Most Frequently asked DSA questions in MAANG 10-20 LPA

DSA Questions

1. Find the maximum element in an array.

To solve this, scan through the array while keeping track of the largest element.

Example:

```
def find_max(arr):
    max_val = arr[0]
    for num in arr:
        if num > max_val:
            max_val = num
    return max_val

# Example
arr = [3, 1, 7, 4, 9, 2]
print(find_max(arr))  # Output:
9
```

Explanation:

- Start with the first element as the current maximum.
- Compare each element with the current maximum.
- Update the maximum if a larger element is found.
- At the end, return the maximum element.
- **Complexity:** O(n) time, O(1) space.

2. Reverse an array in-place.

To solve this, use two pointers: one starting at the beginning, the other at the end, and swap until they meet in the middle.

```
def reverse_array(arr):
    left, right = 0, len(arr) - 1
    while left < right:
        arr[left], arr[right] = arr[right], arr[left]
        left += 1
        right -= 1
    return arr

# Example
arr = [1, 2, 3, 4, 5]
print(reverse_array(arr))  # Output: [5, 4, 3, 2, 1]
```

- Initialize two pointers: left at the start, right at the end.
- Swap elements at these positions.
- Move left forward and right backward until they cross.
- The array is reversed in-place, without using extra space.
- **Complexity:** O(n) time, O(1) space.

3. Check if a string is a palindrome.

To solve this, use two pointers: one starting at the left end, the other at the right end, and compare characters until they meet.

```
def is_palindrome(s):
    left, right = 0, len(s) - 1
    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True

# Example
print(is_palindrome("madam"))   # Output: True
print(is_palindrome("hello"))   # Output: False
```

Explanation:

- A palindrome reads the same forward and backward.
- Use two pointers: compare characters at the beginning and end.
- If all pairs match, it’s a palindrome; otherwise, it’s not.
- **Complexity:** O(n) time, O(1) space.

4. Find the maximum sum of any contiguous subarray of size k.

To solve this, use the sliding window technique: compute the sum of the first k elements, then slide the window one step at a time adding the new element and removing the outgoing element.

```
def max_sum_subarray_k(arr, k):
    if k > len(arr) or k == 0:
        return None  # or raise ValueError

    # initial window sum
    window_sum = sum(arr[:k])
    max_sum = window_sum

    # slide the window
    for i in range(k, len(arr)):
        window_sum += arr[i] - arr[i - k]
        if window_sum > max_sum:
            max_sum = window_sum

    return max_sum
```

```
# Example
arr = [2, 1, 5, 1, 3, 2]
k = 3
print(max_sum_subarray_k(arr, k))  # Output: 9
```

Explanation:

- Groups employees by department.
- Compute the sum of the first k elements as the initial window.
- For each next index i (from k to n-1), add arr[i] (new entering element) and subtract arr[i-k] (exiting element) to update the window sum in O(1).
- Track the maximum window sum seen so far.
- Return the maximum after scanning once.
- Filters groups where the count of employees is **more than 5**.
- **Complexity:** O(n) time, O(1) space.

5. Find the length of the longest substring without repeating characters.

To solve this, use a sliding-window with a hashmap (char → last index) to move the window start when a duplicate appears.

```
def longest_unique_substring(s):
    last_index = {}          # stores last index of each character
    start = 0                # left boundary of window
    max_len = 0

    for i, ch in enumerate(s):
        # if ch seen and its last position is within current window, move start
        if ch in last_index and last_index[ch] >= start:
            start = last_index[ch] + 1
        last_index[ch] = i
        max_len = max(max_len, i - start + 1)

    return max_len

# Example
print(longest_unique_substring("abcabcbb"))  # Output: 3  (substring "abc")
print(longest_unique_substring("bbbbb"))     # Output: 1  (substring "b")
print(longest_unique_substring("pwwkew"))    # Output: 3  (substring "wke")
```

Explanation:

- last_index remembers the most recent index of each character.
- start is the left pointer of the current window (beginning of the substring without repeats).
- When a character repeats and its last seen index is inside the window (>= start), move start to last_index[ch] + 1 to exclude the earlier occurrence.
- Update last_index[ch] = i and compute the current window length i - start + 1. Keep the maximum.
- **Complexity:** O(n) time, O(min(n, Σ)) space where Σ is character alphabet size.

6. Find the top k most frequent elements in an array.

To solve this, count frequencies with a hashmap then use a min-heap of size k (or use bucket sort for O(n) time) to retrieve the top k frequent elements.

```
from collections import Counter
import heapq

def top_k_frequent(nums, k):
    if k == 0:
        return []

    freq = Counter(nums)  # element -> count
    # use a min-heap of (count, element); keep size k
    heap = []
    for num, cnt in freq.items():
        if len(heap) < k:
            heapq.heappush(heap, (cnt, num))
        else:
            if cnt > heap[0][0]:
                heapq.heapreplace(heap, (cnt, num))

    # extract elements from heap
    return [num for cnt, num in heap]

# Example
nums = [1,1,1,2,2,3,3,3,3,4]
print(top_k_frequent(nums, 2))  # Output: [1, 3]  (order may vary)
```

Explanation

a) Counter(nums) builds a frequency map.
Maintain a min-heap of size k keyed by frequency so the heap root is the current k-th most frequent element.
For each (num, cnt), push until heap has k items; afterwards, only replace the root when you find a higher frequency.
At the end, the heap contains the top k frequent elements.

b) **Complexity:**
Heap approach: O(n log k) time, O(n) space (for frequency map + heap of size k).
Bucket sort approach: O(n) time, O(n) space.

7. Search for a target value in a rotated sorted array (no duplicates)..

To solve this, use a modified binary search: determine which half is sorted and decide which side to continue searching.

```
Def search_rotated(nums, target):
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid

        # If left half is sorted
        if nums[left] <= nums[mid]:
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        # Right half must be sorted
        else:
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1

    return -1

# Examples
print(search_rotated([4,5,6,7,0,1,2], 0))  # Output: 4
print(search_rotated([4,5,6,7,0,1,2], 3))  # Output: -1
```

Explanation:

• A rotated sorted array is a sorted array shifted at some pivot (e.g., [0,1,2,4,5,6,7] → [4,5,6,7,0,1,2]).

• Use binary search to get O(log n) time. At each step, check which side (left→mid or mid→right) is normally sorted by comparing endpoint values.

• If the target lies within the sorted half, move the search to that half; otherwise, search the other half.

• Continue until you find the target or the window becomes empty.

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

8. Merge all overlapping intervals in a list of intervals.

To solve this, sort intervals by start time, then iterate and merge an interval with the previous one if they overlap.

```
def merge_intervals(intervals):
    if not intervals:
        return []

    # sort by start time
    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]

    for current in intervals[1:]:
        last = merged[-1]
        # if current overlaps with last, merge them
        if current[0] <= last[1]:
            last[1] = max(last[1], current[1])
        else:
            merged.append(current)

    return merged

# Example
intervals = [[1,3], [2,6], [8,10], [15,18]]
print(merge_intervals(intervals))  # Output: [[1,6], [8,10], [15,18]]
```

Explanation:

• Sort intervals by their start times so potential overlaps are adjacent.
• Keep a merged list with the last interval representing the current merged block.
• For each current interval, check if current[0] <= last[1] (overlap). If yes, extend last[1] = max(last[1], current[1]); otherwise append current as a new block.
• At the end, merged contains non-overlapping intervals covering all original intervals.

Complexity: O(n log n) time (sorting) and O(n) space.

9. Find the k-th smallest element in an unsorted array.

To solve this efficiently, use Quickselect (a selection algorithm related to quicksort): pick a pivot, partition the array around it, then recurse only on the side that contains the k-th smallest element..

```
def partition(arr, left, right, pivot_index):
    pivot_value = arr[pivot_index]
    # move pivot to end
    arr[pivot_index], arr[right] = arr[right], arr[pivot_index]
    store = left
    for i in range(left, right):
        if arr[i] < pivot_value:
            arr[store], arr[i] = arr[i], arr[store]
```

```
        store += 1
    # move pivot to its final place
    arr[store], arr[right] = arr[right], arr[store]
    return store

def quickselect(arr, left, right, k):
    """
    Return the k-th smallest element (0-based k).
    """
    if left == right:
        return arr[left]

    pivot_index = random.randint(left, right)
    pivot_index = partition(arr, left, right, pivot_index)

    # number of elements in left partition
    if k == pivot_index:
        return arr[k]
    elif k < pivot_index:
        return quickselect(arr, left, pivot_index - 1, k)
    else:
        return quickselect(arr, pivot_index + 1, right, k)

def kth_smallest(arr, k):
    """
    k is 1-based (1 means smallest). Returns the k-th smallest value or None for invalid k.
    """
    n = len(arr)
    if k < 1 or k > n:
        return None
    # convert to 0-based index for quickselect
    return quickselect(arr[:], 0, n - 1, k - 1)

# Example
arr = [7, 10, 4, 3, 20, 15]
print(kth_smallest(arr, 3))  # Output: 7   (3rd smallest element)
```

Explanation:

- Quickselect chooses a pivot and partitions elements into < pivot and >= pivot.
- After partitioning, the pivot is at its final sorted index p.
- If k-1 == p, pivot is the k-th smallest. If k-1 < p, search left partition; otherwise search right partition.
- Only one side is processed recursively — average time is O(n).

  Complexity:
  Average: O(n) time, O(1) extra space (in-place).
  Worst-case: $O(n^2)$ time (rare if pivot is random).

10. Design and implement an LRU (Least Recently Used) Cache with get(key) and put(key, value) operations in O(1) time..

To solve this, combine a hashmap (key → node) for O(1) access with a doubly-linked list to maintain usage order (most recent at head, least recent at tail). On get move the node to head; on put insert/move to head and remove tail when capacity exceeded.
order_date, product_id, sales_amount

```
class Node:
    def __init__(self, key=None, val=None):
        self.key = key
        self.val = val
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.map = {}  # key -> Node

        # dummy head and tail to avoid edge checks
        self.head = Node()
        self.tail = Node()
        self.head.next = self.tail
        self.tail.prev = self.head

    def _add_to_head(self, node):
        """Insert node right after dummy head (mark as most recent)."""
        node.next = self.head.next
        node.prev = self.head
        self.head.next.prev = node
        self.head.next = node

    def _remove_node(self, node):
        """Disconnect node from list."""
        prev_node = node.prev
        next_node = node.next
        prev_node.next = next_node
        next_node.prev = prev_node

    def _move_to_head(self, node):
        """Move existing node to head (most recent)."""
        self._remove_node(node)
        self._add_to_head(node)

    def _pop_tail(self):
        """Remove least-recent node (before dummy tail) and return it."""
        node = self.tail.prev
        self._remove_node(node)
        return node
```

```python
    def get(self, key: int) -> int:
        if key not in self.map:
            return -1
        node = self.map[key]
        self._move_to_head(node)
        return node.val

    def put(self, key: int, value: int) -> None:
        if key in self.map:
            node = self.map[key]
            node.val = value
            self._move_to_head(node)
        else:
            node = Node(key, value)
            self.map[key] = node
            self._add_to_head(node)

            if len(self.map) > self.capacity:
                tail = self._pop_tail()
                del self.map[tail.key]

# Example usage
cache = LRUCache(2)
cache.put(1, 1)
cache.put(2, 2)
print(cache.get(1))    # returns 1
cache.put(3, 3)        # evicts key 2
print(cache.get(2))    # returns -1 (not found)
cache.put(4, 4)        # evicts key 1
print(cache.get(1))    # returns -1
print(cache.get(3))    # returns 3
print(cache.get(4))    # returns 4
```

Explanation:

- The hashmap gives O(1) access to nodes by key.
- The doubly-linked list keeps items ordered by recent use; head = most recent, tail = least recent.
- On get, if found, move node to head and return value.
- On put, if key exists update value and move to head; otherwise create node and add to head. If capacity exceeded, remove node at tail and delete from hashmap.
- Dummy head/tail nodes simplify insertion/removal edge cases.
- **Complexity:** O(1) average time for get and put, O(n) space for the data structures.

11. Detect if a singly linked list has a cycle (loop).

Definition:

To solve this, use Floyd's Cycle-Finding algorithm (a.k.a. tortoise and hare): move one pointer (slow) by one step and another (fast) by two steps. If they ever meet, there is a cycle; if fast reaches the end, there is no cycle.
It improves readability and simplifies complex subqueries or recursive logic.

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def has_cycle(head):
    if not head or not head.next:
        return False

    slow = head
    fast = head.next

    while fast and fast.next:
        if slow is fast:
            return True
        slow = slow.next
        fast = fast.next.next

    return False

# Example usage
# Create a list 1 -> 2 -> 3 -> 4 -> 2 (cycle back to node with value 2)
n1 = ListNode(1)
n2 = ListNode(2)
n3 = ListNode(3)
n4 = ListNode(4)
n1.next = n2
n2.next = n3
n3.next = n4
n4.next = n2  # cycle

print(has_cycle(n1))  # Output: True

# For an acyclic list:
a = ListNode(1); b = ListNode(2); c = ListNode(3)
a.next = b; b.next = c
print(has_cycle(a))   # Output: False
```

Benefits:

- slow advances one node at a time, fast advances two nodes.
- If there is a loop, fast will eventually "lap" slow and they will point to the same node (detectable via is comparison).

- If fast reaches None (end of list), the list is acyclic.
- **Complexity:** O(n) time, O(1) extra space.

12. Write a program to check if two strings are anagrams of each other.

(An anagram means both strings contain the same characters with the same frequency, e.g., "listen" and "silent".)

```
def are_anagrams(s1, s2):
    # If lengths are not equal, cannot be anagrams
    if len(s1) != len(s2):
        return False

    # Count characters of both strings
    return sorted(s1) == sorted(s2)

# Example usage
print(are_anagrams("listen", "silent"))  # True
print(are_anagrams("hello", "world"))    # False
```

Explanation:

- **Step 1:** Check if lengths of both strings are equal. If not, they cannot be anagrams.
- **Step 2:** Sort both strings and compare them.
- **Step 3:** If sorted versions match, they are anagrams; otherwise, not.

13. Write a program to count subarrays with sum equal to k

Assume input: nums (array of integers), k (target sum)

```
from collections import defaultdict

def count_subarrays_with_sum(nums, k):
    prefix_count = defaultdict(int)
    prefix_count[0] = 1          # one way to have sum 0 (empty prefix)
    curr_sum = 0
    result = 0

    for x in nums:
        curr_sum += x
        # If there's a prefix with sum = curr_sum - k, then subarray(s) ending here sum
to k
        result += prefix_count[curr_sum - k]
        prefix_count[curr_sum] += 1

    return result

# Example
nums = [1, 1, 1]
```

k = 2
print(count_subarrays_with_sum(nums, k)) # Output: 2 (subarrays [1,1] at indices (0,1) and (1,2))

Explanation:
• Maintain curr_sum = sum of elements up to current index.
• A subarray (i..j) sums to k iff prefix_sum[j] - prefix_sum[i-1] == k → prefix_sum[i-1] == curr_sum - k.
• prefix_count stores how many times each prefix sum has occurred. For each curr_sum add prefix_count[curr_sum - k] to result.
• Update prefix_count[curr_sum] after counting.
Complexity: O(n) time, O(n) space.

14. Write a program to compute the product of array except self (without using division).

Assume input: nums (array of integers). Return an array output where output[i] is the product of all elements of nums except nums[i].

```
def product_except_self(nums):
    n = len(nums)
    if n == 0:
        return []

    # output[i] will hold product of elements to the left of i
    output = [1] * n

    # left product pass
    left_prod = 1
    for i in range(n):
        output[i] = left_prod
        left_prod *= nums[i]

    # right product pass (multiply output[i] by product of elements to the right)
    right_prod = 1
    for i in range(n - 1, -1, -1):
        output[i] *= right_prod
        right_prod *= nums[i]

    return output

# Example
nums = [1, 2, 3, 4]
print(product_except_self(nums))  # Output: [24, 12, 8, 6]
```

1. **First pass (left to right): output[i] stores product of all elements left of i.**
2. **Second pass (right to left): multiply output[i] by product of all elements right of i.**
3. **No division is used, so zeros are handled correctly.**
4. **Complexity: O(n) time, O(1) extra space if you ignore the output array (otherwise O(n) space).**

15. Write a program to find two numbers in an array that sum up to a target k.

Assume input: nums (array of integers), k (target sum). Return indices of the two numbers (or values).

```
def two_sum(nums, k):
    seen = {}  # value -> index
    for i, num in enumerate(nums):
        complement = k - num
        if complement in seen:
            return [seen[complement], i]  # indices of the two numbers
        seen[num] = i
    return []

# Example
nums = [2, 7, 11, 15]
k = 9
print(two_sum(nums, k))  # Output: [0, 1]  (nums[0] + nums[1] = 9)
```

Explanation:

Iterate over the array, for each element calculate complement = k - num.
Check if complement was already seen; if yes, return the indices.
Store each number in a hashmap (seen) with its index.

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