# 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.

# 2 Arrays

# 3 Hashing

## 3.1 When to Use?

**Summary**:

- 

## 3.2 Hashing

**Algorithm**:

## 3.3 Common Problems

**Summary**:

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

- **Tricks:**
  - See if target - nums[i] is in the map.
  - If it is, return the index of the target - nums[i] (from prevMap) and i.
    - `prevMap[nums[i]] = i`

# 4 Two Pointers

## 4.1 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.

## 4.2 Slow and Fast Pointers

**Algorithm**:
1.

### 4.2.1 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. | |

## 4.3 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.

### 4.3.1 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. | |

# 5 Sliding Window

## 5.1 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)

```

### 5.1.1 Common Problems

**Summary**:

| Problem | Description: |
|---|---|
| 643. Maximum Average Subarray I | Given an integer array nums and an integer k,<br>return the maximum average value of a subarray of length k. |
| 567. Permutation in String | Given two strings s1 and s2, return true if s2 contains a<br>permutation of s1, or false otherwise. |

- Follow template.

- **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.

## 5.2 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
```

### 5.2.1 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 given 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.

# 6 Binary Search

## 6.1 When to Use?

**Summary**:
- If array is sorted.

# 7 Linked List

**Summary**: Data structure for storing objects in linear order.
- **Object:** Data and a pointer to the next object.

## 7.1 When to Use?

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

## 7.2 Operations

**Summary**:

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

## 7.3 Singly Linked List

**Algorithm**:

## 7.4 Doubly Linked List

**Algorithm**:

## 7.5 Circular Linked List

**Algorithm**:

## 7.6 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: $\underbrace{\text{None}}_{\text{prev}} \rightarrow \underbrace{0}_{\text{curr}} \rightarrow 1 \rightarrow 2$
  - 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.
    - $\underbrace{\text{None}}_{\text{prev}} \leftarrow \underbrace{0}_{\text{curr}} \rightarrow \underbrace{1}_{\text{temp}} \rightarrow 2$
    - $\text{None} \leftarrow \underbrace{0}_{\text{prev=curr}} \rightarrow \underbrace{1}_{\text{curr=temp}} \rightarrow 2$
  - prev will be the new head.