Arrays Basics (4 problems)

1. Find Max & Min
2. Sum of Array
3. Reverse Array
4. Check if Array is Sorted

Arrays Intermediate (4 problems)

5. Move Zeroes
6. Remove Duplicates from Sorted Array
7. Rotate Array
8. Intersection of Two Arrays

Strings Basics (4 problems)

9. Reverse String
10. Valid Anagram
11. Check if strings are rotation
12. Implement strstr() / substring search

Strings Intermediate (4 problems)

13. Longest Common Prefix
14. Count vowels & consonants
15. Remove duplicates in string
16. Most frequent character

Arrays + Strings Advanced (4 problems)

17. Kadane’s Algorithm (max subarray sum)
18. Merge Sorted Arrays
19. Longest Word in a Sentence
20. String Compression

Hashmaps (5 problems)

21. Two Sum (Hashmap version)
22. Contains Duplicate
23. First Unique Character
24. Group Anagrams
25. Word Frequency Counter

Hashmaps (5 problems)

26. Isomorphic Strings
27. Jewels and Stones
28. Find Duplicate in List
29. Top K Frequent Elements
30. Anagram Mapping

Two Pointers (5 problems)

31. Valid Palindrome
32. Palindrome Number
33. Remove Element
34. Pair with Target Sum
35. Square of Sorted Array

Two Pointers (5 problems)

36. 3Sum
37. Container With Most Water
38. Sort Colors (DNF)
39. Trapping Rainwater (concept only)
40. Minimum Difference Pair

Recursion + Math (5 problems)

41. Factorial (recursive)
42. Fibonacci (recursive)
43. Sum of Digits
44. Power of 2
45. Reverse Integer

Sorting (5 problems)

46. Bubble Sort
47. Selection Sort
48. Merge Sort (concept + small problem)
49. Quick Sort (concept + small problem)
50. Kth Smallest Element

Stacks & Queues (6 problems)

51. Valid Parentheses (most common)
52. Implement Stack using Queues
53. Implement Queue using Stacks
54. Min Stack
55. Daily Temperatures (using stack)
56. Next Greater Element

Trees (6 problems)

57. Binary Tree Traversals (in-order/pre/post)
58. Maximum Depth of Binary Tree
59. Invert Binary Tree
60. Symmetric Tree
61. Path Sum
62. Level Order Traversal (BFS)

Graphs (3 problems)

63. Number of Islands
64. Flood Fill
65. BFS Shortest Path (unweighted graph)

Sliding Window (5 problems)

66. Maximum Sum Subarray of Size K
67. Longest Substring Without Repeating Characters
68. Minimum Size Subarray Sum
69. Fruits Into Baskets (variable window)
70. Longest Repeating Character Replacement

1 Find Max & Min in an Array

□ Concept

You're given a list of numbers, like:

```
nums = [3, 10, 6, 2, 8]
```

You need to find:

- Maximum → `10`
- Minimum → `2`

We can do this in **one pass**:

1. Assume the first element is both `current_max` and `current_min`.
2. Traverse the list from left to right.
3. For each element:
    - If it's bigger than `current_max` → update `current_max`
    - If it's smaller than `current_min` → update `current_min`

□ Time: **O(n)**
🎀 Space: **O(1)**

✅ Code

```
def find_min_max(arr):
    """
    Find the minimum and maximum values in a non-empty list 'arr'.
    Returns a tuple: (min_value, max_value)
    """
    if not arr:
        # In interviews, they might specify arr is non-empty.
        # Here we handle empty list explicitly.
        raise ValueError("Array must not be empty")

    # Initialize both min and max with the first element
    current_min = arr[0]
    current_max = arr[0]

    # Start from index 1 because index 0 is already used
    for value in arr[1:]:
        # If current value is smaller than current_min, update current_min
        if value < current_min:
            current_min = value

        # If current value is larger than current_max, update current_max
        if value > current_max:
            current_max = value

    return current_min, current_max
```

```
# Example usage
nums = [3, 10, 6, 2, 8]
print(find_min_max(nums))  # Output: (2, 10)
```

---

2 Sum of an Array

□ Concept

Given:

```
nums = [1, 2, 3, 4]
```

You need: `1 + 2 + 3 + 4 = 10`.

You can:

- Use a **loop** and accumulate the sum.
- Or use Python's built-in `sum()` (but interviews often want the manual method too).

□ Time: **O(n)**
🎁 Space: **O(1)**

✅ Code

```
def sum_of_array(arr):
    """
    Return the sum of all numbers in arr.
    """
    total = 0  # start with 0, neutral element for addition

    # Go through each element and add it to total
    for value in arr:
        total += value  # same as total = total + value

    return total


# Example usage
nums = [1, 2, 3, 4]
print(sum_of_array(nums))  # Output: 10

# Interview tip:
# You can also write: sum(nums)
# But they may ask you to implement the logic yourself (like above).
```

---

3 Reverse an Array

□ Concept

Given:

```
nums = [1, 2, 3, 4, 5]
```

Reversed array:

```
[5, 4, 3, 2, 1]
```

Two common ways:

1. **Two-pointer approach (in-place)**
   - Pointer `left` at start, `right` at end.
   - Swap elements at `left` and `right`.
   - Move `left += 1`, `right -= 1` until they cross.
2. **Python shortcut**: `arr[::-1]` (not always allowed in interviews)

We'll implement the **two-pointer** method (very important pattern).

 Time: **O(n)**
 Space: **O(1)** (in-place)

✅ Code

```
def reverse_array(arr):
    """
    Reverse the list 'arr' in-place (modifies the original list).
    Also returns the same list for convenience.
    """
    left = 0                    # start pointer
    right = len(arr) - 1        # end pointer

    # Continue swapping until the two pointers meet or cross
    while left < right:
        # Swap elements at positions left and right
        arr[left], arr[right] = arr[right], arr[left]

        # Move left pointer to the right
        left += 1

        # Move right pointer to the left
        right -= 1

    return arr


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

If they allow Python shortcuts:

```
reversed_nums = nums[::-1]  # creates a new reversed list (not in-place)
```

---

4 Check if Array is Sorted (Non-decreasing)

□ Concept

You need to check if the array is sorted in **non-decreasing** order, i.e.:

```
[1, 2, 2, 3, 5]  ✅ sorted
[1, 3, 2, 4]     ❌ not sorted
```

Logic:

- Loop from index `0` to `len(arr) - 2`
- Compare each `arr[i]` with `arr[i+1]`
    - If you ever find `arr[i] > arr[i+1]` → it's **not sorted**
- If you never find such a case → it **is sorted**

□ Time: **O(n)**
🎁 Space: **O(1)**

✅ Code

```
def is_sorted(arr):
    """
    Check if the array 'arr' is sorted in non-decreasing order.
    Returns True if sorted, False otherwise.
    """
    # Loop until second last element, because we compare arr[i] with
arr[i+1]
    for i in range(len(arr) - 1):
        # If current element is greater than next element,
        # array is not sorted in non-decreasing order.
        if arr[i] > arr[i + 1]:
            return False

    # If we never found arr[i] > arr[i+1], it's sorted
    return True


# Example usage
print(is_sorted([1, 2, 2, 3, 5]))  # True
print(is_sorted([1, 3, 2, 4]))     # False
```

---

5 Move Zeroes to the End (keep order of non-zero)

□ Concept

Given:

```
nums = [0, 1, 0, 3, 12]
```

Output:

`[1, 3, 12, 0, 0]`

Rules:

- All **non-zero** elements must stay in the **same order**.
- All **zeroes go to the end**.
- Do it **in-place** (modify the same array), O(1) extra space.

Idea: Two-pointer (write position)

- Use `pos` = index where the next non-zero should be written.
- Traverse array with `i`:
  - If `nums[i] != 0` → write it at `nums[pos]`, then `pos += 1`.
- After this, all non-zero elements are at start, from `0` to `pos-1`.
- Fill the rest from `pos` to end with `0`.

 Time: O(n)
 Space: O(1)

✅ Code

```
def move_zeroes(nums):
    """
    Move all zeros in the list 'nums' to the end,
    while keeping the relative order of non-zero elements.
    The operation is done in-place.
    """
    # Position where the next non-zero should be placed
    pos = 0

    # First pass: move all non-zero values to the front
    for i in range(len(nums)):
        if nums[i] != 0:
            nums[pos] = nums[i]
            pos += 1

    # Second pass: fill the remaining positions with zeros
    while pos < len(nums):
        nums[pos] = 0
        pos += 1

    return nums  # returning for convenience


# Example
nums = [0, 1, 0, 3, 12]
print(move_zeroes(nums))  # [1, 3, 12, 0, 0]
```

---

6 Remove Duplicates from Sorted Array

Very common in interviews.

□ Concept

Given a **sorted** array:

`nums = [1, 1, 2, 2, 2, 3, 4, 4]`

We want:

- Unique elements **at the start**: `[1, 2, 3, 4, ...]`
- Return the **count of unique elements** (say `k`)
- The first `k` indices of `nums` will hold the unique values.

(LeetCode version returns `k`, but here we'll just return the sliced unique list.)

Idea: Two-pointer (slow–fast)

- `slow` → position to write the next unique element.
- `fast` → scans the array.
- Start `slow = 0`.
- For each `fast` from 1 to end:
  - If `nums[fast] != nums[slow]`:
    - `slow += 1`
    - `nums[slow] = nums[fast]`
- Unique count = `slow + 1`.

□ Time: O(n)
🛍 Space: O(1)

✅ Code

```
def remove_duplicates_sorted(nums):
    """
    Given a sorted list 'nums', remove duplicates in-place so that
    each element appears only once.
    Returns the list of unique elements (for understanding),
    but in interviews they often want the length (slow + 1).
    """
    if not nums:
        return []

    # slow is the index of the last unique element
    slow = 0

    # fast scans the array
    for fast in range(1, len(nums)):
        # When we find a new unique value
        if nums[fast] != nums[slow]:
            slow += 1                  # move slow forward
            nums[slow] = nums[fast]    # write the unique value at slow

    # Unique elements are now from index 0 to slow
    unique_part = nums[:slow + 1]
    return unique_part
```

```
# Example
nums = [1, 1, 2, 2, 2, 3, 4, 4]
print(remove_duplicates_sorted(nums))  # [1, 2, 3, 4]
```

---

7 Rotate Array Right by k Steps

□ Concept

Given:

```
nums = [1, 2, 3, 4, 5, 6, 7]
k = 3
```

Output:

```
[5, 6, 7, 1, 2, 3, 4]
```

Each step rotates to the right once:

- 1 step: [7, 1, 2, 3, 4, 5, 6]
- 2 steps: [6, 7, 1, 2, 3, 4, 5]
- 3 steps: [5, 6, 7, 1, 2, 3, 4]

Smart idea: Reverse method (common interview pattern)

Rotate right by `k`:

1. Normalize: `k = k % n` (in case k > n)
2. Reverse whole array.
3. Reverse first `k` elements.
4. Reverse remaining `n-k` elements.

Example:

- Original: [1, 2, 3, 4, 5, 6, 7], k=3
- Step1: Reverse all → [7, 6, 5, 4, 3, 2, 1]
- Step2: Reverse first 3 → [5, 6, 7, 4, 3, 2, 1]
- Step3: Reverse from 3 to end → [5, 6, 7, 1, 2, 3, 4]

□ Time: O(n)
🛍 Space: O(1)

✅ Code

```
def rotate_array(nums, k):
    """
    Rotate the list 'nums' to the right by 'k' steps in-place.
    Uses the reverse method.
    """
```

```
    n = len(nums)
    if n == 0:
        return nums

    # Normalize k to avoid extra full rotations
    k = k % n

    # Helper function to reverse a portion of the array in-place
    def reverse(left, right):
        while left < right:
            nums[left], nums[right] = nums[right], nums[left]
            left += 1
            right -= 1

    # Step 1: Reverse entire array
    reverse(0, n - 1)
    # Step 2: Reverse first k elements
    reverse(0, k - 1)
    # Step 3: Reverse remaining n-k elements
    reverse(k, n - 1)

    return nums


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

---

8 Intersection of Two Arrays (unique elements)

There are 2 common variants:

1. **Intersection without duplicates** → set-based.
2. **Intersection with counts** (like multiset).

We'll do **simple unique intersection**.

□ Concept

Given:

```
nums1 = [1, 2, 2, 1]
nums2 = [2, 2]
```

Output:

```
[2]
```

(only unique 2 is common)

Idea: Use sets

- Convert lists to sets to remove duplicates.

- Take intersection → `set1 & set2`.
- Convert back to list.

 Time: O(n + m) average (because set operations)
 Space: O(n + m)

✅ Code

```
def intersection_arrays(nums1, nums2):
    """
    Return the unique intersection of two lists.
    Each element in the result must be unique.
    """
    set1 = set(nums1)
    set2 = set(nums2)

    # Set intersection gives elements common to both sets
    result_set = set1 & set2

    # Convert back to list (order not guaranteed)
    return list(result_set)


# Example
nums1 = [1, 2, 2, 1]
nums2 = [2, 2]
print(intersection_arrays(nums1, nums2))  # [2]
```

---

9 Reverse String

🧠 Concept

Given a string:

```
s = "hello"
```

Output:

```
"olleh"
```

Two common ways:

1. Python shortcut: `s[::-1]`
2. Two-pointer (good for interviews, same pattern as arrays)

We'll implement **two-pointer**:

- Convert string → list of chars (because strings are immutable).
- Use `left = 0`, `right = len(s) - 1`.
- Swap characters at `left` and `right`, move inward until `left >= right`.
- Join back to string.

 Time: O(n)
 Space: O(n) (because of list)

✅ Code

```
def reverse_string(s):
    """
    Reverse the input string s and return the reversed string.
    Uses two-pointer technique.
    """
    # Strings are immutable, so convert to list of characters
    chars = list(s)
    left = 0
    right = len(chars) - 1

    # Swap characters from both ends
    while left < right:
        chars[left], chars[right] = chars[right], chars[left]
        left += 1
        right -= 1

    # Join list back into a string
    return "".join(chars)


# Example
print(reverse_string("hello"))   # "olleh"
print(reverse_string("abc"))     # "cba"
```

If they allow Python shortcuts:

```
def reverse_string_pythonic(s):
    return s[::-1]
```

---

10 Valid Anagram

□ Concept

Two strings are **anagrams** if:

- They contain the **same characters**
- With the **same frequency**
- Order does **not** matter.

Example:

```
s = "listen"
t = "silent"   → anagram ✅

s = "rat"
t = "car"      → not anagram ❌
```

Idea: Use frequency counting

- If lengths differ → immediately `False`.
- Count frequency of each character in `s`.
- Subtract frequency using characters from `t`.
- In the end, all counts must be zero.

We'll use a **dictionary (hashmap)** for counting.

 Time: O(n)
 Space: O(1) or O(k) (k = unique chars, bounded)

✅ Code

```
def is_anagram(s, t):
    """
    Check if strings s and t are anagrams of each other.
    Returns True if they are anagrams, False otherwise.
    """
    # Quick length check: anagrams must have same length
    if len(s) != len(t):
        return False

    # Dictionary to count characters in s
    freq = {}

    # Count chars in s
    for ch in s:
        freq[ch] = freq.get(ch, 0) + 1

    # Subtract counts using chars from t
    for ch in t:
        if ch not in freq:
            return False
        freq[ch] -= 1
        if freq[ch] < 0:
            # More of this char in t than in s
            return False

    # Optional: verify all zero (usually guaranteed now)
    # for val in freq.values():
    #     if val != 0:
    #         return False

    return True


# Example
print(is_anagram("listen", "silent"))  # True
print(is_anagram("rat", "car"))        # False
```

---

11 Check if One String is a Rotation of Another

🔹 Concept

Example:

```
s1 = "abcd"
s2 = "cdab"   → rotation ✅ (shifted)
s2 = "acbd"   → not rotation ❌
```

Definition: `s2` is a rotation of `s1` if we can rotate `s1` (move some chars from front to back) and get `s2`.

Smart Trick:

`s2` is a rotation of `s1` **iff**:

- `len(s1) == len(s2)`
- and `s2` is a **substring** of `s1 + s1`

Why?

`"abcd" + "abcd" = "abcdabcd"`
All possible rotations of `"abcd"` exist inside `"abcdabcd"`:

- "abcd"
- "bcda"
- "cdab"
- "dabc"

 Time: O(n²) worst case for `in` substring check (but fine for interviews).
🎁 Space: O(n)

✅ Code

```
def are_rotations(s1, s2):
    """
    Check if s2 is a rotation of s1.
    Example: s1 = 'abcd', s2 = 'cdab' -> True
    """
    # Lengths must be equal, else impossible
    if len(s1) != len(s2):
        return False

    # Concatenate s1 with itself
    doubled = s1 + s1

    # If s2 is a substring of doubled, it's a rotation
    return s2 in doubled


# Example
print(are_rotations("abcd", "cdab"))  # True
print(are_rotations("abcd", "acbd"))  # False
```

---

12 Implement `strstr()` / Substring Search

□ Concept

Re-creating the functionality of:

Find the index of first occurrence of `needle` in `haystack`
If not found, return `-1`.

Example:

```
haystack = "hello world"
needle   = "world"   → output: 6

haystack = "aaaaa"
needle   = "bba"     → output: -1
```

We'll implement the **naive approach**:

- Let `n = len(haystack)`, `m = len(needle)`
- Try all starting positions `i` from `0` to `n - m`
- For each `i`, compare `haystack[i:i+m]` with `needle`
- If equal → return `i`
- If none match → return `-1`

Special case: if `needle` is empty, many definitions return `0`.

□ Time: O(n * m) (fine for learning)
🛍 Space: O(1)

✅ Code

```
def strstr(haystack, needle):
    """
    Return the index of the first occurrence of 'needle' in 'haystack'.
    If 'needle' is not found, return -1.
    Behaves similar to haystack.find(needle).
    """

    # Edge case: empty needle usually returns 0
    if needle == "":
        return 0

    n = len(haystack)
    m = len(needle)

    # We only need to check positions where needle can fully fit
    # from 0 to n - m
    for i in range(n - m + 1):
        # Compare substring haystack[i:i+m] with needle
        if haystack[i:i + m] == needle:
            return i
```

```
    # If not found
    return -1


# Example
print(strstr("hello world", "world"))  # 6
print(strstr("aaaaa", "bba"))          # -1
print(strstr("abc", ""))               # 0
```

---

13 Longest Common Prefix (LCP)

□ Concept

Given a list of strings, find the **longest prefix** common to all of them.

Example:

```
["flower", "flow", "flight"] → "fl"
["dog", "racecar", "car"]    → ""  (no common prefix)
```

Simple idea (vertical scan):

- Take the **first string** as reference.
- For each character position `i` in that string:
  - Compare `s[0][i]` with `s[1][i]`, `s[2][i]`, ... for all strings.
  - If any mismatch OR any string finishes → stop.
- The prefix up to just before mismatch is your answer.

□ Time: O(N * L) (N = number of strings, L = length of smallest string)
🎁 Space: O(1) extra (ignoring output)

✅ Code

```
def longest_common_prefix(strs):
    """
    Given a list of strings, return the longest common prefix.
    If there is no common prefix, return an empty string "".
    """
    if not strs:
        return ""

    # Use the first string as the base reference
    first = strs[0]

    # We will build the prefix character by character
    prefix = ""

    # Loop over each character index in the first string
    for i in range(len(first)):
        current_char = first[i]

        # Check if this current_char is present at position i in all
strings
```

```
        for s in strs[1:]:
            # If index is out of range OR mismatch found -> stop
            if i >= len(s) or s[i] != current_char:
                return prefix  # prefix found so far

        # If loop didn't return, this char is common in all strings at
position i
        prefix += current_char

    return prefix


# Examples
print(longest_common_prefix(["flower", "flow", "flight"]))  # "fl"
print(longest_common_prefix(["dog", "racecar", "car"]))     # ""
```

---

14 Count Vowels & Consonants

□ Concept

Given a string, count:

- Number of **vowels**: `a, e, i, o, u` (case-insensitive)
- Number of **consonants**: letters that are alphabetic but not vowels

We'll:

1. Convert to lowercase.
2. For each character:
    - If it's alphabetic (`ch.isalpha()`):
        - If in `vowels set` → increment vowel count
        - Else → consonant count

□ Time: O(n)
🎁 Space: O(1)

✅ Code

```
def count_vowels_consonants(s):
    """
    Count the number of vowels and consonants in string s.
    Only alphabetic characters are considered.
    Returns a tuple: (vowel_count, consonant_count)
    """
    vowels = set("aeiou")
    vowel_count = 0
    consonant_count = 0

    for ch in s.lower():
        if ch.isalpha():  # consider only alphabets
            if ch in vowels:
                vowel_count += 1
            else:
```

```
                consonant_count += 1

    return vowel_count, consonant_count


# Example
text = "Hello World!"
print(count_vowels_consonants(text))  # (3, 7) -> vowels: e,o,o;
consonants: h,l,l,w,r,l,d
```

---

15 Remove Duplicates in a String (Keep First Occurrence)

□ Concept

Given:

```
s = "aaabbcddde"
```

Result:

```
"abcde"
```

We want to remove **repeated characters**, but **keep their first occurrence**, and maintain the **order**.

Idea:

- Use a `set` **seen** to track characters we've already added.
- Traverse string from left to right:
  - If char not in `seen` → add to result + add to `seen`.
  - If already in `seen` → skip.

□ Time: O(n)
🛍 Space: O(k) (k = unique characters)

✅ Code

```
def remove_duplicates_in_string(s):
    """
    Remove duplicate characters from s while preserving the order of
    first occurrence. Return the resulting string.
    Example: "aaabbcddde" -> "abcde"
    """
    seen = set()
    result_chars = []

    for ch in s:
        if ch not in seen:
            seen.add(ch)          # mark this char as seen
            result_chars.append(ch)  # keep the first occurrence

    # Join list of characters back into a string
```

```
    return "".join(result_chars)


# Example
print(remove_duplicates_in_string("aaabbcddde"))  # "abcde"
print(remove_duplicates_in_string("programming")) # "progamin"
```

---

16 Most Frequent Character in a String

□ Concept

Given:

```
s = "hello world"
```

Frequencies:

- h:1, e:1, l:3, o:2, w:1, r:1, d:1 → most frequent: `l`

We need to:

- Count frequency of all characters (often asked as: ignore spaces).
- Return the character with the **maximum frequency**.

We'll:

1. Use a dictionary `freq` → char → count
2. Traverse string, update counts.
3. Find key with **maximum value**.

(You can decide whether to ignore spaces or not; here I'll **ignore spaces**—common in practical use.)

□ Time: O(n)
🛍 Space: O(k)

✅ Code

```
def most_frequent_char(s):
    """
    Return the character that appears most frequently in the string s.
    Spaces are ignored.
    If multiple characters have the same max frequency, the first one
    reaching max while scanning will be returned.
    """
    freq = {}

    for ch in s:
        if ch == " ":
            continue  # ignore spaces
        freq[ch] = freq.get(ch, 0) + 1
```

```
    # If string is empty or only spaces
    if not freq:
        return None

    # Find character with maximum frequency
    max_char = None
    max_count = 0

    for ch, count in freq.items():
        if count > max_count:
            max_count = count
            max_char = ch

    return max_char, max_count


# Example
print(most_frequent_char("hello world"))      # ('l', 3)
print(most_frequent_char("aabbbccccdd"))      # ('c', 4)
```

---

17 Kadane's Algorithm (Maximum Subarray Sum)

□ Concept

Given an array:

```
nums = [-2,1,-3,4,-1,2,1,-5,4]
```

We want the **maximum sum of any continuous subarray**:

Subarray `[4, -1, 2, 1]` gives sum `6`.

Idea:

Maintain:

- `current_sum` → best sum ending at current position
- `max_sum` → best seen so far

At each element:

- `current_sum = max(nums[i], current_sum + nums[i])`
- `max_sum = max(max_sum, current_sum)`

□ Time: O(n)
🎁 Space: O(1)

✅ Code

```
def max_subarray(nums):
    """
    Return the maximum sum of any contiguous subarray using Kadane's
Algorithm.
    """
    current_sum = nums[0]
    max_sum = nums[0]

    for num in nums[1:]:
        # Either extend the previous subarray or start new one at current
num
        current_sum = max(num, current_sum + num)
        # Track the best seen so far
        max_sum = max(max_sum, current_sum)

    return max_sum


# Example
print(max_subarray([-2,1,-3,4,-1,2,1,-5,4]))  # 6
```

---

18 Merge Two Sorted Arrays

□ Concept

Given two **sorted** arrays:

```
a = [1, 3, 5]
b = [2, 4, 6]
```

Merged output:

```
[1, 2, 3, 4, 5, 6]
```

Two-pointer approach:

- Pointer `i` on `a`, `j` on `b`
- Compare `a[i]` and `b[j]`
- Append the smaller one
- Move pointer
- After loop, append remaining elements

□ Time: O(n + m)
🎁 Space: O(n + m)

✅ Code

```
def merge_sorted_arrays(a, b):
    """
    Merge two sorted lists into a single sorted list using two pointers.
    """
    i = 0  # pointer for a
```

```
    j = 0  # pointer for b
    result = []

    # Traverse both arrays
    while i < len(a) and j < len(b):
        if a[i] < b[j]:
            result.append(a[i])
            i += 1
        else:
            result.append(b[j])
            j += 1

    # Append remaining elements in either list
    result.extend(a[i:])
    result.extend(b[j:])

    return result


# Example
print(merge_sorted_arrays([1, 3, 5], [2, 4, 6]))  # [1,2,3,4,5,6]
```

---

19 Longest Word in a Sentence

□ Concept

Given a sentence:

```
s = "I love programming a lot"
```

Output:

```
"programming"  (length 11)
```

Idea:

- Split the sentence by spaces → words
- Track the longest by length

□ Time: O(n)
🎁 Space: O(k) (k = words)

✅ Code

```
def longest_word(sentence):
    """
    Return the longest word in the given sentence.
    If multiple words have same length, returns the first one.
    """
    words = sentence.split()
    longest = ""

    for word in words:
```

```
        if len(word) > len(longest):
            longest = word

    return longest


# Example
print(longest_word("I love programming a lot"))  # programming
```

---

20 String Compression (Run-Length Encoding – Basic)

□ Concept

Given:

```
s = "aaabbcddd"
```

Output:

```
"a3b2c1d3"
```

Rules:

- For each repeating character, store `char + count`.
- Single characters still get count `1`.

Approach:

- Loop through string with a counter.
- Whenever current char != previous char:
  - Append previous char + count
  - Reset count

□ Time: O(n)
🎁 Space: O(n)

✅ Code

```
def string_compression(s):
    """
    Compress string s using basic run-length encoding.
    Example: "aaabbcddd" -> "a3b2c1d3"
    """
    if not s:
        return ""

    result = []
    count = 1
```

```
    # Loop from index 1 to end
    for i in range(1, len(s)):
        if s[i] == s[i - 1]:
            count += 1
        else:
            # Append previous character and its count
            result.append(s[i - 1] + str(count))
            count = 1  # reset for new char

    # Append last group
    result.append(s[-1] + str(count))

    return "".join(result)


# Example
print(string_compression("aaabbcddd"))  # "a3b2c1d3"
print(string_compression("abc"))        # "a1b1c1"
```

---

21 Two Sum (Hashmap Version)

□ Concept

Given:

```
nums = [2, 7, 11, 15], target = 9
```

We want indices:

```
0, 1  (because 2 + 7 = 9)
```

Efficient Hashmap Approach

- Use dictionary:
  `value → index`
- For each element:
  - Compute `needed = target - nums[i]`
  - If `needed` in hashmap → return indices
  - Else store `nums[i]: index`

□ Time: **O(n)**
🎁 Space: **O(n)**

✅ Code

```
def two_sum(nums, target):
    """
    Return indices of the two numbers that add up to the target.
    Uses a hashmap for O(n) time.
    """
    seen = {}  # value -> index
```

```
    for i, num in enumerate(nums):
        needed = target - num

        # Check if needed value already seen
        if needed in seen:
            return [seen[needed], i]

        # Store current num with index
        seen[num] = i

    return []  # if no pair found


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

---

22 Contains Duplicate (Hashset approach)

□ Concept

Given:

```
nums = [1,2,3,1]
```

Output → True (duplicate exists).

Idea:

- Use a **set** to track seen numbers.
- If a number ever appears again → duplicate found.

□ Time: O(n)
🛍 Space: O(n)

✅ Code

```
def contains_duplicate(nums):
    """
    Return True if any number appears at least twice.
    """
    seen = set()

    for num in nums:
        if num in seen:
            return True
        seen.add(num)

    return False


# Example
print(contains_duplicate([1,2,3,1]))  # True
print(contains_duplicate([1,2,3]))    # False
```

23 First Unique Character in a String

□ Concept

Given:

```
s = "leetcode"
```

Output → index of first non-repeating character:

```
0  ('l' is unique)
```

Approach:

1. Count character frequencies using dictionary.
2. Traverse string again → first character whose freq = 1.

□ Time: O(n)
🎁 Space: O(1) (alphabet limited)

✅ Code

```
def first_unique_char(s):
    """
    Return the index of the first non-repeating character in s.
    If none exist, return -1.
    """
    freq = {}

    # First pass: count frequency
    for ch in s:
        freq[ch] = freq.get(ch, 0) + 1

    # Second pass: find first char with freq 1
    for i, ch in enumerate(s):
        if freq[ch] == 1:
            return i

    return -1


# Example
print(first_unique_char("leetcode"))  # 0
print(first_unique_char("aabb"))      # -1
```

---

24 Group Anagrams

□ Concept

Input:

```
["eat", "tea", "tan", "ate", "nat", "bat"]
```

Output groups:

```
[
  ["eat","tea","ate"],
  ["tan","nat"],
  ["bat"]
]
```

Key Idea:

- Anagrams have the same **sorted string**.
- Use sorted characters as dictionary key:
    - `"eat"` → `"aet"`
    - `"tea"` → `"aet"`
    - `"ate"` → `"aet"`

 Time: O(n * k log k)
 Space: O(n * k)

✅ Code

```
def group_anagrams(words):
    """
    Group words that are anagrams of each other.
    Returns list of grouped lists.
    """
    groups = {}  # key: sorted word, value: list of anagrams

    for word in words:
        # Sort characters to form key
        key = "".join(sorted(word))

        if key not in groups:
            groups[key] = []

        groups[key].append(word)

    return list(groups.values())


# Example
print(group_anagrams(["eat","tea","tan","ate","nat","bat"]))
```

---

25 Anagram Mapping

Concept

Given two arrays:

```
A = [12, 28, 46, 32, 50]
B = [50, 12, 32, 46, 28]
```

We want index mapping of A's values in B:

```
[1, 4, 3, 2, 0]
```

Because:

- A[0] = 12 → B index = 1
- A[1] = 28 → B index = 4
- …

Approach:

- Build hashmap: value → index for B.
- Loop through A, lookup in hashmap.

 Time: O(n)
 Space: O(n)

✅ Code

```
def anagram_mapping(A, B):
    """
    Given two lists A and B which are anagrams of each other,
    return a mapping where result[i] is the index of A[i] in B.
    """
    # Build value -> index mapping for B
    pos = {}
    for i, val in enumerate(B):
        pos[val] = i

    # Build mapping for A
    result = [pos[val] for val in A]
    return result


# Example
print(anagram_mapping([12, 28, 46, 32, 50], [50, 12, 32, 46, 28]))
# [1, 4, 3, 2, 0]
```

---

26 Isomorphic Strings

Concept

Two strings `s` and `t` are **isomorphic** if:

- Each character in `s` can be replaced to get `t`
- Mapping is **one-to-one** and **consistent**

Examples:

```
s = "egg", t = "add"   → True
s = "foo", t = "bar"   → False
s = "paper", t = "title" → True
```

We must ensure:

- Same char in `s` always maps to same char in `t`
- No two chars in `s` map to the same char in `t`

☞ Use two dictionaries:

- `s_to_t` : char in s → char in t
- `t_to_s` : char in t → char in s

 Time: O(n)
🎁 Space: O(1) (fixed set of characters)

✅ Code

```
def is_isomorphic(s, t):
    """
    Check if two strings s and t are isomorphic.
    """
    if len(s) != len(t):
        return False

    s_to_t = {}
    t_to_s = {}

    for ch_s, ch_t in zip(s, t):
        # Check mapping from s to t
        if ch_s in s_to_t:
            if s_to_t[ch_s] != ch_t:
                return False
        else:
            s_to_t[ch_s] = ch_t

        # Check mapping from t to s
        if ch_t in t_to_s:
            if t_to_s[ch_t] != ch_s:
                return False
        else:
            t_to_s[ch_t] = ch_s

    return True


# Examples
print(is_isomorphic("egg", "add"))     # True
print(is_isomorphic("foo", "bar"))     # False
print(is_isomorphic("paper", "title")) # True
```

---

27 Jewels and Stones

□ Concept

You are given:

- `J` = string of **jewel types** (unique chars)
- `S` = string of **stones you have**

Count how many stones are jewels.

Example:

```
J = "aA"
S = "aAAbbbb"
Output = 3  (a, A, A)
```

Idea:

- Put jewels in a **set** for O(1) lookup.
- Count characters of `S` that are in the set.

□ Time: O(n + m)
🎁 Space: O(m)

✅ Code

```
def num_jewels_in_stones(J, S):
    """
    Return how many characters in S are also in J.
    """
    jewel_set = set(J)
    count = 0

    for ch in S:
        if ch in jewel_set:
            count += 1

    return count


# Example
print(num_jewels_in_stones("aA", "aAAbbbb"))  # 3
```

---

28 Find Duplicate Number in Array

□ Concept

Given an array of integers `nums` containing `n+1` integers where each integer is in range `[1, n]`, with only **one repeated number**, return the duplicate.

Example:

```
nums = [1,3,4,2,2] → 2
nums = [3,1,3,4,2] → 3
```

Simple beginner approach (using hashmap):

- Use a `set` to track seen numbers.
- First number that appears again is duplicate.

 Time: O(n)
 Space: O(n)

(This is fine for learning; there are advanced O(1) tricks later like Floyd's cycle detection.)

✅ Code

```
def find_duplicate(nums):
    """
    Return the duplicate number in the list nums.
    Assumes exactly one number is duplicated.
    """
    seen = set()

    for num in nums:
        if num in seen:
            return num
        seen.add(num)

    return None  # theoretically shouldn't reach if input guaranteed valid


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

---

29 Top K Frequent Elements

Concept

Given:

```
nums = [1,1,1,2,2,3], k = 2
```

Output:

```
[1, 2]     # 1 appears 3 times, 2 appears 2 times
```

Steps:

1. Use hashmap to count frequencies.
2. Sort elements based on frequency (descending).

3. Pick top `k`.

(This is good enough for now; later you can learn heap / bucket sort optimizations.)

 Time: O(n log n) due to sorting
 Space: O(n)

✅ Code

```
def top_k_frequent(nums, k):
    """
    Return the k most frequent elements in nums.
    If k = 2, return the two elements with highest frequency.
    """
    freq = {}

    # Count frequencies
    for num in nums:
        freq[num] = freq.get(num, 0) + 1

    # Sort keys by their frequency in descending order
    # key=lambda x: freq[x] means: sort numbers by their count
    sorted_nums = sorted(freq.keys(), key=lambda x: freq[x], reverse=True)

    # Take first k elements from sorted list
    return sorted_nums[:k]


# Example
print(top_k_frequent([1,1,1,2,2,3], 2))  # [1,2]
```

---

30 Word Frequency (Log Analysis Style)

🧠 Concept

Given a string (like a log line or paragraph), count frequency of each word.

Example:

```
text = "apple banana apple orange banana apple"
```

Output (order doesn't matter):

```
apple: 3
banana: 2
orange: 1
```

Steps:

1. Split text into words using `split()`.
2. Use dictionary → `word: count`.
3. Optionally, sort by frequency.

 Time: O(n)
 Space: O(m) (m = unique words)

✅ Code

```
def word_frequency(text):
    """
    Given a text string, count the frequency of each word.
    Returns a dictionary: word -> count.
    """
    words = text.split()
    freq = {}

    for word in words:
        freq[word] = freq.get(word, 0) + 1

    return freq


# Example
text = "apple banana apple orange banana apple"
freq_map = word_frequency(text)
print(freq_map)  # {'apple': 3, 'banana': 2, 'orange': 1}

# If you want sorted by frequency (high to low):
sorted_words = sorted(freq_map.items(), key=lambda x: x[1], reverse=True)
print(sorted_words)  # [('apple', 3), ('banana', 2), ('orange', 1)]
```

---

31 Valid Palindrome (Two-Pointer)

🧠 Concept

Given a string, determine if it is a palindrome **ignoring non-alphanumeric characters** and case differences.

Example:

```
"A man, a plan, a canal: Panama"  → True
"race a car"                      → False
```

Approach (Two Pointers)

- Use `left = 0`, `right = len(s) - 1`
- While `left < right`:
    - Skip non-alphanumeric chars
    - Compare lowercase characters
    - Move pointers inward

 O(n)

✅ Code

```
def valid_palindrome(s):
    """
    Check if s is a palindrome, ignoring non-alphanumeric characters and
case.
    """
    left, right = 0, len(s) - 1

    while left < right:
        # Skip non-alphanumeric characters
        if not s[left].isalnum():
            left += 1
            continue
        if not s[right].isalnum():
            right -= 1
            continue

        # Compare characters (case-insensitive)
        if s[left].lower() != s[right].lower():
            return False

        left += 1
        right -= 1

    return True


# Examples
print(valid_palindrome("A man, a plan, a canal: Panama"))  # True
print(valid_palindrome("race a car"))                      # False
```

---

32 Palindrome Number (No string conversion version included)

□ Concept

Check if a number is palindrome:

```
121 → True
123 → False
```

Approach:

Option 1: Convert number to string → apply two pointers.
Option 2 (interview favorite): **Reverse half of the number** and compare.

Steps:

- Negative numbers → automatically False
- Reverse the second half and compare with first half

□ O(log n) digits

✅ Code (string method for beginners)

```
def is_palindrome_number(n):
    """
    Check if an integer n is a palindrome using string two-pointer method.
    """
    s = str(n)
    left, right = 0, len(s) - 1

    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1

    return True


# Examples
print(is_palindrome_number(121))  # True
print(is_palindrome_number(123))  # False
```

---

33 Remove Element (In-place Two Pointers)

🧠 Concept

Given array and value `val`, remove all occurrences of `val` **in-place**, return new length.

Example:

```
nums = [3,2,2,3], val = 3
Output array: [2,2,_ ,_]
Return length = 2
```

We use **two pointers**:

- `i`: read pointer
- `k`: write pointer
- Write only elements that are ≠ val.

⏱ O(n)

✅ Code

```
def remove_element(nums, val):
    """
    Remove all occurrences of val from nums in-place.
    Return new length (k) while modifying nums[0:k].
    """
    k = 0  # write pointer

    for i in range(len(nums)):
        if nums[i] != val:
```

```
            nums[k] = nums[i]  # keep the element
            k += 1

    return k  # nums[:k] contains all kept elements


# Example
nums = [3,2,2,3]
k = remove_element(nums, 3)
print(k, nums[:k])  # 2 [2,2]
```

---

34 Pair With Target Sum (Two Pointers)

□ Concept

Given a **sorted** array:

```
nums = [1,2,3,4,6], target = 6
```

Find pair that sums to target:

```
[1,5], [2,4], [3,3] ...
```

Best method:

- `left = 0, right = len(nums)-1`
- If sum > target → decrease right
- If sum < target → increase left

□ O(n)

✅ Code

```
def pair_with_target_sum(nums, target):
    """
    Find indices of two numbers whose sum equals target in a sorted list.
    """
    left, right = 0, len(nums) - 1

    while left < right:
        s = nums[left] + nums[right]

        if s == target:
            return [left, right]
        elif s < target:
            left += 1
        else:
            right -= 1

    return None


# Example
```

```
print(pair_with_target_sum([1,2,3,4,6], 6))  # [1,3] (2+4)
```

---

35 Squares of a Sorted Array (Two Pointers)

□ Concept

Array may have negatives:

```
[-4, -1, 0, 3, 10]
```

Squares:

```
[16, 1, 0, 9, 100] → sort → [0,1,9,16,100]
```

Better way:

- Because negatives become positive when squared, largest square is at either **left or right end**.
- Use two pointers and fill result array from end.

□ O(n)

✅ Code

```
def sorted_squares(nums):
    """
    Return the squares of nums sorted in non-decreasing order.
    Uses a two-pointer technique to avoid sorting after squaring.
    """
    n = len(nums)
    result = [0] * n

    left = 0
    right = n - 1
    pos = n - 1  # fill result from the end

    while left <= right:
        left_sq = nums[left] * nums[left]
        right_sq = nums[right] * nums[right]

        if left_sq > right_sq:
            result[pos] = left_sq
            left += 1
        else:
            result[pos] = right_sq
            right -= 1

        pos -= 1

    return result
```

```
# Example
print(sorted_squares([-4,-1,0,3,10]))  # [0,1,9,16,100]
```

---

36 3Sum

□ Concept

Given an array `nums`, find **all unique triplets** `(a,b,c)` such that:

```
a + b + c == 0
```

Example:

```
[-1,0,1,2,-1,-4] -> [[-1,-1,2],[-1,0,1]]
```

Idea (Sort + Two Pointers)

1. Sort the array.
2. Fix one element `nums[i]`.
3. Use two pointers `left`, `right` to find pairs such that:

```
nums[i] + nums[left] + nums[right] == 0
```

4. Skip duplicates carefully.

□ Time: $O(n^2)$

✅ Code

```
def three_sum(nums):
    """
    Return all unique triplets [a, b, c] such that a + b + c == 0.
    """
    nums.sort()
    result = []

    n = len(nums)

    for i in range(n):
        # Skip duplicate starting elements
        if i > 0 and nums[i] == nums[i - 1]:
            continue

        left = i + 1
        right = n - 1

        while left < right:
            s = nums[i] + nums[left] + nums[right]

            if s == 0:
                result.append([nums[i], nums[left], nums[right]])
```

```
                # Move left and right skipping duplicates
                left += 1
                right -= 1

                # Skip same left value
                while left < right and nums[left] == nums[left - 1]:
                    left += 1

                # Skip same right value
                while left < right and nums[right] == nums[right + 1]:
                    right -= 1

            elif s < 0:
                left += 1
            else:
                right -= 1

    return result


# Example
print(three_sum([-1, 0, 1, 2, -1, -4]))
```

---

37 Container With Most Water

□ Concept

Given an array `height`, each value is a vertical line.
Two lines + x-axis form a container → you want **max water**.

Example:

```
height = [1,8,6,2,5,4,8,3,7]
Output = 49
```

Area between `i` and `j`:

```
min(height[i], height[j]) * (j - i)
```

Two Pointer Idea

- Start with `left = 0`, `right = n-1` (widest container).
- Track max area.
- Move the pointer with **smaller height** inward:
    - Because height is limiting factor.

□ Time: O(n)

✅ Code

```
def max_area(height):
    """
```

```
    Given a list of heights, find the maximum area of water that can be
contained.
    """
    left, right = 0, len(height) - 1
    best = 0

    while left < right:
        h = min(height[left], height[right])
        width = right - left
        area = h * width
        best = max(best, area)

        # Move the pointer with smaller height
        if height[left] < height[right]:
            left += 1
        else:
            right -= 1

    return best


# Example
print(max_area([1,8,6,2,5,4,8,3,7]))  # 49
```

---

38 Sort Colors (Dutch National Flag)

□ Concept

Array has only `0, 1, 2`. Sort in-place so that:

```
[2,0,2,1,1,0] -> [0,0,1,1,2,2]
```

Constraints:

- One pass
- O(1) space

Three Pointers: `low`, `mid`, `high`

- `low` → boundary for 0s
- `high` → boundary for 2s
- `mid` → current index

Rules:

- If `nums[mid] == 0`: swap with `low`, `low++`, `mid++`
- If `nums[mid] == 1`: just `mid++`
- If `nums[mid] == 2`: swap with `high`, `high--` (do NOT mid++ immediately)

□ Time: O(n)

✅ Code

```
def sort_colors(nums):
    """
    Sort an array with values 0, 1, 2 using Dutch National Flag algorithm.
    """
    low = 0
    mid = 0
    high = len(nums) - 1

    while mid <= high:
        if nums[mid] == 0:
            # Swap to front
            nums[low], nums[mid] = nums[mid], nums[low]
            low += 1
            mid += 1
        elif nums[mid] == 1:
            # Correct region already
            mid += 1
        else:  # nums[mid] == 2
            # Swap to end
            nums[mid], nums[high] = nums[high], nums[mid]
            high -= 1

    return nums


# Example
print(sort_colors([2,0,2,1,1,0]))  # [0,0,1,1,2,2]
```

---

39 Trapping Rainwater

□ Concept

Each element is a bar height. After raining, water is trapped between bars.

Example:

```
height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output = 6
```

Water above each bar:

```
water[i] = min(max_left[i], max_right[i]) - height[i] (if positive)
```

Two Pointer Optimized Solution

Instead of using 2 extra arrays, we can:

- Use `left` and `right` pointers.
- Maintain `left_max` and `right_max`.
- Always move the pointer with **smaller** height:
    - If `height[left] < height[right]`:
        - If `height[left] >= left_max`: update `left_max`
        - Else: water += left_max - height[left]

        - Move `left++`
    - Else do symmetric for right.

 Time: O(n)
 Space: O(1)

✅ Code

```
def trap_rainwater(height):
    """
    Given heights of bars, compute how much water is trapped.
    Uses two-pointer optimized method.
    """
    if not height:
        return 0

    left, right = 0, len(height) - 1
    left_max = 0
    right_max = 0
    water = 0

    while left < right:
        if height[left] < height[right]:
            # Left side is the limiting factor
            if height[left] >= left_max:
                left_max = height[left]  # update barrier
            else:
                water += left_max - height[left]  # water trapped
            left += 1
        else:
            # Right side is the limiting factor
            if height[right] >= right_max:
                right_max = height[right]
            else:
                water += right_max - height[right]
            right -= 1

    return water


# Example
print(trap_rainwater([0,1,0,2,1,0,1,3,2,1,2,1]))  # 6
```

---

40 Minimum Difference Pair (Sorted + Two Pointer)

🧠 Concept

Given an array, find the **minimum absolute difference** between any two elements.

Example:

```
nums = [4, 2, 1, 9]
Sorted → [1, 2, 4, 9]
Differences: 1, 2, 5 → min = 1
```

Idea:

- Sort the array.
- Only adjacent elements can give minimum difference.
- Check diff between `nums[i]` and `nums[i+1]`.

 Time: O(n log n)

✅ Code

```
def min_difference(nums):
    """
    Return the minimum absolute difference between any two elements in
nums.
    """
    if len(nums) < 2:
        return 0  # or None, depending on definition

    nums.sort()
    min_diff = float("inf")

    for i in range(len(nums) - 1):
        diff = nums[i+1] - nums[i]
        if diff < min_diff:
            min_diff = diff

    return min_diff


# Example
print(min_difference([4, 2, 1, 9]))  # 1
```

---

41 Factorial (Recursive)

🧠 Concept

Factorial is defined as:

```
n! = n × (n-1) × (n-2) ... × 1
0! = 1
```

Recursive definition:

```
n! = n × (n-1)!
```

Base case:

```
if n == 0: return 1
```

Recursive step:

```
return n × factorial(n-1)
```

□ Time: O(n)

✅ Code

```
def factorial(n):
    """
    Recursively compute factorial of n.
    Base case: 0! = 1
    Recursive: n! = n * (n-1)!
    """
    if n == 0:
        return 1

    return n * factorial(n - 1)


# Example
print(factorial(5))  # 120
```

---

42 Fibonacci (Recursive + Optimized)

□ Concept

Recursive definition:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)
```

⚠ Problem:

Recursive version is slow → exponential time $O(2^n)$.

🔥 Optimized version: Memoization

Store results in dictionary.

Version 1: Simple Recursive (for learning)

```
def fib_recursive(n):
    """
    Compute Fibonacci number using basic recursion.
    Very slow for large n (exponential time).
    """
    if n <= 1:
        return n
    return fib_recursive(n-1) + fib_recursive(n-2)


print(fib_recursive(6))  # 8
```

Version 2: Optimized Recursive (Memoization)

```
def fib_memo(n, memo={}):
    """
    Compute Fibonacci using recursion + memoization.
    Time: O(n)
    """
    if n <= 1:
        return n

    if n in memo:
        return memo[n]

    memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo)
    return memo[n]


print(fib_memo(10))  # 55
```

---

43 Sum of Digits (Recursive)

□ Concept

Example:

```
n = 12345
Output = 1+2+3+4+5 = 15
```

Recursive logic:

```
sum_digits(n) = last digit + sum_digits(n // 10)
```

Base case:

```
if n == 0: return 0
```

□ O(d) where d = number of digits

✅ Code

```
def sum_of_digits(n):
    """
    Recursively compute the sum of digits of n.
    """
    if n == 0:
        return 0

    return (n % 10) + sum_of_digits(n // 10)


# Example
```

```
print(sum_of_digits(12345))  # 15
```

---

44 Power of 2 (Check if number is power-of-two)

□ Concept

A number is power of 2 if:

```
1, 2, 4, 8, 16, 32, ...
```

Binary representation has exactly **one 1**:

Example:

```
16 = 10000 (power of 2)
18 = 10010 (not a power of 2)
```

Conditions:

- n > 0
- n & (n-1) == 0

□ O(1)

✅ Code

```
def is_power_of_two(n):
    """
    Return True if n is a power of two.
    Uses bit manipulation.
    """
    if n <= 0:
        return False
    return (n & (n - 1)) == 0


# Examples
print(is_power_of_two(16))  # True
print(is_power_of_two(18))  # False
```

---

45 Reverse Integer

□ Concept

Given a number:

```
123 → 321
-456 → -654
```

Steps:

1. Extract last digit: `n % 10`
2. Reduce number: `n //= 10`
3. Build reversed number: `rev = rev * 10 + digit`

We handle signs and overflow (LeetCode 32-bit constraint).

□ O(d)

✅ Code

```
def reverse_integer(n):
    """
    Reverse digits of an integer.
    Handles negative numbers as well.
    """
    negative = n < 0
    n = abs(n)

    rev = 0
    while n > 0:
        digit = n % 10
        rev = rev * 10 + digit
        n //= 10

    return -rev if negative else rev


# Examples
print(reverse_integer(123))   # 321
print(reverse_integer(-456))  # -654
```

---

46 Bubble Sort

□ Concept

Bubble Sort repeatedly compares adjacent elements and swaps them if out of order.

Example:

```
[5,1,4,2]
Pass 1 → largest goes to end
Pass 2 → next largest positions
...
```

We do:

- Outer loop → number of passes

- Inner loop → compare neighbors
- Use a `swapped` flag to stop early if array already sorted

 Time: O(n²) (worst), O(n) (best if sorted)
🛍 Space: O(1)

✅ Code

```
def bubble_sort(arr):
    """
    Sort arr using bubble sort algorithm.
    """
    n = len(arr)

    for i in range(n):
        swapped = False

        # Last i elements are already sorted
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                # Swap neighbors
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        # If no swaps, list is already sorted
        if not swapped:
            break

    return arr


# Example
print(bubble_sort([5,1,4,2,8]))  # [1,2,4,5,8]
```

---

47 Selection Sort

🧠 Concept

Selection Sort finds the **minimum** element and places it at the left.

Process:

1. Find minimum in unsorted part
2. Swap with current index
3. Move to next index

 Time: O(n²)
🛍 Space: O(1)

✅ Code

```
def selection_sort(arr):
```

```
    """
    Sort arr using selection sort algorithm.
    """
    n = len(arr)

    for i in range(n):
        min_index = i

        # Find index of minimum element
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j

        # Swap
        arr[i], arr[min_index] = arr[min_index], arr[i]

    return arr


# Example
print(selection_sort([64, 25, 12, 22, 11]))
```

---

48 Merge Sort (Divide & Conquer)

□ Concept

Merge Sort uses **divide & conquer**:

1. Divide array into halves
2. Recursively sort both halves
3. Merge sorted halves

Properties:

- Stable
- Guaranteed O(n log n)
- Used in real-world systems (e.g., Java's TimSort base idea)

Merge Step:

Compare front of both halves and build sorted result.

□ O(n log n)

🎁 Space: O(n)

✅ Code

```
def merge_sort(arr):
    """
    Sort arr using merge sort (recursive divide & conquer).
    """
```

```
    if len(arr) <= 1:
        return arr  # already sorted

    mid = len(arr) // 2

    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)


def merge(left, right):
    """
    Merge two sorted lists into one sorted list.
    """
    result = []
    i = j = 0

    # Compare and merge
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Add remaining elements
    result.extend(left[i:])
    result.extend(right[j:])

    return result


# Example
print(merge_sort([5,2,4,7,1,3]))
```

---

49 Quick Sort (Partition Method)

□ Concept

Quick Sort picks a **pivot**, reorders array so:

- left part = elements < pivot
- right part = elements > pivot

Then recursively sorts subarrays.

Common partition method: **Lomuto partition**.

□

- Best/avg: O(n log n)
- Worst: O(n^2) if pivot chosen poorly

🛍️ Space: O(log n) recursion

✅ Code

```
def quick_sort(arr):
    """
    Sort arr using quick sort (Lomuto partition scheme).
    """
    def partition(low, high):
        pivot = arr[high]  # pick last element as pivot
        i = low - 1

        for j in range(low, high):
            if arr[j] < pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]

        # place pivot in correct position
        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        return i + 1

    def quick(low, high):
        if low < high:
            pi = partition(low, high)
            quick(low, pi - 1)
            quick(pi + 1, high)

    quick(0, len(arr) - 1)
    return arr


# Example
print(quick_sort([10, 7, 8, 9, 1, 5]))
```

---

50 Kth Smallest Element

□ Concept

Task: Find the **K-th smallest element** in an array.

Example:

```
arr = [7,10,4,3,20,15], k=3
Sorted → [3,4,7,10,15,20]
3rd smallest = 7
```

Methods:

1. **Sort and return arr[k-1]**
2. QuickSelect (optimized) → average O(n)

For Day 11, we use the simpler sorting approach.

□ O(n log n)

✅ Code

```
def kth_smallest(arr, k):
    """
    Return the k-th smallest element in the array.
    """
    arr_sorted = sorted(arr)
    return arr_sorted[k - 1]


# Example
print(kth_smallest([7,10,4,3,20,15], 3))  # 7
```

---

51 Valid Parentheses

□ Concept

You are given a string containing:

```
()  {}  []
```

Check if the string is **balanced**.

Example:

```
"()[]{}" → True
"(]"     → False
```

Idea (Stack):

- Push opening brackets onto stack
- When you see closing bracket:
  - Check if top of stack matches its pair
- If mismatch → invalid
- At end, stack must be empty

□ O(n)

✅ Code

```
def is_valid_parentheses(s):
    """
    Check if brackets in s are valid and balanced.
    Uses a stack.
    """
    stack = []
    pairs = {')': '(', ']': '[', '}': '{'}

    for ch in s:
```

```
        # Push opening brackets
        if ch in pairs.values():
            stack.append(ch)
        else:
            # If stack empty or mismatch → invalid
            if not stack or stack[-1] != pairs.get(ch):
                return False
            stack.pop()

    return len(stack) == 0


# Examples
print(is_valid_parentheses("()[]{}"))  # True
print(is_valid_parentheses("(]"))      # False
```

---

52 Implement Stack using Queues

□ Concept

You must simulate a **stack → LIFO** using **queue → FIFO**.

Best method:
Use **one queue** and rotate elements so that the newest element always comes to the front.

Steps for `push(x)`:

- Add x to queue
- Rotate elements until x becomes first

Data Structure:

`collections.deque` → efficient queue

□

Push: O(n)
Pop/Top: O(1)

✅ Code

```
from collections import deque

class MyStack:
    """
    Implement a Stack using a single Queue.
    push: O(n), pop/top: O(1)
    """
    def __init__(self):
        self.q = deque()

    def push(self, x):
```

```
        # Add item to queue
        self.q.append(x)
        # Rotate all elements except new one
        for _ in range(len(self.q) - 1):
            self.q.append(self.q.popleft())

    def pop(self):
        return self.q.popleft()

    def top(self):
        return self.q[0]

    def empty(self):
        return len(self.q) == 0


# Example
st = MyStack()
st.push(1)
st.push(2)
print(st.top())  # 2
print(st.pop())  # 2
print(st.empty())        # False
```

---

53 Implement Queue using Stacks

□ Concept

Simulate **FIFO** queue using two stacks.

- `in_stack` → push here
- `out_stack` → pop/peek here

When popping:

- If `out_stack` is empty → move elements from `in_stack` to `out_stack`.

□ amortized O(1)

✅ Code

```
class MyQueue:
    """
    Implement a Queue using two Stacks.
    push: O(1), pop: Amortized O(1)
    """
    def __init__(self):
        self.in_stack = []
        self.out_stack = []

    def push(self, x):
        self.in_stack.append(x)

    def pop(self):
```

```
        self.peek()  # ensure out_stack has elements
        return self.out_stack.pop()

    def peek(self):
        if not self.out_stack:
            # Move all from in_stack → out_stack
            while self.in_stack:
                self.out_stack.append(self.in_stack.pop())
        return self.out_stack[-1]

    def empty(self):
        return not self.in_stack and not self.out_stack


# Example
q = MyQueue()
q.push(1)
q.push(2)
print(q.peek())  # 1
print(q.pop())   # 1
print(q.empty()) # False
```

---

54 Min Stack

□ Concept

Design a stack that supports:

- `push()`
- `pop()`
- `top()`
- `getMin()` → return minimum in O(1)

Idea

Use **two stacks**:

- main stack → stores values
- min stack → stores running minimums

On push:

```
if val <= min_stack.top(): push to min_stack
```

On pop:

```
if popped == min_stack.top(): pop from min_stack
```

□ O(1) each operation

✅ Code

```
class MinStack:
    """
    Stack that can return minimum element in O(1) time.
    """
    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, x):
        self.stack.append(x)
        if not self.min_stack or x <= self.min_stack[-1]:
            self.min_stack.append(x)

    def pop(self):
        val = self.stack.pop()
        if val == self.min_stack[-1]:
            self.min_stack.pop()
        return val

    def top(self):
        return self.stack[-1]

    def getMin(self):
        return self.min_stack[-1]


# Example
ms = MinStack()
ms.push(3)
ms.push(1)
ms.push(2)
print(ms.getMin())  # 1
ms.pop()
print(ms.getMin())  # 1
```

---

55 Daily Temperatures (Monotonic Stack)

□ Concept

Given temperatures array, for each day return how many days until a **warmer day**.

Example:

```
[73, 74, 75, 71, 69, 72, 76, 73]
Output:
[1, 1, 4, 2, 1, 1, 0, 0]
```

Idea:

Use a **monotonic decreasing stack** storing indices.

- While current temp > temp at stack top → pop and compute wait days
- Otherwise push index

□ O(n)

✅ Code

```
def daily_temperatures(temps):
    """
    For each day, return how many days until a warmer temperature.
    Uses a monotonic decreasing stack.
    """
    n = len(temps)
    result = [0] * n
    stack = []  # store indices

    for i in range(n):
        # Resolve waiting days for colder temperatures
        while stack and temps[i] > temps[stack[-1]]:
            idx = stack.pop()
            result[idx] = i - idx
        stack.append(i)

    return result


# Example
print(daily_temperatures([73,74,75,71,69,72,76,73]))
```

---

56 Next Greater Element (Stack)

□ Concept

Given circular or linear array, find next element **greater than current**.

Example:

```
[2,1,2,4,3]
Output:
[4,2,4,-1,-1]
```

Idea:

Use **monotonic stack**:

- Traverse array
- While current > arr[stack.top] → pop and assign

□ O(n)

✅ Code

```
def next_greater_element(nums):
    """
    Return next greater element for each element in nums.
```

```
    If none exists, return -1 for that position.
    """
    n = len(nums)
    result = [-1] * n
    stack = []  # will store indices

    for i in range(n):
        while stack and nums[i] > nums[stack[-1]]:
            idx = stack.pop()
            result[idx] = nums[i]
        stack.append(i)

    return result


# Example
print(next_greater_element([2,1,2,4,3]))  # [4,2,4,-1,-1]
```

🌳 Binary Tree Node Definition (Used in all problems)

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

---

57 Tree Traversals (DFS)

□ Concept

DFS has 3 recursive orders:

1. Preorder → Root, Left, Right

2. Inorder → Left, Root, Right

3. Postorder → Left, Right, Root

These form the backbone of all tree-based questions.

□ Time: O(n)

🎁 Space: O(h) recursion stack

✅ Code

```
def preorder(root):
```

```
    if not root:
        return []
    return [root.val] + preorder(root.left) + preorder(root.right)


def inorder(root):
    if not root:
        return []
    return inorder(root.left) + [root.val] + inorder(root.right)


def postorder(root):
    if not root:
        return []
    return postorder(root.left) + postorder(root.right) + [root.val]


# Example Tree:
#     1
#    / \
#   2   3

root = TreeNode(1, TreeNode(2), TreeNode(3))
print(preorder(root))   # [1,2,3]
print(inorder(root))    # [2,1,3]
print(postorder(root))  # [2,3,1]
```

---

58 Maximum Depth of Binary Tree

□ Concept

Depth = longest path from root → leaf.

For example:

```
    1
   / \
  2   3
 /
4
```

Depth = 3

Idea:

Depth of a node =

```
1 + max(depth(left), depth(right))
```

□ Time: O(n)

✅ Code

```
def max_depth(root):
    """
    Return the maximum depth (height) of the tree.
    """
    if not root:
        return 0

    left = max_depth(root.left)
    right = max_depth(root.right)

    return 1 + max(left, right)
```

---

59 Invert Binary Tree

□ Concept

Swap left and right subtree for every node.

Example:

```
    4                 4
   / \               / \
  2   7    →        7   2
```

Simple Recursion:

- Swap children
- Recurse left
- Recurse right

□ O(n)

✅ Code

```
def invert_tree(root):
    """
    Swap left and right children recursively.
    """
    if not root:
        return None

    # Swap children
    root.left, root.right = root.right, root.left

    # Recurse
    invert_tree(root.left)
    invert_tree(root.right)

    return root
```

---

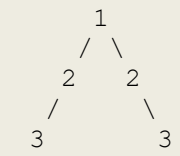
60 Symmetric Tree (Mirror Tree)

□ Concept

A tree is symmetric if:

Left subtree = Mirror of Right subtree.

Example (symmetric):



Recursive check:

```
isMirror(left, right):
- Both null → True
- One null → False
- Compare values
- Compare outer + inner children
```

□ O(n)

✅ Code

```
def is_symmetric(root):
    """
    Check if tree is symmetric around its center.
    """

    def is_mirror(a, b):
        # Both empty → symmetric
        if not a and not b:
            return True
        # One empty → not symmetric
        if not a or not b:
            return False
        # Values must match, and children must mirror
        return (a.val == b.val and
                is_mirror(a.left, b.right) and
                is_mirror(a.right, b.left))

    return is_mirror(root, root)
```

---

61 Level Order Traversal (BFS)

□ Concept

Traverse tree **level by level** using a queue.

Example:

```
Tree:
    3
   / \
  9  20
    /  \
   15   7

Output:
[[3], [9,20], [15,7]]
```

□ O(n)

✅ Code

```
from collections import deque

def level_order(root):
    """
    Return level order traversal of binary tree.
    """
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level = []
        size = len(queue)

        for _ in range(size):
            node = queue.popleft()
            level.append(node.val)

            if node.left:  queue.append(node.left)
            if node.right: queue.append(node.right)

        result.append(level)

    return result
```

---

62 Path Sum (Root → Leaf Path Exists?)

□ Concept

Check if there is a path from root to leaf such that:

```
sum of node values == targetSum
```

Example:

```
Tree:
    5
   / \
  4   8
Sum = 9 → Path exists? YES (5→4)
```

Recursive pattern:

```
targetSum - root.val
```

Recurse left or right.

□ O(n)

✅ Code

```
def has_path_sum(root, target):
    """
    Return True if there is a root-to-leaf path with the given sum.
    """
    if not root:
        return False

    # Leaf node
    if not root.left and not root.right:
        return root.val == target

    # Recurse on children
    return (has_path_sum(root.left, target - root.val) or
            has_path_sum(root.right, target - root.val))
```

---

63 Number of Islands

□ Concept

Given a 2D grid of **1 = land** and **0 = water**, count how many **separate islands** exist.

Example:

```
1 1 0 0 0
1 1 0 0 0
0 0 1 0 0
0 0 0 1 1

Output = 3 islands
```

Key Idea (Graph DFS on grid)

- Traverse every cell
- When you find a `1`:
  - Increase island count
  - Run DFS to **sink** the whole island (mark connected 1s as visited → set to 0)

- DFS explores 4 directions: up, down, left, right

 O(rows × cols)

✅ Code

```
def num_islands(grid):
    """
    Count number of islands in a binary grid.
    Uses DFS to mark visited land.
    """
    if not grid:
        return 0

    rows = len(grid)
    cols = len(grid[0])
    count = 0

    def dfs(r, c):
        # Boundary check OR water check
        if r < 0 or r >= rows or c < 0 or c >= cols or grid[r][c] == "0":
            return

        # Mark visited (sink island)
        grid[r][c] = "0"

        # Explore 4 directions
        dfs(r + 1, c)
        dfs(r - 1, c)
        dfs(r, c + 1)
        dfs(r, c - 1)

    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == "1":   # new island
                count += 1
                dfs(r, c)           # sink the island

    return count


# Example
g = [
    ["1","1","0","0","0"],
    ["1","1","0","0","0"],
    ["0","0","1","0","0"],
    ["0","0","0","1","1"]
]
print(num_islands(g))  # 3
```

---

64 Flood Fill (DFS or BFS)

Concept

Like paint bucket tool in MS Paint.

Given image matrix:

- Starting pixel `(sr, sc)`
- New color `newColor`

Fill all 4-directionally connected pixels with the same original color.

Example:

```
Image:
[1,1,1]
[1,1,0]
[1,0,1]

start = (1,1), newColor = 2

Output:
[2,2,2]
[2,2,0]
[2,0,1]
```

Idea:

- Store original color
- DFS into neighbors
- Change color only where original matches

 O(rows × cols)

✅ Code

```
def flood_fill(image, sr, sc, newColor):
    """
    Perform flood fill on image starting at (sr, sc).
    """
    rows = len(image)
    cols = len(image[0])

    original = image[sr][sc]
    if original == newColor:
        return image  # nothing to change

    def dfs(r, c):
        # Out of bounds or mismatched color
        if r < 0 or r >= rows or c < 0 or c >= cols or image[r][c] !=
original:
            return

        # Change color
        image[r][c] = newColor
```

```
        # Explore 4 neighbors
        dfs(r + 1, c)
        dfs(r - 1, c)
        dfs(r, c + 1)
        dfs(r, c - 1)

    dfs(sr, sc)
    return image


# Example
img = [
    [1,1,1],
    [1,1,0],
    [1,0,1]
]
print(flood_fill(img, 1, 1, 2))
```

---

65 BFS Shortest Path (Unweighted Graph)

□ Concept

In an **unweighted graph**, BFS gives shortest path in terms of edges.

Example graph (adjacency list):

```
A -> [B, C]
B -> [D]
C -> [D, E]
D -> [F]
E -> [F]
```

Find shortest path `A → F`.

BFS Strategy:

- Use queue
- Store `(node, distance)` or `parent mapping`
- First time we reach target → shortest path

□ O(V + E)

✅ Code (Shortest distance)

```
from collections import deque

def bfs_shortest_path(graph, start, target):
    """
    Return shortest distance from start to target in an unweighted graph.
    graph: dict of adjacency list
```

```
    """
    queue = deque([(start, 0)])  # (node, distance)
    visited = set([start])

    while queue:
        node, dist = queue.popleft()

        if node == target:
            return dist

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, dist + 1))

    return -1  # no path


# Example graph
graph = {
    "A": ["B", "C"],
    "B": ["D"],
    "C": ["D", "E"],
    "D": ["F"],
    "E": ["F"],
    "F": []
}

print(bfs_shortest_path(graph, "A", "F"))  # 3 (A → C → E → F)
```

🔥 Version 2: Return actual shortest path (not only distance)

```
def bfs_shortest_path_list(graph, start, target):
    queue = deque([[start]])  # store path list
    visited = set([start])

    while queue:
        path = queue.popleft()
        node = path[-1]

        if node == target:
            return path

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

    return None


print(bfs_shortest_path_list(graph, "A", "F"))
# ['A', 'C', 'E', 'F']
```

---

66 Maximum Sum Subarray of Size K

(Fixed-size sliding window)

□ Concept

Given array & window size `k`, find maximum sum of any subarray of length `k`.

Example:

```
[2,1,5,1,3,2], k=3 → max sum = 9 (5+1+3)
```

Sliding Window Idea

- Compute sum of first `k` elements.
- Slide window to right:
  - Add new element
  - Remove outgoing element
- Track max.

□ O(n)

✅ Code

```
def max_sum_subarray_k(nums, k):
    """
    Return the maximum sum of any contiguous subarray of size k.
    """
    window_sum = sum(nums[:k])
    max_sum = window_sum

    for i in range(k, len(nums)):
        window_sum += nums[i]          # add incoming
        window_sum -= nums[i - k]      # remove outgoing
        max_sum = max(max_sum, window_sum)

    return max_sum


# Example
print(max_sum_subarray_k([2,1,5,1,3,2], 3))  # 9
```

---

67 Longest Substring Without Repeating Characters

(Variable window, expanding and shrinking)

□ Concept

Example:

```
"abcabcbb" → 3 ("abc")
"bbbbb"    → 1 ("b")
```

Sliding Window + Hashmap

- Expand right pointer
- If duplicate found → shrink window from left until no duplicates remain
- Track longest window size

□ O(n)

✅ Code

```
def longest_unique_substring(s):
    """
    Return length of longest substring without repeating characters.
    """
    seen = {}      # char -> last index
    left = 0
    longest = 0

    for right, ch in enumerate(s):
        if ch in seen and seen[ch] >= left:
            # Move left pointer after the last occurrence of ch
            left = seen[ch] + 1

        seen[ch] = right
        longest = max(longest, right - left + 1)

    return longest


# Example
print(longest_unique_substring("abcabcbb"))  # 3
```

---

68 Minimum Size Subarray Sum

(Shrink window from left)

□ Concept

Given array and target sum, find smallest subarray whose sum ≥ target.

Example:

```
target = 7, nums = [2,3,1,2,4,3]
Answer = 2 (subarray [4,3])
```

Sliding Window Idea

- Expand right pointer to increase sum
- When sum ≥ target → shrink from left to make window minimal

 O(n)

✅ Code

```
def min_subarray_len(target, nums):
    """
    Return the minimum length of subarray with sum >= target.
    """
    left = 0
    window_sum = 0
    min_len = float("inf")

    for right in range(len(nums)):
        window_sum += nums[right]

        # Try shrinking window
        while window_sum >= target:
            min_len = min(min_len, right - left + 1)
            window_sum -= nums[left]
            left += 1

    return 0 if min_len == float("inf") else min_len


# Example
print(min_subarray_len(7, [2,3,1,2,4,3]))  # 2
```

---

69 Fruits Into Baskets

(Longest subarray with at most 2 distinct characters)

Concept

Equivalent to:

Longest subarray with at most 2 distinct values.

Example:

```
[1,2,3,2,2] → longest = 4 (2,3,2,2)
```

Sliding Window Idea

- Expand window
- Keep frequency count of fruits
- If >2 types → shrink window until back to ≤2 types
- Track longest window

 O(n)

✅ Code

```
def fruits_into_baskets(nums):
    """
    Return longest subarray length with at most 2 distinct values.
    """
    left = 0
    freq = {}
    longest = 0

    for right, val in enumerate(nums):
        freq[val] = freq.get(val, 0) + 1

        # Shrink window when >2 distinct fruits
        while len(freq) > 2:
            freq[nums[left]] -= 1
            if freq[nums[left]] == 0:
                del freq[nums[left]]
            left += 1

        longest = max(longest, right - left + 1)

    return longest


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

---

70 Longest Repeating Character Replacement

(Variable window + counting)

🧠 Concept

Given string and number `k`, you can replace at most `k` characters to make the longest substring with repeating characters.

Example:

```
s = "AABABBA", k = 1 → longest = 4 ("AABA")
```

Key Insight

Window is valid if:

```
window_size - max_frequency ≤ k
```

Meaning: we can replace the non-majority characters.

□ O(n)

✅ Code

```
def character_replacement(s, k):
    """
    Return the length of the longest substring where we can replace
    at most k characters to make all characters the same.
    """
    freq = {}
    left = 0
    max_freq = 0
    longest = 0

    for right, ch in enumerate(s):
        freq[ch] = freq.get(ch, 0) + 1
        max_freq = max(max_freq, freq[ch])

        # If replacements needed > k → shrink window
        while (right - left + 1) - max_freq > k:
            freq[s[left]] -= 1
            left += 1

        longest = max(longest, right - left + 1)

    return longest


# Example
print(character_replacement("AABABBA", 1))  # 4
```
