1 General

1.1 Interviewer Considerations

Notes:
- How did the candidate **analyze** the problem?
- Did the candidate miss any special or **edge** cases?
- Did the candidate approach the problem **methodically** and logically?
- Does the candidate have a strong foundation in basic computer science **concepts**?
- Did the candidate produce **working code**? Did the candidate **test** the code?
- Is the candidate's code clean and easy to read and **maintain**?
- Can the candidate **explain** their ideas clearly?

1.2 Steps for Success During the Technical Interview

Summary:
1. **Clarify the question**
   (a) Understand what the question is asking and gather example inputs and outputs.
   (b) Clarify constraints such as:
      i. Can numbers be negative or repeated?
      ii. Are values sorted or do we need to sort them?
      iii. Can we assume input validity?
   (c) Asking clarifying questions shows communication skills and prevents missteps.
2. **Design a solution**
   (a) Avoid immediate coding; propose an initial approach and refine it.
   (b) Analyze the algorithm's time and space complexity.
   (c) Consider and address edge cases.
   (d) Think aloud to demonstrate logical reasoning and collaboration.
   (e) Discuss non-optimal ideas to show your thought process.
3. **Write your code**
   (a) Structure the solution using helper functions.
   (b) Confirm API details when uncertain.
   (c) Use your strongest programming language and full syntax.
   (d) Write complete, working code—not pseudocode.
4. **Test your code**
   (a) Validate your solution with 1–2 example test cases.
   (b) Walk through each line using inputs.
   (c) Do not assume correctness—prove it through testing.
   (d) Discuss any further optimizations and their trade-offs.

1.3 Common Mistakes to Avoid

Warning:
1. Starting to code without clarifying the problem.
2. Failing to write or discuss sample inputs and outputs.
3. Using pseudocode instead of fully functional code.
4. Misunderstanding the problem or optimizing prematurely.

1.4 Syntax

Summary:

1. `dict.items()`
    - Returns a view object that displays a list of a dictionary's key-value tuple pairs.
2. `sorted(iterable, key=..., reverse=...)`
    - `iterable`: The sequence or collection (e.g., list, dictionary view) to be sorted.
    - `key=...`: A function that extracts a comparison key from each element. Sorting is performed based on the result of this function.
        - `key=lambda x: x[0]`: Sort by the first element of each tuple.
        - `key=lambda x: x[1]`: Sort by the second element of each tuple.
    - `reverse=...`: A boolean value. If `True`, sorted in descending order; otherwise, sorted in ascending order (default is `False`).

2 Arrays and Hashing

2.1 When to Use?

Summary:

- To count frequencies in $O(n)$ time.
- To check membership in constant time.
- To map keys to values (e.g., index, count, group).
- To group elements by shared features (e.g., anagrams).
- To detect duplicates efficiently.

2.2 Hashing

Algorithm:

```python
def solve_problem(nums):
    # Step 1: Initialize the hashmap (e.g., for frequency, index, or existence check)
    hashmap = {}

    # Step 2: Iterate over the array
    for i, num in enumerate(nums):
        # Step 3: Define your condition (e.g., check complement, existence, frequency)
        if some_condition_based_on_hashmap(num, hashmap):
            # Step 4: Return or process result as needed
            return result_based_on_condition

        # Step 5: Update the hashmap
        hashmap_update_logic(num, i, hashmap)

    # Step 6: Handle the case where the condition is never met
    return final_result_if_needed

# Helper functions (replace with actual logic based on the problem)
def some_condition_based_on_hashmap(num, hashmap):
    # Example: return (target - num) in hashmap
    pass

def hashmap_update_logic(num, i, hashmap):
    # Example: hashmap[num] = i
    pass
```

2.3 Common Problems

Summary:

Problem	**Description:**
217. Contains Duplicate	Given an integer array nums, return true if any value appears at least twice.

- Use a set to store the elements. If an element is already in the set, return True.
- Otherwise, add it to the set.

242. Valid Anagram	Given two strings s and t, return true if t is an anagram of s and false otherwise.

- Use a hashMap to count the frequency of each character in s and t.
- If the frequency maps are equal, return True. Otherwise, return False.

1. Two Sum	Given an array of integers, return indices of the two numbers s.t. they add up to a specific target.

- **Tricks:**
  - Use a hashMap to store the indices of the elements, `prevMap[nums[i]] = i`
  - For each element, check if the target - nums[i] is in the map.
  - If it is, return the index of the target - nums[i] (from prevMap) and i. Otherwise, add target - nums[i].

**49. Group Anagrams	Given an array of strings, group the anagrams together.

- Use a hashMap to store a tuple of count of each char as the key and the list of words as the value.
- For each word, create a tuple of count of each char and add the word to the list in the map.
- Finally, return the values of the map.

**347. Top K Frequent Elements	Given an integer array nums and an integer k, return the k most frequent elements.

- Use a hashMap to count the frequency of each element.
- Sort the map by frequency and return the top k elements.

73. Set Matrix Zeroes	Given an m x n integer matrix, if an element is 0, set its entire row and column to 0.

- **Record Zero Positions:** Iterate through all elements. If `matrix[i][j] == 0`, append `[i, j]` to list.
- **Row/Column Zeroing:** Set all elements in column `col_ind` to zero and all elements in row `row_ind` to zero using two helpers.

3 Two Pointers

3.1 When to Use?

Summary:

- If we need to find a pair of elements that satisfy a condition.
- If we need to find a subarray that satisfies a condition.

3.2 Slow and Fast Pointers

Algorithm:

1.

3.2.1 Common Problems

Summary:

Problem	Description:
15. 3Sum	Given an array of integers, return all the triplets [nums[i], nums[j], nums[k]] s.t. i != j, i != k, and j != k. • **Tricks:**
125. Valid Palindrome	Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases. • `s_new = ".join(char.lower() for char in s if char.isalnum())` to remove non-alphanumeric and lowercase. • Use front and back pointers. If they not equal, return False. If equal move both pointers.
167. Two Sum II - Input array is sorted	Given an array of integers that is already sorted in ascending order, find two numbers such that they add up to a target. • Use front and back pointers. If > target, move back pointer left. If < target, move front pointer right.

3.3 Left and Right Pointers

Algorithm:

1. Initialize two pointers. Some common choices:
   - One at the front and one at the back of the array.
   - Both at the front of the array.
   - Both at the back of the array.

3.3.1 Common Problems

Summary:

Problem	Description:
15. 3Sum	Given an array of integers, return all the triplets [nums[i], nums[j], nums[k]] s.t. i != j, i != k, and j != k.

- **Tricks:**

125. Valid Palindrome	Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

- `s_new = ''.join(char.lower() for char in s if char.isalnum())` to remove non-alphanumeric and lowercase.
- Use front and back pointers. If they not equal, return False. If equal move both pointers.

167. Two Sum II - Input array is sorted	Given an array of integers that is already sorted in ascending order, find two numbers such that they add up to a target.

- Use front and back pointers. If > target, move back pointer left. If < target, move front pointer right.

4 Sliding Window

4.1 Fixed Sliding Window

Summary:

- Find a subarray/substring of a fixed size that satisfies a condition.
- Find the maximum or minimum of a subarray of a fixed size.

Algorithm:

```
initialize window_sum = 0
initialize max_result (or other required value)

# Set up initial window
for i in range(0, k):
    window_sum += arr[i]

max_result = window_sum # Initialize result

# Slide the window
for i in range(k, n):
    window_sum += arr[i] - arr[i - k] # Add new element and remove 1st element of prev window
    max_result = max(max_result, window_sum) (or other computation)

return max_result (or other required value)

```

4.1.1 Common Problems

Summary:

Problem	Description:
643. Maximum Average Subarray I	Given an integer array nums and an integer k, return the maximum average value of a subarray of length k.

- Follow template.

567. Permutation in String	Given two strings s1 and s2, return true if s2 contains a permutation of s1, or false otherwise.

- **Init:** Follow template with window_valid, freqMap_window, freqMap_s1, and fixed size k of len(s1). Rather than sum, get freq of chars.
- **Special Case:** If len(s1) > len(s2), return False.
- **For:** Since contiguous, slide through s2 and update freqMap_window by adding new char and removing old char (make sure to del key if freq = 0).
- **Condition:** If freqMap_window == freqMap_s1, return True.

219. Contains Duplicate II	Given an integer array nums and an integer k, return true if there are two distinct indices i and j in the array such that nums[i] == nums[j] and abs(i - j) <= k.

- **Init:** Follow template with window_freq and fixed size k.
- **Special Case:** If len(nums) < 2, return False.
- **Initial window:** Range(min(k+1, len(nums))) since first window can be smaller than k.

4.2 Dynamic Sliding Window

Summary:

- Find longest or shortest subarray/substring that satisfies a condition.

Algorithm:

```
initialize left = 0
initialize window_state (sum, count, frequency map, etc.)
initialize min_or_max_result

for right in range(n):
    update window_state to include arr[right] # Expand the window

    while window_state violates the condition:
        update min_or_max_result (if needed)
        update window_state to exclude arr[left] # Shrink the window
        move left pointer forward

return min_or_max_result
```

4.2.1 Common Problems

Summary:

Problem	Description:
121. Best Time to Buy and Sell Stock	Given an array where the ith element is the price of a stock on day i, find the maximum profit you can achieve. You may not engage in multiple transactions.

- **Buy low, sell high principle**
  - Use left = buy and right = sell, initialized at 0, 1.
  - If price[right] >= price[left], update max profit. Move right pointer since we can still sell for a profit.
  - If price[right] < price[left], move left pointer since we need to find a lower price to buy.
  - Continue until right pointer reaches the end of the array.

3. Longest Substring W/O Repeating Characters	Given a string s, find the length of the longest substring without repeating characters.

- **Init:** Follow template and use frequency map of chars for window_state.
- **While:** If a char is repeated, move left pointer to right by 1 and adjust freqMap until current char is unique.
- **Change:** Compare substring length outside of while with `max_res = max(max_res, right - left + 1)`.

424. Longest Repeating Character Replacement	Given a string s that consists of only uppercase English letters, you can replace any letter with another letter. Find the length of the longest substr containing the same letter after performing at most k replacements.

- **Init:** Follow template and use freqMap of chars for window_state.
- **While:** If the number of replacements needed exceeds k, i.e. `(r - l + 1) - max_freq > k`
  - Move left pointer to right by 1 and adjust freqMap until the condition is satisfied.
- **Change:** Compare substring length outside of while with `max_res = max(max_res, right - left + 1)`.

**76. Minimum Window Substring	Given two strings s and t, return the minimum window substr of s such that every character in t (including duplicates) is included in the window. If there is no such substring, return ""

- **Init:** Set `left = 0`. Initialize `count_t` as frequency map of `t`, `count_s` for current window, and variables `have = 0`, `required = len(count_t)`, `res = [-1, -1]`, and `resLen = \infty`.
- **For right in range(n):** Expand window by adding `s[right]` to `count_s`. If frequency matches `count_t`, increment `have`.
- **While have == required:**
  - Update result if current window is smaller.
  - Shrink window by decrementing `count_s[s[left]]`; if below `count_t`, decrement `have`; increment `left`.
- **Return:** `s[res[0]:res[1]+1]` if valid window found, else empty string.

239. Sliding Window Maximum	Given an integer array nums and an integer k, return the maximum value in each sliding window of size k.

- **Init:** Use deque to store indices of elements in the current window.
- **For right in range(n):**
  - Remove indices that are out of the current window.
  - Remove indices from the back of the deque while the current element is greater than the element at those indices.
  - Append the current index to the deque.
  - If the window size is reached, append the maximum (element at the front of the deque) to the result list.

5 Binary Search

5.1 When to Use?

Summary:

- Use when the input is **sorted** or can be **monotonically mapped**.
- Common for problems involving **searching for a target**, **finding boundaries**, or **min/max constraints**.
- Works on arrays, answer ranges, or implicit search spaces with $\mathcal{O}(\log n)$ complexity.

Algorithm:

```python
def binary_search(nums, target):
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

5.1.1 Common Problems

Summary:

6 Linked List

Summary: Data structure for storing objects in linear order.

- **Object:** Data and a pointer to the next object.

6.1 When to Use?

Summary:

- Implement other DS: stacks, queues, hash tables.
- Dynamic memory allocation.

6.2 Operations

Summary:

Operation	**Time Complexity**
Search	$O(n)$
Insert	$O(1)$
Delete	$O(1)$
Access	$O(n)$

6.3 Singly Linked List

Algorithm:

6.4 Doubly Linked List

Algorithm:

6.5 Circular Linked List

Algorithm:

6.6 Common Problems

Summary:

Problem	**Description:**
206. Reverse Linked List	Given the head of a singly linked list, reverse the list and return the reversed list.

- **Iterative:**
  - Init: $\underbrace{\text{None}}_{\text{prev}} \to \underbrace{0}_{\text{curr}} \to 1 \to 2$
  - While loop until curr is None. curr will point to prev, then curr will get updated to a temp that has curr.next and prev will be updated to curr.
    * $\underbrace{\text{None}}_{\text{prev}} \leftarrow \underbrace{0}_{\text{curr}} \to \underbrace{1}_{\text{temp}} \to 2$
    * $\text{None} \leftarrow \underbrace{0}_{\text{prev=curr}} \to \underbrace{1}_{\text{curr=temp}} \to 2$
  - prev will be the new head.

7 Images

7.1 Common Problems

Summary:

Problem	Description
661. Image Smoother	Given an image represented by a 2D array, smooth the image by averaging the pixel values of each pixel and its neighbors.

- Loop through the cols and rows of the image, then
  - total sum for each pixel $= \sum_{x,y \in \text{neighbours}} \text{image}[x][y] = \sum_{x=i-1}^{i+1} \sum_{y=j-1}^{j+1} \text{image}[x][y]$
    - If x or y is out of bounds, ignore it.
  - count $= \sum_{x=i-1}^{i+1} \sum_{y=j-1}^{j+1} 1$
  - average = total sum//count
- result$[i][j]$ = average

Problem	Description
832. Flipping an Image	Given a binary matrix, flip the image horizontally and invert it.

- Loop through the rows of the image, then use .reverse() to flip the row horizontally.
- Double for loop to invert image (change 0 to 1 and 1 to 0).

Problem	Description
48. Rotate Image	Given an n x n 2D matrix, rotate the image 90 degrees clockwise.

- Transpose the matrix (swap rows and columns) if $i < j$, then matrix$[i][j] \overset{\text{swap}}{\Longleftrightarrow}$ matrix$[j][i]$.
- Reverse each row.

Problem	Description
**835. Image Overlap	Given two images represented by 2D arrays, find the maximum overlap between the two images.

- Try all possible translations of img1.
- For each translation, calculate the overlap with img2.

8 Trees

8.1 Binary Search Tree (BST)

Summary:

- A binary tree where for each node, left subtree values are smaller, and right subtree values are larger.
- Enables $\mathcal{O}(\log n)$ average-case operations for insert, delete, search on balanced trees.
- Used to implement sets, maps, and ordered data structures.
- Core operations include search, insertion, deletion, in-order traversal, and range queries.

Algorithm:

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def search_bst(root, target):
    if not root or root.val == target:
        return root
    if target < root.val:
        return search_bst(root.left, target)
    else:
        return search_bst(root.right, target)

def insert_bst(root, val):
    if not root:
        return TreeNode(val)
    if val < root.val:
        root.left = insert_bst(root.left, val)
    else:
        root.right = insert_bst(root.right, val)
    return root

def delete_bst(root, key):
    if not root:
        return None
    if key < root.val:
        root.left = delete_bst(root.left, key)
    elif key > root.val:
        root.right = delete_bst(root.right, key)
    else:
        if not root.left:
            return root.right
        if not root.right:
            return root.left
        successor = get_min(root.right)
        root.val = successor.val
        root.right = delete_bst(root.right, successor.val)
    return root

def get_min(node):
    while node.left:
        node = node.left
    return node
```

8.1.1 Common Problems

Summary:

8.1.2 BST-based Sets and Maps

Summary:

- **BST Set**: Stores unique values in sorted order. Supports insert, search, delete.
- **BST Map**: Associates keys with values, maintaining keys in sorted order.
- Can be implemented using self-balancing trees (e.g., AVL, Red-Black Tree) for O(log n) operations.
- Useful for range queries, floor/ceiling lookups, and ordered iteration.

Algorithm:

```python
class BSTSet:
    def __init__(self):
        self.root = None

    def add(self, val):
        self.root = insert_bst(self.root, val)

    def contains(self, val):
        return search_bst(self.root, val) is not None

    def remove(self, val):
        self.root = delete_bst(self.root, val)

class BSTMap:
    def __init__(self):
        self.root = None

    def put(self, key, value):
        self.root = self._put(self.root, key, value)

    def _put(self, node, key, value):
        if not node:
            return TreeNode((key, value))
        if key < node.val[0]:
            node.left = self._put(node.left, key, value)
        elif key > node.val[0]:
            node.right = self._put(node.right, key, value)
        else:
            node.val = (key, value)
        return node

    def get(self, key):
        node = self.root
        while node:
            if key < node.val[0]:
                node = node.left
            elif key > node.val[0]:
                node = node.right
            else:
                return node.val[1]
        return None
```

8.2 Breadth-First Search (BFS)

Summary:

- Use when exploring nodes layer-by-layer, typically in unweighted graphs or grids.
- Ideal for finding the shortest path, level order traversal, or minimum number of steps.
- Queue-based traversal ensures nodes are visited in order of increasing distance from the source.

Algorithm:

```python
from collections import deque

def bfs(start, graph):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft()

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

8.2.1 Common Problems

Summary:

8.3 Depth-First Search (DFS)

Summary:

- Use when traversing all nodes or paths in **trees**, **graphs**, or **matrices**.
- Ideal for problems involving **backtracking**, **recursion**, or exploring all **connected components**.
- Can be implemented recursively or iteratively with a stack.
- Maintain a `visited` set or matrix to avoid revisiting nodes.
- Useful for **topological sorting**, **cycle detection**, and **pathfinding**.

Algorithm:

```python
def dfs(node, visited):
    if node in visited:
        return

    visited.add(node)

    for neighbor in graph[node]:
        dfs(neighbor, visited)
```

8.3.1 Common Problems

Summary: