DSA(PROBLEM SOLVING)

STRING

1. Reverse a String

Brute Force Code

```
function reverseString(str) {
  let reversed = "";
  for (let i = str.length - 1; i >= 0; i--) {
    reversed += str[i];
  }
  return reversed;
}
console.log(reverseString("hello")); // "olleh"
```

Two Pointer Approach

```
function reverseStringOptimal(str) {
  let arr = str.split("");
  let left = 0;
  let right = arr.length - 1;
  while (left < right) {
    // Swap characters
    [arr[left], arr[right]] = [arr[right], arr[left]];
    left++;
    right--;
  }
  return arr.join("");
}
console.log(reverseStringOptimal("hello")); // "olleh"
```

2. Check if a string is a palindrome

```
function isPalindrome(str) {
  let left = 0, right = str.length - 1;

  while (left < right) {
    if (str[left] !== str[right]) return false;
    left++;
    right--;
  }

  return true;
```

```
}
```

3. Anagram Check

Check if two strings are anagrams of each other.
Two strings are anagrams if they have the same characters with the same frequencies, just in a different order.
Time Complexity: `O(n)`
Space Complexity: `O(1)` (if limited to lowercase English letters)

```
function isAnagram(str1, str2) {
  if (str1.length !== str2.length) return false;
  const count = Array(26).fill(0); // for lowercase 'a' to 'z'
  for (let i = 0; i < str1.length; i++) {
    count[str1.charCodeAt(i) - 97]++;
    count[str2.charCodeAt(i) - 97]--;
  }
  return count.every(c => c === 0);
}
console.log(isAnagram("listen", "silent")); // true
console.log(isAnagram("hello", "world"));   // false
```

- It avoids sorting (which is `O(n log n)`).
- It does a single scan over both strings (`O(n)` time).
- It uses fixed-size space (array of 26 for a-z), not a dynamic object.

With a Frequency Hash Map (supports any character set)

```
function isAnagramHash(str1, str2) {
 if (str1.length !== str2.length) return false;
 const map = {};
 for (let char of str1) {
   map[char] = (map[char] || 0) + 1;
 }
 for (let char of str2) {
   if (!map[char]) return false;
   map[char]--;
 }
 return true;
}
```

How it works step by step:
Suppose `str1 = "abbc"`

Initial: `map = {}`
Loop through each character:

1. First `a`:
   - `map['a']` is `undefined`, so `(undefined || 0)` → `0`
   - So `map['a'] = 0 + 1 = 1`
   - Result: `{ a: 1 }`

4. Longest Substring Without Repeating Characters

```
Given a string `s`, return the **length** of the **longest substring**
without repeating characters.
```

Best Approach: Sliding Window + Set

```
function longestUniqueSubstring(str) {
  let left = 0; // Window का बायाँ हिस्सा
  let maxLength = 0; // सबसे लंबी अनूठी सबस्ट्रिंग की लंबाई
  let charSet = new Set(); // Set का उपयोग कर के हम यूनिक characters ट्रैक करेंगे

  for (let right = 0; right < str.length; right++) {
    // अगर चरित्र पहले से मौजूद है तो left pointer को बढ़ाओ
    while (charSet.has(str[right])) {
      // duplicate character को हटाने के लिए बायें pointer को इधर-उधर करेंगे
      charSet.delete(str[left]);
      left++;
    }

    // नए character को सेट में डालना
    charSet.add(str[right]);

    // Max length को अपडेट करना
    maxLength = Math.max(maxLength, right - left + 1);
  }

  return maxLength;
}

// Example
const inputString = "abcdefaxy";
console.log(longestUniqueSubstring(inputString)); // Output: 7

 // example
longestUniqueSubstring("abcabcbb");
// Output: 3
// Longest: "abc"
longestUniqueSubstring("bbbbb");
```

```
// Output: 1
// Longest: "b"
```

5. Remove Duplicates

```
function removeDuplicate(str) {
  return [...new Set(str)].join('');
}
function removeDuplicate(arr) {
  if (!arr.length) return null;
  const newArray = [...new Set(arr)];
  return newArray;
}
```

Method 2: Using `Map` (if you want both uniqueness + frequency)

```
function removeDuplicate(str) {
  const map = new Map();
  let result = '';

  for (const char of str) {
    if (!map.has(char)) {
      map.set(char, 1);
      result += char;
    }
  }

  return result;
}
```

6. Find two distinct elements in the array such that their sum is equal to the target.

Sorted array hai? → Use Two Pointers

```
function findPairWithSum(arr, target) {
let i = 0, j = arr.length - 1;
while (i < j) {
  let sum = arr[i] + arr[j];
  if (sum === target) {
    return [arr[i], arr[j]];
  } else if (sum < target) {
    i++;
  } else {
    j--;
```

```
  }
}
return null;
}
```

Brute Force

```
function findPairBruteForce(arr, target) {
  for (let i = 0; i < arr.length; i++) {
    for (let j = i + 1; j < arr.length; j++) {
      if (arr[i] + arr[j] === target) {
        return [arr[i], arr[j]]; //  Return first pair found
      }
    }
  }

  return null; // ❌ No pair found
}
```

Using Set(O(n))

```
function findPairUsingSet(arr, target) {
  const seen = new Set();

  for (let num of arr) {
    const complement = target - num;
    if (seen.has(complement)) {
      return [complement, num]; //  Pair found
    }
    seen.add(num);
  }
  return null; // ❌ No pair found
}
```

findPairUsingMap (O(n))

```
function findPairUsingMap(arr, target) {
  const map = new Map(); // value → index (optional if just checking for
values)
  for (let i = 0; i < arr.length; i++) {
    const complement = target - arr[i];
    if (map.has(complement)) {
      return [complement, arr[i]];
    }
    map.set(arr[i], i);
```

```
  }

  return null;
```

7. Most Frequent Character

```
function mostFrequentCharBrute(str) {
  let maxChar = '';
  let maxCount = 0;
  for (let i = 0; i < str.length; i++) {
    let count = 0;
    for (let j = 0; j < str.length; j++) {
      if (str[i] === str[j]) {
        count++;
      }
    }
    if (count > maxCount) {
      maxCount = count;
      maxChar = str[i];
    }
  }

  return { char: maxChar, count: maxCount };
}
```

Return the Most Frequent Character

```
function mostFrequentChar(str) {
  const freqMap = new Map();
  let maxChar = '';
  let maxCount = 0;
  for (let char of str) {
    const count = freqMap.has(char) ? freqMap.get(char) + 1 : 1;
    freqMap.set(char, count);
    // 🔍 Keep track of the max while building the map
    if (count > maxCount) {
      maxCount = count;
      maxChar = char;
    }
  }
  return { char: maxChar, count: maxCount };
}
```

8.Valid Parentheses

Given a string containing just the characters (,), {, }, [and], determine if the input string is valid.

Brute-Force Idea Version

```
function isValidParenthesesBrute(s) {
  let prev = '';

  while (s !== prev) {
    prev = s;
    s = s.replace('()', '');
    s = s.replace('{}', '');
    s = s.replace('[]', '');
  }

  return s.length === 0;
}
```

Solution Using Stack (Best Way)

```
function isValidParentheses(s) {
  const stack = [];
  const map = {
    ')': '(',
    '}': '{',
    ']': '[',
  };
  for (let char of s) {
    if (char === '(' || char === '{' || char === '[') {
      stack.push(char);
    } else {
      if (stack.pop() !== map[char]) {
        return false;
      }
    }
  }
  return stack.length === 0;
}
```

9. Longest Common Prefix

Given an array of strings, find the longest common prefix string among them.
If there is no common prefix, return an empty string `""`.

```
Input: ["flower", "flow", "flight"]
Output: "fl"
function longestCommonPrefix(strs) {
```

```
  if (!strs.length) return "";

  let prefix = strs[0]; // मान लो पहला string ही prefix है

  for (let i = 1; i < strs.length; i++) {
    // जब तक current word में prefix नहीं है, एक-एक करके prefix छोटा करो
    while (strs[i].indexOf(prefix) !== 0) {
      prefix = prefix.slice(0, -1); // आखिरी character हटाओ
      if (prefix === "") return ""; // कुछ भी common नहीं मिला
    };
  };

  return prefix;
};
```

10. Count Vowels in a String

Count the number of vowels (a, e, i, o, u) in a given string.

```
function countVowels(str) {
  let vowelsCounter = { a: 0, e: 0, i: 0, o: 0, u: 0 };

  for (let i = 0; i < str.length; i++) {
    let char = str[i].toLowerCase();
    if (vowelsCounter.hasOwnProperty(char)) {
      vowelsCounter[char] += 1;
    }
  }

  return vowelsCounter;
}
```

why we are not using Map

```
function countVowels(str) {
  let vowelsCounter = new Map([
    ["a", 0],
    ["e", 0],
    ["i", 0],
    ["o", 0],
    ["u", 0],
  ]);

  for (let i = 0; i < str.length; i++) {
    let char = str[i].toLowerCase();
    if (vowelsCounter.has(char)) {
      vowelsCounter.set(char, vowelsCounter.get(char) + 1);
```

```
    }
  }

  return vowelsCounter;
}
```

clear comparison between `Map` and plain JavaScript `Object` when it comes to lookup speed and performance
In most real-world cases, `Map` is slightly faster and more reliable for key lookups than plain objects, especially when you have a lot of data or need guaranteed performance.

Why `Map` Can Be Faster:

Feature	`Map`	`Object`
Optimized for key-value ops	Yes	Not specifically
Key types	Any (including objects)	Only strings or symbols
Key order	Maintained in insertion order	Not guaranteed (mostly stable)
Lookup time	Near constant (`O(1)`)	Also `O(1)` , but can degrade
Performance for large data	Better consistently	May degrade with prototype chain

- For small data sizes (like counting vowels), the performance difference is negligible.
  For large key sets (e.g., caching, user lookups, large maps), `Map` tends to outperform objects because it's optimized for frequent additions and lookups.

11. Sum of Array Elements

```
function sum(arr) {
  let sumOfAllElements = arr.reduce((acc, item) => acc + item, 0);
  return sumOfAllElements;
}
//Or even shorter:
const sum = (arr) => arr.reduce((acc, item) => acc + item, 0);
```

12.Return Even Numbers from Array

```
function getEvenNumbers(arr) {
  return arr.filter(num => num % 2 === 0);
};
```

```
function getEvenNumbers(arr) {
  let evenNumber = [];
  for (let element of arr) {
    if (element % 2 === 0) {
      evenNumber.push(element);
    }
  }
  return evenNumber;
};
// Example
const numbers = [1, 2, 3, 4, 5, 6];
const evenNumbers = getEvenNumbers(numbers);
console.log(evenNumbers); // Output: [2, 4, 6]
```

Roman to Integer

Brute Force Solution (O(n))

We can iterate through the string, check each numeral and decide whether we need to subtract (when the current numeral is smaller than the next one).

```
function romanToInt(s) {
  const romanMap = {
    'I': 1,
    'V': 5,
    'X': 10,
    'L': 50,
    'C': 100,
    'D': 500,
    'M': 1000
  };
  let total = 0;
  for (let i = 0; i < s.length; i++) {
    const currentVal = romanMap[s[i]];
    const nextVal = romanMap[s[i + 1]];
    // If current value is smaller than the next, subtract it
    if (currentVal < nextVal) {
      total -= currentVal;
    } else {
      total += currentVal;
    }
  }
  return total;
}
// Example:
console.log(romanToInt("III"));   // 3
console.log(romanToInt("IV"));    // 4
console.log(romanToInt("IX"));    // 9
```

```
console.log(romanToInt("LVIII")); // 58
console.log(romanToInt("MCMXCIV")); // 1994
```

- The `romanMap` object is a dictionary mapping Roman numerals to their integer values
- We iterate over the string, checking the current and next Roman numeral.
- If the current numeral is less than the next, we subtract it (e.g., `IV` → 4). Otherwise, we add it to the total.

Alternative Optimized Approach Using a Loop (O(n))

```
function romanToInt(s) {
  const romanMap = { 'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500,
'M': 1000 };
  let total = 0;
  for (let i = 0; i < s.length; i++) {
    if (romanMap[s[i]] < romanMap[s[i + 1]]) {
      total -= romanMap[s[i]];
    } else {
      total += romanMap[s[i]];
    }
  }

  return total;
}

// Example
console.log(romanToInt("IX"));  // 9
console.log(romanToInt("LVIII")); // 58
```

Arrays

Find Max and Min

Brute Force Approach (O(n))

```
function findMinMaxBruteForce(arr) {
  if (arr.length === 0) return null;
  let max = arr[0];
  let min = arr[0];
  for (let i = 1; i < arr.length; i++) {
    if (arr[i] > max) max = arr[i];
    if (arr[i] < min) min = arr[i];
  }
  return { min, max };
}
```

```
// Example
console.log(findMinMaxBruteForce([3, 5, 1, 9, -2])); // { min: -2, max: 9
}
```

Most Efficient Approach Using Math (Built-in, also O(n) but cleaner)

```
function findMinMaxEfficient(arr) {
  if (arr.length === 0) return null;
  return {
    min: Math.min( ... arr),
    max: Math.max( ... arr)
  };
}
// Example
console.log(findMinMaxEfficient([3, 5, 1, 9, -2])); // { min: -2, max: 9 }
```

> ⚠ **Warning**
>
> ⚠️ Note: Math.min(...arr) spreads the array into individual values. Avoid for very large arrays (e.g., >100,000 elements) due to call stack limits.

Move Zeros to End

Efficient Two-Pointer Approach (In-place, O(n))

```
function moveZerosToEnd(arr) {
  let insertPos = 0;
  // Move non-zero elements forward
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] !== 0) {
      arr[insertPos] = arr[i];
      insertPos++;
    }
  }
  // Fill the rest with zeros
  while (insertPos < arr.length) {
    arr[insertPos] = 0;
    insertPos++;
  }
  return arr;
}
// Example
console.log(moveZerosToEnd([0, 1, 0, 3, 12])); // [1, 3, 12, 0, 0]
```

Brute Force (Using Extra Space, O(n) Time, O(n) Space)

```
function moveZerosToEndBrute(arr) {
  let result = [];
  // First push non-zero
  for (let num of arr) {
    if (num !== 0) result.push(num);
  }
  // Then push zeros
  while (result.length < arr.length) {
    result.push(0);
  }
  return result;
}
```

Find Missing Number

Efficient Math Formula Approach (O(n) Time, O(1) Space)

```
function findMissingNumber(arr) {
  const n = arr.length;
  const expectedSum = (n * (n + 1)) / 2;
  const actualSum = arr.reduce((sum, num) => sum + num, 0);
  return expectedSum - actualSum;
}
```

Brute Force (Using Set)

```
function findMissingNumberBrute(arr) {
  const n = arr.length;
  const numSet = new Set(arr);

  for (let i = 0; i <= n; i++) {
    if (!numSet.has(i)) return i;
  }
}
```

Merge Two Sorted Arrays

Efficient Two-Pointer Approach (O(n + m))

```
function mergeSortedArrays(arr1, arr2) {
  const merged = [];
  let i = 0, j = 0;
  // Compare elements from both arrays
```

```
  while (i < arr1.length && j < arr2.length) {
    if (arr1[i] < arr2[j]) {
      merged.push(arr1[i]);
      i++;
    } else {
      merged.push(arr2[j]);
      j++;
    }
  }
  // Add any remaining elements
  while (i < arr1.length) {
    merged.push(arr1[i]);
    i++;
  }
  while (j < arr2.length) {
    merged.push(arr2[j]);
    j++;
  }
  return merged;
}
// Example
console.log(mergeSortedArrays([1, 3, 5], [2, 4, 6])); // [1, 2, 3, 4, 5,
6]
```

Brute Force (Concatenate and Sort)

```
function mergeSortedArraysBrute(arr1, arr2) {
  return [...arr1, ...arr2].sort((a, b) => a - b);
}
```

> ⚠ **Warning**
>
> ⚠️ Not efficient for large datasets (sorting costs O((n+m) log(n+m))), but it's quick to write.

Count Pairs with Given Sum

Brute Force ($O(n^2)$)

```
function countPairsBrute(arr, target) {
  let count = 0;
  for (let i = 0; i < arr.length; i++) {
    for (let j = i + 1; j < arr.length; j++) {
      if (arr[i] + arr[j] === target) {
        count++;
```

```
      }
    }
  }
  return count;
}
```

Efficient Hash Map Approach (O(n))

```
function countPairsEfficient(arr, target) {
  const map = new Map();
  let count = 0;

  for (let num of arr) {
    const complement = target - num;
    if (map.has(complement)) {
      count += map.get(complement);
    }

    // Store/update current number in map
    map.set(num, (map.get(num) || 0) + 1);
  }

  return count;
}
```

Rotate Array

To rotate an array in JavaScript, you can rotate either:

- Right (clockwise): Last elements move to the front
- Left (counter-clockwise): First elements move to the back

Right Rotate by k (Efficient, O(n) Time, O(1) Space)

```
function rotateRight(arr, k) {
  const n = arr.length;
  k = k % n; // Handle k > n
  if (k === 0) return arr;

  // Reverse helpers
  const reverse = (start, end) => {
    while (start < end) {
      [arr[start], arr[end]] = [arr[end], arr[start]];
      start++;
      end--;
    }
  };
  reverse(0, n - 1);
```

```
  reverse(0, k - 1);
  reverse(k, n - 1);
  return arr;
}

// Example
console.log(rotateRight([1, 2, 3, 4, 5], 2)); // [4, 5, 1, 2, 3]
```

Left Rotate by k (Same logic)

```
function rotateLeft(arr, k) {
  const n = arr.length;
  k = k % n;
  if (k === 0) return arr;
  const reverse = (start, end) => {
    while (start < end) {
      [arr[start], arr[end]] = [arr[end], arr[start]];
      start++;
      end--;
    }
  };
  reverse(0, k - 1);
  reverse(k, n - 1);
  reverse(0, n - 1);
  return arr;
}
// Example
console.log(rotateLeft([1, 2, 3, 4, 5], 2)); // [3, 4, 5, 1, 2]
```

Brute Force Using Extra Space

```
function rotateRightBrute(arr, k) {
  const n = arr.length;
  k = k % n;
  return [...arr.slice(-k), ...arr.slice(0, n - k)];
}
```

Union of Two Arrays

Using Set (Best for unsorted arrays)

```
function unionArrays(arr1, arr2) {
  return [...new Set([...arr1, ...arr2])];
}
```

Brute Force with Loop and includes()

```
function unionBrute(arr1, arr2) {
  const result = [...arr1];
  for (let num of arr2) {
    if (!result.includes(num)) {
      result.push(num);
    }
  }
  return result;
}
```

Two-Pointer Method (Only for sorted arrays)

```
function unionSortedArrays(arr1, arr2) {
  let i = 0, j = 0;
  const result = [];
  while (i < arr1.length && j < arr2.length) {
    if (arr1[i] < arr2[j]) {
      if (result[result.length - 1] !== arr1[i]) result.push(arr1[i]);
      i++;
    } else if (arr1[i] > arr2[j]) {
      if (result[result.length - 1] !== arr2[j]) result.push(arr2[j]);
      j++;
    } else {
      if (result[result.length - 1] !== arr1[i]) result.push(arr1[i]);
      i++;
      j++;
    }
  }
  while (i < arr1.length) {
    if (result[result.length - 1] !== arr1[i]) result.push(arr1[i]);
    i++;
  }
  while (j < arr2.length) {
    if (result[result.length - 1] !== arr2[j]) result.push(arr2[j]);
    j++;
  }
  return result;
}

// Example
console.log(unionSortedArrays([1, 2, 3], [2, 3, 4])); // [1, 2, 3, 4]
```

Sliding Window

Sliding Window (O(n) time, O(1) space)

Given an array of positive integers `nums` and a positive integer `target`, return the minimal length of a contiguous subarray of which the `sum ≥ target`. If no such `subarray` exists, return 0.
The `subarray` [4,3] has the smallest length that sums to at least 7.

Sliding Window Intuition:

We use two pointers:

- `left` → start of the window
- `right` → end of the window

We expand the right pointer to increase the sum,
and shrink the left pointer to minimize the window as soon as the condition `(sum ≥ target)` is met.

Step-by-Step (for input `target = 7`, `nums = [2,3,1,2,4,3]`):

1. `left = 0`, `right = 0`, `sum = 0`, `minLength = Infinity`
2. Move `right` → `sum = 2` → not enough
3. Move `right` → `sum = 5` (2+3) → not enough
4. Move `right` → `sum = 6` (2+3+1) → not enough
5. Move `right` → `sum = 8` (2+3+1+2) ≥ 7 → window size = 4
   → Try to shrink window by moving `left` → `sum = 6`, stop
6. Continue expanding → `right = 4`, `sum = 6+4 = 10`
   → Shrink: remove `nums[1]=3` → `sum = 7` → window = 3
   → Shrink more: remove `nums[2]=1` → `sum = 6` ✗ → stop
7. Move `right = 5`, `sum = 6+3 = 9`
   → Shrink: remove `nums[3]=2` → `sum = 7` → window = 2
   → Shrink: remove `nums[4]=4` → `sum = 3` → stop

Minimum length found = 2

```
function minSubArrayLen(target, nums) {
  let left = 0, sum = 0;
  let minLength = Infinity;
  for (let right = 0; right < nums.length; right++) {
    sum += nums[right];
    while (sum >= target) {
      minLength = Math.min(minLength, right - left + 1);
      sum -= nums[left];
      left++;
    }
  }
  return minLength === Infinity ? 0 : minLength;
```

```
}
```

Count Occurrences of Anagrams

Given a text string `txt` and a pattern string `pat`, return the count of all anagram occurrences of `pat` in `txt`.

```
Input: txt = "cbaebabacd", pat = "abc"
Output: 2
Explanation: Anagrams of "abc" are "cba" and "bac"
```

code:

```
function countAnagrams(txt, pat) {
  const result = [];
  const freqMap = new Map();

  for (const char of pat) {
    freqMap.set(char, (freqMap.get(char) || 0) + 1);
  }

  let count = freqMap.size; // number of unique chars needed
  let k = pat.length;
  let left = 0, right = 0;
  let anagramCount = 0;
  while (right < txt.length) {
    let endChar = txt[right];
    if (freqMap.has(endChar)) {
      freqMap.set(endChar, freqMap.get(endChar) - 1);
      if (freqMap.get(endChar) === 0) count--;
    }
    if (right - left + 1 < k) {
      right++;
    } else if (right - left + 1 === k) {
      if (count === 0) anagramCount++;

      let startChar = txt[left];
      if (freqMap.has(startChar)) {
        if (freqMap.get(startChar) === 0) count++;
        freqMap.set(startChar, freqMap.get(startChar) + 1);
      }
      left++;
      right++;
    }
  }
  return anagramCount;
```

```
}
```

Step 1: Frequency Map of `pat`

```
for (const char of pat) {
  freqMap.set(char, (freqMap.get(char) || 0) + 1);
}
 //For `pat = "abc"`, this gives:
 freqMap = { a: 1, b: 1, c: 1 }
```

Step 2: Initialize pointers and counters

```
let count = freqMap.size; // count = 3, means 3 unique chars needed
let k = pat.length; // k = 3
let left = 0, right = 0;
let anagramCount = 0;
```

Main Loop Begins

We'll slide a window of size `k = 3` across `txt`.

Dry Run on `txt = "cbaebabacd"`

Initial:

- `left = 0`, `right = 0`, `sum = 0`, `count = 3`, `anagramCount = 0`
- `window = txt[left...right]`

Window: `"c"`

- `endChar = c` → exists in `freqMap`
- Reduce its freq → `c: 0` → `count--` → `count = 2`
- Window size < k → just `right++`

Window: "cb"

- endChar = b → exists → b: 0 → count = 1
- Window size < k → right++

Window: `"cba"`

- `endChar = a` → exists → `a: 0` → `count = 0`
- Window size == k → check:
  - `count == 0` → anagram → `anagramCount = 1`
- Now slide window
  - `startChar = c`

    - Put `c` back: `c: 1`, `count++ = 1`
- `left++`, `right++`

Window: `"bae"`

- `endChar = e` → not in map → nothing changes
- Size == k → `count != 0` → ×
- Slide:
    - `startChar = b` → `b: 1`, `count++ = 2`
- `left++`, `right++`

Window: `"aeb"`

- `endChar = b` → `b: 0` → `count-- = 1`
- Still not count == 0 → ×
- Slide:
    - `startChar = a` → `a: 1`, `count++ = 2`
- `left++`, `right++`

Window: `"eba"`

- `endChar = a` → `a: 0` → `count-- = 1`
- Not valid yet → ×
- Slide:
    - `startChar = e` → not in map → nothing changes
- `left++`, `right++`

Window: `"bab"`

- `endChar = b` → `b: -1` → freq below 0 → still `count = 1`
- Slide:
    - `startChar = b` → `b: 0` → no change in `count`
- `left++`, `right++`

Window: `"aba"`

- `endChar = a` → `a: -1` → still `count = 1`
- Slide:
- `startChar = a` → `a: 0` → no change
- `left++`, `right++`

Window: `"bac"`

- `endChar = c` → `c: 0` → `count-- = 0`
- Size == k, `count == 0` → anagram → `anagramCount = 2`

- Slide:
  - `startChar = b` → `b: 1`, `count++ = 1`
- `left++`, `right++`

Window: `"acd"`

- `endChar = d` → not in map
- Slide:
  - `startChar = a` → `a: 1`, `count++ = 2`
- `left++`, `right++`

Permutation in String

The "Permutation in String" problem is very similar to the "Count Occurrences of Anagrams" problem you just asked about — except instead of counting, you're just checking if ***any*** permutation exists in the string.

Problem Statement

Given two strings `s1` and `s2`, return `true` if `s2` contains a permutation of `s1`, or `false` otherwise.
A permutation of `s1` is any rearrangement of its characters.
So we're checking: *"Does `s2` have any substring that is a permutation of `s1`?"*

Optimized Sliding Window Solution (with explanation)

```
function checkInclusion(s1, s2) {
  if (s1.length > s2.length) return false;
  const map = new Array(26).fill(0);
  const aCode = 'a'.charCodeAt(0);
  // Build frequency map of s1
  for (let char of s1) {
    map[char.charCodeAt(0) - aCode]++;
  }
  let left = 0;
  let right = 0;
  let count = s1.length;
  while (right < s2.length) {
    let rIndex = s2.charCodeAt(right) - aCode;
    // If char exists in s1, reduce count
    if (map[rIndex] > 0) {
      count--;
    }
    // Always reduce frequency from the map
    map[rIndex]--;
    right++;
    // If count reaches 0, we found a permutation
    if (count === 0) return true;
```

```
    // If window size is bigger than s1, slide left
    if (right - left === s1.length) {
      let lIndex = s2.charCodeAt(left) - aCode;
      // Restore frequency when moving left pointer
      if (map[lIndex] >= 0) {
        count++;
      }
      map[lIndex]++;
      left++;
    }
  }
  return false;
}
```

Longest Repeating Character Replacement

Problem:

You are given a string `s` and an integer `k`. You can perform at most `k` character replacements. You want to find the length of the longest substring that can be achieved with at most `k` replacements, such that all characters in the substring are the same.

Approach:

We can use the sliding window technique to solve this problem efficiently.

Steps:

1. Use two pointers (`left` and `right`) to represent the window of characters.
2. Expand the window by moving the `right` pointer and count the frequency of each character in the window.
3. If the number of characters that are not the most frequent character exceeds `k`, shrink the window by moving the `left` pointer.
4. Keep track of the maximum window size during this process.

```
function characterReplacement(s, k) {
  const count = new Array(26).fill(0);
  let left = 0;
  let maxCount = 0;
  let maxLength = 0;
  for (let right = 0; right < s.length; right++) {
    const rightCharIndex = s[right].charCodeAt(0) - 'A'.charCodeAt(0);
    count[rightCharIndex]++;
    maxCount = Math.max(maxCount, count[rightCharIndex]);
    // Check if we can still make replacements
    if (right - left + 1 - maxCount > k) {
```

```
      const leftCharIndex = s[left].charCodeAt(0) - 'A'.charCodeAt(0);
      count[leftCharIndex]--;
      left++;
    }
    maxLength = Math.max(maxLength, right - left + 1);
  }

  return maxLength;
}
```

Substrings of Size K with K Distinct Chars

Problem:

You are given a string `s` and an integer `k`. You need to return the number of substrings of size `k` that contain exactly `k` distinct characters.

Approach:

We can use a sliding window technique, maintaining a frequency map of characters within the window of size `k`.

Code:

```
function numKDistinct(s, k) {
  let left = 0, right = 0, result = 0;
  const freqMap = new Map();
  while (right < s.length) {
    const rightChar = s[right];
    freqMap.set(rightChar, (freqMap.get(rightChar) || 0) + 1);
    // Shrink the window if it has more than k distinct characters
    while (freqMap.size > k) {
      const leftChar = s[left];
      freqMap.set(leftChar, freqMap.get(leftChar) - 1);
      if (freqMap.get(leftChar) === 0) {
        freqMap.delete(leftChar);
      }
      left++;
    }
    // If window size equals k and we have exactly k distinct chars
    if (right - left + 1 === k && freqMap.size === k) {
      result++;
    }

    right++;
  }

  return result;
}
```

Binary Subarrays with Sum

Problem:

You are given a binary array `nums` and an integer `goal`. You need to return the number of subarrays that have a sum equal to `goal`.

Approach:

We can use a prefix sum with a hash map to track the number of subarrays that sum up to the target.

```
function numSubarraysWithSum(nums, goal) {
  const prefixSumMap = new Map();
  prefixSumMap.set(0, 1); // To account for subarrays starting from index
0
  let count = 0, prefixSum = 0;

  for (const num of nums) {
    prefixSum += num;

    // Check if there exists a subarray with sum = goal
    if (prefixSumMap.has(prefixSum - goal)) {
      count += prefixSumMap.get(prefixSum - goal);
    }

    // Update the map with the current prefix sum
    prefixSumMap.set(prefixSum, (prefixSumMap.get(prefixSum) || 0) + 1);
  }

  return count;
}
```

Number of Substrings Containing All Three Characters

Problem:

You are given a string `s` containing characters `'a'`, `'b'`, and `'c'`. You need to return the number of substrings that contain all three characters at least once.

Approach:

We can use a sliding window to find substrings that contain all three characters.

```
function numberOfSubstrings(s) {
  let left = 0, count = 0;
  const freqMap = new Map();
```

```
  for (let right = 0; right < s.length; right++) {
    freqMap.set(s[right], (freqMap.get(s[right]) || 0) + 1);

    // If all characters a, b, and c are in the window
    while (freqMap.size === 3) {
      count += s.length - right;
      freqMap.set(s[left], freqMap.get(s[left]) - 1);
      if (freqMap.get(s[left]) === 0) {
        freqMap.delete(s[left]);
      }
      left++;
    }
  }

  return count;
}
```

Two Pointers

Pair with Target Sum

Problem:

Given a sorted array `arr` and a target sum `target`, find if there exists a pair of elements whose sum equals `target`. Return `true` if such a pair exists, otherwise `false`.

Approach:

We can use the two-pointer technique to solve this. Since the array is sorted:

- One pointer (`left`) starts at the beginning of the array.
- The other pointer (`right`) starts at the end.
- We check if the sum of the elements at both pointers equals the target:
  - If the sum is equal to the target, return `true`.
  - If the sum is less than the target, move the left pointer to the right.
  - If the sum is greater than the target, move the right pointer to the left.

```
function hasPairWithTargetSum(arr, target) {
  let left = 0;
  let right = arr.length - 1;

  while (left < right) {
    const sum = arr[left] + arr[right];
    if (sum === target) {
      return true;
    } else if (sum < target) {
```

```
      left++;
    } else {
      right--;
    }
  }

  return false;
}
```

Remove Duplicates from Sorted Array

Problem:

Given a sorted array `arr`, remove the duplicates in-place such that each element appears only once and return the new length of the array.

Approach:

Since the array is sorted, we can use two pointers:

- `i` will point to the current unique element position.
- `j` will iterate through the array to check for duplicates.
- If `arr[i] !== arr[j]`, we move `i` to the next unique position and copy `arr[j]` to `arr[i]`.

```
function removeDuplicates(arr) {
  if (arr.length === 0) return 0;
  let i = 0;
  for (let j = 1; j < arr.length; j++) {
    if (arr[i] !== arr[j]) {
      i++;
      arr[i] = arr[j];
    }
  }
  return i + 1;
}
```

Merge Sorted Arrays

Problem:

Given two sorted arrays `arr1` and `arr2`, merge them into a single sorted array.

Approach:

We can use two pointers, `i` and `j`, to iterate through `arr1` and `arr2`:

- Compare the current elements of both arrays.

- Add the smaller element to the merged array and move the corresponding pointer.
- If one array is exhausted, append the remaining elements from the other array.

```
function mergeSortedArrays(arr1, arr2) {
  let i = 0, j = 0;
  const merged = [];
  while (i < arr1.length && j < arr2.length) {
    if (arr1[i] < arr2[j]) {
      merged.push(arr1[i]);
      i++;
    } else {
      merged.push(arr2[j]);
      j++;
    }
  }
  // Append the remaining elements
  while (i < arr1.length) {
    merged.push(arr1[i]);
    i++;
  }
  while (j < arr2.length) {
    merged.push(arr2[j]);
    j++;
  }

  return merged;
}
```

Container With Most Water

Problem:

Given an array of integers `height`, where each integer represents the height of a vertical line drawn at that index. You need to find two lines that together with the x-axis form a container such that the container contains the most water. Return the maximum amount of water the container can store.

Approach:

We can use the two-pointer technique:

- Place one pointer at the beginning (`left`) and one at the end (`right`) of the array.
- Calculate the area formed between these two lines, and move the pointer pointing to the shorter line inward to potentially find a larger area.
- Continue this process until the two pointers meet.

```
function maxArea(height) {
  let left = 0, right = height.length - 1;
  let maxArea = 0;
  while (left < right) {
    const width = right - left;
    const minHeight = Math.min(height[left], height[right]);
    const area = width * minHeight;
    maxArea = Math.max(maxArea, area);
    if (height[left] < height[right]) {
      left++;
    } else {
      right--;
    }
  }
  return maxArea;
}
```

3Sum Problem

Problem:

Given an array of integers `nums`, find all unique triplets in the array that sum to zero. Each triplet should be unique

Approach:

We can use the two-pointer technique:

1. First, sort the array.
2. Fix one element (`nums[i]`), and use two pointers (`left` and `right`) to find pairs that sum to `-nums[i]`.
3. If a valid triplet is found, add it to the result.
4. Skip duplicate values to ensure uniqueness.

```
function threeSum(nums) {
  const result = [];
  nums.sort((a, b) => a - b);

  for (let i = 0; i < nums.length - 2; i++) {
    if (i > 0 && nums[i] === nums[i - 1]) continue;  // Skip duplicates
    let left = i + 1;
    let right = nums.length - 1;

    while (left < right) {
      const sum = nums[i] + nums[left] + nums[right];
      if (sum === 0) {
        result.push([nums[i], nums[left], nums[right]]);
```

```
        // Skip duplicates
        while (nums[left] === nums[left + 1]) left++;
        while (nums[right] === nums[right - 1]) right--;
        left++;
        right--;
      } else if (sum < 0) {
        left++;
      } else {
        right--;
      }
    }
  }

  return result;
}
```

Trapping Rain Water

Problem:

Given an array `height[]` representing elevation heights, compute how much water can be trapped after raining.

Approach:

Use two pointers, starting from both ends:

- Maintain `leftMax` and `rightMax` to track the highest walls from both ends.
- Water trapped at a position depends on the minimum of leftMax and rightMax, minus current height.

```
function trap(height) {
  let left = 0, right = height.length - 1;
  let leftMax = 0, rightMax = 0;
  let water = 0;
  while (left < right) {
    if (height[left] <= height[right]) {
      if (height[left] >= leftMax) {
        leftMax = height[left];
      } else {
        water += leftMax - height[left];
      }
      left++;
    } else {
      if (height[right] >= rightMax) {
        rightMax = height[right];
      } else {
        water += rightMax - height[right];
```

```
      }
      right--;
    }
  }
  return water;
}
```

Is Subsequence

Problem:

Check if string `s` is a subsequence of string `t`.

Approach:

Use two pointers:

- One for `s`, one for `t`.
- Move both when characters match, else move only `t`.
- If `s` is fully traversed, it's a `subsequence`.

```
function isSubsequence(s, t) {
  let i = 0, j = 0;

  while (i < s.length && j < t.length) {
    if (s[i] === t[j]) i++;
    j++;
  }

  return i === s.length;
}
```

Reverse Vowels of a String

Problem:

Reverse only the vowels in the string.

Approach:

- Use two pointers: one from start, one from end.
- Swap vowels when both pointers point to vowels.
- Move pointers inward.

```
function reverseVowels(s) {
  const vowels = new Set('aeiouAEIOU');
  let arr = s.split('');
  let left = 0, right = s.length - 1;
```

```
  while (left < right) {
    while (left < right && !vowels.has(arr[left])) left++;
    while (left < right && !vowels.has(arr[right])) right--;
    [arr[left], arr[right]] = [arr[right], arr[left]];
    left++;
    right--;
  }

  return arr.join('');
}
```

Backspace String Compare

Problem:

Compare two strings `s` and `t` after simulating backspaces (`#` means delete last char).

Approach:

- Use two pointers from end of both strings.
- Skip backspaces while moving backward and compare characters.

```
function backspaceCompare(s, t) {
  function nextValidCharIndex(str, index) {
    let skip = 0;
    while (index >= 0) {
      if (str[index] === '#') {
        skip++;
      } else if (skip > 0) {
        skip--;
      } else {
        break;
      }
      index--;
    }
    return index;
  }

  let i = s.length - 1, j = t.length - 1;
  while (i >= 0 || j >= 0) {
    i = nextValidCharIndex(s, i);
    j = nextValidCharIndex(t, j);

    if (i >= 0 && j >= 0 && s[i] !== t[j]) return false;
    if ((i >= 0) !== (j >= 0)) return false;

    i--; j--;
```

```
  }

  return true;
}
```

Valid Palindrome

Problem:

Check if a string is a valid palindrome (ignore non-alphanumeric characters and case).

Approach:

- Use two pointers from both ends.
- Skip non-alphanumeric chars.
- Compare lowercase versions.

```
function isPalindrome(s) {
  let left = 0, right = s.length - 1;

  while (left < right) {
    while (left < right && !isAlphaNum(s[left])) left++;
    while (left < right && !isAlphaNum(s[right])) right--;

    if (s[left].toLowerCase() !== s[right].toLowerCase()) return false;

    left++;
    right--;
  }

  return true;
}

function isAlphaNum(c) {
  return /^[a-z0-9]$/i.test(c);
}
```

Stack & Queue

Valid Parentheses

Given a string `s` containing just the characters `'(', ')', '{', '}', '[' and ']'`, determine if the input string is valid.

Rules:

- Open brackets must be closed by the same type.

- Open brackets must be closed in the correct order.
- Every close must match the most recent unmatched open.

Brute Force Approach

This approach keeps removing valid pairs until nothing is left or no more can be removed.
Idea:

- Repeatedly replace `()`, `{}`, and `[]` with an empty string.
- If the final string is empty, it's valid.

```
function isValidBrute(s) {
  let prev;
  do {
    prev = s;
    s = s.replace("()", "").replace("{}", "").replace("[]", "");
  } while (s !== prev);

  return s.length === 0;
}
```

Efficient Approach (Using Stack)

Idea:

- Use a stack to keep track of open brackets.
- Push open brackets.
- For each closing bracket, check the top of the stack:
  - If it matches the corresponding open bracket, pop it.
  - If not, return false.
- At the end, stack should be empty.

```
function isValid(s) {
  const stack = [];
  const map = {
    ')': '(',
    '}': '{',
    ']': '['
  };

  for (let char of s) {
    if (char === '(' || char === '{' || char === '[') {
      stack.push(char);
    } else {
      if (stack.length === 0 || stack.pop() !== map[char]) {
```

```
        return false;
      }
    }
  }

  return stack.length === 0;
}
```

Min Stack

Design a stack that supports these operations in O(1) time:

- `push(x)`
- `pop()`
- `top()`
- `getMin()` — retrieves the minimum element in the stack

Brute Force Approach (Inefficient)

Idea:

- Use a regular array as a stack.
- For `getMin()`, loop through the array each time to find the minimum.

```
class MinStackBrute {
  constructor() {
    this.stack = [];
  }
  push(x) {
    this.stack.push(x);
  }
  pop() {
    this.stack.pop();
  }
  top() {
    return this.stack[this.stack.length - 1];
  }
  getMin() {
    let min = Infinity;
    for (let num of this.stack) {
      if (num < min) min = num;
    }
    return min;
  }
}
```

Efficient Approach (Optimal — Two Stacks)

Idea:

- Use two stacks:
  - `stack`: stores all values
  - `minStack`: keeps track of the minimum value so far
    Trick:
- On `push`, also push the new min to `minStack`
- On `pop`, pop both `stack` and `minStack`

```
class MinStack {
  constructor() {
    this.stack = [];
    this.minStack = [];
  }

  push(x) {
    this.stack.push(x);
    const min = this.minStack.length === 0
      ? x
      : Math.min(x, this.minStack[this.minStack.length - 1]);
    this.minStack.push(min);
  }
  pop() {
    this.stack.pop();
    this.minStack.pop();
  }

  top() {
    return this.stack[this.stack.length - 1];
  }

  getMin() {
    return this.minStack[this.minStack.length - 1];
  }
}
```

Implement Queue using Stacks

Brute Force (Inefficient)

- Use a single stack.
- On `pop()` or `peek()`, reverse the stack into a temp array, access the front, and reverse it back.

```
class MyQueueBrute {
  constructor() {
    this.stack = [];
```

```
  }
  push(x) {
    this.stack.push(x); // O(1)
  }

  pop() {
    const temp = [];
    while (this.stack.length > 1) {
      temp.push(this.stack.pop());
    }
    const front = this.stack.pop(); // front of queue
    while (temp.length) {
      this.stack.push(temp.pop()); // restore
    }
    return front;
  }

  peek() {
    const temp = [];
    while (this.stack.length > 1) {
      temp.push(this.stack.pop());
    }
    const front = this.stack[this.stack.length - 1];
    while (temp.length) {
      this.stack.push(temp.pop());
    }
    return front;
  }

  empty() {
    return this.stack.length === 0;
  }
}
```

Optimal Approach (Using Two Stacks)

Use two stacks:

- `inStack`: for **push**
- `outStack`: for **pop**/**peek** — reversed order
  Only move elements when `outStack` is empty!

```
class MyQueue {
  constructor() {
    this.inStack = [];
    this.outStack = [];
  }
```

```
  push(x) {
    this.inStack.push(x);
  }
  pop() {
    if (this.outStack.length === 0) {
      while (this.inStack.length > 0) {
        this.outStack.push(this.inStack.pop());
      }
    }
    return this.outStack.pop();
  }
  peek() {
    if (this.outStack.length === 0) {
      while (this.inStack.length > 0) {
        this.outStack.push(this.inStack.pop());
      }
    }
    return this.outStack[this.outStack.length - 1];
  }

  empty() {
    return this.inStack.length === 0 && this.outStack.length === 0;
  }
}
```

Daily Temperatures

Given: an array `temperatures[]` where `temperatures[i]` is the temperature on day `i`.
Return: an array `answer[]` such that `answer[i]` is the number of days until a warmer temperature. If no such day exists, put `0`.

Brute Force Approach

```
function dailyTemperaturesBrute(temperatures) {
  const result = new Array(temperatures.length).fill(0);

  for (let i = 0; i < temperatures.length; i++) {
    for (let j = i + 1; j < temperatures.length; j++) {
      if (temperatures[j] > temperatures[i]) {
        result[i] = j - i;
        break;
      }
    }
  }

  return result;
```

```
}
```

Efficient Approach (Monotonic Stack)

Use a **monotonic decreasing stack** (stores indices).
Process from **left to right**:

- At each step, check if the current temperature is **greater** than the one at the **top of the stack**.
- If it is, that means we've found a warmer day → **calculate the difference**.
- Push current index on stack.

```
function dailyTemperatures(temperatures) {
  const result = new Array(temperatures.length).fill(0);
  const stack = []; // holds indices

  for (let i = 0; i < temperatures.length; i++) {
    while (
      stack.length > 0 &&
      temperatures[i] > temperatures[stack[stack.length - 1]]
    ) {
      const prevIndex = stack.pop();
      result[prevIndex] = i - prevIndex;
    }
    stack.push(i);
  }

  return result;
}
```

Next Greater Element I

Given two arrays:

- `nums1`: a subset of `nums2`
- `nums2`: a list of integers

Goal: For each element in `nums1`, find the **next greater element** in `nums2`.
If none exists, return `-1`.

```
function nextGreaterElementBrute(nums1, nums2) {
  const result = [];

  for (let num of nums1) {
    const index = nums2.indexOf(num);
```

```
    let found = -1;

    for (let i = index + 1; i < nums2.length; i++) {
      if (nums2[i] > num) {
        found = nums2[i];
        break;
      }
    }

    result.push(found);
  }

  return result;
}
```

Efficient Approach (Monotonic Stack + Hash Map)

- Use a monotonic decreasing stack to precompute next greater for all items in `nums2`
- Store results in a hash map
- Lookup the answer for each item in `nums1`

```
function nextGreaterElement(nums1, nums2) {
  const stack = [];
  const map = {}; // num -> next greater

  for (let i = nums2.length - 1; i >= 0; i--) {
    const num = nums2[i];

    // Maintain decreasing stack
    while (stack.length > 0 && stack[stack.length - 1] <= num) {
      stack.pop();
    }

    map[num] = stack.length === 0 ? -1 : stack[stack.length - 1];
    stack.push(num);
  }

  return nums1.map(num => map[num]);
}
```

Evaluate Reverse Polish Notation (RPN)

You're given an array of strings, `tokens[]`, representing a Reverse Polish Notation expression.

Return: the final evaluated integer result.

RPN Rules:

- Operators: `+`, `-`, `*`, `/`
- Evaluate from left to right
- Apply operators on the **two most recent operands**

Efficient Stack-Based Approach

```
function evalRPN(tokens) {
  const stack = [];

  for (let token of tokens) {
    if (["+", "-", "*", "/"].includes(token)) {
      const b = stack.pop();
      const a = stack.pop();
      let result;

      switch (token) {
        case "+": result = a + b; break;
        case "-": result = a - b; break;
        case "*": result = a * b; break;
        case "/":
          result = Math.trunc(a / b); // truncate toward zero
          break;
      }

      stack.push(result);
    } else {
      stack.push(Number(token));
    }
  }

  return stack.pop();
}
```

Implement Stack using Queues

Implement a **stack (LIFO)** using one or two **queues (FIFO)**.

Efficient Code (Using 2 Queues):

```
class MyStack {
  constructor() {
    this.q1 = [];
    this.q2 = [];
```

```
  }

  push(x) {
    this.q2.push(x); // step 1
    while (this.q1.length > 0) {
      this.q2.push(this.q1.shift()); // step 2: move old items
    }
    [this.q1, this.q2] = [this.q2, this.q1]; // swap
  }

  pop() {
    return this.q1.shift();
  }

  top() {
    return this.q1[0];
  }

  empty() {
    return this.q1.length === 0;
  }
}
```

Decode String

Decode string like `"3[a2[c]]"` → `"accaccacc"`

```
function decodeString(s) {
  const strStack = [];
  const numStack = [];
  let currentStr = '';
  let num = 0;

  for (let ch of s) {
    if (!isNaN(ch)) {
      num = num * 10 + Number(ch); // build full number
    } else if (ch === '[') {
      numStack.push(num);
      strStack.push(currentStr);
      currentStr = '';
      num = 0;
    } else if (ch === ']') {
      const repeatTimes = numStack.pop();
      const prevStr = strStack.pop();
      currentStr = prevStr + currentStr.repeat(repeatTimes);
    } else {
      currentStr += ch;
    }
```

```
  }

  return currentStr;
}
```

Sliding Window Maximum

Given `nums[]` and a window size `k`, return an array of the **maximum** in every sliding window of size `k`.

```
function maxSlidingWindow(nums, k) {
  const deque = [], result = [];

  for (let i = 0; i < nums.length; i++) {
    // remove indices out of window
    if (deque.length && deque[0] <= i - k) deque.shift();

    // remove smaller values from back
    while (deque.length && nums[i] > nums[deque[deque.length - 1]]) {
      deque.pop();
    }

    deque.push(i);

    if (i >= k - 1) {
      result.push(nums[deque[0]]);
    }
  }

  return result;
}
```

Remove K Digits

Given:
A non-negative integer as a string `num` and an integer `k`.
Goal:
Remove exactly `k` digits from the number so that the resulting number is the **smallest possible**.
Return: result as a string, removing any leading zeros. If the result is empty, return `"0"`.

```
function removeKdigits(num, k) {
  const stack = [];
```

```
  for (let digit of num) {
    while (k > 0 && stack.length > 0 && stack[stack.length - 1] > digit) {
      stack.pop();
      k--;
    }
    stack.push(digit);
  }

  // Still need to remove more? Remove from end
  while (k > 0) {
    stack.pop();
    k--;
  }

  // Convert to string and remove leading zeros
  const result = stack.join('').replace(/^0+/, '');

  return result === '' ? '0' : result;
}
Input: "1432219", k = 3

Stack: []

Digit: '1' → stack: ['1']
Digit: '4' → stack: ['1', '4']
Digit: '3' → pop '4' (k=2) → stack: ['1', '3']
Digit: '2' → pop '3' (k=1) → stack: ['1', '2']
Digit: '2' → stack: ['1', '2', '2']
Digit: '1' → pop '2' (k=0) → stack: ['1', '2', '1']
Digit: '9' → stack: ['1', '2', '1', '9']

Result → "1219"
```

Binary Search

1. Binary Search in Sorted Array

```
function binarySearch(arr, target) {
  let left = 0, right = arr.length - 1;

  while (left <= right) {
    let mid = Math.floor((left + right) / 2);
    if (arr[mid] === target) return mid;
    else if (arr[mid] < target) left = mid + 1;
    else right = mid - 1;
  }

  return -1;
```

```
}
```

2. Search in Rotated Sorted Array

```
function searchRotated(arr, target) {
  let left = 0, right = arr.length - 1;

  while (left <= right) {
    let mid = Math.floor((left + right) / 2);
    if (arr[mid] === target) return mid;

    if (arr[left] <= arr[mid]) {
      if (arr[left] <= target && target < arr[mid]) right = mid - 1;
      else left = mid + 1;
    } else {
      if (arr[mid] < target && target <= arr[right]) left = mid + 1;
      else right = mid - 1;
    }
  }

  return -1;
}
```

3. Find First and Last Position of Element

```
function searchRange(arr, target) {
  const first = findIndex(arr, target, true);
  const last = findIndex(arr, target, false);
  return [first, last];
}

function findIndex(arr, target, isFirst) {
  let left = 0, right = arr.length - 1, result = -1;

  while (left <= right) {
    let mid = Math.floor((left + right) / 2);
    if (arr[mid] === target) {
      result = mid;
      isFirst ? (right = mid - 1) : (left = mid + 1);
    } else if (arr[mid] < target) left = mid + 1;
    else right = mid - 1;
  }

  return result;
```

```
}
```

4. Count Occurrences in Sorted Array

```
function countOccurrences(arr, target) {
  const first = findIndex(arr, target, true);
  if (first === -1) return 0;
  const last = findIndex(arr, target, false);
  return last - first + 1;
}
```

5. Square Root of Number (Floor)

```
function sqrtFloor(x) {
  if (x < 2) return x;
  let left = 1, right = Math.floor(x / 2), ans = 1;

  while (left <= right) {
    let mid = Math.floor((left + right) / 2);
    if (mid * mid <= x) {
      ans = mid;
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }

  return ans;
}
```

6. Peak Element

```
function findPeakElement(nums) {
  let left = 0, right = nums.length - 1;

  while (left < right) {
    let mid = Math.floor((left + right) / 2);
    if (nums[mid] > nums[mid + 1]) right = mid;
    else left = mid + 1;
  }

  return left;
}
```

7. Single Element in Sorted Array

```
function singleNonDuplicate(nums) {
  let left = 0, right = nums.length - 1;

  while (left < right) {
    let mid = Math.floor((left + right) / 2);
    if (mid % 2 === 1) mid--;

    if (nums[mid] === nums[mid + 1]) left = mid + 2;
    else right = mid;
  }

  return nums[left];
}
```

8. Kth Missing Positive Number

```
function findKthMissing(arr, k) {
  let left = 0, right = arr.length - 1;

  while (left <= right) {
    let mid = Math.floor((left + right) / 2);
    const missing = arr[mid] - (mid + 1);

    if (missing < k) left = mid + 1;
    else right = mid - 1;
  }

  return left + k;
}
```

9. Find Position to Insert Element (lower_bound)

```
function searchInsert(arr, target) {
  let left = 0, right = arr.length - 1;

  while (left <= right) {
    let mid = Math.floor((left + right) / 2);
    if (arr[mid] < target) left = mid + 1;
    else right = mid - 1;
  }

  return left;
```

```
}
```

10. Search in 2D Matrix

```
function searchMatrix(matrix, target) {
  if (!matrix.length || !matrix[0].length) return false;
  let rows = matrix.length, cols = matrix[0].length;
  let left = 0, right = rows * cols - 1;

  while (left <= right) {
    let mid = Math.floor((left + right) / 2);
    let midVal = matrix[Math.floor(mid / cols)][mid % cols];

    if (midVal === target) return true;
    else if (midVal < target) left = mid + 1;
    else right = mid - 1;
  }

  return false;
}
```

Summary Table

#	Problem	Time	Space
1	Binary Search	O(log n)	O(1)
2	Rotated Array Search	O(log n)	O(1)
3	First & Last Position	O(log n)	O(1)
4	Count Occurrences	O(log n)	O(1)
5	Sqrt Floor	O(log n)	O(1)
6	Peak Element	O(log n)	O(1)
7	Single Element	O(log n)	O(1)
8	Kth Missing Positive Number	O(log n)	O(1)
9	Insert Position	O(log n)	O(1)
10	Search in 2D Matrix	O(log n)	O(1)

Recursion/Backtracking problems

1. Factorial using Recursion

```
function factorial(n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}
```

2. Fibonacci Series (Recursion)

```
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

3. Power of a Number (x^n)

```
function power(x, n) {
  if (n === 0) return 1;
  if (n < 0) return 1 / power(x, -n);
  return n % 2 === 0 ? power(x * x, n / 2) : x * power(x, n - 1);
}
```

4. Permutations of a String

```
function permute(str) {
  const res = [];
  const chars = str.split('');

  function backtrack(start) {
    if (start === chars.length) {
      res.push(chars.join(''));
      return;
    }
    for (let i = start; i < chars.length; i++) {
      [chars[start], chars[i]] = [chars[i], chars[start]];
      backtrack(start + 1);
      [chars[start], chars[i]] = [chars[i], chars[start]];
    }
  }

  backtrack(0);
  return res;
}
```

5. Subsets of Array

```
function subsets(nums) {
  const res = [];

  function backtrack(index, path) {
    res.push([ ... path]);
    for (let i = index; i < nums.length; i++) {
      path.push(nums[i]);
      backtrack(i + 1, path);
      path.pop();
    }
  }

  backtrack(0, []);
  return res;
}
```

6. Palindrome Partitioning

```
function partition(s) {
  const res = [];

  function isPalindrome(str, l, r) {
    while (l < r) {
      if (str[l++] !== str[r--]) return false;
    }
    return true;
  }

  function backtrack(start, path) {
    if (start === s.length) {
      res.push([ ... path]);
      return;
    }

    for (let end = start; end < s.length; end++) {
      if (isPalindrome(s, start, end)) {
        path.push(s.slice(start, end + 1));
        backtrack(end + 1, path);
        path.pop();
      }
    }
  }

  backtrack(0, []);
  return res;
}
```

7. Combination Sum

```
function combinationSum(candidates, target) {
  const res = [];

  function backtrack(start, path, total) {
    if (total === target) {
      res.push([ ... path]);
      return;
    }
    if (total > target) return;

    for (let i = start; i < candidates.length; i++) {
      path.push(candidates[i]);
      backtrack(i, path, total + candidates[i]);
      path.pop();
    }
  }

  backtrack(0, [], 0);
  return res;
}
```

8. Generate Parentheses

```
function generateParenthesis(n) {
  const res = [];

  function backtrack(open, close, path) {
    if (path.length === n * 2) {
      res.push(path);
      return;
    }

    if (open < n) backtrack(open + 1, close, path + "(");
    if (close < open) backtrack(open, close + 1, path + ")");
  }

  backtrack(0, 0, "");
  return res;
}
```

9. Word Search (Matrix DFS)

```
function exist(board, word) {
  const rows = board.length;
  const cols = board[0].length;

  function dfs(i, j, k) {
    if (k === word.length) return true;
    if (i < 0 || j < 0 || i >= rows || j >= cols || board[i][j] !==
word[k]) return false;

    const temp = board[i][j];
    board[i][j] = '#';

    const found = dfs(i + 1, j, k + 1) ||
                  dfs(i - 1, j, k + 1) ||
                  dfs(i, j + 1, k + 1) ||
                  dfs(i, j - 1, k + 1);

    board[i][j] = temp;
    return found;
  }

  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      if (dfs(i, j, 0)) return true;
    }
  }

  return false;
}
```

10. N-Queens Problem

```
function solveNQueens(n) {
  const board = Array.from({ length: n }, () => Array(n).fill('.'));
  const res = [];

  function isValid(row, col) {
    for (let i = 0; i < row; i++) {
      if (board[i][col] === 'Q') return false;
      if (col - (row - i) >= 0 && board[i][col - (row - i)] === 'Q')
return false;
      if (col + (row - i) < n && board[i][col + (row - i)] === 'Q') return
false;
    }
    return true;
  }
```

```
  function backtrack(row) {
    if (row === n) {
      res.push(board.map(row => row.join('')));
      return;
    }

    for (let col = 0; col < n; col++) {
      if (isValid(row, col)) {
        board[row][col] = 'Q';
        backtrack(row + 1);
        board[row][col] = '.';
      }
    }
  }

  backtrack(0);
  return res;
}
```

Sorting & Searching

Sort Colors (Dutch National Flag Algorithm)

```
function sortColors(nums) {
  let low = 0, mid = 0, high = nums.length - 1;

  while (mid <= high) {
    if (nums[mid] === 0) {
      [nums[low], nums[mid]] = [nums[mid], nums[low]];
      low++;
      mid++;
    } else if (nums[mid] === 1) {
      mid++;
    } else {
      [nums[mid], nums[high]] = [nums[high], nums[mid]];
      high--;
    }
  }
}
```

Find Kth Largest Element in an Array

```
function findKthLargest(nums, k) {
  k = nums.length - k;

  function quickSelect(left, right) {
```

```
    const pivot = nums[right];
    let p = left;

    for (let i = left; i < right; i++) {
      if (nums[i] <= pivot) {
        [nums[i], nums[p]] = [nums[p], nums[i]];
        p++;
      }
    }
    [nums[p], nums[right]] = [nums[right], nums[p]];

    if (p === k) return nums[p];
    else if (p < k) return quickSelect(p + 1, right);
    else return quickSelect(left, p - 1);
  }

  return quickSelect(0, nums.length - 1);
}
```

Top K Frequent Elements

```
function topKFrequent(nums, k) {
  const freqMap = new Map();
  for (const num of nums) {
    freqMap.set(num, (freqMap.get(num) || 0) + 1);
  }

  const buckets = Array(nums.length + 1).fill().map(() => []);
  for (const [num, freq] of freqMap.entries()) {
    buckets[freq].push(num);
  }

  const res = [];
  for (let i = buckets.length - 1; i >= 0 && res.length < k; i--) {
    res.push( ...buckets[i]);
  }

  return res.slice(0, k);
}
```

Sort Characters by Frequency

```
function frequencySort(s) {
  const map = new Map();

  for (const char of s) {
```

```
    map.set(char, (map.get(char) || 0) + 1);
  }

  return [...map.entries()]
    .sort((a, b) => b[1] - a[1])
    .map(([char, freq]) => char.repeat(freq))
    .join('');
}
```

Check if Array is Sorted and Rotated

```
function check(nums) {
  let count = 0;
  const n = nums.length;

  for (let i = 0; i < n; i++) {
    if (nums[i] > nums[(i + 1) % n]) {
      count++;
      if (count > 1) return false;
    }
  }

  return true;
}
```

Hashing problems

Two Sum

```
function twoSum(nums, target) {
  const map = new Map();

  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
    if (map.has(complement)) return [map.get(complement), i];
    map.set(nums[i], i);
  }
}
```

. Intersection of Two Arrays (Unique elements)

```
function intersection(nums1, nums2) {
  const set1 = new Set(nums1);
  const set2 = new Set(nums2);
```

```
  return [...set1].filter(num => set2.has(num));
}
```

. Longest Consecutive Sequence

```
function longestConsecutive(nums) {
  const set = new Set(nums);
  let maxLen = 0;

  for (let num of set) {
    if (!set.has(num - 1)) {
      let curr = num, count = 1;
      while (set.has(curr + 1)) {
        curr++;
        count++;
      }
      maxLen = Math.max(maxLen, count);
    }
  }

  return maxLen;
}
```

Count Distinct Elements

```
function countDistinct(arr) {
  return new Set(arr).size;
}
```

. First Non-Repeating Character

```
function firstUniqChar(s) {
  const map = new Map();

  for (let ch of s) {
    map.set(ch, (map.get(ch) || 0) + 1);
  }

  for (let i = 0; i < s.length; i++) {
    if (map.get(s[i]) === 1) return i;
  }

  return -1;
```

```
}
```

. Group Anagrams

```
function groupAnagrams(strs) {
  const map = new Map();

  for (let word of strs) {
    const key = word.split('').sort().join('');
    if (!map.has(key)) map.set(key, []);
    map.get(key).push(word);
  }

  return [...map.values()];
}
```

. Subarray Sum Equals K

```
function subarraySum(nums, k) {
  const map = new Map();
  map.set(0, 1);

  let sum = 0, count = 0;

  for (let num of nums) {
    sum += num;
    if (map.has(sum - k)) count += map.get(sum - k);
    map.set(sum, (map.get(sum) || 0) + 1);
  }

  return count;
}
```

. Majority Element (> n/2 times)

```
function majorityElement(nums) {
  let count = 0, candidate = null;

  for (let num of nums) {
    if (count === 0) candidate = num;
    count += (num === candidate ? 1 : -1);
  }

  return candidate;
```

```
}
```

. Isomorphic Strings'

```
function isIsomorphic(s, t) {
  if (s.length !== t.length) return false;

  const mapST = new Map();
  const mapTS = new Map();

  for (let i = 0; i < s.length; i++) {
    const a = s[i], b = t[i];
    if ((mapST.has(a) && mapST.get(a) !== b) ||
        (mapTS.has(b) && mapTS.get(b) !== a)) {
      return false;
    }
    mapST.set(a, b);
    mapTS.set(b, a);
  }

  return true;
}
```

. Word Pattern Match

```
function wordPattern(pattern, s) {
  const words = s.split(" ");
  if (pattern.length !== words.length) return false;

  const charMap = new Map();
  const wordMap = new Map();

  for (let i = 0; i < pattern.length; i++) {
    const ch = pattern[i], word = words[i];

    if ((charMap.has(ch) && charMap.get(ch) !== word) ||
        (wordMap.has(word) && wordMap.get(word) !== ch)) {
      return false;
    }

    charMap.set(ch, word);
    wordMap.set(word, ch);
  }

  return true;
```

```
}
```