```
// 🟦 1. Array Utilities
class ArrayUtils {
  // Reverse an array
  static reverseArray(arr) {
    const result = [];
    for (let i = arr.length - 1; i >= 0; i--) {
      result.push(arr[i]);
    }
    return result;
  }

  // Rotate array by k positions
  static rotateArray(arr, k) {
    if (arr.length === 0) return arr;
    k = k % arr.length;
    const result = new Array(arr.length);
    for (let i = 0; i < arr.length; i++) {
      result[(i + k) % arr.length] = arr[i];
    }
    return result;
  }

  // Find max and min
  static findMaxMin(arr) {
    if (arr.length === 0) return [null, 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 [max, min];
  }

  // Remove duplicates
  static removeDuplicates(arr) {
    const seen = new Set();
    const result = [];
    for (const item of arr) {
      if (!seen.has(item)) {
        seen.add(item);
        result.push(item);
      }
    }
    return result;
  }

  // Check if array is sorted
  static isSorted(arr) {
    for (let i = 1; i < arr.length; i++) {
      if (arr[i] < arr[i - 1]) return false;
    }
    return true;
  }
}

// 🟦 2. String Utilities
class StringUtils {
  // Reverse a string
  static reverseString(str) {
    let result = '';
    for (let i = str.length - 1; i >= 0; i--) {
      result += str[i];
    }
    return result;
  }

  // Check palindrome
  static isPalindrome(str) {
    let left = 0;
    let right = str.length - 1;
    while (left < right) {
      if (str[left] !== str[right]) return false;
      left++;
      right--;
    }
    return true;
  }

  // Count character frequency
  static charFrequency(str) {
    const freq = {};
    for (const char of str) {
      freq[char] = (freq[char] || 0) + 1;
    }
    return freq;
  }

  // Check anagram
  static isAnagram(str1, str2) {
    if (str1.length !== str2.length) return false;
    const freq1 = this.charFrequency(str1);
    const freq2 = this.charFrequency(str2);

    for (const char in freq1) {
      if (freq1[char] !== freq2[char]) return false;
    }
    return true;
  }

  // Find first non-repeating character
  static firstNonRepeating(str) {
    const freq = this.charFrequency(str);
    for (const char of str) {
      if (freq[char] === 1) return char;
    }
    return null;
  }
}

// 🟦 3. Searching Algorithms
class SearchAlgorithms {
  // Linear search
  static linearSearch(arr, target) {
    for (let i = 0; i < arr.length; i++) {
      if (arr[i] === target) return i;
    }
    return -1;
  }
  // Binary search
  static binarySearch(arr, target) {
    let left = 0;
    let right = arr.length - 1;

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

  // First occurrence
  static firstOccurrence(arr, target) {
    let left = 0;
    let right = arr.length - 1;
    let result = -1;

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

  // Last occurrence
  static lastOccurrence(arr, target) {
    let left = 0;
    let right = arr.length - 1;
    let result = -1;

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

  // Count occurrences in sorted array
  static countOccurrences(arr, target) {
    const first = this.firstOccurrence(arr, target);
    if (first === -1) return 0;
    const last = this.lastOccurrence(arr, target);
    return last - first + 1;
  }
}

// 🟦 4. Sorting Helpers
class SortingHelpers {
  // Check if array is sorted
  static isSorted(arr) {
    return ArrayUtils.isSorted(arr);
  }

  // Custom sort using key
  static customSort(arr, keyFunc) {
    return arr.slice().sort((a, b) => {
      const keyA = keyFunc(a);
      const keyB = keyFunc(b);
      return keyA < keyB ? -1 : keyA > keyB ? 1 : 0;
    });
  }

  // Sort by frequency
  static sortByFrequency(arr) {
    const freq = {};
    for (const num of arr) {
      freq[num] = (freq[num] || 0) + 1;
    }

    return arr.slice().sort((a, b) => {
      if (freq[a] !== freq[b]) {
        return freq[b] - freq[a];
      }
      return a - b;
    });
  }

  // Sort strings by length
  static sortStringsByLength(strings) {
    return strings.slice().sort((a, b) => a.length - b.length);
  }

  // Kth smallest element
  static kthSmallest(arr, k) {
    if (k <= 0 || k > arr.length) return null;

    // Using selection sort approach
    for (let i = 0; i < k; i++) {
      let minIndex = i;
      for (let j = i + 1; j < arr.length; j++) {
        if (arr[j] < arr[minIndex]) {
          minIndex = j;
        }
      }
      [arr[i], arr[minIndex]] = [arr[minIndex], arr[i]];
    }
    return arr[k - 1];
  }
}
```

```
// 🟦 5. Two Pointers Patterns
class TwoPointers {
  // Pair sum in sorted array
  static pairSumSorted(arr, target) {
    const result = [];
    let left = 0;
    let right = arr.length - 1;

    while (left < right) {
      const sum = arr[left] + arr[right];
      if (sum === target) {
        result.push([arr[left], arr[right]]);
        left++;
        right--;
      } else if (sum < target) {
        left++;
      } else {
        right--;
      }
    }
    return result;
  }

  // Remove duplicates in-place
  static removeDuplicatesInPlace(arr) {
    if (arr.length === 0) return 0;

    let j = 1;
    for (let i = 1; i < arr.length; i++) {
      if (arr[i] !== arr[i - 1]) {
        arr[j] = arr[i];
        j++;
      }
    }
    return arr.slice(0, j);
  }

  // Reverse vowels
  static reverseVowels(str) {
    const vowels = new Set(['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O',
'U']);
    const chars = str.split(");
    let left = 0;
    let right = chars.length - 1;

    while (left < right) {
      while (left < right && !vowels.has(chars[left])) {
        left++;
      }
      while (left < right && !vowels.has(chars[right])) {
        right--;
      }
      [chars[left], chars[right]] = [chars[right], chars[left]];
      left++;
      right--;
    }
    return chars.join(");
  }

  // Merge two sorted arrays
  static mergeSortedArrays(arr1, arr2) {
    const result = [];
    let i = 0, j = 0;

    while (i < arr1.length && j < arr2.length) {
      if (arr1[i] <= arr2[j]) {
        result.push(arr1[i]);
        i++;
      } else {
        result.push(arr2[j]);
        j++;
      }
    }

    while (i < arr1.length) {
      result.push(arr1[i]);
      i++;
    }

    while (j < arr2.length) {
      result.push(arr2[j]);
      j++;
    }

    return result;
  }

  // Check palindrome ignoring non-alphanumeric
  static isPalindromeAlphaNum(str) {
    let left = 0;
    let right = str.length - 1;

    while (left < right) {
      while (left < right && !this.isAlphanumeric(str[left])) {
        left++;
      }
      while (left < right && !this.isAlphanumeric(str[right]))
{
        right--;
      }

      if (str[left].toLowerCase() !==
str[right].toLowerCase()) {
        return false;
      }

      left++;
      right--;
    }
    return true;
  }

  static isAlphanumeric(char) {
    return /^[a-zA-Z0-9]$/.test(char);
  }
}

// 🟦 6. Sliding Window Patterns
class SlidingWindow {
  // Maximum sum subarray of size k
```

```
static maxSumSubarray(arr, k) {
  if (arr.length < k) return 0;

  let windowSum = 0;
  for (let i = 0; i < k; i++) {
    windowSum += arr[i];
  }

  let maxSum = windowSum;
  for (let i = k; i < arr.length; i++) {
    windowSum += arr[i] - arr[i - k];
    maxSum = Math.max(maxSum, windowSum);
  }

  return maxSum;
}

// Longest substring without repeating characters
static longestUniqueSubstring(str) {
  const seen = new Map();
  let maxLen = 0;
  let start = 0;

  for (let end = 0; end < str.length; end++) {
    if (seen.has(str[end]) && seen.get(str[end]) >= start) {
      start = seen.get(str[end]) + 1;
    }
    seen.set(str[end], end);
    maxLen = Math.max(maxLen, end - start + 1);
  }

  return maxLen;
}

// Longest substring with k distinct characters
static longestSubstringKDistinct(str, k) {
  if (k === 0) return 0;

  const freq = new Map();
  let maxLen = 0;
  let left = 0;

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

    while (freq.size > k) {
      freq.set(str[left], freq.get(str[left]) - 1);
      if (freq.get(str[left]) === 0) {
        freq.delete(str[left]);
      }
      left++;
    }

    maxLen = Math.max(maxLen, right - left + 1);
  }

  return maxLen;
}

// Count subarrays with given sum
static countSubarraysWithSum(arr, target) {
  let count = 0;

  for (let start = 0; start < arr.length; start++) {
    let sum = 0;
    for (let end = start; end < arr.length; end++) {
      sum += arr[end];
      if (sum === target) {
        count++;
      }
    }
  }

  return count;
}

// Minimum window substring
static minWindow(s, t) {
  if (s.length === 0 || t.length === 0) return '';

  const targetFreq = {};
  const windowFreq = {};

  for (const char of t) {
    targetFreq[char] = (targetFreq[char] || 0) + 1;
  }

  let required = Object.keys(targetFreq).length;
  let formed = 0;
  let left = 0, right = 0;
  let minLen = Infinity;
  let minStart = 0;

  while (right < s.length) {
    const char = s[right];
    windowFreq[char] = (windowFreq[char] || 0) + 1;

    if (targetFreq[char] && windowFreq[char] ===
targetFreq[char]) {
      formed++;
    }

    while (left <= right && formed === required) {
      if (right - left + 1 < minLen) {
        minLen = right - left + 1;
        minStart = left;
      }

      const leftChar = s[left];
      windowFreq[leftChar]--;
      if (targetFreq[leftChar] && windowFreq[leftChar] <
targetFreq[leftChar]) {
        formed--;
      }
      left++;
    }

    right++;
  }
```

```
    return minLen === Infinity ? '' : s.substring(minStart,
minStart + minLen);
  }
}

// 🟦 7. Hashing / Dictionary Usage
class HashingUtils {
  // Frequency counter
  static frequencyCounter(arr) {
    const freq = {};
    for (const item of arr) {
      freq[item] = (freq[item] || 0) + 1;
    }
    return freq;
  }

  // Two sum
  static twoSum(nums, target) {
    const seen = new Map();

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

    return [];
  }

  // Group anagrams
  static groupAnagrams(strs) {
    const groups = new Map();

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

    return Array.from(groups.values());
  }

  // Longest consecutive sequence
  static longestConsecutive(nums) {
    const numSet = new Set(nums);
    let longest = 0;

    for (const num of numSet) {
      if (!numSet.has(num - 1)) {
        let current = num;
        let length = 1;

        while (numSet.has(current + 1)) {
          current++;
          length++;
        }
```

```
        longest = Math.max(longest, length);
      }
    }

    return longest;
  }

  // Find duplicates
  static findDuplicates(nums) {
    const seen = new Set();
    const duplicates = [];

    for (const num of nums) {
      if (seen.has(num)) {
        duplicates.push(num);
      } else {
        seen.add(num);
      }
    }

    return duplicates;
  }
}

// 🟦 8. Stack Utilities
class StackUtils {
  // Valid parentheses
  static isValidParentheses(str) {
    const stack = [];
    const pairs = {
      ')': '(',
      ']': '[',
      '}': '{'
    };

    for (const char of str) {
      if (char === '(' || char === '[' || char === '{') {
        stack.push(char);
      } else if (char === ')' || char === ']' || char === '}') {
        if (stack.length === 0 || stack.pop() !== pairs[char]) {
          return false;
        }
      }
    }

    return stack.length === 0;
  }

  // Next greater element
  static nextGreaterElement(nums) {
    const result = new Array(nums.length).fill(-1);
    const stack = [];

    for (let i = nums.length - 1; i >= 0; i--) {
      while (stack.length > 0 && stack[stack.length - 1] <=
nums[i]) {
        stack.pop();
      }
      if (stack.length > 0) {
        result[i] = stack[stack.length - 1];
      }

      stack.push(nums[i]);
    }

    return result;
  }

  // Previous smaller element
  static previousSmallerElement(nums) {
    const result = new Array(nums.length).fill(-1);
    const stack = [];

    for (let i = 0; i < nums.length; i++) {
      while (stack.length > 0 && stack[stack.length - 1] >=
nums[i]) {
        stack.pop();
      }

      if (stack.length > 0) {
        result[i] = stack[stack.length - 1];
      }

      stack.push(nums[i]);
    }

    return result;
  }

  // Evaluate postfix expression
  static evaluatePostfix(tokens) {
    const stack = [];

    for (const token of tokens) {
      if (this.isOperator(token)) {
        const b = stack.pop();
        const a = stack.pop();

        switch (token) {
          case '+': stack.push(a + b); break;
          case '-': stack.push(a - b); break;
          case '*': stack.push(a * b); break;
          case '/': stack.push(Math.floor(a / b)); break;
        }
      } else {
        stack.push(parseInt(token, 10));
      }
    }

    return stack.pop();
  }

  static isOperator(token) {
    return token === '+' || token === '-' || token === '*' ||
token === '/';
  }

  // Remove adjacent duplicates
  static removeAdjacentDuplicates(str) {
    const stack = [];

    for (const char of str) {
      if (stack.length > 0 && stack[stack.length - 1] ===
char) {
        stack.pop();
      } else {
        stack.push(char);
      }
    }

    return stack.join('');
  }
}

// 🟦 9. Queue & Deque
class QueueUtils {
  constructor() {
    this.items = [];
  }

  enqueue(item) {
    this.items.push(item);
  }

  dequeue() {
    return this.items.shift();
  }

  isEmpty() {
    return this.items.length === 0;
  }

  size() {
    return this.items.length;
  }

  // Sliding window maximum
  static slidingWindowMaximum(nums, k) {
    if (nums.length === 0) return [];

    const result = [];
    const deque = [];

    for (let i = 0; i < nums.length; i++) {
      while (deque.length > 0 && deque[0] < i - k + 1) {
        deque.shift();
      }

      while (deque.length > 0 && nums[deque[deque.length
- 1]] < nums[i]) {
        deque.pop();
      }

      deque.push(i);

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

  return result;
}

// First negative number in window
static firstNegativeInWindow(arr, k) {
  if (arr.length < k) return [];

  const result = [];
  const deque = [];

  for (let i = 0; i < arr.length; i++) {
    if (deque.length > 0 && deque[0] < i - k + 1) {
      deque.shift();
    }

    if (arr[i] < 0) {
      deque.push(i);
    }

    if (i >= k - 1) {
      result.push(deque.length > 0 ? arr[deque[0]] : 0);
    }
  }

  return result;
}

// Level order traversal (for trees)
static levelOrderTraversal(root) {
  if (!root) return [];

  const result = [];
  const queue = [root];

  while (queue.length > 0) {
    const levelSize = queue.length;
    const level = [];

    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift();
      level.push(node.val);

      if (node.left) queue.push(node.left);
      if (node.right) queue.push(node.right);
    }

    result.push(level);
  }

  return result;
}

// Generate binary numbers
static generateBinaryNumbers(n) {
  const result = [];
  const queue = ['1'];

  for (let i = 0; i < n; i++) {
      const current = queue.shift();
      result.push(current);

      queue.push(current + '0');
      queue.push(current + '1');
    }

    return result;
  }
}

// 🟦 10. Recursion Basics
class RecursionBasics {
  // Factorial
  static factorial(n) {
    if (n <= 1) return 1;
    return n * this.factorial(n - 1);
  }

  // Fibonacci
  static fibonacci(n) {
    if (n <= 1) return n;
    return this.fibonacci(n - 1) + this.fibonacci(n - 2);
  }

  // Reverse string recursively
  static reverseStringRecursive(str) {
    if (str.length <= 1) return str;
    return this.reverseStringRecursive(str.slice(1)) + str[0];
  }

  // Power function
  static power(x, n) {
    if (n === 0) return 1;
    if (n < 0) return 1 / this.power(x, -n);
    return x * this.power(x, n - 1);
  }

  // Check palindrome recursively
  static isPalindromeRecursive(str) {
    if (str.length <= 1) return true;
    if (str[0] !== str[str.length - 1]) return false;
    return this.isPalindromeRecursive(str.slice(1, -1));
  }
}

// 🟦 11. Backtracking Patterns
class Backtracking {
  // Generate subsets
  static subsets(nums) {
    const result = [];

    function backtrack(start, current) {
      result.push([...current]);

      for (let i = start; i < nums.length; i++) {
        current.push(nums[i]);
        backtrack(i + 1, current);
        current.pop();
      }
    }

    backtrack(0, []);
    return result;
  }

  // Generate permutations
  static permutations(nums) {
    const result = [];

    function backtrack(current) {
      if (current.length === nums.length) {
        result.push([...current]);
        return;
      }

      for (let i = 0; i < nums.length; i++) {
        if (current.includes(nums[i])) continue;
        current.push(nums[i]);
        backtrack(current);
        current.pop();
      }
    }

    backtrack([]);
    return result;
  }

  // N-Queens
  static nQueens(n) {
    const result = [];
    const board = Array(n).fill().map(() => Array(n).fill('.'));

    function isSafe(row, col) {
      for (let i = 0; i < row; i++) {
        if (board[i][col] === 'Q') return false;
      }

      for (let i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--)
{
        if (board[i][j] === 'Q') return false;
      }

      for (let i = row - 1, j = col + 1; i >= 0 && j < n; i--, j+
+) {
        if (board[i][j] === 'Q') return false;
      }

      return true;
    }

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

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

    backtrack(0);
    return result;
  }

  // Combination sum
  static combinationSum(candidates, target) {
    const result = [];

    function backtrack(start, current, sum) {
      if (sum === target) {
        result.push([...current]);
        return;
      }

      if (sum > target) return;

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

    backtrack(0, [], 0);
    return result;
  }

  // Word search
  static wordSearch(board, word) {
    const rows = board.length;
    const cols = board[0].length;

    function dfs(r, c, index) {
      if (index === word.length) return true;
      if (r < 0 || r >= rows || c < 0 || c >= cols || board[r][c] !
== word[index]) {
        return false;
      }

      const temp = board[r][c];
      board[r][c] = '#';

      const found = dfs(r + 1, c, index + 1) ||
                    dfs(r - 1, c, index + 1) ||
                    dfs(r, c + 1, index + 1) ||
                    dfs(r, c - 1, index + 1);

      board[r][c] = temp;
      return found;
    }

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

    return false;
  }
}

// 🟦 12. Binary Search Patterns
class BinarySearchPatterns {
  // Lower bound (first element >= target)
  static lowerBound(arr, target) {
    let left = 0;
    let right = arr.length;

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

    return left;
  }

  // Upper bound (first element > target)
  static upperBound(arr, target) {
    let left = 0;
    let right = arr.length;

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

    return left;
  }

  // Search in rotated array
  static searchRotated(nums, target) {
    let left = 0;
    let right = nums.length - 1;

    while (left <= right) {
      const mid = Math.floor((left + right) / 2);

      if (nums[mid] === target) return mid;

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

    return -1;
  }

  // Find peak element
  static findPeakElement(nums) {
    let left = 0;
    let right = nums.length - 1;

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

    return left;
  }

  // Find minimum in rotated array
  static findMinRotated(nums) {
    let left = 0;
    let right = nums.length - 1;

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

    return nums[left];
  }
}

// 🟦 13. Heap / Priority Queue
class MinHeap {
  constructor() {
    this.heap = [];
  }

  insert(val) {
    this.heap.push(val);
    this.bubbleUp();
  }

  extractMin() {
    if (this.heap.length === 0) return null;
    if (this.heap.length === 1) return this.heap.pop();
    const min = this.heap[0];
    this.heap[0] = this.heap.pop();
    this.bubbleDown();
    return min;
  }

  bubbleUp() {
    let index = this.heap.length - 1;
    while (index > 0) {
      const parentIndex = Math.floor((index - 1) / 2);
      if (this.heap[parentIndex] <= this.heap[index]) break;
      [this.heap[parentIndex], this.heap[index]] =
[this.heap[index], this.heap[parentIndex]];
      index = parentIndex;
    }
  }

  bubbleDown() {
    let index = 0;
    const length = this.heap.length;

    while (true) {
      let leftChild = 2 * index + 1;
      let rightChild = 2 * index + 2;
      let swap = null;

      if (leftChild < length && this.heap[leftChild] <
this.heap[index]) {
        swap = leftChild;
      }

      if (rightChild < length &&
          (swap === null && this.heap[rightChild] <
this.heap[index]) ||
          (swap !== null && this.heap[rightChild] <
this.heap[leftChild])) {
        swap = rightChild;
      }

      if (swap === null) break;
      [this.heap[index], this.heap[swap]] = [this.heap[swap],
this.heap[index]];
      index = swap;
    }
  }

  size() {
    return this.heap.length;
  }

  peek() {
    return this.heap[0];
  }
}

class HeapUtils {
  // Kth largest element
  static kthLargest(nums, k) {
    // Using quickselect approach
    return this.quickSelect(nums, 0, nums.length - 1,
nums.length - k);
  }

  static quickSelect(nums, left, right, k) {
    if (left === right) return nums[left];

    const pivotIndex = this.partition(nums, left, right);

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

  static partition(nums, left, right) {
    const pivot = nums[right];
    let i = left;

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

    [nums[i], nums[right]] = [nums[right], nums[i]];
    return i;
  }

  // Top k frequent elements
  static topKFrequent(nums, k) {
    const freq = {};
    for (const num of nums) {
      freq[num] = (freq[num] || 0) + 1;
    }

    const unique = Object.keys(freq).map(Number);
    return this.quickSelectFrequent(unique, 0, unique.length
- 1, unique.length - k, freq);
  }

  static quickSelectFrequent(arr, left, right, k, freq) {
    if (left === right) return arr.slice(k);

    const pivotIndex = this.partitionFrequent(arr, left, right,
freq);

    if (k === pivotIndex) {
      return arr.slice(k);
    } else if (k < pivotIndex) {
      return this.quickSelectFrequent(arr, left, pivotIndex - 1,
k, freq);
    } else {
      return this.quickSelectFrequent(arr, pivotIndex + 1,
right, k, freq);
    }
  }

  static partitionFrequent(arr, left, right, freq) {
    const pivot = arr[right];
    let i = left;

    for (let j = left; j < right; j++) {
      if (freq[arr[j]] <= freq[pivot]) {
        [arr[i], arr[j]] = [arr[j], arr[i]];
        i++;
      }
    }

    [arr[i], arr[right]] = [arr[right], arr[i]];
    return i;
  }

  // Merge k sorted lists (simplified array version)
  static mergeKSortedLists(lists) {
    const heap = new MinHeap();
    const result = [];

    for (let i = 0; i < lists.length; i++) {
      if (lists[i].length > 0) {
        heap.insert({ value: lists[i][0], listIndex: i,
elementIndex: 0 });
      }
    }

    while (heap.size() > 0) {
      const { value, listIndex, elementIndex } =
heap.extractMin();
      result.push(value);

      if (elementIndex + 1 < lists[listIndex].length) {
        heap.insert({
          value: lists[listIndex][elementIndex + 1],
          listIndex,
          elementIndex: elementIndex + 1
        });
      }
    }

    return result;
  }

  // Sort nearly sorted array
  static sortNearlySorted(arr, k) {
    const heap = new MinHeap();
    const result = [];

    for (let i = 0; i <= k && i < arr.length; i++) {
      heap.insert(arr[i]);
    }

    for (let i = k + 1; i < arr.length; i++) {
      result.push(heap.extractMin());
      heap.insert(arr[i]);
    }
```

```
    while (heap.size() > 0) {
      result.push(heap.extractMin());
    }

    return result;
  }

  // Task scheduling
  static leastInterval(tasks, n) {
    const freq = {};
    for (const task of tasks) {
      freq[task] = (freq[task] || 0) + 1;
    }

    const frequencies = Object.values(freq).sort((a, b) => b -
a);
    const maxFreq = frequencies[0];

    let idleTime = (maxFreq - 1) * n;

    for (let i = 1; i < frequencies.length; i++) {
      idleTime -= Math.min(maxFreq - 1, frequencies[i]);
    }

    idleTime = Math.max(0, idleTime);
    return tasks.length + idleTime;
  }
}

// 🟦 14. Linked List Utilities
class ListNode {
  constructor(val = 0, next = null) {
    this.val = val;
    this.next = next;
  }
}

class LinkedListUtils {
  // Reverse linked list
  static reverseList(head) {
    let prev = null;
    let current = head;

    while (current !== null) {
      const next = current.next;
      current.next = prev;
      prev = current;
      current = next;
    }

    return prev;
  }

  // Detect cycle
  static hasCycle(head) {
    if (!head || !head.next) return false;

    let slow = head;
    let fast = head;
  while (fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;

    if (slow === fast) return true;
  }

  return false;
}

// Find middle node
static findMiddle(head) {
  if (!head) return null;

  let slow = head;
  let fast = head;

  while (fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;
  }

  return slow;
}

// Merge two sorted lists
static mergeTwoLists(l1, l2) {
  const dummy = new ListNode();
  let current = dummy;

  while (l1 !== null && l2 !== null) {
    if (l1.val <= l2.val) {
      current.next = l1;
      l1 = l1.next;
    } else {
      current.next = l2;
      l2 = l2.next;
    }
    current = current.next;
  }

  if (l1 !== null) {
    current.next = l1;
  } else {
    current.next = l2;
  }

  return dummy.next;
}

// Remove nth node from end
static removeNthFromEnd(head, n) {
  const dummy = new ListNode(0, head);
  let slow = dummy;
  let fast = dummy;

  for (let i = 0; i <= n; i++) {
    fast = fast.next;
  }
```

```
    while (fast !== null) {
      slow = slow.next;
      fast = fast.next;
    }

    slow.next = slow.next.next;
    return dummy.next;
  }
}

// 🟦 15. Tree Traversals
class TreeNode {
  constructor(val = 0, left = null, right = null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}

class TreeTraversals {
  // Inorder traversal
  static inorderTraversal(root) {
    const result = [];

    function traverse(node) {
      if (!node) return;
      traverse(node.left);
      result.push(node.val);
      traverse(node.right);
    }

    traverse(root);
    return result;
  }

  // Preorder traversal
  static preorderTraversal(root) {
    const result = [];

    function traverse(node) {
      if (!node) return;
      result.push(node.val);
      traverse(node.left);
      traverse(node.right);
    }

    traverse(root);
    return result;
  }

  // Postorder traversal
  static postorderTraversal(root) {
    const result = [];

    function traverse(node) {
      if (!node) return;
      traverse(node.left);
      traverse(node.right);
      result.push(node.val);
    }
    traverse(root);
    return result;
  }

  // Level order traversal
  static levelOrderTraversal(root) {
    return QueueUtils.levelOrderTraversal(root);
  }

  // Height of tree
  static treeHeight(root) {
    if (!root) return -1;
    const leftHeight = this.treeHeight(root.left);
    const rightHeight = this.treeHeight(root.right);
    return Math.max(leftHeight, rightHeight) + 1;
  }
}

// 🟦 16. Graph Algorithms
class Graph {
  constructor(vertices) {
    this.vertices = vertices;
    this.adjList = new Map();
    for (let i = 0; i < vertices; i++) {
      this.adjList.set(i, []);
    }
  }

  addEdge(u, v) {
    this.adjList.get(u).push(v);
    this.adjList.get(v).push(u);
  }
}

class GraphAlgorithms {
  // BFS traversal
  static BFS(graph, start) {
    const visited = new Set();
    const result = [];
    const queue = [start];
    visited.add(start);

    while (queue.length > 0) {
      const vertex = queue.shift();
      result.push(vertex);

      for (const neighbor of graph.adjList.get(vertex)) {
        if (!visited.has(neighbor)) {
          visited.add(neighbor);
          queue.push(neighbor);
        }
      }
    }

    return result;
  }

  // DFS traversal
  static DFS(graph, start) {
    const visited = new Set();
    const result = [];

    function dfs(vertex) {
      visited.add(vertex);
      result.push(vertex);

      for (const neighbor of graph.adjList.get(vertex)) {
        if (!visited.has(neighbor)) {
          dfs(neighbor);
        }
      }
    }

    dfs(start);
    return result;
  }

  // Detect cycle in undirected graph
  static hasCycleGraph(graph) {
    const visited = new Set();

    function dfs(vertex, parent) {
      visited.add(vertex);

      for (const neighbor of graph.adjList.get(vertex)) {
        if (!visited.has(neighbor)) {
          if (dfs(neighbor, vertex)) return true;
        } else if (neighbor !== parent) {
          return true;
        }
      }

      return false;
    }

    for (let vertex = 0; vertex < graph.vertices; vertex++) {
      if (!visited.has(vertex)) {
        if (dfs(vertex, -1)) return true;
      }
    }

    return false;
  }

  // Count connected components
  static countConnectedComponents(graph) {
    const visited = new Set();
    let count = 0;

    function dfs(vertex) {
      visited.add(vertex);
      for (const neighbor of graph.adjList.get(vertex)) {
        if (!visited.has(neighbor)) {
          dfs(neighbor);
        }
      }
    }

    for (let vertex = 0; vertex < graph.vertices; vertex++) {
      if (!visited.has(vertex)) {
        count++;
        dfs(vertex);
      }
    }

    return count;
  }

  // Shortest path in unweighted graph
  static shortestPathUnweighted(graph, start, end) {
    if (start === end) return [start];

    const visited = new Set();
    const parent = new Map();
    const queue = [start];
    visited.add(start);

    while (queue.length > 0) {
      const vertex = queue.shift();

      for (const neighbor of graph.adjList.get(vertex)) {
        if (!visited.has(neighbor)) {
          visited.add(neighbor);
          parent.set(neighbor, vertex);
          queue.push(neighbor);

          if (neighbor === end) {
            const path = [end];
            while (path[path.length - 1] !== start) {
              path.push(parent.get(path[path.length - 1]));
            }
            return path.reverse();
          }
        }
      }
    }

    return [];
  }
}

// 🟦 17. Dynamic Programming (1D)
class DynamicProgramming1D {
  // Fibonacci with memoization
  static fibonacciMemo(n, memo = {}) {
    if (n <= 1) return n;
    if (memo[n]) return memo[n];
    memo[n] = this.fibonacciMemo(n - 1, memo) +
this.fibonacciMemo(n - 2, memo);
    return memo[n];
  }

  // Climbing stairs
  static climbStairs(n) {
    if (n <= 2) return n;
    const dp = new Array(n + 1);
    dp[1] = 1;
    dp[2] = 2;
    for (let i = 3; i <= n; i++) {
      dp[i] = dp[i - 1] + dp[i - 2];
    }

    return dp[n];
  }

  // House robber
  static rob(nums) {
    if (nums.length === 0) return 0;
    if (nums.length === 1) return nums[0];

    const dp = new Array(nums.length);
    dp[0] = nums[0];
    dp[1] = Math.max(nums[0], nums[1]);

    for (let i = 2; i < nums.length; i++) {
      dp[i] = Math.max(dp[i - 1], dp[i - 2] + nums[i]);
    }

    return dp[nums.length - 1];
  }

  // Maximum subarray sum
  static maxSubArray(nums) {
    if (nums.length === 0) return 0;

    let maxSum = nums[0];
    let currentSum = nums[0];

    for (let i = 1; i < nums.length; i++) {
      currentSum = Math.max(nums[i], currentSum +
nums[i]);
      maxSum = Math.max(maxSum, currentSum);
    }

    return maxSum;
  }

  // Coin change (minimum coins)
  static coinChange(coins, amount) {
    const dp = new Array(amount + 1).fill(amount + 1);
    dp[0] = 0;

    for (let i = 1; i <= amount; i++) {
      for (const coin of coins) {
        if (coin <= i) {
          dp[i] = Math.min(dp[i], dp[i - coin] + 1);
        }
      }
    }

    return dp[amount] > amount ? -1 : dp[amount];
  }
}

// 🟦 18. Dynamic Programming (2D)
class DynamicProgramming2D {
  // Longest common subsequence
  static longestCommonSubsequence(text1, text2) {
    const m = text1.length;
    const n = text2.length;
    const dp = Array(m + 1).fill().map(() => Array(n +
1).fill(0));

    for (let i = 1; i <= m; i++) {
      for (let j = 1; j <= n; j++) {
        if (text1[i - 1] === text2[j - 1]) {
          dp[i][j] = dp[i - 1][j - 1] + 1;
        } else {
          dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
        }
      }
    }

    return dp[m][n];
  }

  // Longest common substring
  static longestCommonSubstring(text1, text2) {
    const m = text1.length;
    const n = text2.length;
    const dp = Array(m + 1).fill().map(() => Array(n +
1).fill(0));
    let maxLength = 0;

    for (let i = 1; i <= m; i++) {
      for (let j = 1; j <= n; j++) {
        if (text1[i - 1] === text2[j - 1]) {
          dp[i][j] = dp[i - 1][j - 1] + 1;
          maxLength = Math.max(maxLength, dp[i][j]);
        }
      }
    }

    return maxLength;
  }

  // Edit distance
  static editDistance(word1, word2) {
    const m = word1.length;
    const n = word2.length;
    const dp = Array(m + 1).fill().map(() => Array(n +
1).fill(0));

    for (let i = 0; i <= m; i++) {
      dp[i][0] = i;
    }
    for (let j = 0; j <= n; j++) {
      dp[0][j] = j;
    }

    for (let i = 1; i <= m; i++) {
      for (let j = 1; j <= n; j++) {
        if (word1[i - 1] === word2[j - 1]) {
          dp[i][j] = dp[i - 1][j - 1];
        } else {
          dp[i][j] = 1 + Math.min(
            dp[i - 1][j - 1],
            dp[i - 1][j],
            dp[i][j - 1]
          );
        }
      }
    }

    return dp[m][n];
  }

  // Unique paths
  static uniquePaths(m, n) {
    const dp = Array(m).fill().map(() => Array(n).fill(1));

    for (let i = 1; i < m; i++) {
      for (let j = 1; j < n; j++) {
        dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
      }
    }

    return dp[m - 1][n - 1];
  }

  // 0/1 Knapsack
  static knapsack01(weights, values, capacity) {
    const n = weights.length;
    const dp = Array(n + 1).fill().map(() => Array(capacity
+ 1).fill(0));

    for (let i = 1; i <= n; i++) {
      for (let w = 0; w <= capacity; w++) {
        if (weights[i - 1] <= w) {
          dp[i][w] = Math.max(
            dp[i - 1][w],
            dp[i - 1][w - weights[i - 1]] + values[i - 1]
          );
        } else {
          dp[i][w] = dp[i - 1][w];
        }
      }
    }

    return dp[n][capacity];
  }
}

// 🟦 19. Bit Manipulation
class BitManipulation {
  // Check if number is power of two
  static isPowerOfTwo(n) {
    return n > 0 && (n & (n - 1)) === 0;
  }

  // Count set bits
  static countSetBits(n) {
    let count = 0;
    while (n > 0) {
      count += n & 1;
      n >>= 1;
    }
    return count;
  }

  // Find single non-repeating number
  static singleNumber(nums) {
    let result = 0;
    for (const num of nums) {
      result ^= num;
    }
    return result;
  }

  // Toggle ith bit
  static toggleBit(n, i) {
    return n ^ (1 << i);
  }

  // Generate subsets using bitmask
  static subsetsBitmask(nums) {
    const n = nums.length;
    const total = 1 << n;
    const result = [];

    for (let mask = 0; mask < total; mask++) {
      const subset = [];
      for (let i = 0; i < n; i++) {
        if (mask & (1 << i)) {
          subset.push(nums[i]);
        }
      }
      result.push(subset);
    }

    return result;
  }
}

// 🟦 20. Math & Number Theory
class MathUtils {
  // GCD and LCM
  static gcd(a, b) {
    while (b !== 0) {
      [a, b] = [b, a % b];
    }
    return a;
  }

  static lcm(a, b) {
    return (a * b) / this.gcd(a, b);
  }

  // Prime check
  static isPrime(n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 === 0 || n % 3 === 0) return false;

    for (let i = 5; i * i <= n; i += 6) {
      if (n % i === 0 || n % (i + 2) === 0) return false;
    }
```

```
    return true;
  }

  // Sieve of Eratosthenes
  static sieveOfEratosthenes(n) {
    const isPrime = new Array(n + 1).fill(true);
    isPrime[0] = false;
    isPrime[1] = false;

    for (let p = 2; p * p <= n; p++) {
      if (isPrime[p]) {
        for (let i = p * p; i <= n; i += p) {
          isPrime[i] = false;
        }
      }
    }

    const primes = [];
    for (let i = 2; i <= n; i++) {
      if (isPrime[i]) {
        primes.push(i);
      }
    }

    return primes;
  }

  // Fast exponentiation
  static fastExponentiation(x, n) {
    if (n === 0) return 1;
    if (n < 0) {
      x = 1 / x;
      n = -n;
    }

    let result = 1;
    while (n > 0) {
      if (n & 1) {
        result *= x;
      }
      x *= x;
      n >>= 1;
    }

    return result;
  }

  // Modular inverse (using extended Euclidean algorithm)
  static modularInverse(a, m) {
    let m0 = m;
    let y = 0, x = 1;

    if (m === 1) return 0;

    while (a > 1) {
      const q = Math.floor(a / m);
      let t = m;
      m = a % m;
      a = t;
      t = y;
      y = x - q * y;
      x = t;
    }

    if (x < 0) x += m0;
    return x;
  }
}
```