Two Sum & Beyond

A SOLVED PROBLEM SET · JAVA EDITION

★ IMPORTANT = high-frequency interview pattern, worth mastering first

21 Problems • Java • Compiled July 24, 2026

Solutions

1. 3Sum ★ IMPORTANT MEDIUM

APPROACH

Sort the array. Fix one element, then two-pointer the remainder to find pairs summing to its negation. Skip duplicate values at every level.

SOLUTION

```
public static List<List<Integer>> threeSum(int[] nums) {
    Arrays.sort(nums);
    List<List<Integer>> res = new ArrayList<>();
    int n = nums.length;
    for (int i = 0; i < n; i++) {
        if (i > 0 && nums[i] == nums[i - 1]) continue;
        if (nums[i] > 0) break;
        int l = i + 1, r = n - 1;
        while (l < r) {
            int sum = nums[i] + nums[l] + nums[r];
            if (sum < 0) {
                l++;
            } else if (sum > 0) {
                r--;
            } else {
                res.add(Arrays.asList(nums[i], nums[l], nums[r]));
                l++;
                r--;
                while (l < r && nums[l] == nums[l - 1]) l++;
                while (l < r && nums[r] == nums[r + 1]) r--;
            }
        }
    }
    return res;
}
```

Time: **O(n²)** Space: **O(1)** extra

2. 4Sum ★ IMPORTANT MEDIUM

APPROACH

Fix two elements with nested loops, two-pointer scan the rest. Use `long` for the running sum to avoid int overflow. Duplicate skipping applied at all four positions.

SOLUTION

```
public static List<List<Integer>> fourSum(int[] nums, int target) {
    Arrays.sort(nums);
    int n = nums.length;
    List<List<Integer>> res = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        if (i > 0 && nums[i] == nums[i - 1]) continue;
        for (int j = i + 1; j < n; j++) {
            if (j > i + 1 && nums[j] == nums[j - 1]) continue;
            int l = j + 1, r = n - 1;
            while (l < r) {
                long sum = (long) nums[i] + nums[j] + nums[l] + nums[r];
                if (sum < target) {
                    l++;
                } else if (sum > target) {
                    r--;
                } else {
                    res.add(Arrays.asList(nums[i], nums[j], nums[l], nums[r]));
                    l++;
                    r--;
                    while (l < r && nums[l] == nums[l - 1]) l++;
                    while (l < r && nums[r] == nums[r + 1]) r--;
                }
            }
        }
    }
    return res;
}
```

Time: **O(n³)** Space: **O(1)** extra

3. Two Sum II – Input Array Is Sorted ★ MEDIUM

IMPORTANT

APPROACH

Array is pre-sorted, so a single two-pointer pass finds the pair in linear time.

SOLUTION

```
public static int[] twoSum(int[] numbers, int target) {
    int l = 0, r = numbers.length - 1;
    while (l < r) {
        int sum = numbers[l] + numbers[r];
        if (sum == target) {
            return new int[]{l + 1, r + 1};
        } else if (sum < target) {
            l++;
        } else {
            r--;
        }
    }
    return new int[]{};
}
```

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

4. Two Sum III – Data Structure Design

EASY

APPROACH

Running frequency map. On `find`, check every stored number's complement, handling the case where a value pairs with a duplicate of itself.

SOLUTION

```
class TwoSum {
    private Map<Integer, Integer> counts = new HashMap<>();

    public void add(int number) {
        counts.put(number, counts.getOrDefault(number, 0) + 1);
    }

    public boolean find(int value) {
        for (int num : counts.keySet()) {
            int complement = value - num;
            if (counts.containsKey(complement)) {
                if (complement != num || counts.get(num) > 1) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

add: **O(1)** find: **O(n)**

---

5. Subarray Sum Equals K

★ IMPORTANT

MEDIUM

APPROACH

Prefix sums in a hashmap. A subarray sums to k exactly when `prefixSum - k` has been seen before.

SOLUTION

```
public static int subarraySum(int[] nums, int k) {
    Map<Integer, Integer> seen = new HashMap<>();
    seen.put(0, 1);
    int count = 0, prefixSum = 0;
    for (int num : nums) {
        prefixSum += num;
        count += seen.getOrDefault(prefixSum - k, 0);
        seen.put(prefixSum, seen.getOrDefault(prefixSum, 0) + 1);
    }
    return count;
}
```

Time: **O(n)** Space: **O(n)**

6. Two Sum IV – Input Is a BST

EASY

APPROACH

DFS with a visited-values set; at each node check whether its complement has already been seen.

SOLUTION

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

public static boolean findTarget(TreeNode root, int k) {
    Set<Integer> seen = new HashSet<>();
    return dfs(root, k, seen);
}

private static boolean dfs(TreeNode node, int k, Set<Integer> seen) {
    if (node == null) return false;
    if (seen.contains(k - node.val)) return true;
    seen.add(node.val);
    return dfs(node.left, k, seen) || dfs(node.right, k, seen);
}
```

Time: **O(n)** Space: **O(n)**

---

7. Two Sum Less Than K

EASY

APPROACH

Sort, two-pointer inward. Sum below k is a candidate — record and grow left; otherwise shrink right.

SOLUTION

```
public static int twoSumLessThanK(int[] nums, int k) {
    Arrays.sort(nums);
    int l = 0, r = nums.length - 1, best = -1;
    while (l < r) {
        int sum = nums[l] + nums[r];
        if (sum < k) {
            best = Math.max(best, sum);
            l++;
        } else {
            r--;
        }
    }
    return best;
}
```

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

---

8. Max Number of K-Sum Pairs ★ IMPORTANT MEDIUM

APPROACH

Sort and two-pointer: matching pair removes both ends; too small advances left; too large retreats right.

SOLUTION

```
public static int maxOperations(int[] nums, int k) {
    Arrays.sort(nums);
    int l = 0, r = nums.length - 1, ops = 0;
    while (l < r) {
        int sum = nums[l] + nums[r];
        if (sum == k) {
            ops++;
            l++;
            r--;
        } else if (sum < k) {
            l++;
        } else {
            r--;
        }
    }
    return ops;
}
```

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

9. Count Good Meals

MEDIUM

APPROACH

A pair is "good" if its sum is a power of two. For each item, check against every power of two up to 2·max using a running frequency map.

SOLUTION

```
public static int countGoodMeals(int[] deliciousness) {
    final int MOD = 1_000_000_007;
    Map<Integer, Integer> count = new HashMap<>();
    long res = 0;
    int maxVal = 0;
    for (int d : deliciousness) maxVal = Math.max(maxVal, d);
    int maxSum = 2 * maxVal;
    List<Integer> powers = new ArrayList<>();
    for (int p = 1; p <= maxSum; p *= 2) powers.add(p);
    for (int d : deliciousness) {
        for (int p : powers) {
            int complement = p - d;
            if (count.containsKey(complement)) {
                res = (res + count.get(complement)) % MOD;
            }
        }
        count.put(d, count.getOrDefault(d, 0) + 1);
    }
    return (int) res;
}
```

Time: **O(n log(maxSum))** Space: **O(n)**

---

10. Count Number of Pairs With Absolute Difference K

EASY

APPROACH

Frequency map, one pass. Both `num - k` and `num + k` counts (seen so far) add to the answer.

SOLUTION

```
public static int countKDifference(int[] nums, int k) {
    Map<Integer, Integer> count = new HashMap<>();
    int res = 0;
    for (int num : nums) {
        res += count.getOrDefault(num - k, 0) + count.getOrDefault(num + k, 0);
        count.put(num, count.getOrDefault(num, 0) + 1);
    }
    return res;
}
```

Time: **O(n)** Space: **O(n)**

---

11. Number of Pairs of Strings With Concatenation Equal to Target

MEDIUM

APPROACH

Constraints are small (n ≤ 100); a direct double loop checking every ordered pair's concatenation is simplest and fast enough.

SOLUTION

```
public static int numOfPairs(String[] nums, String target) {
    int res = 0;
    int n = nums.length;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i != j && (nums[i] + nums[j]).equals(target)) {
                res++;
            }
        }
    }
    return res;
}
```

Time: **O(n² · L)** Space: **O(1)**

---

12. Find All K-Distant Indices in an Array

EASY

APPROACH

Collect indices holding key. An index qualifies if it lies within k of any key-index.

SOLUTION

```
public static List<Integer> findKDistantIndices(int[] nums, int key, int k) {
    int n = nums.length;
    List<Integer> keyIndices = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        if (nums[i] == key) keyIndices.add(i);
    }
    List<Integer> res = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        for (int j : keyIndices) {
            if (Math.abs(i - j) <= k) {
                res.add(i);
                break;
            }
        }
    }
    return res;
}
```

Time: **O(n · m)** Space: **O(m)**, m = key occurrences

---

13. First Letter to Appear Twice

EASY

APPROACH

Single pass with a seen-set; return the first character already present.

SOLUTION

```
public static char repeatedCharacter(String s) {
    Set<Character> seen = new HashSet<>();
    for (char ch : s.toCharArray()) {
        if (seen.contains(ch)) return ch;
        seen.add(ch);
    }
    return ' ';
}
```

Time: **O(n)** Space: **O(1)** (bounded alphabet)

---

14. Number of Excellent Pairs

★ IMPORTANT HARD

APPROACH

Only distinct values matter. Group by bit popcount (`Integer.bitCount`), then for every pair of popcount buckets whose sum ≥ k, multiply bucket sizes — popcount is bounded (≤ 30), so the nested loop is effectively constant time.

SOLUTION

```
public static long countExcellentPairs(int[] nums, int k) {
    Set<Integer> distinct = new HashSet<>();
    for (int num : nums) distinct.add(num);
    int[] freqs = new int[61];
    for (int num : distinct) {
        freqs[Integer.bitCount(num)]++;
    }
    long res = 0;
    for (int c1 = 0; c1 <= 60; c1++) {
        if (freqs[c1] == 0) continue;
        for (int c2 = 0; c2 <= 60; c2++) {
            if (freqs[c2] == 0) continue;
            if (c1 + c2 >= k) {
                res += (long) freqs[c1] * freqs[c2];
            }
        }
    }
    return res;
}
```

Time: **O(D + 30²)**, D = distinct count Space: **O(D)**

15. Number of Arithmetic Triplets

EASY

APPROACH

Put all values in a set. For each x, check whether both `x + diff` and `x + 2·diff` exist.

SOLUTION

```
public static int arithmeticTriplets(int[] nums, int diff) {
    Set<Integer> s = new HashSet<>();
    for (int num : nums) s.add(num);
    int count = 0;
    for (int x : nums) {
        if (s.contains(x + diff) && s.contains(x + 2 * diff)) {
            count++;
        }
    }
    return count;
}
```

Time: **O(n)** Space: **O(n)**

---

16. Node With Highest Edge Score

MEDIUM

APPROACH

Each node i has an edge to edges[i]. Accumulate incoming scores (use `long` to be safe), then scan for the max.

SOLUTION

```
public static int edgeScore(int[] edges) {
    int n = edges.length;
    long[] score = new long[n];
    for (int i = 0; i < n; i++) {
        score[edges[i]] += i;
    }
    int best = 0;
    for (int i = 1; i < n; i++) {
        if (score[i] > score[best]) best = i;
    }
    return best;
}
```

Time: **O(n)** Space: **O(n)**

---

17. Check Distances Between Same Letters

EASY

APPROACH

Record the first index of each letter. On its second occurrence, verify the gap matches the required distance.

SOLUTION

```
public static boolean checkDistances(String s, int[] distance) {
    int[] first = new int[26];
    Arrays.fill(first, -1);
    for (int i = 0; i < s.length(); i++) {
        int idx = s.charAt(i) - 'a';
        if (first[idx] != -1) {
            if (i - first[idx] - 1 != distance[idx]) return false;
        } else {
            first[idx] = i;
        }
    }
    return true;
}
```

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

---

18. Find Subarrays With Equal Sum

EASY

APPROACH

Track sums of adjacent pairs in a set; a repeat means two equal-sum subarrays exist.

SOLUTION

```
public static boolean findSubarrays(int[] nums) {
    Set<Integer> seen = new HashSet<>();
    for (int i = 0; i < nums.length - 1; i++) {
        int sum = nums[i] + nums[i + 1];
        if (seen.contains(sum)) return true;
        seen.add(sum);
    }
    return false;
}
```

Time: **O(n)** Space: **O(n)**

---

19. Largest Positive Integer That Exists With Its Negative

EASY

APPROACH

Put all values in a set. For every positive number, check whether its negation is present, tracking the largest match.

SOLUTION

```
public static int findMaxK(int[] nums) {
    Set<Integer> s = new HashSet<>();
    for (int num : nums) s.add(num);
    int best = -1;
    for (int num : nums) {
        if (num > 0 && s.contains(-num)) {
            best = Math.max(best, num);
        }
    }
    return best;
}
```

Time: **O(n)** Space: **O(n)**

---

20. Number of Distinct Averages

EASY

APPROACH

Sort, then repeatedly pair the smallest remaining value with the largest, tracking distinct averages in a set.

SOLUTION

```
public static int distinctAverages(int[] nums) {
    Arrays.sort(nums);
    Set<Double> averages = new HashSet<>();
    int l = 0, r = nums.length - 1;
    while (l < r) {
        averages.add((nums[l] + nums[r]) / 2.0);
        l++;
        r--;
    }
    return averages.size();
}
```

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

---

21. Count Pairs Whose Sum Is Less Than Target ★ IMPORTANT

EASY

APPROACH

Sort, two-pointer inward. If `nums[l] + nums[r] < target`, every index between l and r also pairs validly with l — add `r - l` pairs at once and advance l. Otherwise shrink from the right.

SOLUTION

```
public static int countPairsLessThanTarget(int[] nums, int target) {
    Arrays.sort(nums);
    int l = 0, r = nums.length - 1, count = 0;
    while (l < r) {
        if (nums[l] + nums[r] < target) {
            count += r - l;
            l++;
        } else {
            r--;
        }
    }
    return count;
}
```

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