20 Top Array-Based Interview Questions — Step-by-step Java Solutions

A compact, copy-paste friendly collection of 20 common array interview problems with a clear approach, complexity analysis, and a runnable Java snippet for each. Save the snippets as separate `.java` files if you want to run them individually.

Problem 1 — Find the largest and smallest element in an array

Approach (step-by-step) 1. Validate input (non-empty array). 2. Initialize `min` and `max` as the first element. 3. Scan the array once: update `min` and `max` when you find smaller/larger values.

Complexity: O(n) time, O(1) extra space.

```java
// Problem1_MinMax.java
import java.util.Arrays;

class Problem1_MinMax {
    public static int[] minMax(int[] a){
        if(a == null || a.length == 0)
            throw new IllegalArgumentException("Array must not be 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};
    }

    public static void main(String[] args){
        int[] arr = {6,2,3,5,7,11,8,9,10,13,4};
        int[] res = minMax(arr);
        System.out.println("Array: " + Arrays.toString(arr));
        System.out.println("Min: " + res[0] + ", Max: " + res[1]);
    }
}
```

Problem 2 — Second largest and second smallest element

Approach (step-by-step) 1. Use two nullable `Integer` references for largest and secondLargest (and similarly for smallest). 2. Iterate once: update candidates while skipping duplicates. 3. After the loop check whether second exists.

Complexity: O(n) time, O(1) space.

```
// Problem2_SecondMinMax.java
import java.util.Arrays;

class Problem2_SecondMinMax {
    public static Integer[] secondMinMax(int[] a){
        if(a == null || a.length == 0) return new Integer[]{null, null, null,
null};
        Integer largest = null, secondLargest = null;
        Integer smallest = null, secondSmallest = null;
        for(int v : a){
            // largest
            if(largest == null || v > largest){
                secondLargest = largest;
                largest = v;
            } else if(v != largest && (secondLargest == null || v >
secondLargest)){
                secondLargest = v;
            }
            // smallest
            if(smallest == null || v < smallest){
                secondSmallest = smallest;
                smallest = v;
            } else if(v != smallest && (secondSmallest == null || v <
secondSmallest)){
                secondSmallest = v;
            }
        }
        return new Integer[]{secondSmallest, smallest, secondLargest, largest};
    }

    public static void main(String[] args){
        int[] arr = {4,2,5,1,3,5,2};
        Integer[] r = secondMinMax(arr);
        System.out.println("Array: " + Arrays.toString(arr));
        System.out.println("Second smallest: " + r[0] + ", Smallest: " + r[1]);
        System.out.println("Second largest: " + r[2] + ", Largest: " + r[3]);
```

```
    }
}
```

Problem 3 — Reverse an array (in-place)

Approach: Use two pointers (start, end) and swap until they meet.

Complexity: O(n) time, O(1) space.

```
// Problem3_Reverse.java
import java.util.Arrays;

class Problem3_Reverse {
    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};
        System.out.println("Before: " + Arrays.toString(a));
        reverse(a);
        System.out.println("After : " + Arrays.toString(a));
    }
}
```

Problem 4 — Check if an array is a palindrome

Approach: Two pointers compare elements from both ends.

Complexity: O(n) time, O(1) space.

```
// Problem4_IsPalindrome.java

class Problem4_IsPalindrome {
    public static boolean isPalindrome(int[] a){
        if(a == null) return true;
```

```
        int i=0, j=a.length-1;
        while(i<j){
            if(a[i] != a[j]) return false;
            i++; j--;
        }
        return true;
    }

    public static void main(String[] args){
        int[] a = {1,2,3,2,1};
        System.out.println(isPalindrome(a)); // true
    }
}
```

Problem 5 — Rotate an array by k steps (left & right) — in-place

Approach: Use the reversal trick (three reversals) to rotate in O(n) time and O(1) space.

Complexity: O(n), O(1).

```
// Problem5_Rotate.java
import java.util.Arrays;

class Problem5_Rotate {
    private static void reverse(int[] a, int l, int r){
        while(l < r){ int t = a[l]; a[l++] = a[r]; a[r--] = t; }
    }

    public static void rotateRight(int[] a, int k){
        int n = a.length; if(n == 0) return;
        k = ((k % n) + n) % n; // normalize
        reverse(a, 0, n-1);
        reverse(a, 0, k-1);
        reverse(a, k, n-1);
    }

    public static void rotateLeft(int[] a, int k){
        int n = a.length; if(n == 0) return;
        k = ((k % n) + n) % n;
        reverse(a, 0, k-1);
        reverse(a, k, n-1);
        reverse(a, 0, n-1);
    }
```

```
    public static void main(String[] args){
        int[] a = {1,2,3,4,5,6,7};
        rotateRight(a, 3);
        System.out.println("Right rotate by 3 -> " + Arrays.toString(a));
        rotateLeft(a, 3);
        System.out.println("Left rotate by 3  -> " + Arrays.toString(a));
    }
}
```

Problem 6 — Find the missing number in array containing numbers from 1..n (one missing)

Approach: Use XOR of all indices 1..n and XOR with array elements. The leftover is the missing number (no overflow).

Complexity: O(n) time, O(1) space.

```
// Problem6_MissingNumber.java

class Problem6_MissingNumber {
    public static int findMissing(int[] a){
        int n = a.length + 1; // one number missing
        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 = {1,2,4,5,6}; // missing 3, n = 6
        System.out.println("Missing = " + findMissing(a));
    }
}
```

Problem 7 — Find duplicate elements in an array

Approach: Use a `HashSet` while scanning; duplicates are when the `add()` returns false.

Complexity: O(n) time, O(n) space.

```java
// Problem7_Duplicates.java
import java.util.*;

class Problem7_Duplicates {
    public static List<Integer> findDuplicates(int[] a){
        Set<Integer> seen = new HashSet<>();
        List<Integer> duplicates = new ArrayList<>();
        for(int v : a){
            if(!seen.add(v)) duplicates.add(v);
        }
        return duplicates;
    }

    public static void main(String[] args){
        int[] a = {1,2,3,2,4,3,5};
        System.out.println(findDuplicates(a)); // [2,3]
    }
}
```

Problem 8 — Remove duplicates from sorted / unsorted array

Approach: - **Sorted array:** two-pointer in-place removal (like `unique`). - **Unsorted:** use a `LinkedHashSet` to preserve insertion order.

Complexity: Sorted: O(n) time, O(1) extra. Unsorted: O(n) time, O(n) space.

```java
// Problem8_RemoveDuplicates.java
import java.util.*;

class Problem8_RemoveDuplicates {
    // For sorted arrays — returns new length and modifies in place
    public static int removeDuplicatesSorted(int[] a){
        if(a == null || a.length == 0) return 0;
        int write = 1;
        for(int i=1;i<a.length;i++){
            if(a[i] != a[write-1]) a[write++] = a[i];
        }
        return write; // valid elements in a[0..write-1]
    }

    // For unsorted arrays — return array with unique elements (preserve order)
    public static int[] removeDuplicatesUnsorted(int[] a){
        if(a == null) return new int[0];
```

```
        LinkedHashSet<Integer> set = new LinkedHashSet<>();
        for(int v: a) set.add(v);
        int[] res = new int[set.size()];
        int i=0; for(int v: set) res[i++] = v;
        return res;
    }

    public static void main(String[] args){
        int[] sorted = {1,1,2,2,3,4,4};
        int newLen = removeDuplicatesSorted(sorted);
        System.out.println("Sorted unique: " +
Arrays.toString(Arrays.copyOf(sorted, newLen)));

        int[] unsorted = {3,1,2,3,2,4};
        System.out.println("Unsorted unique: " +
Arrays.toString(removeDuplicatesUnsorted(unsorted)));
    }
}
```

Problem 9 — Frequency of each element

Approach: Use a `HashMap<Integer,Integer>` to count frequencies.

Complexity: O(n) time, O(n) space.

```
// Problem9_Frequency.java
import java.util.*;

class Problem9_Frequency {
    public static Map<Integer, Integer> freq(int[] a){
        Map<Integer,Integer> map = new HashMap<>();
        for(int v: a) map.put(v, map.getOrDefault(v,0) + 1);
        return map;
    }

    public static void main(String[] args){
        int[] a = {1,2,2,3,3,3};
        System.out.println(freq(a)); // {1=1, 2=2, 3=3}
    }
}
```

Problem 10 — Sum of all elements in an array

Approach: Accumulate in a loop. For very large arrays consider `long` to avoid overflow.

Complexity: O(n), O(1).

```
// Problem10_Sum.java

class Problem10_Sum {
    public static long sum(int[] a){
        long s = 0;
        for(int v: a) s += v;
        return s;
    }
    public static void main(String[] args){
        int[] a = {1,2,3,4};
        System.out.println(sum(a)); // 10
    }
}
```

Problem 11 — Maximum subarray (Kadane's Algorithm)

Approach: Maintain `currentMax` and `globalMax` while scanning. Reset `currentMax` when it becomes negative.

Complexity: O(n), O(1).

```
// Problem11_Kadane.java

class Problem11_Kadane {
    public static int maxSubarray(int[] a){
        int maxSoFar = Integer.MIN_VALUE, curMax = 0;
        for(int v : a){
            curMax = curMax + v;
            if(maxSoFar < curMax) maxSoFar = curMax;
            if(curMax < 0) curMax = 0;
        }
        return maxSoFar;
    }

    public static void main(String[] args){
        int[] a = {-2,1,-3,4,-1,2,1,-5,4};
        System.out.println(maxSubarray(a)); // 6 (subarray [4,-1,2,1])
```

```
    }
}
```

Problem 12 — Intersection of two arrays

Approach: Use `HashSet` for one array and iterate the other. Decide whether you want **unique** intersection or with duplicates. Below: unique intersection.

Complexity: O(n + m) time, O(min(n,m)) space.

```java
// Problem12_Intersection.java
import java.util.*;

class Problem12_Intersection {
    public static int[] intersection(int[] a, int[] b){
        Set<Integer> s = new HashSet<>();
        for(int v: a) s.add(v);
        Set<Integer> inter = new HashSet<>();
        for(int v: b) if(s.contains(v)) inter.add(v);
        int[] r = new int[inter.size()]; int i=0; for(int v: inter) r[i++] = v;
        return r;
    }

    public static void main(String[] args){
        int[] a = {1,2,2,3};
        int[] b = {2,2,4};
        System.out.println(Arrays.toString(intersection(a,b))); // [2]
    }
}
```

Problem 13 — Union of two arrays

Approach: Use a `LinkedHashSet` to preserve first-seen order and collect elements from both arrays.

Complexity: O(n + m), O(n + m) space.

```java
// Problem13_Union.java
import java.util.*;

class Problem13_Union {
    public static int[] union(int[] a, int[] b){
```

```
        LinkedHashSet<Integer> set = new LinkedHashSet<>();
        for(int v: a) set.add(v);
        for(int v: b) set.add(v);
        int[] r = new int[set.size()]; int i=0; for(int v: set) r[i++] = v;
        return r;
    }

    public static void main(String[] args){
        int[] a = {1,2,3};
        int[] b = {2,4};
        System.out.println(Arrays.toString(union(a,b))); // [1,2,3,4]
    }
}
```

Problem 14 — Sort an array without built-in functions (QuickSort)

Approach: Implement QuickSort (divide & conquer). You can also use MergeSort; QuickSort is provided here.

Complexity: Average O(n log n), worst O(n^2) (if pivot poor). Space: O(log n) recursion.

```
// Problem14_QuickSort.java
import java.util.Arrays;

class Problem14_QuickSort {
    public static void quickSort(int[] a){
        qs(a, 0, a.length-1);
    }
    private static void qs(int[] a, int lo, int hi){
        if(lo >= hi) return;
        int p = partition(a, lo, hi);
        qs(a, lo, p-1);
        qs(a, p+1, hi);
    }
    private static int partition(int[] a, int lo, int hi){
        int pivot = a[hi];
        int i = lo;
        for(int j=lo;j<hi;j++){
            if(a[j] <= pivot){
                int t = a[i]; a[i] = a[j]; a[j] = t; i++;
            }
        }
        int t = a[i]; a[i] = a[hi]; a[hi] = t;
        return i;
```

```
    }

    public static void main(String[] args){
        int[] a = {3,6,1,5,2,4};
        quickSort(a);
        System.out.println(Arrays.toString(a));
    }
}
```

Problem 15 — Sort array of 0s, 1s, 2s (Dutch National Flag)

Approach: Maintain pointers `low`, `mid`, `high` and swap appropriately in one pass.

Complexity: O(n), O(1).

```
// Problem15_DutchFlag.java
import java.util.Arrays;

class Problem15_DutchFlag {
    public static void sort012(int[] a){
        int low = 0, mid = 0, high = a.length - 1;
        while(mid <= high){
            if(a[mid] == 0) { int t = a[low]; a[low] = a[mid]; a[mid] = t; low+
+; mid++; }
            else if(a[mid] == 1) { mid++; }
            else { int t = a[mid]; a[mid] = a[high]; a[high] = t; high--; }
        }
    }

    public static void main(String[] args){
        int[] a = {2,0,2,1,1,0};
        sort012(a);
        System.out.println(Arrays.toString(a)); // [0,0,1,1,2,2]
    }
}
```

Problem 16 — Find majority element (> n/2 times) (Boyer-Moore)

Approach: First pass find candidate via Boyer-Moore; second pass verify the candidate.

Complexity: O(n), O(1).

```java
// Problem16_MajorityElement.java

class Problem16_MajorityElement {
    public static Integer majority(int[] a){
        int count = 0, candidate = 0;
        for(int v: a){
            if(count == 0){ candidate = v; count = 1; }
            else if(candidate == v) count++;
            else count--;
        }
        // verify
        int freq = 0;
        for(int v: a) if(v == candidate) freq++;
        return freq > a.length/2 ? candidate : null;
    }

    public static void main(String[] args){
        int[] a = {2,2,1,1,2,2,2};
        System.out.println(majority(a)); // 2
    }
}
```

Problem 17 — Leaders in an array (element greater than all to its right)

Approach: Scan from right to left keeping a running `maxFromRight`. If current > max, it's a leader.

Complexity: O(n), O(k) to store leaders.

```java
// Problem17_Leaders.java
import java.util.*;

class Problem17_Leaders {
    public static List<Integer> leaders(int[] a){
        List<Integer> res = new ArrayList<>();
        int max = Integer.MIN_VALUE;
        for(int i=a.length-1; i>=0; i--){
            if(a[i] > max){ res.add(a[i]); max = a[i]; }
        }
        Collections.reverse(res);
        return res;
    }
```

```
    public static void main(String[] args){
        int[] a = {16,17,4,3,5,2};
        System.out.println(leaders(a)); // [17,5,2]
    }
}
```

Problem 18 — Equilibrium index (left sum == right sum)

Approach: Compute total sum, scan while maintaining left sum; at index i check `left == total - left - a[i]`.

Complexity: O(n), O(1).

```
// Problem18_Equilibrium.java
import java.util.*;

class Problem18_Equilibrium {
    public static List<Integer> equilibriumIndices(int[] a){
        List<Integer> res = new ArrayList<>();
        long total = 0;
        for(int v: a) total += v;
        long left = 0;
        for(int i=0;i<a.length;i++){
            if(left == total - left - a[i]) res.add(i);
            left += a[i];
        }
        return res;
    }

    public static void main(String[] args){
        int[] a = {1,3,5,2,2};
        System.out.println(equilibriumIndices(a)); // [2]
    }
}
```

Problem 19 — Pair of elements with given sum

Approach: Use a `HashSet` to check complements; for multiple pairs you can store results in a set of ordered pairs.

Complexity: O(n) time, O(n) space.

```java
// Problem19_PairSum.java
import java.util.*;

class Problem19_PairSum {
    public static List<int[]> pairsWithSum(int[] a, int target){
        Set<Integer> seen = new HashSet<>();
        Set<String> added = new HashSet<>();
        List<int[]> res = new ArrayList<>();
        for(int v: a){
            int need = target - v;
            if(seen.contains(need)){
                int x = Math.min(v, need), y = Math.max(v, need);
                String key = x+":"+y;
                if(!added.contains(key)){
                    res.add(new int[]{x,y});
                    added.add(key);
                }
            }
            seen.add(v);
        }
        return res;
    }

    public static void main(String[] args){
        int[] a = {2,4,3,3,1};
        List<int[]> pairs = pairsWithSum(a, 6);
        for(int[] p: pairs) System.out.println(Arrays.toString(p)); // [2,4],
[3,3]
    }
}
```

Problem 20 — Merge two sorted arrays without extra space (Gap method)

Approach (step-by-step): 1. Consider two arrays `a` (size m) and `b` (size n). Treat them as one virtual array of length `m+n`. 2. Start with `gap = ceil((m+n)/2)`. Compare elements at `i` and `i+gap` and swap if out of order. Decrease gap until 0 using `gap = ceil(gap/2)`. 3. This sorts in-place across the two arrays.

Complexity: O((m+n) log(m+n)) in practice for the gap reductions, O(1) extra space.

```java
// Problem20_MergeWithoutExtraSpace.java
import java.util.*;
```

```java
class Problem20_MergeWithoutExtraSpace {
    private static int nextGap(int gap){
        if(gap <= 1) return 0;
        return (gap / 2) + (gap % 2);
    }

    public static void merge(int[] a, int[] b){
        int m = a.length, n = b.length;
        int gap = nextGap(m + n);
        while(gap > 0){
            int i = 0;
            // compare in a
            for(i = 0; i + gap < m; i++){
                if(a[i] > a[i + gap]){ int t = a[i]; a[i] = a[i+gap]; a[i+gap]
= t; }
            }
            // compare across a and b
            int j = gap > m ? gap - m : 0; // starting index in b
            for(; i < m && j < n; i++, j++){
                if(a[i] > b[j]){ int t = a[i]; a[i] = b[j]; b[j] = t; }
            }
            // compare in b
            if(j < n){
                for(j = 0; j + gap < n; j++){
                    if(b[j] > b[j + gap]){ int t = b[j]; b[j] = b[j+gap];
b[j+gap] = t; }
                }
            }
            gap = nextGap(gap);
        }
    }

    public static void main(String[] args){
        int[] a = {1,4,7,8,10};
        int[] b = {2,3,9};
        merge(a,b);
        System.out.println("a: " + Arrays.toString(a));
        System.out.println("b: " + Arrays.toString(b));
    }
}
```

Final notes

- Each snippet is self-contained and easy to copy into a single `.java` file for quick testing.
- If you want, I can:
- Combine all snippets into a single project with tests.

- Provide JUnit tests for selected problems.
- Walk through any one problem step-by-step live and explain corner cases.

Which would you like next? Want me to run through Problem 11 (Kadane) with several example arrays, or generate test cases for all 20?"