Most Asked DSA Interview Questions (0-3 Years) 5-20 LPA

DSA Questions

1. Rotate array right by k (in-place)

Problem: Given an array nums and integer k, rotate the array to the right by k steps in-place.

Example:

```
def rotate(nums, k):
    n = len(nums)
    k %= n
    def rev(i, j):
        while i < j:
            nums[i], nums[j] = nums[j], nums[i]
            i += 1; j -= 1
    rev(0, n-1)
    rev(0, k-1)
    rev(k, n-1)

# Example
arr = [1,2,3,4,5,6,7]
rotate(arr, 3)
print(arr)  # [5,6,7,1,2,3,4]
```

Explanation:

- Reverse whole array, reverse first k, reverse rest..
- Complexity: O(n) time, O(1) extra space
- **Tip:** Handles large k with k %= n.

2. Merge overlapping intervals.

Given intervals [[s,e], ...], merge overlapping intervals.

Example: [[1,3],[2,6],[8,10],[15,18]] → [[1,6],[8,10],[15,18]]

```
def merge_intervals(intervals):
    if not intervals:
        return []
    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]
    for s,e in intervals[1:]:
        last_s, last_e = merged[-1]
        if s <= last_e:
```

```
            merged[-1][1] = max(last_e, e)
        else:
            merged.append([s,e])
    return merged

print(merge_intervals([[1,3],[2,6],[8,10],[15,18]]))
# [[1,6],[8,10],[15,18]];
```

- **Explaination**: Sort by start; iterate merging when overlap.
- **Complexity**: O(n log n) due to sort.

3. Longest Common Prefix

Problem: Find the longest common prefix string amongst an array of strings.)

```
def longest_common_prefix(strs):
    if not strs:
        return ""
    prefix = list(strs[0])
    for s in strs[1:]:
        i = 0
        while i < len(prefix) and i < len(s) and prefix[i] == s[i]:
            i += 1
        prefix = prefix[:i]
        if not prefix:
            return ""
    return "".join(prefix)

print(longest_common_prefix(["flower","flow","flight"]))
```

Example: ["flower","flow","flight"] → "fl"
Approach: Vertical scanning or binary search; here vertical scan with two pointers char-by-char.
Complexity: O(S) where S is total chars.

4. Longest Increasing Subsequence (LIS).

Problem: Given nums, return length of LIS.

```
import bisect

def length_of_lis(nums):
    tails = []  # tails[i] = smallest
tail of all increasing subseqs of
length i+1
    for x in nums:
        pos =
bisect.bisect_left(tails, x)
        if pos == len(tails):
            tails.append(x)
        else:
            tails[pos] = x
    return len(tails)

print(length_of_lis([10,9,2,5,3,7,
101,18]))  # 4
```

Explanation:

- **Example:** [10,9,2,5,3,7,101,18] → 4 (2,3,7,101)
  Approach: Patience sorting (tails array + binary search).
- **Complexity:** O(n log n).

Tip: Good to implement in interviews and explain intuition.

5. Product of Array Except Self

Problem: Return array where each element is product of all other elements (no division).
Example: [1,2,3,4] → [24,12,8,6]
Approach: Prefix and suffix products in two passes.

```
def product_except_self(nums):
    n = len(nums)
    res = [1]*n
    prefix = 1
    for i in range(n):
        res[i] = prefix
        prefix *= nums[i]
    suffix = 1
    for i in range(n-1, -1, -1):
        res[i] *= suffix
        suffix *= nums[i]
    return res

print(product_except_self([1,2,3,4]))  # [24,12,8,6]
```

Complexity: O(n), O(1) extra (output not counted)

HashMap + Sliding Window

6. Two Sum

Problem: Given nums and target, return indices of the two numbers adding to target.
Example: nums=[2,7,11,15], target=9 → [0,1]
Approach: Hashmap of value→index while iterating.A window function performs calculations across a set of table rows related to the current row — without collapsing rows like GROUP BY.

```
def two_sum(nums, target):
    seen = {}
    for i, x in enumerate(nums):
        want = target - x
        if want in seen:
            return [seen[want], i]
        seen[x] = i
    return []

print(two_sum([2,7,11,15], 9))  # [0,1]
```

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

7. Subarray Sum Equals K (count)

Problem: Count subarrays with sum == k.
Example: nums=[1,1,1], k=2 → 2
Approach: Prefix sums + hashmap counting occurrences.

```
from collections import
defaultdict

def subarray_sum(nums, k):
    cnt = defaultdict(int)
    cnt[0] = 1
    cur = 0
    res = 0
    for x in nums:
        cur += x
        res += cnt[cur - k]
        cnt[cur] += 1
    return res

print(subarray_sum([1,1,1], 2))
# 2
```

8. Explain the difference between UNION and UNION ALL.

Feature	UNION	UNION ALL
Duplicates	Removes duplicates	Keeps all rows, including duplicates
Performance	Slower (because of sorting)	Faster (no de-duplication)
Use case	When you want distinct rows	When duplicates are meaningful

Example:

```
SELECT city FROM customers
UNION
SELECT city FROM vendors;
```

→ Returns a unique list of cities.

```
SELECT city FROM customers
UNION ALL
SELECT city FROM vendors;
```

→ Returns **all cities**, including duplicates.

9. Longest Substring Without Repeating Characters

Problem: Length of the longest substring without repeating characters.
Example: "abcabcbb" → 3 ("abc")
Approach: Sliding window with hashmap of last index.

```
def length_of_longest_substring(s):
    last = {}
    start = 0
    best = 0
    for i, ch in enumerate(s):
        if ch in last and last[ch] >= start:
            start = last[ch] + 1
        last[ch] = i
        best = max(best, i - start + 1)
    return best

print(length_of_longest_substring("abcabcbb"))  # 3 CASE
        WHEN salary >= 100000 THEN 'High'
        WHEN salary >= 50000 THEN 'Medium'
        ELSE 'Low'
      END AS salary_category
FROM employees;
```

Explanation:

- **Complexity:** O(n).
  Tip: Use start to shrink window when duplicates found.

10. Minimum Window Substring

Problem: Length of the longest substring without repeating characters.
Example: "abcabcbb" → 3 ("abc")
Approach: Sliding window with hashmap of last index.

```
Def length_of_longest_substring(s):
    last = {}
    start = 0
    best = 0
    for i, ch in enumerate(s):
        if ch in last and last[ch] >= start:
            start = last[ch] + 1
        last[ch] = i
        best = max(best, i - start + 1)
    return best

print(length_of_longest_substring("abcabcbb"))  # 3
```

Explanation:

- **Complexity:** O(n).
  Tip: Use start to shrink window when duplicates found.

11. Minimum Window Substring

Problem: Given strings s and t, find minimum window in s containing all chars of t. Return "" if none**.**
Example: s="ADOBECODEBANC", t="ABC" → "BANC"
Approach: Sliding window + counts of required chars; expand then contract.A CTE (Common Table Expression) is a temporary, named result set that you can reference within a SQL query. It improves readability and simplifies complex subqueries or recursive logic.

Syntax:

```
from collections import Counter

def min_window(s, t):
    need = Counter(t)
    missing = len(t)
    left = 0
    best = (0, float('inf'))
    for right, ch in enumerate(s):
        if need[ch] > 0:
            missing -= 1
        need[ch] -= 1
        while missing == 0:
            if right - left < best[1] - best[0]:
                best = (left, right)
            # try to move left
            need[s[left]] += 1
            if need[s[left]] > 0:
                missing += 1
            left += 1
    l, r = best
    return "" if r == float('inf') else s[l:r+1]

print(min_window("ADOBECODEBANC", "ABC"))  # "BANC"
```

Benefits:

- **Complexity:** O(|s| + |t|) typical, O(1) alphabet assumption.
  Tip: Explaining need and missing succinctly is crucial in interviews.

12. Longest Subarray with At Most K Distinct

Problem: Given array (or string) find longest subarray with at most k distinct elements.
Example: [1,2,1,2,3], k=2 → length 4 (1,2,1,2)
Approach: Sliding window + hashmap counting distinct in window.

```
from collections import defaultdict

def longest_at_most_k_distinct(nums, k):
    cnt = defaultdict(int)
    left = 0
    distinct = 0
    best = 0
    for right, x in enumerate(nums):
        if cnt[x] == 0:
            distinct += 1
        cnt[x] += 1
        while distinct > k:
            cnt[nums[left]] -= 1
            if cnt[nums[left]] == 0:
                distinct -= 1
            left += 1
        best = max(best, right - left + 1)
    return best

print(longest_at_most_k_distinct([1,2,1,2,3], 2))  # 4
```

Explanation:

- **Complexity:** O(n).

Graph Traversal

13. Shortest Path in Unweighted Graph (BFS returning path)

Problem: Given adjacency list, source s and target t, return shortest path nodes (or [] if none).
Example: Graph where 1→2,1→3,2→4; shortest 1→2→4 etc.
Approach: BFS with parent pointers; reconstruct path.

```
from collections import deque

def bfs_shortest_path(adj, s, t):
    q = deque([s])
    parent = {s: None}
    while q:
        u = q.popleft()
        if u == t:
            break
        for v in adj.get(u, []):
            if v not in parent:
                parent[v] = u
                q.append(v)
    if t not in parent:
        return []
    path = []
    cur = t
    while cur is not None:
        path.append(cur)
        cur = parent[cur]
    return path[::-1]

# Example adjacency
adj = {1:[2,3], 2:[4], 3:[], 4:[]}
print(bfs_shortest_path(adj, 1, 4))  # [1,2,4]
```

- **Complexity:** O(V+E).
  Tip: Use BFS for unweighted shortest path problems.

14. Detect Cycle in Directed Graph (DFS)

Problem: Detect if a directed graph has a cycle**.**
Approach: DFS with 3-color marking (0=unvisited,1=visiting,2=visited).

```
def has_cycle_directed(adj):
    nstates = {}
    def dfs(u):
        if nstates.get(u,0) == 1:
            return True
        if nstates.get(u,0) == 2:
            return False
        nstates[u] = 1
        for v in adj.get(u, []):
            if dfs(v):
                return True
        nstates[u] = 2
        return False
    for node in adj:
        if nstates.get(node,0) == 0:
            if dfs(node):
                return True
    return False

print(has_cycle_directed({1:[2], 2:[3], 3:[1]}))
```

Explanation:

- **Complexity:** O(V+E)..
- WHERE t.customer_id IS NULL ensures the customer had **no purchase in the last 6 months**.

15. Number of Connected Components (Union-Find)

Problem: Given n nodes and edge list, count connected components.
Approach: Union-Find (disjoint set union).Using IS NULL / IS NOT NULL:

```
SELECT * FROM employees WHERE manager_id IS NULL;
```

```
class DSU:
    def __init__(self, n):
        self.par = list(range(n))
        self.rank = [0]*n
    def find(self, a):
        while self.par[a] != a:
            self.par[a] = self.par[self.par[a]]
            a = self.par[a]
        return a
    def union(self, a, b):
```

```
        ra, rb = self.find(a), self.find(b)
        if ra == rb:
            return False
        if self.rank[ra] < self.rank[rb]:
            ra, rb = rb, ra
        self.par[rb] = ra
        if self.rank[ra] == self.rank[rb]:
            self.rank[ra] += 1
        return True

def count_components(n, edges):
    dsu = DSU(n)
    for a, b in edges:
        dsu.union(a, b)
    roots = set(dsu.find(i) for i in range(n))
    return len(roots)

print(count_components(5, [(0,1),(1,2),(3,4)]))  # 2
```

Complexity: ~O(α(n)) per op.
Tip: Explain path compression + union by rank.