```

Interviewer discussion

- Explain why reversal works (proof by positions).
- Compare with using extra buffer and the tradeoff O(n) time O(n) space vs in-place O(1).

Complexity

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

---

4. Find missing number in an array (1 to n)

Problem

Given an array of size n-1 containing distinct numbers from 1..n with exactly one missing, find the missing number.

Approach (detailed)

- Use sum formula: expectedSum = n*(n+1)/2; subtract actual sum to get missing. Beware of overflow for large n — use long.
- Alternative: XOR method — XOR all numbers from 1..n and XOR with array elements; result is missing.

Implementation notes

- Validate ranges if input may be corrupted.

Input / Output
Input: n=5, a=[2,3,1,5]
Output: 4

Java code (XOR safe against overflow)

```java
public class MissingNumber {
	public static int missing(int[] a, int n) {
		int xor = 0;
		for (int i = 1; i <= n; i++) xor ^= i;
		for (int v : a) xor ^= v;
		return xor;
	}
	public static void main(String[] args) {
		int[] a = {2,3,1,5};
		System.out.println(missing(a,5)); // 4
	}
}
```

Interviewer discussion

- Explain overflow when using sum; show XOR avoids overflow.
- Discuss duplicate or corrupted inputs — how to detect multiple missing values.

Complexity

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

---

5. Find duplicate number in an array

Problem
Given an array of n+1 integers where each integer is between 1 and n (inclusive), there is at least one duplicate. Find any duplicate.

Approach (detailed)

- Multiple approaches:
  - Using HashSet/extra space: O(n) time, O(n) space.
  - Sorting the array and scanning adjacent: O(n log n) time, O(1) extra space (if allowed to modify array).
  - Floyd's Tortoise and Hare (cycle detection): treat indices/values as linked list to find duplicate in O(n) time and O(1) space without modifying array — commonly asked in interviews.

Floyd's cycle detection (sketch)

- Define f(x) = a[x] (or the value at index x). Start tortoise = a[0], hare = a[0]; advance tortoise by 1 step, hare by 2 steps until they meet; then find entry point of cycle — that value is duplicate.

Input / Output
Input: [3,1,3,4,2]
Output: 3 (one possible duplicate)

Java code (using Floyd)

```java
public class FindDuplicate {
	public static int findDuplicate(int[] a) {
		int tortoise = a[0];
		int hare = a[0];
		do {
			tortoise = a[tortoise];
			hare = a[a[hare]];
		} while (tortoise != hare);
		tortoise = a[0];
		while (tortoise != hare) {
			tortoise = a[tortoise];
			hare = a[hare];
		}
		return hare;
	}
	public static void main(String[] args) {
		int[] a = {3,1,3,4,2};
		System.out.println(findDuplicate(a));
	}
}
```

```

Interviewer discussion

- Explain why the array values/index mapping creates a cycle.
- Offer alternative solutions and tradeoffs (space vs time vs modifying input).

Complexity

- Time: O(n)
- Space: O(1) (Floyd) or O(n) (hash-based)

---

6. Find intersection of two arrays

Problem
Given two arrays, return their intersection (common elements). Define whether duplicates should be included (set intersection vs multiset intersection); clarify with interviewer.

Approach (detailed)

- If duplicates not required and order irrelevant: put smaller array into HashSet, iterate larger array and collect matches.
- If duplicates required (each element as many times as it appears in both), use HashMap to store counts, then iterate second array to emit min counts.
- If arrays sorted: use two-pointer merge-like approach O(n+m) time and O(1) extra space.

Implementation notes

- Clarify expected output format: array, list, unique set, sorted or original order.

Input / Output
Input: a=[1,2,2,3], b=[2,2,4]
Output (with duplicates): [2,2]

Java code (duplicates allowed)

```java
import java.util.*;

public class ArrayIntersection {
	public static List<Integer> intersect(int[] a, int[] b) {
		if (a.length > b.length) return intersect(b, a);
		Map<Integer,Integer> cnt = new HashMap<>();
		for (int x : a) cnt.put(x, cnt.getOrDefault(x,0)+1);
		List<Integer> res = new ArrayList<>();
		for (int y : b) {
```
```

```
                        Integer c = cnt.get(y);
                        if (c != null && c > 0) {
                                res.add(y);
                                cnt.put(y, c-1);
                        }
                }
                return res;
        }
        public static void main(String[] args) {
                int[] a = {1,2,2,3};
                int[] b = {2,2,4};
                System.out.println(intersect(a,b));
        }
}
```

Interviewer discussion

- Clarify expected behavior for duplicates and ordering.
- Mention sorting-based approach if arrays are large and memory is constrained.

Complexity

- Time: O(n + m)
- Space: O(min(n,m)) for hash-based approach

---

7. Find subarray with given sum

Problem
Given an array of non-negative integers and a target sum, find a contiguous subarray that sums to the target. (If negative numbers allowed, clarify and adapt.)

Approach (detailed)

- For non-negative numbers: sliding window / two-pointer technique — maintain current window sum and expand/contract to reach target.
- For arrays with negatives: sliding window doesn't work reliably; either use prefix sums + hashmap to find i,j where prefix[j]-prefix[i]=target (O(n) time, O(n) space), or use more complex algorithms.

Input / Output
Input: a=[1,2,3,7,5], target=12
Output: subarray [5,7] or indices (2,4) depending 0-indexing

Java code (non-negative numbers, return indices)

```java
import java.util.*;

public class SubarrayWithSum {
	public static int[] findWithSum(int[] a, int target) {
		int l = 0, curr = 0;
		for (int r = 0; r < a.length; r++) {
			curr += a[r];
			while (curr > target && l <= r) {
				curr -= a[l++];
			}
			if (curr == target) return new int[]{l, r};
		}
		return null;
	}
	public static void main(String[] args) {
		int[] a = {1,2,3,7,5};
		int[] res = findWithSum(a,12);
		System.out.println(Arrays.toString(res));
	}
}
```

Interviewer discussion

- Distinguish cases where numbers may be negative.
- Discuss prefix-sum + hashmap for negatives and how to return earliest solution.

Complexity

- Time: O(n)
- Space: O(1) for sliding-window (non-negative). For prefix-sum hashmap: O(n) space.

---

8. Maximum subarray sum (Kadane’s algorithm)

Problem
Find the contiguous subarray with the largest sum.

Approach (detailed)

- Kadane's algorithm maintains currentSum = max(a[i], currentSum + a[i]) and maxSoFar = max(maxSoFar, currentSum).
- Also track start/end indices if required: when currentSum becomes a[i], set tentative start=i; when maxSoFar updates set start/end accordingly.

Input / Output

Input: [-2,1,-3,4,-1,2,1,-5,4]
Output: maximum sum = 6 with subarray [4,-1,2,1]

Java code (with indices)

```java
public class Kadane {
	public static int[] maxSubarray(int[] a) {
		int maxSoFar = Integer.MIN_VALUE, curr = 0;
		int start = 0, s = 0, end = 0;
		for (int i = 0; i < a.length; i++) {
			curr += a[i];
			if (curr > maxSoFar) {
				maxSoFar = curr; start = s; end = i;
			}
			if (curr < 0) {
				curr = 0; s = i+1;
			}
		}
		return new int[]{maxSoFar, start, end};
	}
	public static void main(String[] args) {
		int[] a = {-2,1,-3,4,-1,2,1,-5,4};
		int[] r = maxSubarray(a);
		System.out.println("max="+r[0]+" start="+r[1]+" end="+r[2]);
	}
}
```

Interviewer discussion

- Explain why resetting current sum when negative is correct.
- Variants: require non-empty subarray or allow empty (sum 0) — handle accordingly.

Complexity

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

---

9. Find leaders in an array

Problem
An element is a leader if it is greater than all elements to its right. Return all leaders in order of appearance (or reverse order depending on requirement).

Approach (detailed)

- Scan from right to left keeping maxSoFar. If current element > maxSoFar, it's a leader; update maxSoFar.
- Collect leaders while scanning and reverse at the end if you want left-to-right order.

Input / Output
Input: [16,17,4,3,5,2]
Output: [17,5,2] (or [17,5,2] depending ordering preference)

Java code

```java
import java.util.*;

public class Leaders {
	public static List<Integer> leaders(int[] a) {
		List<Integer> out = new ArrayList<>();
		int max = Integer.MIN_VALUE;
		for (int i = a.length - 1; i >= 0; i--) {
			if (a[i] > max) { out.add(a[i]); max = a[i]; }
		}
		Collections.reverse(out);
		return out;
	}
	public static void main(String[] args) {
		int[] a = {16,17,4,3,5,2};
		System.out.println(leaders(a));
	}
}
```

Interviewer discussion

- Clarify ordering asked for. Explain right-to-left scan.

Complexity

- Time: O(n)
- Space: O(k) where k is number of leaders (O(n) worst-case)

---

10. Trapping Rainwater problem

Problem
Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Approach (detailed)

- Several solutions:
  - Brute force: for each index, find max left and max right, water = min(leftMax,rightMax) - height[i]. O(n^2).
  - Precompute leftMax and rightMax arrays: two passes to fill arrays then one pass to compute water. O(n) time, O(n) space.
  - Two-pointer optimized: maintain left and right pointers and running leftMax/rightMax; move the smaller side inward and accumulate water, achieving O(n) time and O(1) extra space.

Two-pointer intuition

- Water trapped at a point is limited by the smaller of the maximum boundaries on both sides; by moving the side with smaller boundary inward we can safely compute trapped water using the current boundary.

Input / Output
Input: [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6

Java code (two-pointer)

```java
public class TrappingRain {
	public static int trap(int[] h) {
		int l = 0, r = h.length - 1;
		int leftMax = 0, rightMax = 0, res = 0;
		while (l <= r) {
			if (h[l] <= h[r]) {
				if (h[l] >= leftMax) leftMax = h[l]; else res += leftMax - h[l];
				l++;
			} else {
				if (h[r] >= rightMax) rightMax = h[r]; else res += rightMax - h[r];
				r--;
			}
		}
		return res;
	}
	public static void main(String[] args) {
		int[] h = {0,1,0,2,1,0,1,3,2,1,2,1};
		System.out.println(trap(h));
	}
}
```

Interviewer discussion

- Explain why two-pointer is correct and how invariants hold.
- Discuss edge cases: flat bars, monotonic arrays, empty input.

Complexity

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

---

2. Strings

Below are full, interview-ready writeups for the 11–20 string problems. Each entry includes: problem statement, detailed approach with edge-cases, a runnable Java implementation (small class with main), sample input/output, interviewer discussion points, alternate/optimized solutions, and time & space complexity.

---

11. Check if a string is palindrome

Problem
Given a string, determine if it reads the same backward as forward. Consider whether to ignore case, punctuation and spaces — clarify with interviewer.

Approach (detailed)

- Two-pointer technique: left at 0 and right at n-1. Move inward comparing characters. If ignoring non-alphanumerics and case, skip non-alphanumerics and compare lowercase versions.
- Handle empty and single-character strings (palindrome).

Edge cases

- Unicode normalization, combining characters, and grapheme clusters if supporting full Unicode.

Input / Output
Input: "A man, a plan, a canal: Panama"
Output: true (when ignoring spaces and punctuation)

Java code

```java
public class IsPalindrome {
	public static boolean isPalindrome(String s) {
		if (s == null) return false;
		int l = 0, r = s.length() - 1;
```

```
                while (l < r) {
                        while (l < r && !Character.isLetterOrDigit(s.charAt(l))) l++;
                        while (l < r && !Character.isLetterOrDigit(s.charAt(r))) r--;
                        if (Character.toLowerCase(s.charAt(l)) !=
Character.toLowerCase(s.charAt(r))) return false;
                        l++; r--;
                }
                return true;
        }
        public static void main(String[] args) {
                System.out.println(isPalindrome("A man, a plan, a canal: Panama"));
        }
}
```

Interviewer discussion

- Clarify normalization (case/punctuation). Discuss Unicode concerns and tradeoffs between simplicity and full correctness.

Complexity

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

---

12. Reverse words in a string

Problem
Given a string containing words separated by spaces, reverse the order of the words. Handle extra spaces at ends and between words.

Approach (detailed)

- Trim leading/trailing spaces, split on whitespace, reverse array of words and join with single spaces.
- In-place approach for char array: reverse whole string, then reverse each word to restore word characters.

Input / Output
Input: " the sky is blue "
Output: "blue is sky the"

Java code (in-place char array approach sketch)

```java
import java.util.*;
```

```
public class ReverseWords {
	public static String reverseWords(String s) {
		if (s == null) return null;
		String[] parts = s.trim().split("\\s+");
		Collections.reverse(Arrays.asList(parts));
		return String.join(" ", parts);
	}
	public static void main(String[] args) {
		System.out.println("" + reverseWords("  the sky   is blue  ") + "");
	}
}
```

Interviewer discussion

- Discuss splitting vs manual parse to avoid extra memory. For large strings, prefer streaming/parsing to avoid intermediate arrays.

Complexity

- Time: O(n)
- Space: O(n) (due to split/array) or O(1) extra for in-place char-array method

---

13. Find first non-repeating character in a string

Problem
Return the index (or character) of the first non-repeating character in a string. If none exists, return -1 or a sentinel.

Approach (detailed)

- Two-pass solution: first build frequency count (array of size 256 for ASCII or HashMap for Unicode), second pass find first char with count 1.

Input / Output
Input: "leetcode"
Output: 'l' (index 0)

Java code

```java
import java.util.*;

public class FirstUniqueChar {
	public static int firstUniqChar(String s) {
```

```
        int[] cnt = new int[256];
        for (char c : s.toCharArray()) cnt[c]++;
        for (int i = 0; i < s.length(); i++) if (cnt[s.charAt(i)] == 1) return i;
        return -1;
    }
    public static void main(String[] args) {
        System.out.println(firstUniqChar("leetcode"));
    }
}
```

Interviewer discussion

- Discuss Unicode and memory tradeoffs; streaming option using LinkedHashMap to preserve order.

Complexity

- Time: O(n)
- Space: O(1) for fixed alphabet or O(k) for HashMap where k is distinct chars

---

14. Check if two strings are anagrams

Problem
Determine if two strings are anagrams (contain same characters with same counts). Clarify whether to ignore whitespace/case.

Approach (detailed)

- For limited alphabets: count frequency arrays and compare.
- For generic Unicode: sort both strings and compare (O(n log n)) or use HashMap counts (O(n)).

Input / Output
Input: s="anagram", t="nagaram"
Output: true

Java code (count approach for lowercase letters)

```java
import java.util.*;

public class AnagramCheck {
    public static boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) return false;
        int[] cnt = new int[26];
```

```
                for (int i = 0; i < s.length(); i++) {
                        cnt[s.charAt(i) - 'a']++;
                        cnt[t.charAt(i) - 'a']--;
                }
                for (int v : cnt) if (v != 0) return false;
                return true;
        }
        public static void main(String[] args) {
                System.out.println(isAnagram("anagram","nagaram"));
        }
}
```

Interviewer discussion

- Ask about character set and normalization. Sorting is simple but slower; counting is linear and preferred when alphabet is known.

Complexity

- Time: O(n)
- Space: O(1) for fixed alphabet

---

15. Longest common prefix

Problem
Given an array of strings, find the longest common prefix among them.

Approach (detailed)

- Horizontal scanning: take first string as prefix and iteratively trim using startsWith.
- Vertical scanning: check character by character across strings.
- Divide and conquer: split array and compute prefix recursively; useful for parallelization.

Input / Output
Input: ["flower","flow","flight"]
Output: "fl"

Java code (vertical scan)

```java
public class LongestCommonPrefix {
        public static String longestCommonPrefix(String[] strs) {
                if (strs == null || strs.length == 0) return "";
                for (int i = 0; i < strs[0].length(); i++) {
                        char c = strs[0].charAt(i);
```

```
                        for (int j = 1; j < strs.length; j++) {
                                if (i == strs[j].length() || strs[j].charAt(i) != c) return
strs[0].substring(0, i);
                        }
                }
                return strs[0];
        }
        public static void main(String[] args) {
                System.out.println(longestCommonPrefix(new
String[]{"flower","flow","flight"}));
        }
}
```

Interviewer discussion

- Explain tradeoffs between methods and how divide-and-conquer or binary search on prefix length can help when strings are long.

Complexity

- Time: O(S) where S is total chars across all strings (vertical), Space: O(1)

---

16. Longest substring without repeating characters

Problem
Find the length (and optionally the substring) of the longest substring without repeating characters.

Approach (detailed)

- Sliding window with a HashMap/array to track last seen index of characters. Move left pointer to max(lastSeen+1, left) when encountering a repeat.

Input / Output
Input: "abcabcbb"
Output: 3 ("abc")

Java code

```java
import java.util.*;

public class LongestUniqueSubstr {
        public static int lengthOfLongestSubstring(String s) {
                int[] last = new int[256]; Arrays.fill(last, -1);
```

```
                int res = 0, start = 0;
                for (int i = 0; i < s.length(); i++) {
                        start = Math.max(start, last[s.charAt(i)] + 1);
                        res = Math.max(res, i - start + 1);
                        last[s.charAt(i)] = i;
                }
                return res;
        }
        public static void main(String[] args) {
                System.out.println(lengthOfLongestSubstring("abcabcbb"));
        }
}
```

Interviewer discussion

- Discuss using array vs HashMap for Unicode. Mention recovering the substring (track start/end).

Complexity

- Time: O(n)
- Space: O(1) for fixed alphabet or O(k) otherwise

---

17. Count vowels and consonants

Problem
Given a string, count vowels and consonants (depending on language rules).

Approach (detailed)

- Iterate characters, classify via Character.isLetter and vowel set. Handle uppercase by lowercasing.

Input / Output
Input: "Hello World"
Output: vowels=3, consonants=7

Java code

```java
public class VowelConsonantCount {
        public static int[] count(String s) {
                int vowels = 0, cons = 0;
                for (char ch : s.toLowerCase().toCharArray()) {
                        if (!Character.isLetter(ch)) continue;
```

```
                        if ("aeiou".indexOf(ch) >= 0) vowels++; else cons++;
                }
                return new int[]{vowels, cons};
        }
        public static void main(String[] args) {
                int[] r = count("Hello World");
                System.out.println("vowels="+r[0]+" consonants="+r[1]);
        }
}
```

Interviewer discussion

- Clarify locale-specific vowels (y sometimes vowel) and Unicode letters.

Complexity

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

---

18. Implement strstr() (substring search)

Problem
Implement function to find the first occurrence of needle in haystack (return index or -1). Many algorithms: naive, KMP, Rabin-Karp.

Approach (detailed)

- KMP (Knuth-Morris-Pratt) builds LPS (longest proper prefix which is suffix) to skip comparisons and achieve O(n + m) time.
- Rabin-Karp uses rolling hash and is average O(n + m) but has hash collision considerations.

Input / Output
Input: haystack="hello", needle="ll"
Output: 2

Java code (simple KMP)

```java
import java.util.*;

public class StrStrKMP {
        public static int strStr(String hay, String need) {
                if (need.length() == 0) return 0;
                int[] lps = buildLPS(need);
```

```
                int i = 0, j = 0;
                while (i < hay.length()) {
                        if (hay.charAt(i) == need.charAt(j)) { i++; j++; if (j == need.length())
return i - j; }
                        else if (j > 0) j = lps[j-1]; else i++;
                }
                return -1;
        }
        private static int[] buildLPS(String p) {
                int n = p.length(); int[] lps = new int[n]; int len = 0; int i = 1;
                while (i < n) {
                        if (p.charAt(i) == p.charAt(len)) { lps[i++] = ++len; }
                        else if (len > 0) len = lps[len-1]; else lps[i++] = 0;
                }
                return lps;
        }
        public static void main(String[] args) {
                System.out.println(strStr("hello","ll"));
        }
}
```

Interviewer discussion

- Explain LPS array intuition and time complexity. Discuss when Rabin-Karp is appropriate.

Complexity

- Time: O(n + m)
- Space: O(m)

---

19. Longest Palindromic Substring

Problem
Find the longest palindromic substring in a given string.

Approach (detailed)

- Expand-around-center: for each center (2n-1 centers counting between chars), expand and keep longest — O(n^2) worst-case but simple.
- Manacher's algorithm yields O(n) time using clever palindrome radius transformations — more complex to write but optimal.

Input / Output
Input: "babad"
Output: "bab" or "aba"

Java code (expand-around-center)

```java
public class LongestPalSubstr {
	public static String longestPalindrome(String s) {
		if (s == null || s.length() < 1) return "";
		int start = 0, end = 0;
		for (int i = 0; i < s.length(); i++) {
			int len1 = expand(s, i, i);
			int len2 = expand(s, i, i+1);
			int len = Math.max(len1, len2);
			if (len > end - start + 1) {
				start = i - (len-1)/2;
				end = i + len/2;
			}
		}
		return s.substring(start, end+1);
	}
	private static int expand(String s, int l, int r) {
		while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) { l--; r++; }
		return r - l - 1;
	}
	public static void main(String[] args) {
		System.out.println(longestPalindrome("babad"));
	}
}
```

Interviewer discussion

- Explain center expansion and when Manacher's is worth implementing (very long strings / strict time limits).

Complexity

- Time: O(n^2) (expand), O(n) for Manacher's
- Space: O(1) (apart from result)

---

20. Minimum edit distance (Levenshtein)

Problem
Given two strings, compute the minimum number of operations (insert, delete, replace) required to convert one string into the other.

Approach (detailed)

- Dynamic programming: dp[i][j] = min operations to convert prefix A[0..i) to B[0..j).
Recurrence uses dp[i-1][j] (delete), dp[i][j-1] (insert), dp[i-1][j-1] (+0 if equal else +1 replace).
- Space optimized to two rows for O(min(n,m)) space.

Input / Output
Input: word1="kitten", word2="sitting"
Output: 3

Java code (space-optimized)

```java
import java.util.*;

public class EditDistance {
	public static int minDistance(String a, String b) {
		int n = a.length(), m = b.length();
		if (n < m) return minDistance(b, a); // ensure n >= m to reduce space
		int[] prev = new int[m+1], curr = new int[m+1];
		for (int j = 0; j <= m; j++) prev[j] = j;
		for (int i = 1; i <= n; i++) {
			curr[0] = i;
			for (int j = 1; j <= m; j++) {
				int cost = a.charAt(i-1) == b.charAt(j-1) ? 0 : 1;
				curr[j] = Math.min(Math.min(prev[j] + 1, curr[j-1] + 1), prev[j-1] +
cost);
			}
			int[] tmp = prev; prev = curr; curr = tmp;
		}
		return prev[m];
	}
	public static void main(String[] args) {
		System.out.println(minDistance("kitten","sitting"));
	}
}
```

Interviewer discussion

- Show derivation of recurrence and how to recover edit operations by storing choices.
Discuss weighted edits.

Complexity

- Time: O(n*m)
- Space: O(min(n,m)) after optimization

---

3. Searching & Sorting

Below are expanded, interview-ready writeups for problems 21–30 (Searching & Sorting). Each entry includes: a short contract (inputs/outputs, success conditions), detailed approach with edge-cases, one or more Java implementations (self-contained small classes), sample input/output, interviewer discussion points and tradeoffs, possible optimizations, and time/space complexity. These are more detailed than the earlier sections and suitable for a deep interview discussion.

---

21. Binary Search (iterative & recursive)

Contract

- Input: sorted array `a[]` of n elements and a target `x`.
- Output: index of `x` in `a` (any matching index), or -1 if not found.
- Success: returns a valid index when `x` exists; handles empty arrays.

Approach (detailed)

- Binary search repeatedly halves the search range using mid = left + (right-left)/2 to avoid overflow. Compare a[mid] to target and move left/right pointers. Use iterative for performance and recursion for clarity; both O(log n).
- Edge cases: duplicates (returns any index), empty array, integer overflow in mid computation, searching first/last element.

Java code (iterative + recursive)

```java
public class BinarySearch {
	public static int iterative(int[] a, int target) {
		int l = 0, r = a.length - 1;
		while (l <= r) {
			int mid = l + (r - l) / 2;
			if (a[mid] == target) return mid;
			if (a[mid] < target) l = mid + 1; else r = mid - 1;
		}
		return -1;
	}
	public static int recursive(int[] a, int l, int r, int target) {
		if (l > r) return -1;
		int mid = l + (r - l) / 2;
		if (a[mid] == target) return mid;
		if (a[mid] < target) return recursive(a, mid + 1, r, target);
		return recursive(a, l, mid - 1, target);
	}
```

```
	public static void main(String[] args) {
		int[] a = {1,2,4,7,9};
		System.out.println(iterative(a,7));
		System.out.println(recursive(a,0,a.length-1,4));
	}
}
```

Interviewer discussion

- Discuss variant for first/last occurrence, off-by-one bugs, and applications (search in monotonic functions). Mention using BigInteger or long for extreme ranges.

Complexity

- Time: O(log n)
- Space: O(1) iterative, O(log n) recursion stack for recursive version

---

22. First and last occurrence of an element (binary search variant)

Contract

- Input: sorted array `a[]` and target `x`.
- Output: pair of indices [first, last] where `x` appears (or [-1,-1] if not present).

Approach (detailed)

- Use two binary searches: one to find the leftmost index where a[i] == x (search boundary when a[mid] >= x), and one to find rightmost (search boundary when a[mid] <= x). Careful with loop invariants and mid calculation.

Java code

```java
import java.util.*;

public class FirstLastOccurrence {
	public static int findFirst(int[] a, int x) {
		int l = 0, r = a.length - 1, ans = -1;
		while (l <= r) {
			int m = l + (r - l) / 2;
			if (a[m] >= x) { if (a[m] == x) ans = m; r = m - 1; }
			else l = m + 1;
		}
		return ans;
	}
```

```
	public static int findLast(int[] a, int x) {
		int l = 0, r = a.length - 1, ans = -1;
		while (l <= r) {
			int m = l + (r - l) / 2;
			if (a[m] <= x) { if (a[m] == x) ans = m; l = m + 1; }
			else r = m - 1;
		}
		return ans;
	}
	public static void main(String[] args) {
		int[] a = {1,2,2,2,3,4};
		System.out.println(Arrays.toString(new int[]{findFirst(a,2), findLast(a,2)}));
	}
}
```

Interviewer discussion

- Demonstrate correctness by loop invariants. Mention STL equivalents (lower_bound/upper_bound) and use when counting frequency of a value.

Complexity

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

---

23. Find peak element in an array

Contract

- Input: array `a[]` where a[i] != a[i+1] (or allow equals if specified).
- Output: index i such that a[i] > a[i-1] and a[i] > a[i+1] (peak). Edges can be considered peaks if greater than neighbor.

Approach (detailed)

- Use binary search on monotonic slopes: if a[mid] < a[mid+1], peak is to the right; else peak is to the left (including mid). This finds any peak in O(log n).

Java code

```java
public class PeakElement {
	public static int findPeak(int[] a) {
		int l = 0, r = a.length - 1;
		while (l < r) {
```

```
                int m = l + (r - l) / 2;
                if (a[m] < a[m+1]) l = m + 1; else r = m;
            }
            return l;
        }
        public static void main(String[] args) {
            int[] a = {1,2,1,3,5,6,4};
            System.out.println(findPeak(a));
        }
}
```

Interviewer discussion

- Explain intuition: array is a sequence of increasing/decreasing runs; gradient points to a peak. Discuss ties or plateau handling if equal elements exist.

Complexity

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

---

24. Search in rotated sorted array

Contract

- Input: array `a[]` that was sorted and rotated at unknown pivot (no duplicates for simpler variant) and target `x`.
- Output: index of `x` or -1.

Approach (detailed)

- Modified binary search: at each mid determine which half is sorted. If left half sorted and x in that range, search left; else search right. Carefully handle duplicates if allowed (then worst-case degrades).

Java code

```java
public class SearchRotated {
        public static int search(int[] a, int target) {
            int l = 0, r = a.length - 1;
            while (l <= r) {
                int m = l + (r - l) / 2;
                if (a[m] == target) return m;
                if (a[l] <= a[m]) { // left sorted
```

```
                        if (a[l] <= target && target < a[m]) r = m - 1; else l = m + 1;
                    } else { // right sorted
                        if (a[m] < target && target <= a[r]) l = m + 1; else r = m - 1;
                    }
                }
                return -1;
        }
        public static void main(String[] args) {
                int[] a = {4,5,6,7,0,1,2};
                System.out.println(search(a,0));
        }
}
```

Interviewer discussion

- Discuss handling duplicates (e.g., when a[l]==a[m] you must advance l++ to skip equal values, causing worst-case O(n)). Also mention finding pivot explicitly then binary searching two halves as alternative.

Complexity

- Time: O(log n) without duplicates, O(n) worst-case with duplicates
- Space: O(1)

---

25. Find square root using binary search

Contract

- Input: non-negative integer `x`.
- Output: floor(sqrt(x)) as integer (or double with precision if requested).

Approach (detailed)

- Binary search integer range [0, x] using mid and compare mid*mid to x; be careful with overflow (use long). For fractional sqrt, binary search on double range with precision epsilon.

Java code (integer floor)

```java
public class IntSqrt {
        public static int mySqrt(int x) {
                if (x < 2) return x;
                int l = 1, r = x/2, ans = 1;
                while (l <= r) {
                        int m = l + (r - l) / 2;
```

```
                long sq = 1L * m * m;
                if (sq == x) return m;
                if (sq < x) { ans = m; l = m + 1; } else r = m - 1;
            }
            return ans;
        }
        public static void main(String[] args) { System.out.println(mySqrt(8)); }
}
```

Interviewer discussion

- Explain overflow avoidance. For floating sqrt, binary search stops when r-l < eps. Mention Newton's method as an alternative with faster convergence.

Complexity

- Time: O(log x)
- Space: O(1)

---

26. Merge Sort

Contract

- Input: array `a[]` of n elements.
- Output: sorted array (stable sort).

Approach (detailed)

- Divide-and-conquer: split array in half, recursively sort both halves, then merge. Merging uses two pointers and requires O(n) extra space. Stable and predictable O(n log n) time.

Java code (top-level)

```java
import java.util.*;

public class MergeSort {
        public static void sort(int[] a) { mergeSort(a, 0, a.length-1, new int[a.length]); }
        private static void mergeSort(int[] a, int l, int r, int[] tmp) {
            if (l >= r) return;
            int m = l + (r - l) / 2;
            mergeSort(a, l, m, tmp);
            mergeSort(a, m+1, r, tmp);
            int i = l, j = m+1, k = l;
            while (i <= m || j <= r) {
```

```
                        if (i <= m && (j > r || a[i] <= a[j])) tmp[k++] = a[i++]; else tmp[k++] =
a[j++];
                }
                for (k = l; k <= r; k++) a[k] = tmp[k];
        }
        public static void main(String[] args) { int[] a = {5,2,4,6,1,3}; sort(a);
System.out.println(Arrays.toString(a)); }
}
```

Interviewer discussion

- Discuss in-place merge variants (complex and rarely used), stability, and when to choose mergesort vs quicksort (stable vs average-case performance).

Complexity

- Time: O(n log n)
- Space: O(n) auxiliary

---

27. Quick Sort

Contract

- Input: array `a[]`.
- Output: sorted array (not stable by default unless carefully implemented).

Approach (detailed)

- Pick pivot (randomized pivot recommended), partition array into < pivot, = pivot, > pivot (Lomuto or Hoare partition). Recurse on partitions. Worst-case O(n^2) on already sorted input if pivot bad; randomization reduces that to expected O(n log n).

Java code (randomized quicksort with Lomuto partition)

```java
import java.util.*;

public class QuickSort {
        private static final Random rnd = new Random();
        public static void sort(int[] a) { qs(a, 0, a.length-1); }
        private static void qs(int[] a, int l, int r) {
                if (l >= r) return;
                int p = l + rnd.nextInt(r - l + 1);
                int pivot = a[p];
                int i = l, lt = l, gt = r;
```

```
                while (i <= gt) {
                        if (a[i] < pivot) swap(a, lt++, i++);
                        else if (a[i] > pivot) swap(a, i, gt--);
                        else i++;
                }
                qs(a, l, lt-1); qs(a, gt+1, r);
        }
        private static void swap(int[] a, int i, int j) { int t = a[i]; a[i] = a[j]; a[j] = t; }
        public static void main(String[] args) { int[] a = {3,6,8,10,1,2,1}; sort(a);
System.out.println(Arrays.toString(a)); }
}
```

Interviewer discussion

- Discuss pivot choice, tail recursion elimination, and introsort (switch to heapsort when recursion too deep) used in standard libraries.

Complexity

- Average Time: O(n log n), Worst Time: O(n^2) (rare with random pivot)
- Space: O(log n) average recursion stack

---

28. Heap Sort

Contract

- Input: array `a[]`.
- Output: sorted array (in-place, not stable).

Approach (detailed)

- Build max-heap in-place (heapify), then repeatedly swap heap root with last element and reduce heap size, sift-down to restore max-heap. O(n log n) time, O(1) extra space.

Java code

```java
import java.util.*;

public class HeapSort {
        public static void sort(int[] a) {
                int n = a.length;
                for (int i = n/2 - 1; i >= 0; i--) siftDown(a, i, n-1);
                for (int end = n-1; end > 0; end--) {
                        int t = a[0]; a[0] = a[end]; a[end] = t;
```

```
                siftDown(a, 0, end-1);
            }
        }
        private static void siftDown(int[] a, int i, int end) {
            while (2*i + 1 <= end) {
                int j = 2*i + 1;
                if (j+1 <= end && a[j+1] > a[j]) j++;
                if (a[i] >= a[j]) break;
                int t = a[i]; a[i] = a[j]; a[j] = t; i = j;
            }
        }
        public static void main(String[] args) { int[] a = {4,10,3,5,1}; sort(a);
System.out.println(Arrays.toString(a)); }
}
```

Interviewer discussion

- Mention that heapsort is in-place and guaranteed O(n log n), but not stable and usually slower in practice than quicksort due to cache behavior.

Complexity

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

---

29. Count Inversions in array

Contract

- Input: array `a[]`.
- Output: number of inversion pairs (i < j and a[i] > a[j]).

Approach (detailed)

- Use modified merge sort: while merging, when taking element from right half before left, add number of remaining left elements to inversion count. This yields O(n log n).

Java code

```java
public class CountInversions {
        public static long count(int[] a) { return mergeCount(a, 0, a.length-1, new
int[a.length]); }
        private static long mergeCount(int[] a, int l, int r, int[] tmp) {
            if (l >= r) return 0;
```

```
        int m = l + (r - l) / 2;
        long cnt = mergeCount(a, l, m, tmp) + mergeCount(a, m+1, r, tmp);
        int i = l, j = m+1, k = l;
        while (i <= m || j <= r) {
            if (i <= m && (j > r || a[i] <= a[j])) tmp[k++] = a[i++];
            else { tmp[k++] = a[j++]; cnt += (m - i + 1); }
        }
        for (k = l; k <= r; k++) a[k] = tmp[k];
        return cnt;
    }
    public static void main(String[] args) { int[] a = {2,4,1,3,5};
System.out.println(count(a)); }
}
```

Interviewer discussion

- Explain why the merge step counts inversions and show correctness with small example. Also mention Fenwick/Segment tree approaches for online counting when values are bounded.

Complexity

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

---

30. Kth largest/smallest element in an array

Contract

- Input: array `a[]`, integer k.
- Output: k-th largest (or smallest) element. Clarify 0/1-based k semantics.

Approach (detailed)

- Several options:
  - Sort and pick: O(n log n) time, O(1) extra space.
  - Quickselect (randomized selection): expected O(n) time, O(1) space; uses partitioning like quicksort and recurses on one side.
  - Use heap: min-heap of size k for k-th largest (O(n log k) time, O(k) space) — good when k << n or input is a stream.

Java code (Quickselect)

```java
import java.util.*;
```

```
public class KthElement {
	private static final Random rnd = new Random();
	public static int quickselect(int[] a, int k) { // returns k-th smallest (1-based)
		int l = 0, r = a.length - 1;
		while (true) {
			int p = l + rnd.nextInt(r - l + 1);
			int pivot = a[p];
			int i = l, lt = l, gt = r;
			while (i <= gt) {
				if (a[i] < pivot) swap(a, lt++, i++);
				else if (a[i] > pivot) swap(a, i, gt--);
				else i++;
			}
			if (k-1 < lt) r = lt-1; else if (k-1 > gt) l = gt+1; else return pivot;
		}
	}
	private static void swap(int[] a, int i, int j) { int t = a[i]; a[i] = a[j]; a[j] = t; }
	public static void main(String[] args) { int[] a = {3,2,1,5,6,4};
System.out.println(quickselect(a,2)); }
}
```

Interviewer discussion

- Discuss Quickselect average-case linear time and pathological inputs. For streaming input, use heap approach. If stable ranking or duplicates matter, clarify semantics.

Complexity

- Quickselect: Average O(n), Worst O(n^2) (rare with random pivot)
- Heap approach: O(n log k) time, O(k) space

---

4. Recursion & Backtracking

The Recursion & Backtracking section dives into problems that are ideal for demonstrating recursion, state management, pruning, and search-space optimization. Each entry below follows a small contract, an in-depth approach, a working Java implementation (small class with a main), sample I/O, interviewer talking points, possible optimizations, and time & space complexity.

---

31. Factorial using recursion

Contract

- Input: non-negative integer n (assume n small enough to avoid overflow).
- Output: n! as long (or BigInteger for large n).

Approach (detailed)

- Recursive definition: n! = n * (n-1)! with base case 0! = 1. Explain stack growth and tail recursion concept; Java does not optimize tail recursion, so iterative version is preferred for large n.

Java code

```java
import java.math.BigInteger;

public class Factorial {
	public static BigInteger factorial(int n) {
		if (n < 0) throw new IllegalArgumentException("n must be >= 0");
		if (n == 0) return BigInteger.ONE;
		return BigInteger.valueOf(n).multiply(factorial(n-1));
	}
	public static void main(String[] args) {
		System.out.println(factorial(20));
	}
}
```

Interviewer discussion

- Explain recursion depth O(n) and stack overflow risk; show iterative or BigInteger tradeoffs.

Complexity

- Time: O(n)
- Space: O(n) recursion stack (O(1) iterative)

---

32. Fibonacci using recursion

Contract

- Input: integer n >= 0.
- Output: F(n) where F(0)=0, F(1)=1.

Approach (detailed)

- Naive recursion directly follows definition but is exponential due to repeated work. Discuss memoization (top-down DP) or bottom-up iterative DP to reduce to O(n) time. Also mention matrix exponentiation or fast doubling for O(log n) time.

Java code (memoized)

```java
import java.util.*;

public class Fibonacci {
	private static Map<Integer, Long> memo = new HashMap<>();
	public static long fib(int n) {
		if (n < 0) throw new IllegalArgumentException();
		if (n <= 1) return n;
		if (memo.containsKey(n)) return memo.get(n);
		long val = fib(n-1) + fib(n-2);
		memo.put(n, val);
		return val;
	}
	public static void main(String[] args) {
		System.out.println(fib(50));
	}
}
```

Interviewer discussion

- Discuss time/space tradeoffs of memoization vs iterative DP vs fast doubling. Mention overflow and BigInteger if required.

Complexity

- Naive recursion: O(2^n)
- Memoized / DP: Time O(n), Space O(n)

---

33. N-Queens problem

Contract

- Input: integer n (size of board and number of queens).
- Output: all distinct solutions (list of placements) or count of solutions.

Approach (detailed)

- Classic backtracking: place queens row by row, checking column, diagonal, and anti-diagonal conflicts. Use boolean arrays for columns and diagonals for O(1) conflict checks. Prune early when conflict detected.

Java code (return count of solutions)

```java
import java.util.*;

public class NQueens {
	public static int totalNQueens(int n) {
		boolean[] cols = new boolean[n];
		boolean[] diag = new boolean[2*n]; // r+c
		boolean[] anti = new boolean[2*n]; // r-c + n
		return backtrack(0, n, cols, diag, anti);
	}
	private static int backtrack(int r, int n, boolean[] cols, boolean[] diag, boolean[] anti) {
		if (r == n) return 1;
		int count = 0;
		for (int c = 0; c < n; c++) {
			if (cols[c] || diag[r+c] || anti[r-c + n]) continue;
			cols[c] = diag[r+c] = anti[r-c + n] = true;
			count += backtrack(r+1, n, cols, diag, anti);
			cols[c] = diag[r+c] = anti[r-c + n] = false;
		}
		return count;
	}
	public static void main(String[] args) {
		System.out.println(totalNQueens(8));
	}
}
```

Interviewer discussion

- Talk about symmetry optimizations (only explore half columns for first row when n>1 and multiply), bitmask optimizations to speed up for large n, and outputting board configurations vs counting.

Complexity

- Time: exponential; approximate O(n!) in worst case, but pruning reduces the search significantly for practical n
- Space: O(n) recursion depth + O(n) state arrays

---

34. Rat in a Maze

Contract

- Input: 2D grid (matrix) of 0/1 where 1 is open and 0 is blocked; start at (0,0), goal (n-1,m-1).
- Output: one path (list of coordinates) or boolean indicating existence.

Approach (detailed)

- DFS/backtracking through 4 (or 8) directions, mark visited to avoid cycles, backtrack on dead-ends. Prefer iterative stack or recursion depending on constraints. For shortest path use BFS instead.

Java code (return boolean path exists and print path)

```java
import java.util.*;

public class RatInMaze {
	private static int[][] dirs = {{1,0},{0,1},{-1,0},{0,-1}};
	public static List<int[]> findPath(int[][] maze) {
		int n = maze.length, m = maze[0].length;
		boolean[][] vis = new boolean[n][m];
		List<int[]> path = new ArrayList<>();
		if (dfs(0,0,maze,vis,path)) return path;
		return Collections.emptyList();
	}
	private static boolean dfs(int r, int c, int[][] maze, boolean[][] vis, List<int[]> path) {
		int n = maze.length, m = maze[0].length;
		if (r < 0 || c < 0 || r >= n || c >= m || vis[r][c] || maze[r][c] == 0) return false;
		vis[r][c] = true; path.add(new int[]{r,c});
		if (r == n-1 && c == m-1) return true;
		for (int[] d : dirs) {
			if (dfs(r + d[0], c + d[1], maze, vis, path)) return true;
		}
		path.remove(path.size()-1); vis[r][c] = false;
		return false;
	}
	public static void main(String[] args) {
		int[][] maze = {{1,0,0},{1,1,0},{0,1,1}};
		List<int[]> p = findPath(maze);
		for (int[] xy : p) System.out.println(Arrays.toString(xy));
	}
}
```

Interviewer discussion

- Explain choice of DFS for any path vs BFS for shortest path. Discuss avoiding revisits and marking/unmarking visited during backtracking.

Complexity

- Time: O(n*m) in worst case, Space: O(n*m) for visited + recursion stack

---

35. Word Search in 2D grid

Contract

- Input: board of characters and a target word.
- Output: boolean whether word exists as a sequence of adjacent letters (4 directions) without revisiting a cell.

Approach (detailed)

- Backtracking start from cells that match first char, DFS with visited marking, backtracking upon mismatch. Prune by early mismatch and optionally by letter frequency heuristics.

Java code

```java
public class WordSearch {
	private static int[][] dirs = {{1,0},{-1,0},{0,1},{0,-1}};
	public static boolean exist(char[][] board, String word) {
		int n = board.length, m = board[0].length;
		boolean[][] vis = new boolean[n][m];
		for (int i = 0; i < n; i++) for (int j = 0; j < m; j++)
			if (board[i][j] == word.charAt(0) && dfs(board, word, 0, i, j, vis)) return
true;
		return false;
	}
	private static boolean dfs(char[][] b, String w, int idx, int r, int c, boolean[][] vis) {
		if (idx == w.length()) return true;
		int n = b.length, m = b[0].length;
		if (r < 0 || c < 0 || r >= n || c >= m || vis[r][c] || b[r][c] != w.charAt(idx)) return
false;
		vis[r][c] = true;
		for (int[] d : dirs) if (dfs(b, w, idx+1, r + d[0], c + d[1], vis)) return true;
		vis[r][c] = false;
		return false;
	}
	public static void main(String[] args) {
		char[][] b = {{'A','B','C','E'},{'S','F','C','S'},{'A','D','E','E'}};
		System.out.println(exist(b, "ABCCED"));
```

```
        }
}
```

Interviewer discussion

- Mention optimization: check letter counts to quickly reject impossible words; consider ordering starts by rare first letters to prune earlier.

Complexity

- Time: O(N * 4^L) worst-case where N is number of cells and L is word length; heavy pruning in practice
- Space: O(L) recursion depth + O(N) visited

---

36. Generate all subsets of a set (power set)

Contract

- Input: array/list of distinct elements.
- Output: list of all subsets.

Approach (detailed)

- Backtracking: decide include/exclude each element recursively. Alternatively iterative bitmask method enumerates 2^n subsets. For large n, streaming generation or limits required.

Java code (backtracking)

```java
import java.util.*;

public class Subsets {
        public static List<List<Integer>> subsets(int[] a) {
                List<List<Integer>> res = new ArrayList<>();
                backtrack(0, a, new ArrayList<>(), res);
                return res;
        }
        private static void backtrack(int idx, int[] a, List<Integer> cur, List<List<Integer>> res)
{
                if (idx == a.length) { res.add(new ArrayList<>(cur)); return; }
                // exclude
                backtrack(idx+1, a, cur, res);
                // include
                cur.add(a[idx]); backtrack(idx+1, a, cur, res); cur.remove(cur.size()-1);
```

```
        }
        public static void main(String[] args) { System.out.println(subsets(new int[]{1,2,3})); }
}
```

Interviewer discussion

- Discuss ordering (lexicographic vs size), handling duplicates (use sorting and skip duplicates), and iterative bitmask generation when memory is tight.

Complexity

- Time: O(n _ 2^n) to build all subsets, Space: O(2^n _ n) to store results (or O(n) stack if streaming)

---

37. Generate permutations of a string/array

Contract

- Input: array/string of elements (may have duplicates depending on variant).
- Output: list of permutations (unique if requested).

Approach (detailed)

- Backtracking with swapping for arrays or used[] boolean for building permutations incrementally. For duplicates, sort and skip equal choices at the same recursion level.

Java code (unique permutations for array with possible duplicates)

```java
import java.util.*;

public class Permutations {
        public static List<List<Integer>> permuteUnique(int[] a) {
                Arrays.sort(a);
                List<List<Integer>> res = new ArrayList<>();
                boolean[] used = new boolean[a.length];
                backtrack(a, used, new ArrayList<>(), res);
                return res;
        }
        private static void backtrack(int[] a, boolean[] used, List<Integer> cur,
List<List<Integer>> res) {
                if (cur.size() == a.length) { res.add(new ArrayList<>(cur)); return; }
                for (int i = 0; i < a.length; i++) {
                        if (used[i]) continue;
                        if (i > 0 && a[i] == a[i-1] && !used[i-1]) continue; // skip duplicates
```

```
                used[i] = true; cur.add(a[i]);
                backtrack(a, used, cur, res);
                used[i] = false; cur.remove(cur.size()-1);
            }
        }
        public static void main(String[] args) { System.out.println(permuteUnique(new
int[]{1,1,2})); }
}
```

Interviewer discussion

- Discuss factorial growth, pruning, generating permutations in lexicographic order, and in-place swapping vs used[] approaches.

Complexity

- Time: O(n! _ n) to generate and copy each permutation, Space: O(n! _ n) to store results (or O(n) stack if streaming)

---

38. Sudoku Solver

Contract

- Input: 9x9 board with digits '1'-'9' or '.' for empty cells.
- Output: filled board that satisfies Sudoku constraints (row/col/3x3 boxes) or boolean fail.

Approach (detailed)

- Backtracking with constraint checks; accelerate using bitmasks to track used digits in rows, columns and boxes allowing O(1) checks. Order empty cells by fewest legal choices (MRV heuristic) to prune faster.

Java code (basic backtracking with bitmasks)

```java
import java.util.*;

public class SudokuSolver {
    public static boolean solve(char[][] board) {
        int[] rows = new int[9], cols = new int[9], boxes = new int[9];
        List<int[]> empties = new ArrayList<>();
        for (int r = 0; r < 9; r++) for (int c = 0; c < 9; c++) {
            if (board[r][c] == '.') empties.add(new int[]{r,c});
            else {
                int d = board[r][c] - '1';
```

```
                                rows[r] |= 1 << d; cols[c] |= 1 << d; boxes[(r/3)*3 + c/3] |= 1 <<
d;
                        }
                }
                return backtrack(board, empties, 0, rows, cols, boxes);
        }
        private static boolean backtrack(char[][] b, List<int[]> empties, int idx, int[] rows, int[]
cols, int[] boxes) {
                if (idx == empties.size()) return true;
                int[] cell = empties.get(idx); int r = cell[0], c = cell[1], box = (r/3)*3 + c/3;
                int mask = ~(rows[r] | cols[c] | boxes[box]) & 0x1FF; // available digits
                while (mask != 0) {
                        int pick = Integer.numberOfTrailingZeros(mask);
                        mask &= mask - 1;
                        int bit = 1 << pick;
                        rows[r] |= bit; cols[c] |= bit; boxes[box] |= bit; b[r][c] = (char)('1' + pick);
                        if (backtrack(b, empties, idx+1, rows, cols, boxes)) return true;
                        rows[r] &= ~bit; cols[c] &= ~bit; boxes[box] &= ~bit; b[r][c] = '.';
                }
                return false;
        }
        public static void main(String[] args) {
                char[][] board = new char[9][9]; // populate with problem or test
                // ...example omitted for brevity
                // System.out.println(solve(board));
        }
}
```

Interviewer discussion

- Highlight bitmask acceleration and MRV heuristic. Mention that naive backtracking still works for typical Sudoku puzzles but heuristics make it fast for worst-case.

Complexity

- Exponential in worst-case but fast in practice with heuristics. Space: O(81) for state + recursion.

---

5. Linked List

This section expands the linked list problems (39–46) with concrete contracts, careful approaches, complete Java implementations (each includes a small `ListNode` helper), sample input/output, interviewer talking points, alternate/optimized solutions, and time & space complexity. Linked lists are a common interview topic for pointer manipulation, in-place algorithms, and two-pointer techniques.

---

39. Reverse a linked list (iterative & recursive)

Contract

- Input: head of singly-linked list.
- Output: head of reversed list.

Approach (detailed)

- Iterative: maintain three pointers prev, curr, next; iterate and flip curr.next = prev. Very memory efficient and commonly expected.
- Recursive: reverse rest of list and append head at the end; elegant but uses recursion stack O(n).

Java code (includes ListNode)

```java
public class ReverseLinkedList {
	static class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
	public static ListNode reverseIter(ListNode head) {
		ListNode prev = null, curr = head;
		while (curr != null) {
			ListNode nx = curr.next;
			curr.next = prev;
			prev = curr; curr = nx;
		}
		return prev;
	}
	public static ListNode reverseRec(ListNode head) {
		if (head == null || head.next == null) return head;
		ListNode p = reverseRec(head.next);
		head.next.next = head;
		head.next = null;
		return p;
	}
	// sample driver builds 1->2->3 and prints reversed list
	public static void main(String[] args) {
		ListNode h = new ListNode(1); h.next = new ListNode(2); h.next.next = new
ListNode(3);
		ListNode r = reverseIter(h);
		while (r != null) { System.out.print(r.val + " "); r = r.next; }
	}
}
```

Interviewer discussion

- Talk about in-place vs recursion, stack depth, and how to handle very long lists. Show how to reverse sublists (from m to n) by pointer juggling.

Complexity

- Time: O(n), Space: O(1) iterative, O(n) recursion stack for recursive

---

40. Detect cycle in linked list

Contract

- Input: head of singly-linked list.
- Output: boolean (cycle exists) and optionally the node where cycle begins.

Approach (detailed)

- Floyd's Tortoise and Hare: move slow by 1 and fast by 2; if they meet a cycle exists. To find cycle start, reset slow to head and advance both by 1 until they meet — that meeting node is cycle entry.

Java code

```java
public class DetectCycle {
	static class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
	public static ListNode detectCycle(ListNode head) {
		ListNode slow = head, fast = head;
		while (fast != null && fast.next != null) {
			slow = slow.next; fast = fast.next.next;
			if (slow == fast) {
				slow = head;
				while (slow != fast) { slow = slow.next; fast = fast.next; }
				return slow; // entry
			}
		}
		return null;
	}
	public static void main(String[] args) {
		ListNode a = new ListNode(3); a.next = new ListNode(2); a.next.next = new
ListNode(0); a.next.next.next = new ListNode(-4);
		a.next.next.next.next = a.next; // create cycle at node with value 2
		System.out.println(detectCycle(a).val);
	}
}
```

```

Interviewer discussion

- Explain why moving pointers at different speeds finds a meeting point. Derive why resetting one pointer to head finds the cycle start.

Complexity

- Time: O(n), Space: O(1)

---

41. Find middle of linked list

Contract

- Input: head of singly-linked list.
- Output: middle node (for even length, return the second middle by common convention) or both depending on spec.

Approach (detailed)

- Fast/slow pointers: advance fast by 2 and slow by 1; when fast reaches end, slow is at middle. Handle edge cases for even/odd lengths.

Java code

```java
public class MiddleOfList {
	static class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
	public 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;
	}
	public static void main(String[] args) {
		ListNode h = new ListNode(1); h.next = new ListNode(2); h.next.next = new
ListNode(3); h.next.next.next = new ListNode(4);
		System.out.println(middleNode(h).val); // prints 3 (second middle)
	}
}
```

Interviewer discussion
```

- Discuss returning first middle vs second middle and how to change loop conditions. Mention uses: splitting list for merge sort.

Complexity

- Time: O(n), Space: O(1)

---

42. Merge two sorted linked lists

Contract

- Input: two sorted singly-linked lists l1, l2.
- Output: merged sorted list (new head), typically in-place by reusing nodes.

Approach (detailed)

- Iterative two-pointer merge using a dummy head; attach smaller node and advance pointer. Works in-place by relinking nodes.

Java code

```java
public class MergeSortedLists {
	static class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
	public static ListNode merge(ListNode a, ListNode b) {
		ListNode dummy = new ListNode(0), tail = dummy;
		while (a != null && b != null) {
			if (a.val <= b.val) { tail.next = a; a = a.next; }
			else { tail.next = b; b = b.next; }
			tail = tail.next;
		}
		tail.next = (a != null) ? a : b;
		return dummy.next;
	}
	public static void main(String[] args) {
		ListNode a = new ListNode(1); a.next = new ListNode(3); a.next.next = new
ListNode(5);
		ListNode b = new ListNode(2); b.next = new ListNode(4);
		ListNode h = merge(a,b);
		while (h != null) { System.out.print(h.val + " "); h = h.next; }
	}
}
```

Interviewer discussion

- Talk about in-place vs creating new nodes, stable merge, and using recursion for a succinct solution (but recursion uses stack).

Complexity

- Time: O(n + m), Space: O(1) extra

---

43. Remove Nth node from end

Contract

- Input: head of list and integer n (1-based: remove nth from end).
- Output: head of list after removal.

Approach (detailed)

- Two-pointer gap method: advance fast by n steps, then move both slow (starts at dummy) and fast until fast reaches end; slow.next is node to remove. Use dummy to simplify removing head.

Java code

```java
public class RemoveNthFromEnd {
	static class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
	public 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; // assume n is valid
		while (fast.next != null) { fast = fast.next; slow = slow.next; }
		// slow.next is to remove
		slow.next = slow.next.next;
		return dummy.next;
	}
	public static void main(String[] args) {
		ListNode h = new ListNode(1); h.next = new ListNode(2); h.next.next = new ListNode(3); h.next.next.next = new ListNode(4);
		ListNode r = removeNthFromEnd(h, 2);
		while (r != null) { System.out.print(r.val + " "); r = r.next; }
	}
}
```

Interviewer discussion

- Clarify n validity, show why dummy node simplifies deleting head, and discuss one-pass vs two-pass alternatives.

Complexity

- Time: O(n), Space: O(1)

---

44. Intersection point of two linked lists

Contract

- Input: heads of two singly-linked lists that may merge at a node (share tail).
- Output: reference to intersection node or null.

Approach (detailed)

- Two-pointer switching trick: traverse both lists with pointers p and q; when one reaches null, switch to other head. If lists intersect, pointers meet at intersection after at most 2*(lenA+lenB) steps; else both become null.

Java code

```java
public class GetIntersectionNode {
	static class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
	public static ListNode getIntersection(ListNode a, ListNode b) {
		if (a == null || b == null) return null;
		ListNode p = a, q = b;
		while (p != q) {
			p = (p == null) ? b : p.next;
			q = (q == null) ? a : q.next;
		}
		return p;
	}
	public static void main(String[] args) {
		ListNode a = new ListNode(1); a.next = new ListNode(2); ListNode c = new
ListNode(3); a.next.next = c; c.next = new ListNode(4);
		ListNode b = new ListNode(9); b.next = c; // intersect at c
		System.out.println(getIntersection(a,b).val);
	}
}
```

Interviewer discussion

- Explain correctness of pointer switching and compare with length-difference alignment method. Discuss cases with cycles and if inputs could be cyclic.

Complexity

- Time: O(n + m), Space: O(1)

---

45. Add two numbers represented by linked lists

Contract

- Input: two non-empty linked lists representing non-negative integers in reverse order (each node a digit).
- Output: linked list representing sum in same reverse-digit format.

Approach (detailed)

- Simulate grade-school addition with carry; iterate through both lists, sum digits and carry, create new nodes (or modify one list in-place if allowed). Handle final carry.

Java code

```java
public class AddTwoNumbers {
	static class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
	public static ListNode addTwoNumbers(ListNode l1, ListNode l2) {
		ListNode dummy = new ListNode(0), cur = dummy;
		int carry = 0;
		while (l1 != null || l2 != null || carry != 0) {
			int sum = carry + (l1 != null ? l1.val : 0) + (l2 != null ? l2.val : 0);
			carry = sum / 10;
			cur.next = new ListNode(sum % 10);
			cur = cur.next;
			if (l1 != null) l1 = l1.next; if (l2 != null) l2 = l2.next;
		}
		return dummy.next;
	}
	public static void main(String[] args) {
		ListNode a = new ListNode(2); a.next = new ListNode(4); a.next.next = new ListNode(3); // 342
		ListNode b = new ListNode(5); b.next = new ListNode(6); b.next.next = new ListNode(4); // 465
		ListNode s = addTwoNumbers(a,b); while (s != null) { System.out.print(s.val); s = s.next; } // prints 708 (reverse list 7->0->8)
	}
}
```

```

Interviewer discussion

- Discuss in-place modification to save allocations, handling different lengths, and variants where digits are stored forward (requires reversal or stack).

Complexity

- Time: O(max(n,m)), Space: O(max(n,m)) for result (O(1) extra if in-place allowed)

---

46. Flatten a linked list

Problem variants

- Many problems use the phrase "flatten a linked list": common variant is a list where each node has next and child pointers (multilevel list) and the task is to flatten depth-first; another is flattening a linked list of lists.

Contract (multilevel list, depth-first)

- Input: head of multilevel linked list with `next` and `child` pointers.
- Output: flattened single-level list following depth-first order in-place.

Approach (detailed)

- Use iterative stack or recursion: traverse nodes, when child exists push next onto stack, link child as next, null child; when reach end and stack not empty pop and attach.

Java code (simple multilevel flatten)

```java
import java.util.*;
public class FlattenMultilevelList {
	static class Node { int val; Node next, child; Node(int v){val=v;} }
	public static Node flatten(Node head) {
		if (head == null) return null;
		Node cur = head;
		Deque<Node> st = new ArrayDeque<>();
		while (cur != null) {
			if (cur.child != null) {
				if (cur.next != null) st.push(cur.next);
				cur.next = cur.child; cur.child = null;
			}
			if (cur.next == null && !st.isEmpty()) cur.next = st.pop();
			cur = cur.next;
```
```

```
                }
                return head;
        }
        public static void main(String[] args) {
                // build small example omitted for brevity
        }
}
```

Interviewer discussion

- Clarify which flatten variant is expected. Discuss in-place vs building new list, recursion depth, and preserving original child pointers if needed.

Complexity

- Time: O(n) where n is total nodes across levels, Space: O(d) stack where d is depth (O(n) worst-case)

---

6. Stack & Queue

This section expands common stack & queue problems (47–55). For each problem I give a small contract, a clear approach, a complete Java implementation (self-contained), sample input/output, interviewer talking points, possible optimizations, and time & space complexity. Stack and queue problems test understanding of LIFO/FIFO behavior, amortized analysis, and use of auxiliary structures.

---

47. Implement stack using array/linked list

Contract

- Implement push, pop, peek/top, isEmpty operations with expected time complexity.

Approach (detailed)

- Two standard implementations:
  - Array-backed stack: fixed capacity or dynamic resizing (like ArrayList) — O(1) amortized push with resizing.
  - Linked-list-backed stack: push/pop at head pointer — O(1) worst-case and no resizing.

Java code (array-backed with dynamic resize)

```java
public class ArrayStack<T> {
```

```
        private Object[] data; private int size = 0;
        public ArrayStack(int cap) { data = new Object[Math.max(4, cap)]; }
        public void push(T x) {
                if (size == data.length) resize(data.length * 2);
                data[size++] = x;
        }
        @SuppressWarnings("unchecked")
        public T pop() {
                if (size == 0) throw new RuntimeException("Empty");
                T v = (T) data[--size]; data[size] = null;
                if (size > 0 && size == data.length / 4) resize(data.length / 2);
                return v;
        }
        @SuppressWarnings("unchecked") public T peek() { if (size==0) throw new
RuntimeException("Empty"); return (T)data[size-1]; }
        public boolean isEmpty() { return size == 0; }
        private void resize(int n) { Object[] d = new Object[n];
System.arraycopy(data,0,d,0,size); data = d; }
}
```

Interviewer discussion

- Compare array vs linked-list tradeoffs, memory locality (array faster), and resizing amortized guarantees.

Complexity

- Push/pop/peek: O(1) amortized for dynamic array, O(1) worst-case for linked list; Space: O(n)

---

48. Implement queue using array/linked list

Contract

- Implement enqueue, dequeue, peek/front, isEmpty operations.

Approach (detailed)

- Circular buffer (array with head/tail indices) yields O(1) operations and good cache locality. Linked list with head/tail pointers is simple and unbounded.

Java code (circular buffer)

```java
public class CircularQueue<T> {
```

```
        private Object[] data; private int head = 0, tail = 0, size = 0;
        public CircularQueue(int cap) { data = new Object[Math.max(4, cap)]; }
        public void enqueue(T x) {
                if (size == data.length) resize(data.length * 2);
                data[tail] = x; tail = (tail + 1) % data.length; size++;
        }
        @SuppressWarnings("unchecked")
        public T dequeue() {
                if (size == 0) throw new RuntimeException("Empty");
                T v = (T) data[head]; data[head] = null; head = (head + 1) % data.length;
size--;
                if (size > 0 && size == data.length/4) resize(data.length/2);
                return v;
        }
        public boolean isEmpty() { return size == 0; }
        private void resize(int cap) {
                Object[] d = new Object[cap];
                for (int i = 0; i < size; i++) d[i] = data[(head + i) % data.length];
                data = d; head = 0; tail = size;
        }
}
```

Interviewer discussion

- Emphasize circular indexing, resizing behavior, and differences with linked-list implementation when memory predictability matters.

Complexity

- Enqueue/dequeue: O(1) amortized, Space: O(n)

---

49. Implement two stacks in an array

Contract

- Store two stacks in a single array without overlap, support push/pop for both stacks.

Approach (detailed)

- Start one stack from left (idx L) and the other from right (idx R). Grow toward each other; detect overflow when L > R.

Java code

```java
```

```
public class TwoStacks {
	private int[] data; private int left = -1, right;
	public TwoStacks(int cap) { data = new int[cap]; right = cap; }
	public void push1(int x) { if (left + 1 == right) throw new RuntimeException("Full");
data[++left] = x; }
	public void push2(int x) { if (left + 1 == right) throw new RuntimeException("Full");
data[--right] = x; }
	public int pop1() { if (left < 0) throw new RuntimeException("Empty1"); return
data[left--]; }
	public int pop2() { if (right == data.length) throw new RuntimeException("Empty2");
return data[right++]; }
}
```

Interviewer discussion

- Discuss alternative: fixed partition vs dynamic and handling rebalancing. Mention that this keeps O(1) operations and minimal extra memory.

Complexity

- Push/pop: O(1), Space: O(n) shared

---

50. Implement min stack

Contract

- Stack supporting push, pop, top, and getMin in O(1) time.

Approach (detailed)

- Two main patterns:
  - Maintain auxiliary stack that stores current minimums (push min when new value <= current min, pop when value equals top of min stack).
  - Store (value, currentMin) pair on main stack.

Java code (aux stack)

```java
import java.util.*;
public class MinStack {
	private Deque<Integer> st = new ArrayDeque<>();
	private Deque<Integer> mins = new ArrayDeque<>();
	public void push(int x) { st.push(x); if (mins.isEmpty() || x <= mins.peek())
mins.push(x); }
	public int pop() { int v = st.pop(); if (v == mins.peek()) mins.pop(); return v; }
```

```
        public int top() { return st.peek(); }
        public int getMin() { return mins.peek(); }
}
```

Interviewer discussion

- Discuss equality handling (<=) to support duplicates. Show variant storing difference to compress space.

Complexity

- All ops O(1) time, Space: O(n) worst-case for auxiliary stack

---

51. Next Greater Element

Contract

- Input: array `a[]`; output: for each element find next greater element to its right (or -1 if none).

Approach (detailed)

- Monotonic stack: iterate left to right, maintain stack of indices whose next greater hasn't been found; when current > top, pop and set result for popped index. Works in O(n).

Java code

```java
import java.util.*;
public class NextGreater {
        public static int[] nextGreater(int[] a) {
                int n = a.length; int[] res = new int[n]; Arrays.fill(res, -1);
                Deque<Integer> st = new ArrayDeque<>();
                for (int i = 0; i < n; i++) {
                        while (!st.isEmpty() && a[i] > a[st.peek()]) res[st.pop()] = a[i];
                        st.push(i);
                }
                return res;
        }
}
```

Interviewer discussion

- Explain monotonic stack invariant and variations for circular arrays (wrap-around) and previous greater element.

Complexity

- Time: O(n), Space: O(n)

---

52. Balanced Parentheses

Contract

- Input: string of brackets ((), {}, []).
- Output: boolean whether string is validly balanced and properly nested.

Approach (detailed)

- Use stack: push opening brackets, when encountering closing bracket check stack top matches expected pair. Early reject on mismatch or leftover opens at end.

Java code

```java
import java.util.*;
public class BalancedParentheses {
	public static boolean isValid(String s) {
		Deque<Character> st = new ArrayDeque<>();
		for (char c : s.toCharArray()) {
			if (c == '(' || c == '{' || c == '[') st.push(c);
			else {
				if (st.isEmpty()) return false;
				char t = st.pop();
				if ((c==')' && t!='(') || (c=='}' && t!='{') || (c==']' && t!='[')) return
false;
			}
		}
		return st.isEmpty();
	}
}
```

Interviewer discussion

- Mention extensions: longest valid parentheses, count of valid substrings, or handling non-bracket chars.

Complexity

- Time: O(n), Space: O(n)

---

53. Implement LRU Cache

Contract

- Support get(key) and put(key,value) in O(1) time, evict least-recently-used entry when at capacity.

Approach (detailed)

- Standard pattern: Doubly-linked list for usage order + HashMap from key to node for O(1) access. Move accessed/updated node to front; evict tail when capacity exceeded.

Java code (concise)

```java
import java.util.*;
public class LRUCache {
	static class Node { int k,v; Node prev,next; Node(int k,int v){this.k=k;this.v=v;} }
	private int cap; private Map<Integer,Node> map = new HashMap<>(); private Node head, tail;
	public LRUCache(int capacity){cap=capacity; head=new Node(-1,-1); tail=new Node(-1,-1); head.next=tail; tail.prev=head;}
	private void add(Node n){ n.next=head.next; n.prev=head; head.next.prev=n; head.next=n; }
	private void remove(Node n){ n.prev.next=n.next; n.next.prev=n.prev; }
	public int get(int key){ Node n=map.get(key); if(n==null) return -1; remove(n); add(n); return n.v; }
	public void put(int key,int value){ Node n=map.get(key); if(n!=null){ remove(n); n.v=value; add(n); } else { if(map.size()==cap){ Node del=tail.prev; remove(del); map.remove(del.k);} Node nn=new Node(key,value); add(nn); map.put(key,nn);} }
}
```

Interviewer discussion

- Discuss thread-safety, capacity resizing, using LinkedHashMap as simple alternative, and differences between LRU and LFU.

Complexity

- get/put: O(1) amortized, Space: O(capacity)

---

54. Sliding Window Maximum

Contract

- Input: array `a[]` and window size k. Output: array of maximums for each sliding window.

Approach (detailed)

- Monotonic deque: store indices in decreasing value order; front is current max. When sliding, pop indices out of range and maintain decreasing order by value.

Java code

```java
import java.util.*;
public class SlidingWindowMax {
	public static int[] maxSlidingWindow(int[] a, int k) {
		int n = a.length; if (n==0) return new int[0];
		int[] res = new int[n - k + 1]; Deque<Integer> dq = new ArrayDeque<>();
		for (int i = 0; i < n; i++) {
			while (!dq.isEmpty() && dq.peekFirst() <= i - k) dq.pollFirst();
			while (!dq.isEmpty() && a[dq.peekLast()] <= a[i]) dq.pollLast();
			dq.offerLast(i);
			if (i >= k - 1) res[i - k + 1] = a[dq.peekFirst()];
		}
		return res;
	}
}
```

Interviewer discussion

- Explain deque invariants and amortized O(1) operations. Mention variants: sum/average in sliding window (use prefix sums) and streaming data.

Complexity

- Time: O(n), Space: O(k)

---

55. Circular Queue implementation

Contract

- FIFO queue with fixed capacity behaving circularly, support enqueue, dequeue, isFull/isEmpty, front/rear operations.

Approach (detailed)

- Similar to `CircularQueue` above but fixed capacity not resizing. Use head/tail and size or use (tail+1)%cap == head to detect full.

Java code

```java
public class MyCircularQueue {
	private int[] data; private int head = 0, tail = 0, size = 0;
	public MyCircularQueue(int k) { data = new int[k]; }
	public boolean enQueue(int value){ if (size == data.length) return false;
data[tail]=value; tail=(tail+1)%data.length; size++; return true; }
	public boolean deQueue(){ if (size==0) return false; head=(head+1)%data.length;
size--; return true; }
	public int Front(){ return size==0 ? -1 : data[head]; }
	public int Rear(){ return size==0 ? -1 : data[(tail-1+data.length)%data.length]; }
	public boolean isEmpty(){ return size==0; }
	public boolean isFull(){ return size==data.length; }
}
```

Interviewer discussion

- Clarify wrap-around edge cases, off-by-one errors, and alternatives (linked list) where dynamic size required.

Complexity

- All ops O(1) time, Space: O(k)

---

7. Trees & Binary Search Trees (BST)

56. Preorder, Inorder, Postorder traversals (recursive & iterative)

Problem
Given the root of a binary tree, produce the preorder (root,left,right), inorder (left,root,right) and postorder (left,right,root) traversals. Provide both recursive and iterative implementations.

Approach (detailed)

- Recursive implementations follow the exact order with straightforward recursion and are easy to reason about. They use O(h) stack space where h is tree height.

- Iterative implementations use an explicit stack. Preorder is natural with a stack by pushing right then left. Inorder uses a stack to traverse left spine then visit nodes. Postorder can be implemented with two stacks or a single stack with a visited flag or by reversing a modified preorder (root,right,left -> reverse to get left,right,root).
- Edge cases: empty tree (return empty list), very deep trees (recursion may overflow Java stack), and skewed trees (height = n).

Java code (all three, iterative + recursive)

```java
import java.util.*;

public class TreeTraversals {
	static class TreeNode { int val; TreeNode left, right; TreeNode(int v){val=v;} }

	// Recursive inorder
	public static List<Integer> inorderRec(TreeNode root) {
		List<Integer> res = new ArrayList<>();
		inorderRecHelper(root, res);
		return res;
	}
	private static void inorderRecHelper(TreeNode r, List<Integer> out) {
		if (r == null) return;
		inorderRecHelper(r.left, out);
		out.add(r.val);
		inorderRecHelper(r.right, out);
	}

	// Iterative inorder
	public static List<Integer> inorderIter(TreeNode root) {
		List<Integer> res = new ArrayList<>(); Deque<TreeNode> st = new
ArrayDeque<>(); TreeNode cur = root;
		while (cur != null || !st.isEmpty()) {
			while (cur != null) { st.push(cur); cur = cur.left; }
			cur = st.pop(); res.add(cur.val); cur = cur.right;
		}
		return res;
	}

	// Iterative preorder
	public static List<Integer> preorderIter(TreeNode root) {
		List<Integer> res = new ArrayList<>(); if (root == null) return res;
		Deque<TreeNode> st = new ArrayDeque<>(); st.push(root);
		while (!st.isEmpty()) {
			TreeNode n = st.pop(); res.add(n.val);
			if (n.right != null) st.push(n.right);
			if (n.left != null) st.push(n.left);
		}
```

```
        return res;
    }

    // Iterative postorder using modified preorder then reverse
    public static List<Integer> postorderIter(TreeNode root) {
        List<Integer> res = new ArrayList<>(); if (root == null) return res;
        Deque<TreeNode> st = new ArrayDeque<>(); st.push(root);
        while (!st.isEmpty()) {
            TreeNode n = st.pop(); res.add(n.val);
            if (n.left != null) st.push(n.left);
            if (n.right != null) st.push(n.right);
        }
        Collections.reverse(res);
        return res;
    }

    // sample driver
    public static void main(String[] args) {
        TreeNode r = new TreeNode(1);
        r.left = new TreeNode(2); r.right = new TreeNode(3);
        r.left.left = new TreeNode(4); r.left.right = new TreeNode(5);
        System.out.println("inorderRec=" + inorderRec(r));
        System.out.println("inorderIter=" + inorderIter(r));
        System.out.println("preorderIter=" + preorderIter(r));
        System.out.println("postorderIter=" + postorderIter(r));
    }
}
```

Interviewer discussion

- Explain why the stack simulation matches recursion (call stack). Discuss space: recursion uses implicit stack; iterative uses explicit stack—both O(h).
- For postorder show tradeoffs between two-stack, one-stack-with-flag, and reverse-preorder techniques.

Complexity

- Time: O(n) for all traversals
- Space: O(h) extra (recursion or stack), O(n) worst-case for skewed trees

---

57. Level Order Traversal (BFS)

Problem
Return nodes level-by-level (breadth-first) for a binary tree.

Approach (detailed)

- Use a queue and push root. For each level, record queue size and dequeue that many nodes, appending their children. This yields a list of lists (each inner list is one level). Useful variants: return single flattened list, zigzag order, or record level sums.
- Edge cases: empty tree, very wide trees (queue may hold O(width)).

Java code (level-order with per-level grouping)

```java
import java.util.*;

public class LevelOrder {
	static class TreeNode { int val; TreeNode left, right; TreeNode(int v){val=v;} }
	public static List<List<Integer>> levelOrder(TreeNode root) {
		List<List<Integer>> res = new ArrayList<>(); if (root == null) return res;
		Deque<TreeNode> q = new ArrayDeque<>(); q.add(root);
		while (!q.isEmpty()) {
			int sz = q.size(); List<Integer> level = new ArrayList<>();
			for (int i = 0; i < sz; i++) {
				TreeNode n = q.poll(); level.add(n.val);
				if (n.left != null) q.add(n.left);
				if (n.right != null) q.add(n.right);
			}
			res.add(level);
		}
		return res;
	}
}
```

Interviewer discussion

- Discuss memory bound by maximum width and when BFS is appropriate (shortest path in unweighted trees/graphs). Mention iterative level separation vs storing level in node pairs.

Complexity

- Time: O(n), Space: O(w) where w is maximum width (O(n) worst-case)

---

58. Height of a tree

Problem
Compute the height (max depth) of a binary tree.

Approach (detailed)

- Recursive depth-first: height = 1 + max(height(left), height(right)). Iterative BFS counts levels by processing queue size per level. Be careful with definition: height of empty tree is 0 or -1 depending on convention; here we use number of nodes on longest root-to-leaf path.

Java code (recursive + iterative)

```java
public class TreeHeight {
	static class TreeNode { int val; TreeNode left, right; TreeNode(int v){val=v;} }
	public static int heightRec(TreeNode r) { return r == null ? 0 : 1 + Math.max(heightRec(r.left), heightRec(r.right)); }
	public static int heightBFS(TreeNode r) {
		if (r == null) return 0; Deque<TreeNode> q = new ArrayDeque<>(); q.add(r); int h = 0;
		while (!q.isEmpty()) { int sz = q.size(); for (int i=0;i<sz;i++){ TreeNode n=q.poll(); if(n.left!=null) q.add(n.left); if(n.right!=null) q.add(n.right);} h++; }
		return h;
	}
}
```

Interviewer discussion

- Discuss recursion depth and tail recursion (not optimized in Java). For very deep trees, iterative BFS avoids stack overflow but may use more memory.

Complexity

- Time: O(n), Space: O(h) recursion or O(w) BFS

---

59. Diameter of a binary tree

Problem
Return the diameter (longest path between any two nodes measured in number of nodes or edges). Clarify nodes vs edges; here we return number of nodes on longest path.

Approach (detailed)

- Single-pass DFS: at each node compute height(left) and height(right); candidate diameter = leftHeight + rightHeight + 1 (nodes). Track global maximum while computing heights. This yields O(n) time.
- Edge cases: empty tree diameter 0.

Java code (single-pass)

```java
public class Diameter {
	static class TreeNode { int val; TreeNode left, right; TreeNode(int v){val=v;} }
	private static int best = 0;
	public static int diameter(TreeNode root) { best = 0; depth(root); return best; }
	private static int depth(TreeNode r) {
		if (r == null) return 0;
		int L = depth(r.left), R = depth(r.right);
		best = Math.max(best, L + R + 1);
		return 1 + Math.max(L, R);
	}
}
```

Interviewer discussion

- Explain why combining heights yields longest path through a node. Mention variant returning number of edges (subtract 1 from node count).

Complexity

- Time: O(n), Space: O(h)

---

60. Lowest Common Ancestor (LCA)

Problem
Given root of binary tree (or BST) and two nodes p and q, find their lowest common ancestor (deepest node that is ancestor of both).

Approach (detailed)

- For general binary tree: recursive DFS that returns node when it finds p or q; if both sides return non-null, current node is LCA. This is O(n) time.
- For BST: exploit ordering—if both p and q < root go left; if both > root go right; else root is LCA—this is O(h).
- Edge cases: one node being ancestor of the other, nodes not present (clarify with interviewer).

Java code (general tree + BST variant)

```java
public class LCA {
	static class TreeNode { int val; TreeNode left, right; TreeNode(int v){val=v;} }
	// general binary tree
	public static TreeNode lca(TreeNode root, TreeNode p, TreeNode q) {
		if (root == null || root == p || root == q) return root;
```

```
        TreeNode L = lca(root.left, p, q);
        TreeNode R = lca(root.right, p, q);
        if (L != null && R != null) return root;
        return L != null ? L : R;
    }

    // BST optimized
    public static TreeNode lcaBST(TreeNode root, TreeNode p, TreeNode q) {
        while (root != null) {
            if (p.val < root.val && q.val < root.val) root = root.left;
            else if (p.val > root.val && q.val > root.val) root = root.right;
            else return root;
        }
        return null;
    }
}
```

Interviewer discussion

- Discuss assumptions (nodes exist, unique values). Show proof for BST variant using ordering. Mention alternative parent-pointer method if parent links given.

Complexity

- General: Time O(n), Space O(h)
- BST: Time O(h), Space O(1)

---

61. Check if tree is balanced

Problem
Determine if a binary tree is height-balanced: for every node, heights of left and right subtrees differ by at most 1.

Approach (detailed)

- Efficient single-pass recursion returns -1 when subtree unbalanced or returns height otherwise. At each node check child heights and propagate failure early. Naive approach computing heights repeatedly is O(n^2); single-pass is O(n).

Java code (single-pass)

```java
public class IsBalanced {
    static class TreeNode { int val; TreeNode left, right; TreeNode(int v){val=v;} }
    public static boolean isBalanced(TreeNode root) { return check(root) != -1; }
```

```
        private static int check(TreeNode r) {
                if (r == null) return 0;
                int L = check(r.left); if (L == -1) return -1;
                int R = check(r.right); if (R == -1) return -1;
                if (Math.abs(L - R) > 1) return -1;
                return 1 + Math.max(L, R);
        }
}
```

Interviewer discussion

- Explain why early -1 propagation avoids repeated work. Mention definition variants (allow difference <=1) and discuss balancing rotations in AVL/Red-Black trees as next step for self-balancing BSTs.

Complexity

- Time: O(n), Space: O(h)

---

62. Serialize & Deserialize a binary tree

Problem
Design algorithms to serialize a binary tree to a string and deserialize back to a tree. Useful for storing or transmitting tree structure.

Approach (detailed)

- Use a DFS preorder with sentinel (e.g., "#") for nulls and comma-separated values. Deserialization splits tokens and consumes them in order to rebuild tree. BFS (level-order) with null markers is also common and keeps balanced representation.
- Edge cases: node values containing separators (choose safe encoding), empty tree, very large trees (string size). Consider using streaming readers to avoid building huge intermediate arrays.

Java code (preorder with null marker)

```java
import java.util.*;
public class Codec {
        static class TreeNode { int val; TreeNode left, right; TreeNode(int v){val=v;} }
        public String serialize(TreeNode root) {
                StringBuilder sb = new StringBuilder();
                serializeRec(root, sb);
                return sb.toString();
        }
```

```
        private void serializeRec(TreeNode r, StringBuilder sb) {
                if (r == null) { sb.append("#,"); return; }
                sb.append(r.val).append(',');
                serializeRec(r.left, sb); serializeRec(r.right, sb);
        }
        public TreeNode deserialize(String data) {
                if (data == null || data.length() == 0) return null;
                String[] toks = data.split(",");
                Deque<String> dq = new ArrayDeque<>(Arrays.asList(toks));
                return deserializeRec(dq);
        }
        private TreeNode deserializeRec(Deque<String> dq) {
                if (dq.isEmpty()) return null;
                String t = dq.pollFirst();
                if (t.equals("#")) return null;
                TreeNode node = new TreeNode(Integer.parseInt(t));
                node.left = deserializeRec(dq); node.right = deserializeRec(dq);
                return node;
        }
}
```

Interviewer discussion

- Explain choice of traversal and null sentinel. Discuss tradeoffs between preorder and level-order serialization and robustness when node values contain delimiters.

Complexity

- Time: O(n) for both serialize and deserialize, Space: O(n) for output string and token array (or streaming alternatives)

---

63. BST insert, search, delete

Problem
Implement core BST operations: insert a value, search for a value, and delete a node by value while preserving BST properties.

Approach (detailed)

- Search and insert are standard recursive or iterative traversals guided by value comparisons. Delete requires three cases: leaf (remove), one child (replace with child), two children (replace node with inorder successor or predecessor then remove that successor). Maintain links carefully.

Java code (recursive implementations)

```java
public class BSTOps {
	static class TreeNode { int val; TreeNode left, right; TreeNode(int v){val=v;} }
	public static TreeNode search(TreeNode root, int key) {
		if (root == null || root.val == key) return root;
		return key < root.val ? search(root.left, key) : search(root.right, key);
	}
	public static TreeNode insert(TreeNode root, int key) {
		if (root == null) return new TreeNode(key);
		if (key < root.val) root.left = insert(root.left, key);
		else if (key > root.val) root.right = insert(root.right, key);
		return root;
	}
	public static TreeNode delete(TreeNode root, int key) {
		if (root == null) return null;
		if (key < root.val) root.left = delete(root.left, key);
		else if (key > root.val) root.right = delete(root.right, key);
		else {
			if (root.left == null) return root.right;
			if (root.right == null) return root.left;
			// two children: find inorder successor
			TreeNode succ = root.right; while (succ.left != null) succ = succ.left;
			root.val = succ.val;
			root.right = delete(root.right, succ.val);
		}
		return root;
	}
}
```

Interviewer discussion

- Discuss balancing issues (BST degenerates to linked list). Mention AVL/Red-Black trees for guaranteed O(log n) operations. Discuss iterative delete to avoid recursion if stack is a concern.

Complexity

- Average Time: O(h) where h is height (O(log n) for balanced), Worst O(n) for degenerate BST; Space: O(h) recursion

---

64. Check if binary tree is BST

Problem

Given root of binary tree, determine whether it is a valid BST (left subtree values < node < right subtree values).

Approach (detailed)

- Use recursion with min/max bounds propagated down: each node must satisfy (min, max) exclusive range. Alternatively, inorder traversal should produce a strictly increasing sequence.
- Edge cases: duplicate values allowed or not (clarify comparator). Use long bounds to avoid overflow when comparing extremes.

Java code (bounds approach)

```java
public class IsBST {
	static class TreeNode { int val; TreeNode left, right; TreeNode(int v){val=v;} }
	public static boolean isValidBST(TreeNode root) { return valid(root,
Long.MIN_VALUE, Long.MAX_VALUE); }
	private static boolean valid(TreeNode r, long lo, long hi) {
		if (r == null) return true;
		if (r.val <= lo || r.val >= hi) return false;
		return valid(r.left, lo, r.val) && valid(r.right, r.val, hi);
	}
}
```

Interviewer discussion

- Show a counterexample where local checks (node > left.val and node < right.val) fail because subtree extremes violate BST property. Highlight inorder method as alternative.

Complexity

- Time: O(n), Space: O(h)

---

65. Top View / Bottom View of Binary Tree

Problem
Return the set of nodes visible from the top (or bottom) of the binary tree when viewed vertically. Nodes are grouped by horizontal distance (HD) from root; for top view pick the first node seen at each HD, for bottom view pick the last.

Approach (detailed)

- BFS with tracking horizontal distance: enqueue pairs (node, hd). Use a map hd -> value. For top view, record value for hd the first time it appears. For bottom view, overwrite the map each time. Finally, extract entries sorted by hd.
- Edge cases: multiple nodes share same hd and level; BFS ensures level order so top view picks topmost.

Java code (top & bottom views)

```java
import java.util.*;
public class VerticalViews {
	static class TreeNode { int val; TreeNode left, right; TreeNode(int v){val=v;} }
	public static List<Integer> topView(TreeNode root) {
		if (root == null) return Collections.emptyList();
		Map<Integer,Integer> map = new TreeMap<>(); Deque<Object[]> q = new
ArrayDeque<>(); q.add(new Object[]{root, 0});
		while (!q.isEmpty()) {
			Object[] cur = q.poll(); TreeNode n = (TreeNode)cur[0]; int hd =
(int)cur[1];
			map.putIfAbsent(hd, n.val);
			if (n.left != null) q.add(new Object[]{n.left, hd-1});
			if (n.right != null) q.add(new Object[]{n.right, hd+1});
		}
		return new ArrayList<>(map.values());
	}
	public static List<Integer> bottomView(TreeNode root) {
		if (root == null) return Collections.emptyList();
		Map<Integer,Integer> map = new TreeMap<>(); Deque<Object[]> q = new
ArrayDeque<>(); q.add(new Object[]{root, 0});
		while (!q.isEmpty()) {
			Object[] cur = q.poll(); TreeNode n = (TreeNode)cur[0]; int hd =
(int)cur[1];
			map.put(hd, n.val); // overwrite so last (lowest) stays
			if (n.left != null) q.add(new Object[]{n.left, hd-1});
			if (n.right != null) q.add(new Object[]{n.right, hd+1});
		}
		return new ArrayList<>(map.values());
	}
}
```

Interviewer discussion

- Discuss the choice of TreeMap vs capturing min/max hd and writing values into an array. Explain why BFS (level order) is essential to preserving top vs bottom selection by level.

Complexity

- Time: O(n log m) if using TreeMap (m distinct HDs), or O(n) with offset array when range known; Space: O(m) map + O(w) queue

---

8. Heaps & Priority Queue

66. Implement Min Heap & Max Heap

Problem
Design and implement a binary heap supporting insert, peek, extract (pop), and optionally decrease/increase-key. Provide both min-heap and max-heap variants.

Approach (detailed)

- A binary heap is usually implemented with an array where parent/child indices map: parent(i)=(i-1)/2, left=2*i+1, right=2*i+2. Insert appends at the end and siftUp; extract replaces root with last element and siftDown. Heap operations are O(log n).
- Consider dynamic resizing for array-backed storage, and provide comparator-based generic implementation to support min or max behavior.
- Edge cases: empty heap (peek/ pop should signal empty), many duplicate values, and concurrency concerns if used in multi-threaded context.

Java code (generic heap wrapper using comparator)

```java
import java.util.*;

public class BinaryHeap<T> {
	private List<T> data = new ArrayList<>();
	private final Comparator<? super T> cmp;
	public BinaryHeap(Comparator<? super T> cmp) { this.cmp = cmp; }
	public boolean isEmpty() { return data.isEmpty(); }
	public int size() { return data.size(); }
	public void add(T v) { data.add(v); siftUp(data.size()-1); }
	public T peek() { return data.isEmpty() ? null : data.get(0); }
	public T poll() { if (data.isEmpty()) return null; T root = data.get(0); T last = data.remove(data.size()-1); if (!data.isEmpty()) { data.set(0, last); siftDown(0); } return root; }
	private void siftUp(int i) {
		while (i > 0) {
			int p = (i - 1) >>> 1;
			if (cmp.compare(data.get(i), data.get(p)) >= 0) break;
			Collections.swap(data, i, p); i = p;
		}
	}
	private void siftDown(int i) {
		int n = data.size(); while (true) {
			int l = (i<<1) + 1; if (l >= n) break;
```

```
                        int r = l + 1; int j = l;
                        if (r < n && cmp.compare(data.get(r), data.get(l)) < 0) j = r;
                        if (cmp.compare(data.get(j), data.get(i)) >= 0) break;
                        Collections.swap(data, i, j); i = j;
                }
        }
        // convenience factories
        public static <T extends Comparable<T>> BinaryHeap<T> minHeap() { return new
BinaryHeap<>(Comparator.naturalOrder()); }
        public static <T extends Comparable<T>> BinaryHeap<T> maxHeap() { return new
BinaryHeap<>(Comparator.reverseOrder()); }

        // simple test driver
        public static void main(String[] args) {
                BinaryHeap<Integer> min = BinaryHeap.minHeap();
                min.add(5); min.add(1); min.add(3); min.add(2);
                while (!min.isEmpty()) System.out.print(min.poll() + " "); // 1 2 3 5
        }
}
```

Interviewer discussion

- Explain array mapping of indices and why siftUp/siftDown maintain heap invariant. Compare binary heap with pairing/ fibonacci heaps for specialized operations (amortized decrease-key).

Complexity

- add/poll: O(log n), peek: O(1), Space: O(n)

---

67. Kth largest/smallest element using heap

Problem
Return the k-th largest (or smallest) element from an unsorted array or data stream.

Approach (detailed)

- Use a min-heap of size k for k-th largest: push elements and when size > k pop smallest; the heap root ends up being k-th largest. For k-th smallest use max-heap similarly. For arrays Quickselect gives expected O(n) time and O(1) space.
- For streaming data where elements arrive online, keep a size-k min-heap and update as elements arrive.

Java code (min-heap streaming style)

```java
import java.util.*;

public class KthHeap {
	public static int findKthLargest(int[] a, int k) {
		PriorityQueue<Integer> pq = new PriorityQueue<>();
		for (int v : a) {
			pq.offer(v);
			if (pq.size() > k) pq.poll();
		}
		return pq.peek();
	}
	public static void main(String[] args) {
		int[] a = {3,2,1,5,6,4};
		System.out.println(findKthLargest(a, 2)); // 5
	}
}
```

Interviewer discussion

- Compare heap approach O(n log k) vs Quickselect average O(n). For very large n with small k, heap is preferred. Discuss stability and duplicates.

Complexity

- Time: O(n log k), Space: O(k)

---

68. Merge K sorted arrays

Problem
Given k sorted arrays, merge them into one sorted array efficiently.

Approach (detailed)

- Use a min-heap (priority queue) that holds the current smallest element from each array along with its origin (array index and element index). Repeatedly extract min and push next element from that array. This is O(N log k) where N is total elements.
- Alternative: pairwise merge using divide-and-conquer reduces overhead and can be implemented with iterative merges (like tournament tree), or use k-way merge using specialized heap structures for faster constant factors.

Java code (k-way merge)

```java
import java.util.*;
```

```
public class MergeKSorted {
	static class Node { int val; int arrIdx; int idxInArr; Node(int v,int a,int
i){val=v;arrIdx=a;idxInArr=i;} }
	public static List<Integer> merge(List<int[]> arrays) {
		PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(n
-> n.val));
		int total = 0;
		for (int i = 0; i < arrays.size(); i++) if (arrays.get(i).length > 0) { pq.offer(new
Node(arrays.get(i)[0], i, 0)); total += arrays.get(i).length; }
		List<Integer> out = new ArrayList<>(total);
		while (!pq.isEmpty()) {
			Node n = pq.poll(); out.add(n.val);
			int next = n.idxInArr + 1;
			if (next < arrays.get(n.arrIdx).length) pq.offer(new
Node(arrays.get(n.arrIdx)[next], n.arrIdx, next));
		}
		return out;
	}
}
```

Interviewer discussion

- Discuss memory locality vs sequential merging and when divide-and-conquer merging is preferable. For extremely large data, use streaming and external merge (merge runs on disk).

Complexity

- Time: O(N log k), Space: O(k + output)

---

69. Find median from data stream

Problem
Design a data structure that supports adding numbers from a stream and querying the median at any time.

Approach (detailed)

- Maintain two heaps: a max-heap for the lower half and a min-heap for the upper half. Keep sizes balanced (difference <=1). The median is either top of one heap (odd count) or average of both tops (even count). This yields O(log n) insertion and O(1) median query.
- Edge cases: even/odd counts, integer vs floating median, large streams (memory), and duplicate values.

Java code (two-heap median finder)

```java
import java.util.*;

public class MedianFinder {
	private PriorityQueue<Integer> lo = new
PriorityQueue<>(Comparator.reverseOrder()); // max-heap
	private PriorityQueue<Integer> hi = new PriorityQueue<>(); // min-heap
	public void addNum(int num) {
		if (lo.isEmpty() || num <= lo.peek()) lo.offer(num); else hi.offer(num);
		// rebalance
		if (lo.size() > hi.size() + 1) hi.offer(lo.poll());
		else if (hi.size() > lo.size() + 1) lo.offer(hi.poll());
	}
	public double findMedian() {
		if (lo.size() == hi.size()) return lo.isEmpty() ? 0.0 : ((lo.peek() + hi.peek()) /
2.0);
		return lo.size() > hi.size() ? lo.peek() : hi.peek();
	}
	public static void main(String[] args) {
		MedianFinder mf = new MedianFinder();
		mf.addNum(1); mf.addNum(2); System.out.println(mf.findMedian()); // 1.5
		mf.addNum(3); System.out.println(mf.findMedian()); // 2
	}
}
```

Interviewer discussion

- Discuss balancing invariants and numeric precision for median (integers vs floats). For very large streams that exceed memory, discuss reservoir sampling or streaming approximations (t-digest) for approximate quantiles.

Complexity

- addNum: O(log n), findMedian: O(1), Space: O(n)

---

9. Graphs

Below are interview-ready writeups for graph problems 70–78. Each entry contains a short contract, approach with edge-cases, a runnable Java sketch (self-contained small class or method), interviewer talking points, and time/space complexity.

---

70. Represent a graph (adjacency list / adjacency matrix)

Contract

- Input: set of nodes and edges (directed or undirected), possibly weighted.
- Output: data structure that supports fast neighbor iteration and edge queries.

Approach

- Two standard representations:
  - Adjacency list: for each node keep a list of neighbors (and weight if weighted). Excellent for sparse graphs, iteration over neighbors is O(deg(v)).
  - Adjacency matrix: an n x n matrix with boolean/weight entries. Constant-time edge existence checks (O(1)), but uses O(n^2) memory — best for dense graphs or small n.
- Choice depends on |V|, |E| and operations required.

Java code (adjacency list, weighted/unweighted)

```java
import java.util.*;

public class GraphRepr {
	static class Edge { int to; int w; Edge(int t){to=t; w=1;} Edge(int t,int w){to=t; this.w=w;} }
	// adjacency list for 0..n-1
	List<List<Edge>> adj;
	public GraphRepr(int n) { adj = new ArrayList<>(); for (int i=0;i<n;i++) adj.add(new ArrayList<>()); }
	public void addEdgeDirected(int u,int v,int w){ adj.get(u).add(new Edge(v,w)); }
	public void addEdgeUndirected(int u,int v,int w){ addEdgeDirected(u,v,w); addEdgeDirected(v,u,w); }
	public boolean hasEdge(int u,int v){ for (Edge e : adj.get(u)) if (e.to==v) return true; return false; }
	// sample driver
	public static void main(String[] args){ GraphRepr g = new GraphRepr(4); g.addEdgeUndirected(0,1,1); g.addEdgeUndirected(1,2,2); System.out.println(g.hasEdge(0,2)); }
}
```

Interviewer discussion

- Explain tradeoffs (memory/time) and when each representation is preferable. Mention specialized representations (compressed sparse row) for performance.

Complexity

- Adjacency list: build O(n + m) time, neighbor iteration O(deg(v)), space O(n + m).

- Adjacency matrix: O(n^2) space, O(1) edge test.

---

71. BFS & DFS (graph traversal)

Contract

- Input: graph (adj list), start node s.
- Output: visitation order (or distances for BFS), discovery/finish times for DFS, and reachable set.

Approach

- BFS: use queue, mark visited, optionally record distance and parent for shortest path in unweighted graphs.
- DFS: use recursion or explicit stack; useful for ordering, cycle detection, and connectivity queries. Track visited[] to avoid revisiting.

Java code (BFS distances and iterative DFS)

```java
import java.util.*;

public class GraphSearch {
	public static int[] bfs(List<List<Integer>> adj, int src) {
		int n = adj.size(); int[] dist = new int[n]; Arrays.fill(dist, -1);
		Deque<Integer> q = new ArrayDeque<>(); q.add(src); dist[src]=0;
		while(!q.isEmpty()){ int u=q.poll(); for(int v: adj.get(u)) if(dist[v]==-1){
dist[v]=dist[u]+1; q.add(v); } }
		return dist;
	}

	public static List<Integer> dfsIter(List<List<Integer>> adj, int src){
		List<Integer> order = new ArrayList<>(); boolean[] vis = new
boolean[adj.size()]; Deque<Integer> st = new ArrayDeque<>(); st.push(src);
		while(!st.isEmpty()){ int u = st.pop(); if(vis[u]) continue; vis[u]=true;
order.add(u); for(int v: adj.get(u)) if(!vis[v]) st.push(v); }
		return order;
	}
}
```

Interviewer discussion

- Explain BFS yields shortest path in unweighted graphs. Discuss recursion depth for DFS and iterative alternatives. Note ordering of neighbors affects DFS/BFS order and deterministic behavior.

Complexity

- Time: O(n + m), Space: O(n) for visited and queue/stack.

---

72. Detect cycle in graph (directed & undirected)

Contract

- Input: graph (adj list).
- Output: boolean indicating cycle; for directed graphs optionally return a cycle path.

Approach

- Undirected: run DFS, track parent; if you see a neighbor that is visited and not parent -> cycle.
- Directed: use DFS coloring (0=unvisited,1=visiting,2=done) — encountering a node with color 1 indicates a back-edge and a cycle. Can also reconstruct cycle by storing parent chain.

Java code (directed using coloring, returns boolean)

```java
import java.util.*;

public class DetectCycleGraph {
	public static boolean hasCycleDirected(List<List<Integer>> adj){
		int n = adj.size(); int[] color = new int[n];
		for (int i=0;i<n;i++) if (color[i]==0) if (dfs(i, adj, color)) return true;
		return false;
	}
	private static boolean dfs(int u, List<List<Integer>> adj, int[] color){
		color[u]=1; for (int v: adj.get(u)){
			if (color[v]==1) return true;
			if (color[v]==0 && dfs(v, adj, color)) return true;
		}
		color[u]=2; return false;
	}
}
```

Interviewer discussion

- Mention difference between cross/forward/back edges in directed graphs. For undirected graphs highlight parent check. For reconstructing cycles, keep parent pointers.

Complexity

- Time: O(n + m), Space: O(n) recursion/stack + O(n) for color/parent arrays.

---

73. Topological sort

Contract

- Input: directed acyclic graph (DAG).
- Output: an ordering of nodes such that for every edge u->v, u appears before v; if cycle exists, report impossibility.

Approach

- Kahn's algorithm (BFS): compute indegree[], push nodes with indegree 0 into queue, repeatedly pop and reduce neighbors' indegree; if processed count < n -> cycle.
- DFS-postorder: run DFS and output nodes on completion (reverse of finishing order) — also detects cycles with coloring.

Java code (Kahn's algorithm)

```java
import java.util.*;

public class TopologicalSort {
	public static List<Integer> kahn(List<List<Integer>> adj){
		int n = adj.size(); int[] indeg = new int[n];
		for (int u=0; u<n; u++) for (int v: adj.get(u)) indeg[v]++;
		Deque<Integer> q = new ArrayDeque<>(); for (int i=0;i<n;i++) if (indeg[i]==0) q.add(i);
		List<Integer> order = new ArrayList<>();
		while(!q.isEmpty()){ int u=q.poll(); order.add(u); for (int v: adj.get(u)) if (--indeg[v]==0) q.add(v); }
		if (order.size() != n) return Collections.emptyList(); // cycle
		return order;
	}
}
```

Interviewer discussion

- Explain why Kahn's algorithm yields valid order and how it detects cycles. Discuss multiple valid orderings and applications (build systems, instruction scheduling).

Complexity

- Time: O(n + m), Space: O(n).

---

74. Shortest paths: Dijkstra & Bellman-Ford

Contract

- Input: weighted graph (non-negative weights for Dijkstra). For Bellman-Ford allow negative weights but no negative cycles.
- Output: shortest distance array (and optionally predecessors) from source.

Approach

- Dijkstra: priority queue (min-heap) keyed by tentative distance; relax edges when popping node; O((n+m) log n) with binary heap.
- Bellman-Ford: relax all edges repeatedly for n-1 iterations; can detect negative cycles if an extra iteration changes distances.

Java code (Dijkstra)

```java
import java.util.*;

public class ShortestPath {
	static class Edge{int to; int w; Edge(int t,int w){to=t;this.w=w;}}
	public static long[] dijkstra(List<List<Edge>> adj, int src){
		int n = adj.size(); long[] dist = new long[n]; Arrays.fill(dist, Long.MAX_VALUE);
		dist[src]=0; PriorityQueue<int[]> pq = new
PriorityQueue<>(Comparator.comparingLong(a->a[0])); // [dist,node]
		pq.offer(new int[]{0, src});
		while(!pq.isEmpty()){
			int[] top = pq.poll(); long d = top[0]; int u = top[1]; if (d != dist[u])
continue;
			for (Edge e : adj.get(u)){
				if (dist[e.to] > dist[u] + e.w){ dist[e.to] = dist[u] + e.w;
pq.offer(new int[]{(int)dist[e.to], e.to}); }
			}
		}
		return dist;
	}
}
```

Interviewer discussion

- Discuss why Dijkstra requires non-negative weights. Mention optimizations: decrease-key via indexed heap or Fibonacci heaps for theoretical improvements. For dense graphs consider adjacency matrix and O(n^2) Dijkstra without heap.

Complexity

- Dijkstra with binary heap: O((n + m) log n). Bellman-Ford: O(n*m).

---

75. Minimum Spanning Tree (Prim's & Kruskal's)

Contract

- Input: undirected weighted connected graph.
- Output: set of edges forming MST with minimum total weight.

Approach

- Kruskal: sort edges by weight and union them using Union-Find (DSU) avoiding cycles — O(m log m) dominated by sorting.
- Prim: grow tree from a start node using a min-heap keyed by smallest crossing edge; for sparse graphs use adjacency lists + heap O(m log n).

Java code (Kruskal sketch using DSU)

```java
import java.util.*;

public class MST {
	static class Edge{int u,v,w; Edge(int u,int v,int w){this.u=u;this.v=v;this.w=w;}}
	static class DSU{int[] p; DSU(int n){p=new int[n]; for(int i=0;i<n;i++)p[i]=i;} int find(int x){return p[x]==x?x:p[x]=find(p[x]);} boolean union(int a,int b){a=find(a);b=find(b); if(a==b) return false; p[b]=a; return true;}}
	public static List<Edge> kruskal(int n, List<Edge> edges){ Collections.sort(edges, Comparator.comparingInt(e->e.w)); DSU dsu = new DSU(n); List<Edge> mst = new ArrayList<>(); for(Edge e: edges) if(dsu.union(e.u,e.v)) mst.add(e); return mst; }
}
```

Interviewer discussion

- Discuss correctness via cut and cycle properties. Mention union by rank and path compression for near-constant amortized DSU operations. Compare Prim vs Kruskal depending on graph density.

Complexity

- Kruskal: O(m log m) time, Prim (heap): O(m log n). Space O(n + m).

---

76. Number of Islands (grid -> graph connectivity)

Contract

- Input: 2D grid of '1' (land) and '0' (water).
- Output: number of connected components of land (4-directional adjacency).

Approach

- Treat grid as implicit graph. Use DFS or BFS to flood-fill each unvisited '1' and mark visited. Count fills.

Java code (BFS flood-fill)

```java
import java.util.*;

public class NumIslands {
	public static int numIslands(char[][] grid){
		int n = grid.length, m = grid[0].length; int cnt=0;
		int[][] dirs = {{1,0},{-1,0},{0,1},{0,-1}};
		boolean[][] vis = new boolean[n][m];
		for(int i=0;i<n;i++) for(int j=0;j<m;j++) if(grid[i][j]=='1' && !vis[i][j]){
			cnt++; Deque<int[]> q = new ArrayDeque<>(); q.add(new int[]{i,j});
vis[i][j]=true;
			while(!q.isEmpty()){ int[] cur=q.poll(); for(int[] d:dirs){int r=cur[0]+d[0],
c=cur[1]+d[1]; if(r>=0&&c>=0&&r<n&&c<m && !vis[r][c] && grid[r][c]=='1'){ vis[r][c]=true;
q.add(new int[]{r,c}); } } }
		}
		return cnt;
	}
}
```

Interviewer discussion

- Clarify adjacency (4 vs 8 directions). For large grids use union-find or in-place marking to save memory. Discuss recursion depth for DFS and prefer iterative BFS when grid is large.

Complexity

- Time: O(n*m), Space: O(n*m) visited or O(1) if modifying input in-place.

---

77. Clone a graph

Contract

- Input: a reference node of a connected graph (nodes with list of neighbors).
- Output: deep copy of the entire graph preserving edges.

Approach

- BFS or DFS with a map original->copy. When visiting a node for first time, create its copy and enqueue neighbors; for each neighbor ensure its copy exists and add to adjacency list of current copy.

Java code (BFS)

```java
import java.util.*;

public class CloneGraph {
	static class Node{ int val; List<Node> neighbors; Node(int v){val=v; neighbors=new ArrayList<>();} }
	public static Node cloneGraph(Node node){
		if (node==null) return null; Map<Node,Node> map = new HashMap<>(); Deque<Node> q = new ArrayDeque<>(); q.add(node); map.put(node, new Node(node.val));
		while(!q.isEmpty()){ Node u = q.poll(); for(Node v: u.neighbors){ if(!map.containsKey(v)){ map.put(v, new Node(v.val)); q.add(v); } map.get(u).neighbors.add(map.get(v)); } }
		return map.get(node);
	}
}
```

Interviewer discussion

- Explain why map is needed to avoid duplicating nodes and to reconstruct edges. Discuss handling of node labels vs object identity.

Complexity

- Time/Space: O(n + m) for nodes and edges.

---

78. Check if a graph is bipartite

Contract

- Input: undirected graph (adj list).
- Output: boolean indicating whether nodes can be colored with two colors so that no edge has same-colored endpoints.

Approach

- BFS coloring: for each component, if uncolored start BFS and assign colors 0/1 alternately; if a neighbor already colored with same color -> not bipartite. DFS coloring is similar.

Java code (BFS coloring)

```java
import java.util.*;

public class BipartiteCheck {
	public static boolean isBipartite(List<List<Integer>> adj){
		int n = adj.size(); int[] color = new int[n]; Arrays.fill(color, -1);
		for (int s=0;s<n;s++) if (color[s]==-1){
			Deque<Integer> q = new ArrayDeque<>(); q.add(s); color[s]=0;
			while(!q.isEmpty()){ int u=q.poll(); for(int v: adj.get(u)){ if (color[v]==-1){
color[v]=color[u]^1; q.add(v); } else if (color[v]==color[u]) return false; } }
		}
		return true;
	}
}
```

Interviewer discussion

- Discuss relation to odd-length cycles: graph is bipartite iff it has no odd cycle. Mention 2-coloring and extension to k-coloring being NP-hard for k>=3.

Complexity

- Time: O(n + m), Space: O(n) for color and queue.

---

10. Dynamic Programming (DP)

This section covers classic DP problems (79–89). Each entry contains: contract (inputs/outputs), detailed approach with state and recurrence, implementation notes and a runnable Java sketch, interviewer talking points, and time/space complexity.

---

79. Fibonacci with DP (bottom-up and fast doubling)

Contract

- Input: integer n >= 0
- Output: F(n) (nth Fibonacci number)

Approach

- Naive recursion repeats work; DP saves subproblem results. Two practical approaches:
  - Bottom-up iterative DP: compute F(0)..F(n) storing last two values — O(n) time, O(1) space.
  - Fast doubling: use matrix exponentiation identities to compute F(n) in O(log n) time (useful for very large n).

Java code (bottom-up + fast doubling with modular option)

```java
import java.math.BigInteger;

public class FibonacciDP {
	// Bottom-up iterative (O(n) time, O(1) space)
	public static long fibIter(int n) {
		if (n <= 1) return n;
		long a = 0, b = 1;
		for (int i = 2; i <= n; i++) {
			long c = a + b; a = b; b = c;
		}
		return b;
	}

	// Fast doubling using BigInteger for large n
	public static BigInteger[] fibFastDoubling(int n) {
		if (n == 0) return new BigInteger[]{BigInteger.ZERO, BigInteger.ONE};
		BigInteger[] ab = fibFastDoubling(n >> 1);
		BigInteger a = ab[0]; BigInteger b = ab[1];
		BigInteger c = a.multiply(b.shiftLeft(1).subtract(a)); // F(2k) = F(k) * (2*F(k+1) − F(k))
		BigInteger d = a.multiply(a).add(b.multiply(b));     // F(2k+1) = F(k)^2 + F(k+1)^2
		if ((n & 1) == 0) return new BigInteger[]{c, d}; else return new BigInteger[]{d, c.add(d)};
	}

	public static void main(String[] args) {
		System.out.println(fibIter(40));              // 102334155
		System.out.println(fibFastDoubling(100)[0]);   // F(100) as BigInteger
	}
```

```
}
```

Interviewer discussion

- Explain why memoization / bottom-up removes exponential repetition. Show derivation of fast-doubling formulas and when to prefer them (very large n or logarithmic requirement).

Complexity

- Bottom-up: Time O(n), Space O(1). Fast-doubling: Time O(log n), Space O(log n) due to recursion.

---

80. Climbing Stairs (DP simple recurrence)

Contract

- Input: n steps
- Output: number of distinct ways to climb to top when you can take 1 or 2 steps

Approach

- Recurrence: ways[n] = ways[n-1] + ways[n-2] with base ways[0]=1, ways[1]=1. This is Fibonacci-like and solved with iterative DP or constant-space variables.

Java code

```java
public class ClimbingStairs {
	public static int climbStairs(int n) {
		if (n <= 1) return 1;
		int a = 1, b = 1; // ways[0], ways[1]
		for (int i = 2; i <= n; i++) { int c = a + b; a = b; b = c; }
		return b;
	}
	public static void main(String[] args) { System.out.println(climbStairs(10)); }
}
```

Interviewer discussion

- Clarify variation allowing different step sizes or forbidden steps (then generalize to coin-change style DP). Discuss overflow for large n and using BigInteger if needed.

Complexity

- Time O(n), Space O(1).

---

81. Longest Increasing Subsequence (LIS)

Contract

- Input: integer array a[]
- Output: length of LIS (and optionally one LIS)

Approach

- Two standard solutions:
  - DP O(n^2): dp[i] = 1 + max(dp[j]) for all j < i and a[j] < a[i]. Use when n ≤ few thousands.
  - Patience sorting / tails O(n log n): maintain an array tails[] where tails[len] = smallest tail of an increasing subsequence of length len. Use binary search to place each element — yields length; to reconstruct one LIS keep predecessor indices.

Java code (O(n log n) length-only)

```java
import java.util.*;

public class LIS {
	// returns length of LIS
	public static int lengthLIS(int[] a) {
		if (a.length == 0) return 0;
		int[] tails = new int[a.length]; int size = 0;
		for (int x : a) {
			int i = Arrays.binarySearch(tails, 0, size, x);
			if (i < 0) i = -(i+1);
			tails[i] = x;
			if (i == size) size++;
		}
		return size;
	}
	public static void main(String[] args) { System.out.println(lengthLIS(new
int[]{10,9,2,5,3,7,101,18})); }
}
```

Interviewer discussion

- Explain tails invariant and why replacing tails[i] with smaller value is safe. Discuss reconstruction (store predecessors and positions). Compare DP vs n log n solution.

Complexity

- DP: O(n^2) time, O(n) space. Patience sorting: O(n log n) time, O(n) space.

---

82. Longest Common Subsequence (LCS)

Contract

- Input: strings A (length n) and B (length m)
- Output: length (and optionally the sequence) of LCS

Approach

- DP on prefixes: dp[i][j] = length of LCS(A[0..i), B[0..j)). Recurrence: if A[i-1]==B[j-1] dp[i][j]=dp[i-1][j-1]+1 else dp[i][j]=max(dp[i-1][j], dp[i][j-1]). Use 2-row optimization for length; reconstruct sequence by keeping full table or backtracking.

Java code (length with space-optimized rows)

```java
public class LCS {
	public static int lcsLen(String a, String b) {
		int n = a.length(), m = b.length();
		int[] prev = new int[m+1], curr = new int[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)) curr[j] = prev[j-1] + 1;
				else curr[j] = Math.max(prev[j], curr[j-1]);
			}
			int[] tmp = prev; prev = curr; curr = tmp; // reuse arrays
		}
		return prev[m];
	}
	public static void main(String[] args) { System.out.println(lcsLen("abcde","ace")); }
}
```

Interviewer discussion

- Discuss tradeoffs for reconstructing the sequence (keep parent pointers or full dp table). Mention variations (LCS vs LCSubstring) and complexity limits when strings are long.

Complexity

- Time O(n*m), Space O(min(n,m)) for length-only optimization, O(n*m) for reconstruction.

---

83. 0/1 Knapsack Problem

Contract

- Input: n items with weights w[i] and values v[i], capacity W
- Output: maximum value achievable without exceeding capacity; items chosen (optional)

Approach

- Classic DP: dp[i][cap] = best value using first i items with capacity cap. Use 1D rolling array dp[cap] iterating items and capacities backwards to avoid reuse in same item.

Java code (value-only DP, 1D)

```java
public class Knapsack01 {
	public static int knapSack(int[] wt, int[] val, int W) {
		int[] dp = new int[W+1];
		for (int i = 0; i < wt.length; i++) {
			for (int cap = W; cap >= wt[i]; cap--) {
				dp[cap] = Math.max(dp[cap], dp[cap - wt[i]] + val[i]);
			}
		}
		return dp[W];
	}
	public static void main(String[] args) {
		System.out.println(knapSack(new int[]{1,3,4}, new int[]{15,50,60}, 4));
	}
}
```

Interviewer discussion

- Explain why capacity loop goes backward to prevent reusing the same item. Discuss reconstruction (store choices or use 2D table) and variants (unbounded knapsack where inner loop goes forward).

Complexity

- Time O(n*W), Space O(W) with 1D optimization.

---

84. Coin Change (minimum coins / ways)

Contract

- Input: coin denominations coins[] and amount S
- Output: minimum number of coins to make S (or number of ways)

Approach

- Minimum coins (unlimited supply): dp[0]=0; for each coin for cap from coin..S dp[cap]=min(dp[cap], dp[cap-coin]+1). For ways: dpWays[0]=1 and add counts instead of min. Order of loops matters depending on whether permutations or combinations are required.

Java code (min coins)

```java
import java.util.*;

public class CoinChange {
	public static int coinChangeMin(int[] coins, int amount) {
		int INF = amount + 1; int[] dp = new int[amount+1]; Arrays.fill(dp, INF);
dp[0]=0;
		for (int coin : coins) for (int a = coin; a <= amount; a++) dp[a] =
Math.min(dp[a], dp[a-coin] + 1);
		return dp[amount] > amount ? -1 : dp[amount];
	}
	public static void main(String[] args) { System.out.println(coinChangeMin(new
int[]{1,2,5}, 11)); }
}
```

Interviewer discussion

- Clarify whether order matters (combinations vs permutations). Discuss unbounded vs bounded coin supply and using DP vs BFS for large amounts.

Complexity

- Time O(S * k) where k = number of coins, Space O(S).

---

85. Matrix Chain Multiplication (optimal parenthesization)

Contract

- Input: sequence of matrices with dimensions p0 x p1, p1 x p2, ..., pn-1 x pn
- Output: minimum number of scalar multiplications to multiply the chain

Approach

- DP on interval: dp[i][j] = min over k in [i..j-1] of dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j]. Compute increasing interval lengths. Reconstruct optimal split with a split table.

Java code (cost only, O(n^3))

```java
import java.util.*;

public class MatrixChain {
	public static long matrixChain(int[] p) {
		int n = p.length - 1; long[][] dp = new long[n+1][n+1];
		for (int len = 2; len <= n; len++) {
			for (int i = 1; i + len - 1 <= n; i++) {
				int j = i + len - 1; dp[i][j] = Long.MAX_VALUE;
				for (int k = i; k < j; k++) {
					long cost = dp[i][k] + dp[k+1][j] + 1L * p[i-1] * p[k] * p[j];
					if (cost < dp[i][j]) dp[i][j] = cost;
				}
			}
		}
		return dp[1][n];
	}
	public static void main(String[] args) { System.out.println(matrixChain(new
int[]{40,20,30,10,30})); }
}
```

Interviewer discussion

- Explain interval DP and how to derive recurrence. Mention optimization techniques (Knuth optimization) when applicable and complexity tradeoffs.

Complexity

- Time O(n^3), Space O(n^2).

---

86. Word Break (DP)

Contract

- Input: string s and dictionary set words
- Output: boolean whether s can be segmented into space-separated sequence of dictionary words (optionally return one segmentation)

Approach

- DP on prefix: dp[i] = true if there exists j < i with dp[j] true and s[j..i) in dict. Use a boolean array and optionally keep backpointers to reconstruct a solution. Use word length bounds to limit inner loop.

Java code

```java
import java.util.*;

public class WordBreak {
	public static boolean wordBreak(String s, Set<String> dict) {
		int n = s.length(); boolean[] dp = new boolean[n+1]; dp[0] = true;
		int maxLen = 0; for (String w: dict) maxLen = Math.max(maxLen, w.length());
		for (int i = 1; i <= n; i++) {
			for (int l = 1; l <= Math.min(i, maxLen); l++) {
				if (!dp[i-l]) continue;
				if (dict.contains(s.substring(i-l, i))) { dp[i] = true; break; }
			}
		}
		return dp[n];
	}
	public static void main(String[] args) {
		Set<String> dict = new HashSet<>(Arrays.asList("leet","code"));
		System.out.println(wordBreak("leetcode", dict));
	}
}
```

Interviewer discussion

- Discuss reconstruction using parent pointers. Mention trie-based acceleration and BFS variant for large dicts or to produce shortest segmentation.

Complexity

- Time O(n * maxWordLen), Space O(n).

---

87. Maximum Product Subarray

Contract

- Input: integer array a[] (may contain negative numbers)
- Output: maximum product of any contiguous subarray

Approach

- Because negative numbers flip sign, track both maxEndingHere and minEndingHere at each position: newMax = max(a[i], a[i]*prevMax, a[i]*prevMin); newMin similar. Update global max.

Java code

```java
public class MaxProductSubarray {
	public static int maxProduct(int[] a) {
		int best = a[0]; int curMax = a[0], curMin = a[0];
		for (int i = 1; i < a.length; i++) {
			int x = a[i];
			int t1 = Math.max(x, Math.max(curMax * x, curMin * x));
			int t2 = Math.min(x, Math.min(curMax * x, curMin * x));
			curMax = t1; curMin = t2; best = Math.max(best, curMax);
		}
		return best;
	}
	public static void main(String[] args) { System.out.println(maxProduct(new
int[]{2,3,-2,4})); }
}
```

Interviewer discussion

- Explain why tracking min as well as max is necessary. Discuss overflow and when using long/BigInteger is required.

Complexity

- Time O(n), Space O(1).

---

88. Edit Distance (Levenshtein) — DP revisited

Note: Edit distance was introduced earlier, here we add a concise DP table reconstruction note.

Approach (recap)

- dp[i][j] stores min ops to convert A[0..i) to B[0..j). Use choices insert/delete/replace. To reconstruct the edit sequence, maintain parent pointers indicating which operation produced dp[i][j]. Use space optimization only when reconstruction is not required.

Java code (full DP with backtracking to recover operations)

```java
```

```
import java.util.*;

public class EditDistanceWithOps {
	static class Op { int i,j; String action; }
	public static List<String> getEdits(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]));
		}
		List<String> ops = new ArrayList<>(); int i=n,j=m;
		while (i>0 || j>0) {
			if (i>0 && j>0 && a.charAt(i-1)==b.charAt(j-1)) { i--; j--; }
			else if (i>0 && j>0 && dp[i][j]==dp[i-1][j-1]+1) { ops.add("Replace " +
a.charAt(i-1) + "->" + b.charAt(j-1)); i--; j--; }
			else if (i>0 && dp[i][j]==dp[i-1][j]+1) { ops.add("Delete " +
a.charAt(i-1)); i--; }
			else { ops.add("Insert " + b.charAt(j-1)); j--; }
		}
		Collections.reverse(ops); return ops;
	}
	public static void main(String[] args) { System.out.println(getEdits("kitten","sitting")); }
}
```

Complexity

- Time O(n*m), Space O(n*m) for reconstruction; optimized to O(min(n,m)) if only distance required.

---

89. Palindrome Partitioning (minimum cuts)

Contract

- Input: string s
- Output: minimum number of cuts needed to partition s into palindromic substrings

Approach

- Precompute isPal[i][j] via DP (expand or fill table) then dp[i] = min cuts for prefix s[0..i]: dp[i] = 0 if s[0..i] is palindrome else min over j<=i of dp[j-1]+1 if s[j..i] palindrome.

Java code

```java
```

```
import java.util.*;

public class PalindromePartition {
	public static int minCut(String s) {
		int n = s.length(); boolean[][] isPal = new boolean[n][n];
		for (int len=1; len<=n; len++) for (int i=0; i+len-1<n; i++) {
			int j = i+len-1;
			if (s.charAt(i)==s.charAt(j) && (len<=2 || isPal[i+1][j-1])) isPal[i][j]=true;
		}
		int[] dp = new int[n]; Arrays.fill(dp, Integer.MAX_VALUE);
		for (int i = 0; i < n; i++) {
			if (isPal[0][i]) dp[i]=0;
			else {
				for (int j = 1; j <= i; j++) if (isPal[j][i]) dp[i] = Math.min(dp[i],
dp[j-1] + 1);
			}
		}
		return dp[n-1];
	}
	public static void main(String[] args) { System.out.println(minCut("aab")); }
}
```

Interviewer discussion

- Mention precomputing palindromes vs expand-around-center on the fly. Discuss tradeoffs and possible O(n) Manacher-based accelerations for special cases.

Complexity

- Time: O(n^2) (palindrome table + cuts), Space: O(n^2) for palindrome table or O(n) if using online checks with extra cost.

---

11. Advanced Topics

90. Disjoint Set Union (DSU / Union-Find)

Problem
Design a data structure that supports union and find operations on disjoint sets. Common uses: Kruskal's MST, connected components, dynamic connectivity.

Approach (detailed)

- Maintain parent[] and size[] (or rank[]). Each node points to its parent; root nodes represent sets. find(x) follows parent links until root; apply path compression to flatten tree by pointing nodes directly to root during find.
- union(a,b) attaches smaller tree under larger (union by size/rank) to keep depth small.
- This yields near-constant amortized time: inverse-Ackermann α(n) per operation.

Edge-cases

- Repeated unions of already-connected nodes — return early.
- Large N: use iterative find to avoid deep recursion.

Java code

```java
import java.util.*;

public class DSU {
	private int[] parent, size;
	public DSU(int n){ parent = new int[n]; size = new int[n]; for(int i=0;i<n;i++){
parent[i]=i; size[i]=1; } }
	public int find(int x){ while (x != parent[x]) { parent[x] = parent[parent[x]]; x = parent[x];
} return x; }
	public boolean union(int a, int b){ int ra = find(a), rb = find(b); if (ra==rb) return false; if
(size[ra] < size[rb]) { parent[ra]=rb; size[rb]+=size[ra]; } else { parent[rb]=ra;
size[ra]+=size[rb]; } return true; }
	public static void main(String[] args){ DSU d = new DSU(5); d.union(0,1);
d.union(3,4); System.out.println(d.find(1)==d.find(0)); System.out.println(d.find(2)); }
}
```

Interviewer discussion

- Explain path compression and union by size/rank. Show amortized complexity and common pitfalls (not compressing, union order causing tall trees).

Complexity

- Amortized time: roughly O(α(n)) per op, Space: O(n).

---

91. Trie (Prefix Tree)

Problem
Implement a prefix tree supporting insert, search (exact), and startsWith (prefix query). Useful for autocomplete, dictionary, and efficient prefix lookups.

Approach (detailed)

- Each node holds children map (fixed alphabet array for small alphabets) and an end-of-word flag. Insert walks/creates nodes per char. Search and prefix check walk nodes and inspect flags.
- Optimize memory with arrays for small alphabets or HashMap/TreeMap for sparse/unicode alphabets. Consider compressing tries (radix/trie) for memory savings.

Java code

```java
import java.util.*;

public class Trie {
	static class Node { Node[] ch = new Node[26]; boolean end; }
	private final Node root = new Node();
	public void insert(String s){ Node cur = root; for(char c: s.toCharArray()){ int i=c-'a';
if(cur.ch[i]==null) cur.ch[i]=new Node(); cur=cur.ch[i]; } cur.end=true; }
	public boolean search(String s){ Node cur = root; for(char c: s.toCharArray()){ cur =
cur.ch[c-'a']; if(cur==null) return false; } return cur.end; }
	public boolean startsWith(String p){ Node cur=root; for(char c: p.toCharArray()){ cur =
cur.ch[c-'a']; if(cur==null) return false; } return true; }
	public static void main(String[] args){ Trie t=new Trie(); t.insert("apple");
System.out.println(t.search("apple")); System.out.println(t.startsWith("app")); }
}
```

Interviewer discussion

- Discuss memory tradeoffs, compressed tries (radix), and using tries for lexicographic ordering or auto-suggestions (store top-k at nodes).

Complexity

- Insert/Search/Prefix: O(L) where L is string length; Space: O(total chars) (can be large).

---

92. Segment Tree (Range Sum / Range Min)

Problem
Support range queries (sum/min) and point or range updates on an array efficiently.

Approach (detailed)

- Build a binary segment tree with nodes storing aggregated info (sum/min). Queries traverse O(log n) nodes; point updates update leaf and ancestors. For range updates use lazy propagation to defer updates and keep O(log n) per operation.

Java code (range sum, point update)

```java
import java.util.*;

public class SegmentTree {
	private final int n; private final long[] tree;
	public SegmentTree(int[] a){ n = a.length; tree = new long[4*n]; build(1,0,n-1,a); }
	private void build(int v,int l,int r,int[] a){ if(l==r){ tree[v]=a[l]; } else{ int m=(l+r)/2;
build(v*2,l,m,a); build(v*2+1,m+1,r,a); tree[v]=tree[v*2]+tree[v*2+1]; } }
	public long query(int ql,int qr){ return query(1,0,n-1,ql,qr); }
	private long query(int v,int l,int r,int ql,int qr){ if(ql>r||qr<l) return 0; if(ql<=l&&r<=qr)
return tree[v]; int m=(l+r)/2; return query(v*2,l,m,ql,qr)+query(v*2+1,m+1,r,ql,qr); }
	public void update(int idx,int val){ update(1,0,n-1,idx,val); }
	private void update(int v,int l,int r,int idx,int val){ if(l==r){ tree[v]=val; } else{ int
m=(l+r)/2; if(idx<=m) update(v*2,l,m,idx,val); else update(v*2+1,m+1,r,idx,val);
tree[v]=tree[v*2]+tree[v*2+1]; } }
	public static void main(String[] args){ int[] a={1,2,3,4,5}; SegmentTree st=new
SegmentTree(a); System.out.println(st.query(1,3)); st.update(2,10);
System.out.println(st.query(1,3)); }
}
```

Interviewer discussion

- Explain tree layout, query recursion, and lazy propagation for range updates. Compare Fenwick tree for simpler sum queries vs segment tree for more complex associative operations.

Complexity

- Query/point-update: O(log n), Build: O(n), Space: O(n) (4n array).

---

93. Fenwick Tree (Binary Indexed Tree)

Problem
Support prefix-sum queries and point updates on an array in O(log n) time with lower constant factor than segment tree and less memory.

Approach (detailed)

- Fenwick tree stores partial sums in an array using least-significant-bit jumps. sum(idx) accumulates by subtracting lowbit, update adds value at indices by adding lowbit.

Java code

```java
public class Fenwick {
	private final long[] bit; private final int n;
	public Fenwick(int n){ this.n=n; bit = new long[n+1]; }
	public void add(int idx,long val){ for(int i=idx;i<=n;i+=i&-i) bit[i]+=val; }
	public long sum(int idx){ long s=0; for(int i=idx;i>0;i-=i&-i) s+=bit[i]; return s; }
	public long rangeSum(int l,int r){ return sum(r)-sum(l-1); }
	public static void main(String[] args){ Fenwick f=new Fenwick(5); f.add(1,1);
f.add(2,2); f.add(3,3); System.out.println(f.rangeSum(1,3)); }
}
```

Interviewer discussion

- Discuss 1-indexed implementation, lowbit trick, and limitations (no easy range-update without dual trees). Compare to segment trees for non-commutative ops.

Complexity

- Update and prefix-sum: O(log n), Space: O(n).

---

94. KMP Algorithm (Knuth-Morris-Pratt)

Problem
Find the first occurrence of a pattern (needle) in text (haystack) efficiently in O(n + m) time.

Approach (detailed)

- Build LPS (longest proper prefix which is suffix) array for pattern; then scan text while maintaining matched length using LPS to skip comparisons when mismatch occurs.

Java code

```java
public class KMP {
	public static int strStr(String text, String pat){ if(pat.length()==0) return 0; int[] lps =
buildLPS(pat); int i=0,j=0;
		while(i<text.length()){
			if(text.charAt(i)==pat.charAt(j)){ i++; j++; if(j==pat.length()) return i-j; }
			else if(j>0) j = lps[j-1]; else i++; }
		return -1;
	}
	private static int[] buildLPS(String p){ int n=p.length(); int[] lps=new int[n]; int
len=0,i=1; while(i<n){ if(p.charAt(i)==p.charAt(len)) lps[i++]=++len; else if(len>0)
len=lps[len-1]; else lps[i++]=0; } return lps; }
	public static void main(String[] args){ System.out.println(strStr("hello","ll")); }
```

}
```

Interviewer discussion

- Explain LPS computation and correctness proof. Compare with Rabin-Karp when average-case hashing is acceptable and discuss worst-case guarantees.

Complexity

- Time: O(n + m), Space: O(m).

---

95. Rabin-Karp Algorithm

Problem
String search using rolling hash to locate pattern occurrences; useful for multiple-pattern search with hashing structures.

Approach (detailed)

- Compute rolling hash for window of length m in text and compare with pattern hash; on hash match verify by direct compare to avoid false positives. Choose base and modulus to reduce collisions; for long texts use 64-bit rolling hash with care.

Java code (simple 64-bit rolling hash)

```java
public class RabinKarp {
        public static int search(String text, String pat){ int n=text.length(), m=pat.length();
if(m==0) return 0; long base=911382323, mod=(1L<<61)-1; long ph=0, th=0, pow=1;
                for(int i=0;i<m;i++){ ph = (ph*base + pat.charAt(i)) % mod; th = (th*base +
text.charAt(i)) % mod; if(i>0) pow = (pow*base)%mod; }
                if(ph==th && text.substring(0,m).equals(pat)) return 0;
                for(int i=m;i<n;i++){ th = (th - (long)text.charAt(i-m) * pow % mod + mod) %
mod; th = (th*base + text.charAt(i)) % mod; int start = i-m+1; if(th==ph &&
text.substring(start, start+m).equals(pat)) return start; }
                return -1; }
        public static void main(String[] args){ System.out.println(search("hello","ll")); }
}
```

Interviewer discussion

- Discuss hash collisions, choice of modulus/base, and verifying matches. Mention multi-hash or strong hashing when collisions are costly. Compare to KMP for guaranteed linear time without hashes.
```

Complexity

- Average O(n + m), worst-case O(n*m) if every hash collides (rare with good hash); Space O(1).

---

96. Boyer-Moore Voting Algorithm (Majority Element)

Problem
Given an array, find the element that appears more than n/2 times (guaranteed to exist) in O(n) time and O(1) space.

Approach (detailed)

- Use Boyer-Moore majority vote: maintain candidate and count; sweep once to find candidate, optional second pass to verify frequency.

Java code

```java
public class MajorityElement {
	public static int majority(int[] a){ int cand=0, cnt=0; for(int x:a){ if(cnt==0){ cand=x; cnt=1; } else if(cand==x) cnt++; else cnt--; } return cand; }
	public static void main(String[] args){ System.out.println(majority(new int[]{2,2,1,1,2,2,2})); }
}
```

Interviewer discussion

- Explain why algorithm works (pairwise cancelation). For k-majority generalization (elements > n/k) use HashMap or Misra-Gries algorithm.

Complexity

- Time: O(n), Space: O(1).

---

97. Median of Two Sorted Arrays

Problem
Given two sorted arrays, find the median of the combined sorted array in O(log(min(n,m))) time.

Approach (detailed)

- Binary search on partition: partition both arrays so left parts contain half of combined elements and max(lefts) <= min(rights). Adjust partition using binary search on smaller array until invariant holds. Handle even/odd combined length.

Java code (outline)

```java
public class MedianTwoSorted {
	public static double findMedian(int[] A,int[] B){ if(A.length>B.length) return findMedian(B,A);
		int n=A.length,m=B.length, half=(n+m+1)/2; int lo=0,hi=n;
		while(lo<=hi){ int i=(lo+hi)/2; int j=half-i; int Aleft = (i==0)?Integer.MIN_VALUE:A[i-1]; int Aright=(i==n)?Integer.MAX_VALUE:A[i];
			int Bleft=(j==0)?Integer.MIN_VALUE:B[j-1]; int Bright=(j==m)?Integer.MAX_VALUE:B[j];
			if(Aleft<=Bright && Bleft<=Aright){ if(((n+m)&1)==1) return Math.max(Aleft,Bleft); else return (Math.max(Aleft,Bleft)+Math.min(Aright,Bright))/2.0; }
			else if(Aleft>Bright) hi=i-1; else lo=i+1;
		}
		return 0.0; }
	public static void main(String[] args){ System.out.println(findMedian(new int[]{1,3}, new int[]{2})); }
}
```

Interviewer discussion

- Walk through partition invariants and edge cases (empty arrays). Compare with merging approach O(n+m) when arrays small.

Complexity

- Time: O(log(min(n,m))), Space: O(1).

---

98. Maximum Flow (Ford-Fulkerson / Edmonds-Karp)

Problem
Given a directed graph with capacities, compute max flow from source to sink.

Approach (detailed)

- Ford-Fulkerson: repeatedly find augmenting path in residual graph and push flow; Edmonds-Karp uses BFS for shortest augmenting paths (in edges) which guarantees O(V*E^2) worst-case bound. Implement residual capacities and augment along path. Use Dinic for better performance in practice.

Java code (Edmonds-Karp sketch)

```java
import java.util.*;
public class MaxFlow {
	static class Edge{int v; int cap; int rev; Edge(int v,int cap,int rev){this.v=v;this.cap=cap;this.rev=rev;}}
	int n; List<Edge>[] g;
	public MaxFlow(int n){ this.n=n; g=new List[n]; for(int i=0;i<n;i++) g[i]=new ArrayList<>(); }
	void addEdge(int u,int v,int cap){ g[u].add(new Edge(v,cap,g[v].size())); g[v].add(new Edge(u,0,g[u].size()-1)); }
	public int maxFlow(int s,int t){ int flow=0; while(true){ int[] parent = new int[n]; Edge[] parentEdge = new Edge[n]; Arrays.fill(parent,-1); Deque<Integer> q = new ArrayDeque<>(); parent[s]=s; q.add(s);
			while(!q.isEmpty() && parent[t]==-1){ int u=q.poll(); for(Edge e: g[u]) if(parent[e.v]==-1 && e.cap>0){ parent[e.v]=u; parentEdge[e.v]=e; q.add(e.v);} }
			if(parent[t]==-1) break; int aug = Integer.MAX_VALUE; for(int v=t; v!=s; v=parent[v]) aug = Math.min(aug, parentEdge[v].cap);
			for(int v=t; v!=s; v=parent[v]){ Edge e=parentEdge[v]; e.cap -= aug; g[v].get(e.rev).cap += aug; }
			flow += aug;
		} return flow; }
	public static void main(String[] args){ MaxFlow mf=new MaxFlow(4); mf.addEdge(0,1,3); mf.addEdge(0,2,2); mf.addEdge(1,2,1); mf.addEdge(1,3,2); mf.addEdge(2,3,4); System.out.println(mf.maxFlow(0,3)); }
}
```

Interviewer discussion

- Discuss residual graph, augmenting paths, complexity of Edmonds-Karp vs Dinic, and applications (bipartite matching, circulation). Mention integer capacities assumption for Ford-Fulkerson termination.

Complexity

- Edmonds-Karp: O(V _ E^2), Dinic: O(E _ sqrt(V)) for unit networks or O(min(V^{2/3}, sqrt(E)) * E) variants; Space O(V + E).

---

99. LRU / LFU Cache Design (brief)

Problem
Design caches with eviction policies: LRU (least recently used) and LFU (least frequently used) supporting O(1) operations.

Approach (detailed)

- LRU: double-linked list + hashmap from key->node for O(1) get/put and move-to-front. Java's LinkedHashMap provides a ready implementation.
- LFU: maintain frequency lists (map from freq -> ordered doubly-linked list of keys) and a map key->(value,freq,node). Update freq on access and evict from lowest freq list; careful handling of ties.

Java code (LRU using LinkedHashMap)

```java
import java.util.*;
public class SimpleLRU {
	private final LinkedHashMap<Integer,Integer> map;
	public SimpleLRU(int cap){ map = new
LinkedHashMap<Integer,Integer>(16,0.75f,true){ protected boolean
removeEldestEntry(Map.Entry<Integer,Integer> e){ return size()>cap; } }; }
	public Integer get(int k){ return map.getOrDefault(k,-1); }
	public void put(int k,int v){ map.put(k,v); }
}
```

Interviewer discussion

- Explain data structures for O(1) ops and tradeoffs. For LFU discuss maintaining minFreq and per-frequency lists and edge cases when updating frequencies.

Complexity

- get/put: O(1) amortized, Space: O(capacity).

-