50 Subjective DSA Interview Questions on Arrays

Topics Covered: Array Basics, Prefix Sum, Sliding Window, Two Pointer Technique, 2-sum/3-sum, Kadane's Algorithm, Max Subarray Sum, Merge Sort (Inversion Count)

Q1. (Google) What is the difference between static arrays and dynamic arrays in terms of memory allocation and time complexity?

Answer: Static arrays are fixed in size and allocated at compile-time, while dynamic arrays can grow and shrink at runtime. Static arrays provide O(1) access and allocation time, whereas dynamic arrays may require O(n) time for resizing.

Q2. (Amazon) How do you compute a prefix sum array and use it to answer range sum queries efficiently?

Answer: A prefix sum array is built such that prefix[i] = sum(arr[0] to arr[i]). To get the sum of a range [l, r], use prefix[r] - prefix[l-1]. This allows O(1) query time after O(n) preprocessing.

Q3. (Microsoft) Explain how the sliding window technique improves time complexity in subarray sum problems.

Answer: Instead of recalculating the sum for each subarray, we slide the window by subtracting the outgoing element and adding the incoming element. This reduces the time complexity from O(n*k) to O(n).

Q4. (Facebook) How do you solve the 2-sum problem in a sorted array using the two-pointer technique?

Answer: Start two pointers at the beginning and end. If sum < target, increment left. If sum > target, decrement right. Stop when sum == target. Time complexity is O(n).

Q5. (Adobe) Describe Kadane's Algorithm for finding the maximum sum subarray.

Answer: Kadane's Algorithm maintains a current sum and resets it when it drops below 0. The maximum seen so far is updated during iteration. This runs in O(n) time.

PREFIX SUM (Q6–Q10)

Q6. (Amazon)
What are the time and space complexities of prefix sum preprocessing and query operations?

Answer: Preprocessing takes O(n) time to build the prefix sum array. Querying the sum of any subarray can be done in O(1) time using prefix[r] - prefix[l - 1]. Space complexity is O(n) for storing prefix sums.

Q7. (Flipkart)
Write code to compute prefix sums for a given array.
Answer:

```
int[] prefix = new int[arr.length];
prefix[0] = arr[0];
for (int i = 1; i < arr.length; i++) {
    prefix[i] = prefix[i - 1] + arr[i];
}
```

Q8. (Samsung)
How can prefix sums be used to find the number of subarrays with sum equal to a given value?
Answer: Use a HashMap to store prefix sum frequencies. For each prefix sum s, check if s - target exists. Add its count to the result. This gives O(n) time.

Q9. (TCS Digital)
What is the time and space complexity of prefix sum preprocessing and query operations?
Answer: Same as Q6 — O(n) time and space for preprocessing, O(1) time for queries.

Q10. (Google)
How would you handle prefix sum logic if the array contains negative numbers?
Answer: Prefix sum works fine with negative numbers. However, when counting subarrays with a given sum, use a HashMap since fixed window length doesn't apply.

SLIDING WINDOW (Q11–Q15)

Q11. (Microsoft)
Explain the sliding window technique for finding the maximum sum of k consecutive elements.
Answer: Calculate the sum of the first k elements, then slide the window forward by subtracting the element going out and adding the new element. Time: O(n).

Q12. (Adobe)
How do you use sliding window to find the smallest subarray with sum > target?
Answer: Use two pointers (start and end). Expand end until the sum ≥ target, then shrink start to minimize length while maintaining the sum constraint.

Q13. (Walmart Labs)
Explain how to use sliding window for problems involving strings or arrays.

Answer: Use fixed or dynamic-size windows to check conditions (e.g., length, frequency) in substrings/subarrays. Maintain auxiliary data like character count maps.

Q14. (Paytm)
What are the advantages of using sliding window over nested loops?
Answer: Reduces time complexity from $O(n^2)$ to O(n) or O(n log n). Useful in continuous subarray or substring problems with constraints.

Q15. (Capgemini)
Describe how to adapt sliding window for variable-sized subarray problems.
Answer: Expand right pointer to increase the window. Once a condition is satisfied, shrink from the left while keeping it valid.

TWO POINTER TECHNIQUE (Q16–Q20)

Q16. (Facebook)
How do you solve the 2-sum problem using the two-pointer technique?
Answer: Sort the array. Use two pointers (left and right). If sum is less than target, move left. If more, move right. Time: O(n log n) for sorting + O(n) traversal.

Q17. (Google)
Explain the two-pointer method to remove duplicates from a sorted array.
Answer: Use one pointer for unique position, one to scan. If new element ≠ last unique, update position. O(n) time, O(1) space.

Q18. (Flipkart)
Can the two-pointer technique work on unsorted arrays? Why or why not?
Answer: Not effectively — two-pointer relies on sorted order to decide movement direction. Use hashing for unsorted versions.

Q19. (Amazon)
How do you find the longest subarray with at most two distinct integers?
Answer: Use sliding window + HashMap to track counts of elements. Adjust the left pointer when unique count > 2.

Q20. (Meta)
Compare the two-pointer technique with the sliding window method.
Answer: Two-pointer is specific to sorted data or pair/triplet problems. Sliding window is more general and applies to subarray constraints.

2-SUM / 3-SUM PROBLEMS (Q21–Q25)

Q21. (Google)
Describe a complete approach to solve the 3-sum problem using sorting and two pointers.

Answer: Sort the array. Fix the first element, then apply 2-pointer for the remaining two. Skip duplicates. Time: $O(n^2)$.

Q22. (Adobe)
How do you find all unique triplets in an array that sum to zero?
Answer: As in Q21, skip duplicates at each level. Store results in a Set or skip using conditions.

Q23. (Amazon)
Compare 2-sum using hashing vs two-pointers.
Answer: Hashing: O(n) time and space. Two-pointer: O(n log n) for sorting + O(n) time, O(1) space. Hashing works on unsorted arrays.

Q24. (Qualcomm)
What are the edge cases to consider when solving 3-sum?
Answer: Duplicates, large numbers, empty array, all positives or negatives, and input with < 3 elements.

Q25. (Oracle)
Write code for 2-sum using two-pointer approach on a sorted array.
Answer:

```
int l = 0, r = arr.length - 1;
while (l < r) {
   int sum = arr[l] + arr[r];
   if (sum == target) return true;
   if (sum < target) l++;
   else r--;
}
```

KADANE'S ALGORITHM (Q26–Q30)

Q26. (Microsoft)
Explain Kadane's Algorithm and its use in finding the maximum subarray sum.
Answer: Initialize maxSum and currSum to the first element. Traverse array, set currSum = max(arr[i], currSum + arr[i]). Then maxSum = max(maxSum, currSum). This finds the maximum sum of a contiguous subarray in O(n) time.

Q27. (Apple)
How does Kadane's Algorithm handle arrays with all negative values?
Answer: Kadane's fails with default 0 initialization. To fix, initialize maxSum and currSum with arr[0] and start iteration from index 1.

Q28. (Infosys)
Modify Kadane's Algorithm to also return the start and end indices of the subarray.

Answer: Track temporary start index when currSum becomes arr[i]. When maxSum is updated, update start and end pointers accordingly.

Q29. (Google)
Why does Kadane's Algorithm work in linear time complexity?
Answer: It processes each element exactly once and maintains only current sum and max sum. No nested loops or recursion are needed.

Q30. (TCS Digital)
What is the difference between Kadane's Algorithm and brute-force approach for max subarray?
Answer: Brute-force checks all $O(n^2)$ subarrays. Kadane's works in O(n) by maintaining local and global max while scanning linearly.

MAXIMUM SUBARRAY SUM (Q31–Q35)

Q31. (Amazon)
Compare the prefix sum approach vs Kadane's Algorithm for finding maximum subarray sum.
Answer: Prefix sum requires nested loops to evaluate sum between every pair, giving $O(n^2)$. Kadane's solves it in O(n) with constant space.

Q32. (Flipkart)
How do you find maximum sum of subarrays of size k?
Answer: Use a fixed-size sliding window. Compute sum of first k elements, then slide the window, adding next and removing previous elements in O(1) per step.

Q33. (Samsung)
Write a function to return both the maximum sum and the subarray.
Answer: Extend Kadane's Algorithm by storing start, end, and tempStart indexes and update them whenever the maxSum changes.

Q34. (Microsoft)
What are the limitations of Kadane's Algorithm for circular arrays?
Answer: It fails when maximum subarray wraps around. Solve by:
maxCircular = totalSum - minSubarraySum.
Final answer: max(maxKadane, maxCircular).

Q35. (Google)
Solve the maximum circular subarray sum using modified Kadane's Algorithm.
Answer: Use Kadane for max sum. Then invert array elements and run Kadane again to get min subarray sum, subtract it from total array sum.

MERGE SORT / INVERSION COUNT (Q36–Q40)

Q36. (Adobe)
Explain how inversion count in an array is related to Merge Sort.
Answer: While merging, if left[i] > right[j], then all remaining elements in left (from i onwards) form inversions with right[j].

Q37. (Oracle)
Write a function to count the number of inversions in an array.
Answer: Modify merge step to count inversions.

```
if (left[i] > right[j]) inversions += (mid - i + 1);
```

Q38. (Google)
What is the time complexity of inversion count using Merge Sort?
Answer: O(n log n), due to merge sort recursion and linear merge steps.

Q39. (Wipro)
Why is the brute-force approach to count inversions inefficient?
Answer: It checks every pair (i, j), giving $O(n^2)$ time complexity.

Q40. (Capgemini)
How do you modify the merge step of merge sort to count inversions?
Answer: During merge, if left[i] > right[j], increment inversion count by number of remaining elements in left[].

MIXED / COMBINED APPLICATIONS (Q41–Q45)

Q41. (Facebook)
Combine two-pointer and sliding window techniques to solve a subarray sum problem.
Answer: Use left and right pointers to form a dynamic window. Expand right to increase sum, shrink left if sum > target, check sum == target to count matches.

Q42. (Flipkart)
How can prefix sums and hash maps be used together to find subarrays with target sum?
Answer: For each prefix sum, store it in a map. For current prefix sum s, check if (s - target) exists in map. If yes, subarray with required sum found.

Q43. (Amazon)
Solve a problem requiring 2-sum inside a sliding window of fixed size.
Answer: For each window, apply HashSet-based 2-sum check in O(k) time. Total complexity: O(nk).

Q44. (Google)
When should you prefer prefix sums over Kadane's Algorithm?
Answer: Use prefix sums for range queries and subarray count problems. Use Kadane's when you only need the maximum sum subarray.

Q45. (Meta)
Write a hybrid approach to find the longest subarray with sum less than a target using sliding window + prefix.
Answer: Maintain prefix sum array, and binary search it to get farthest index satisfying prefix[j] - prefix[i] ≤ target.

EDGE CASES / THEORY (Q46–Q50)

Q46. (Microsoft)
What are the edge cases to test while implementing merge sort or prefix sum logic?
Answer: Empty array, single element, all elements equal, already sorted, all negative or positive values, large values for overflow.

Q47. (Adobe)
Discuss integer overflow issues in subarray sum problems and how to prevent them.
Answer: Use long instead of int. In Java, use Math.addExact() or manual overflow checks.

Q48. (Apple)
Explain the impact of array size and initialization in Java vs C++.
Answer: Java initializes array elements to default (e.g., 0), while C++ does not. In C++, uninitialized arrays can lead to garbage values.

Q49. (Google)
When should you prefer recursive vs iterative merge sort?
Answer: Recursive is easier and standard. Iterative saves stack space and can perform better for large data with tail call optimization.

Q50. (Amazon)
How would you optimize memory usage when solving large prefix sum problems?
Answer: Use in-place prefix sums if original array can be modified. Avoid full prefix array if only sum queries are needed occasionally.