Q1: Generate the first 10 numbers in the Fibonacci sequence

Your task is to write a function that returns an array containing the first 10 numbers of the Fibonacci sequence, which is a series where each number is the sum of the two preceding ones

- Initialize the sequence with the first two Fibonacci numbers (0 and 1)
- Return an array containing exactly 10 Fibonacci numbers

```
// Usage Example:
const fibonacci = (count) => {
// Implementation here
}

console.log(fibonacci(10)); // Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Q2: Print numbers from 1 to 100, replacing multiples of 3 with "Fizz," multiples of 5 with "Buzz," and both with "FizzBuzz" (FizzBuzz)

FizzBuzz is a classic programming problem used to teach division and conditionals

Your task is to loop through numbers from 1 to 100 and apply the FizzBuzz rules. Replace numbers divisible by 3 with "Fizz," numbers divisible by 5 with "Buzz," and numbers divisible by both with "FizzBuzz."

```
// Usage Example:
for (let i = 1; i <= 100; i++) {
 console.log(fizzBuzz(i));
}

// Outputs numbers from 1 to 100 with appropriate replacements
```

Q3: Implement a function to traverse a graph by exploring as far as possible before backtracking (Depth-first search)

Depth-first search (DFS) is a fundamental graph traversal algorithm that explores paths to their full depth before backtracking. Used extensively in tree traversal, pathfinding, and solving maze-like problems, DFS excels in scenarios where complete exploration of branches is needed before moving to siblings, making it particularly effective for tasks like topological sorting, cycle detection, and generating game trees

Create a function that performs DFS on a graph represented as an adjacency list. Your implementation should handle cycles and demonstrate both recursive and iterative approaches

- Implement both recursive and iterative solutions
- Track visited nodes to avoid cycles
- Return traversal order array
- Handle invalid inputs

```
const graph = {
 a: ['b', 'c'],
 b: ['d'],
 c: ['e'],
 d: [],
 e: ['f'],
 f: []
```

```
};

dfs(graph, 'a'); // Should visit nodes in depth-first order
```

Q4: Explore nodes layer by layer in a graph (Breadth-first search)

Breadth-first search (BFS) is a graph traversal strategy where we visit all vertices at the current depth before moving deeper. It's commonly used in shortest path algorithms for unweighted graphs, web crawling, and social network analysis where relationships between nodes need to be mapped level by level

Implement a BFS function that processes a graph level by level using a queue data structure. The graph should be represented as an adjacency list, and your solution should handle cycles effectively

- Use a queue for level-order traversal
- Handle cycles in the graph
- Process nodes level by level
- Return complete traversal order

The queue-based approach ensures we maintain the correct level order. While similar to DFS in time complexity, BFS provides the additional guarantee of finding shortest paths in unweighted graphs. The visited set prevents cycles from causing infinite loops, making the algorithm robust for any graph structure

```
const graph = {
 a: ['b', 'c'],
 b: ['d', 'e'],
 c: ['f'],
 d: [],
 e: ['f'],
 f: []
};

bfs(graph, 'a'); // Should visit nodes level by level
```

Q5: Divide and conquer by repeatedly narrowing the search range (Binary Search)

Binary search is a divide-and-conquer algorithm that achieves O(log n) time complexity by leveraging sorted data to repeatedly halve the search space. This makes it exponentially faster than linear search for large datasets, though it requires the input to be sorted first

Implement both iterative and recursive versions of binary search that find the index of a target element in a sorted array. Handle edge cases and verify input validity

- Verify array is sorted
- Implement both approaches
- Handle edge cases properly
- Return -1 if not found

```
const arr = [1, 3, 5, 7, 9, 11];
const target = 7;

console.log(binarySearch(arr, target)); // Expected: 3
```

Q6: Check each element in a list to find a target value (Linear Search)

Linear search is the most straightforward searching algorithm that sequentially checks each element in a collection until finding a match. Despite its O(n) time complexity, it's often practical for small datasets, unsorted collections, or when elements are likely to be found near the start of the array, making it a fundamental building block for more complex search algorithms

Write a function that performs linear search to find a target element in an array. Return the index of the first occurrence or -1 if not found

- Return first occurrence index
- Handle different data types
- Return -1 if not found
- Validate inputs properly

```
const arr = [64, 34, 25, 12, 22, 11, 90];
console.log(linearSearch(arr, 12)); // Expected: 3
```

Q7: Find pairs in array using two pointers technique

The two pointers technique is a powerful pattern for solving array-based problems, where maintaining two index references allows for efficient traversal and comparison of elements. It's particularly useful for sorted arrays and can often transform quadratic solutions into linear ones by eliminating the need for nested loops

Implement a function that finds a pair of numbers in a sorted array that sum to a target value. Return the indices of the numbers or null if no such pair exists

- Ensure input array is sorted
- Return pair indices or null
- Maintain O(n) time complexity
- Handle edge cases properly

The two pointers approach excels when the relationship between elements can guide pointer movement. In sum-finding problems, comparing the current sum with the target tells us which pointer to move, making the solution both elegant and efficient. This pattern can be adapted for many similar problems like finding triplets or subarrays with specific properties

```
const arr = [2, 7, 11, 15];
const target = 9;


console.log(findPairSum(arr, target)); // Expected: [0, 1]
```

Q8: Reverse the order of characters in a string

String reversal is a common programming task that tests understanding of string manipulation, array operations, and handling of special characters including Unicode. This operation appears frequently in text processing, palindrome checking, and as a building block for more complex string algorithms

Implement multiple approaches to reverse a string, demonstrating both built-in methods and manual iteration

- Handle invalid inputs
- Implement multiple approaches
- Consider Unicode characters
- Maintain O(n) complexity

```

```

```
const test = "Hello, World! ";
console.log(reverseString(test));     // Expected: " !dlroW ,olleH"
```

Q9: Check if two strings are made of the same characters (Anagram Check)

Anagram checking is a classic string manipulation problem that tests ability to compare and process characters efficiently. This problem appears in many applications, from word games to text analysis, and offers multiple solution approaches with different performance characteristics and trade-offs

Create a function that determines if two strings are anagrams of each other, handling case sensitivity and special characters appropriately

- Handle case sensitivity
- Manage special characters
- Implement O(n) solution
- Validate inputs properly

```
const str1 = "Tea Time";
const str2 = "Eat Time";


console.log(isAnagram(str1, str2)); // Expected: true
```

Q10: Check if a string can be split into dictionary words (Word Break)

Word Break is a classic dynamic programming challenge that tests the ability to identify and efficiently solve problems with overlapping subproblems by using previously computed results. This problem appears frequently in string parsing and natural language processing applications. Write a function that determines if a string can be segmented into words from a given dictionary. The function should return true if such segmentation is possible

- Use dynamic programming approach
- Handle empty strings and dictionary
- Implement efficient memoization
- Consider case sensitivity

```
const dict = ['leet', 'code'];
const str = "leetcode";


console.log(wordBreak(str, dict)); // Expected: true
```

Q11: Expand and contract a range to find subarrays that meet criteria (Sliding Window)

The sliding window technique is an optimization pattern used to solve array or string problems where we need to track a subset of elements in a specific range. It involves maintaining a window that can expand or contract over the data to find optimal solutions without unnecessary recalculations

Implement a function that finds the maximum sum of any contiguous subarray of size k in an array. The function should efficiently handle the window movement without recalculating the entire sum each time

- Handle invalid inputs (k > array length)
- Maintain O(n) time complexity
- Track current and maximum sums
- Handle edge cases (empty arrays)

```

```

```
const arr = [2, 3, 4, 1, 5];
const k = 2;

console.log(maxSubarraySum(arr, k)); // Expected: 7 (sum of [3,4])
```

Q12: Check if a string has balanced opening and closing brackets (Valid Parentheses)

The parentheses validation problem demonstrates how stack data structures can efficiently track and verify matching pairs in sequences. This fundamental problem appears in many real-world applications, from code editors checking syntax to expression evaluators. Create a function that determines if a string containing different types of brackets is valid, meaning every opening bracket has a corresponding closing bracket in the correct order

- Handle multiple bracket types
- Ensure correct opening/closing order
- Manage empty strings
- Maintain O(n) complexity

Using a stack for this problem provides an elegant solution that naturally handles nested structures. The last-in-first-out nature of stacks perfectly matches the nesting behavior of brackets, and using a hash map for bracket pairs makes the matching logic clean and maintainable. This approach can be extended to handle more complex validation scenarios

```
const str1 = "({[]})";
const str2 = "([)]";

console.log(isValidParentheses(str1)); // Expected: true
console.log(isValidParentheses(str2)); // Expected: false
```

Q13: Find all possible solutions by trying different paths and backtracking when needed (Backtracking)

Backtracking is a systematic approach to finding solutions for complex problems by incrementally building candidates and abandoning those that fail to satisfy constraints. This algorithmic technique is particularly powerful for solving problems that require exploring all possible combinations or permutations while respecting certain rules or limitations

Write a function that solves the N-Queens problem: place N chess queens on an N×N board so that no two queens threaten each other. Return all possible solutions

- Validate queen placements
- Implement efficient backtracking
- Handle various board sizes
- Return all valid solutions

```
console.log(solveNQueens(4));
// Expected: Array of valid board configurations
```

Q14: Split and merge sorted halves to sort an array (Merge Sort)

Merge sort is a classic divide-and-conquer algorithm that demonstrates efficient sorting through recursion and merging. It provides stable sorting with guaranteed O(n log n) time complexity. Implement a merge sort function that takes an unsorted array and returns it in sorted order. The solution should handle the recursive division and merging of subarrays efficiently

- Divide array into smaller subarrays
- Merge sorted subarrays correctly

- Handle edge cases (empty/single element)

Merge sort is particularly efficient for large datasets and linked lists. While its space complexity is O(n), it offers advantages like stability and predictable performance. The algorithm shines when dealing with datasets too large to fit in memory, as it can be adapted for external sorting

```
const arr = [38, 27, 43, 3, 9, 82, 10];
console.log(mergeSort(arr)); // Expected: [3, 9, 10, 27, 38, 43, 82]
```

Q15: Order graph nodes in a sequence respecting dependencies (Topological Sort)

Topological sorting of a Directed Acyclic Graph (DAG) is a linear ordering of its vertices such that for every directed edge uv, vertex u comes before v in the ordering

Implement a function that performs a topological sort on a given DAG

- Use depth-first search to explore nodes
- Detect cycles to ensure the graph is acyclic
- Return a list representing the topological order

```
const graph = {
5: [2, 0],
4: [0, 1],
2: [3],
3: [1],
0: [],
1: []
};

console.log(topologicalSort(graph));
// Expected: [5, 4, 2, 3, 1, 0]
```

Q16: Repeatedly swap adjacent elements to sort an array (Bubble Sort)

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. Your task is to implement the bubble sort algorithm to sort an array of numbers

- Implement nested loops to compare adjacent elements
- Optimize by reducing the range after each pass
- Detect if no swaps are made to terminate early

```
const array = [64, 34, 25, 12, 22, 11, 90];
bubbleSort(array);
console.log(array);
// Expected: [11, 12, 22, 25, 34, 64, 90]
```

Q17: Sort by placing elements into different 'buckets' before sorting each bucket (Bucket Sort)

Bucket sort divides the elements into several buckets, sorts each bucket individually, and then combines them to form the sorted array

Implement bucket sort to sort an array of floating-point numbers between 0 and 1

- Determine the number of buckets and their ranges

- Distribute the elements into appropriate buckets
- Sort individual buckets using another sorting algorithm

While bucket sort can achieve linear time complexity under optimal conditions, it requires knowledge of the data distribution and is most effective when input is uniformly distributed over a range. By dividing elements into buckets and sorting each, we can significantly improve sorting performance for certain datasets

```
const array = [0.42, 0.32, 0.23, 0.52, 0.25, 0.47, 0.51];
console.log(bucketSort(array));
// Expected: [0.23, 0.25, 0.32, 0.42, 0.47, 0.51, 0.52]
```

Q18: Find the smallest unsorted element and move it to the front (Selection Sort)

Selection sort divides the input list into two parts: a sorted sublist and an unsorted sublist. It repeatedly selects the smallest element from the unsorted sublist and moves it to the end of the sorted sublist. Your task is to implement the selection sort algorithm to sort an array of numbers

- Iterate over the array to find the minimum element
- Swap the found minimum element with the first element
- Repeat for the remaining unsorted subarray

```
const array = [64, 25, 12, 22, 11];
selectionSort(array);
console.log(array); // Expected: [11, 12, 22, 25, 64]
```

Q19: Insert each unsorted element into its correct position (Insertion Sort)

Insertion sort builds the final sorted array one item at a time. It takes each element from the input data and finds the position it belongs within the sorted list and inserts it there

Implement the insertion sort algorithm to sort an array of numbers

- Start from the second element and compare it with previous elements
- Shift elements to make space for the inserted element
- Repeat until the array is sorted

```
const array = [12, 11, 13, 5, 6];
insertionSort(array);
console.log(array); // Expected: [5, 6, 11, 12, 13]
```

Q20: Sort using a binary heap structure (Heap Sort)

Heap sort is a comparison-based sorting technique based on a binary heap data structure. It involves building a max heap from the input data and then repeatedly extracting the maximum element from the heap and rebuilding the heap. Your task is to implement heap sort to sort an array of numbers

- Build a max heap from the input data
- Swap the root of the heap with the last element
- Reduce the heap size and heapify the root
- Repeat until the heap size is reduced to one

Heap sort offers an in-place sorting algorithm with a time complexity of O(n log n). While it doesn't require additional memory like merge sort, it is not a stable sort and may not maintain the order of equal elements. It's efficient for large datasets and is widely used when memory usage is a concern

```

```

```
const array = [12, 11, 13, 5, 6, 7];
heapSort(array);
console.log(array); // Expected: [5, 6, 7, 11, 12, 13]
```

Q21: Use a pivot to partition elements into two groups for sorting (Quick Sort)

Quick sort is a divide-and-conquer algorithm that selects a pivot element and partitions the array into two halves based on the pivot, recursively sorting the subarrays

Your task is to write a function that sorts an array using the quick sort algorithm

- Implement the partitioning logic
- Choose an efficient pivot selection strategy
- Ensure the base case handles arrays of size 1 or 0

```
const array = [10, 7, 8, 9, 1, 5];
quickSort(array);
console.log(array); // Expected: [1, 5, 7, 8, 9, 10]
```

Q22: Store strings in a prefix tree structure for fast retrieval (Trie)

A trie, or prefix tree, is a data structure used for efficient retrieval of keys in a large set of strings. Each node represents a common prefix, allowing for fast insertions and lookups. Tries are especially useful for tasks like autocomplete and spell checking

Implement a Trie with methods for inserting words, searching for exact matches, and checking if any words start with a given prefix

- Implement insertion and search operations
- Handle cases for words ending at a node

```
const trie = new Trie();
trie.insert('apple');
console.log(trie.search('apple'));   // Expected: true
console.log(trie.search('app'));     // Expected: false
console.log(trie.startsWith('app')); // Expected: true
trie.insert('app');
console.log(trie.search('app'));     // Expected: true
```

Q23: Compute cumulative sums for subarray calculations (Prefix Sum)

Prefix sums are a technique used to calculate cumulative sums of a sequence of numbers. By precomputing these sums, you can efficiently calculate the sum of any subarray in constant time

Implement a function that computes the prefix sums of an array and uses it to calculate the sum of a given subarray

- Compute the prefix sum array
- Use the prefix sums to calculate subarray sums efficiently
- Handle edge cases like empty arrays

```
const nums = [1, 2, 3, 4, 5];
const prefixSums = computePrefixSums(nums);
console.log(sumRange(prefixSums, 1, 3)); // Expected: 9 (sum of elements at indices 1 to 3)
```

Q24: Find the k-th smallest element in an unsorted array (Quickselect)

Quickselect is an efficient algorithm to find the k-th smallest element in an unordered list. It works similarly to quicksort but focuses only on the part of the array that contains the k-th smallest element

Implement the quickselect algorithm to find the k-th smallest element in an unsorted array

- Use a partitioning method similar to quicksort
- Reduce the problem size in each recursive call
- Optimize for average-case O(n) time complexity

Quickselect is efficient for finding specific order statistics without fully sorting the array, making it practical for large datasets where full sorting is unnecessary

```
const arr = [7, 10, 4, 3, 20, 15];
console.log(quickSelect(arr, 3)); // Expected: 7 (3rd smallest element)
```

Q25: Find the shortest path from a source node to all other nodes (Dijkstra's Algorithm)

Dijkstra's algorithm finds the shortest paths from a source node to all other nodes in a weighted graph with non-negative edge weights. It's widely used in network routing protocols and mapping applications

Implement Dijkstra's algorithm to compute the shortest path distances from a given source node

- Initialize distances and priority queue
- Relax edges and update distances
- Handle graphs represented as adjacency lists or matrices

```
const graph = {
 A: { B: 1, C: 4 },
 B: { A: 1, C: 2, D: 5 },
 C: { A: 4, B: 2, D: 1 },
 D: { B: 5, C: 1 }
};

console.log(dijkstra(graph, 'A'));
// Expected: { A: 0, B: 1, C: 3, D: 4 }
```

Q26: Group nodes by connectivity for union-find operations (Union Find)

Union-Find (Disjoint Set) is a data structure that keeps track of elements partitioned into disjoint subsets, useful for tracking connectivity in networks. Your task is to implement a Union-Find data structure with union and find methods

- Use path compression for optimization
- Detect cycles in undirected graphs

```
const uf = new UnionFind(5);
uf.union(0, 1);
uf.union(1, 2);
console.log(uf.connected(0, 2)); // Expected: true
console.log(uf.connected(0, 3)); // Expected: false
```

Q27: Find the minimum number of coins needed to make change for a given amount using available denominations (Coin Change Problem)

The coin change problem seeks the minimum number of coins needed to make a certain amount of change from a given set of coin denominations. Implement a function that computes the minimum number of coins needed to make change for a given amount

```
const coins = [1, 2, 5];
console.log(coinChange(coins, 11)); // Expected: 3 (11 = 5 + 5 + 1)
```

Q28: Find the shortest path in a weighted graph, using heuristics (A* Algorithm)

The A* algorithm finds the shortest path between nodes in a graph, using heuristics to prioritize paths that are likely to lead to the goal

Your task is to implement the A* algorithm to find the shortest path between a start and goal node in a graph

- Implement a priority queue to select the next node
- Use a heuristic function (e.g., Euclidean distance)
- Handle graphs with varying edge weights

By using heuristics, A* can significantly reduce the number of nodes it needs to examine, making it more efficient than other algorithms like Dijkstra's in certain scenarios. It's widely used in pathfinding and graph traversal applications, especially in game development and AI

```
const graph = {
 A: { B: 1, C: 3 },
 B: { A: 1, C: 1, D: 5 },
 C: { A: 3, B: 1, D: 2 },
 D: { B: 5, C: 2 }
};
const heuristics = {
 A: 4,
 B: 2,
 C: 1,
 D: 0
};

console.log(aStar(graph, heuristics, 'A', 'D')); // Expected: ['A', 'B', 'C', 'D']
```

Q29: Find the minimum spanning tree of a graph (Prim's Algorithm)

Prim's algorithm finds a minimum spanning tree for a connected weighted undirected graph, adding the least expensive edge from the tree to a vertex not yet in the tree

Implement Prim's algorithm to compute the minimum spanning tree of a given graph

- Initialize the tree with a single vertex
- Use a priority queue to select the next edge
- Ensure all vertices are included in the spanning tree

Prim's algorithm is greedy in nature, always picking the minimum weight edge that connects a vertex in the tree to a vertex outside the tree. It's essential in network design, where minimizing the total length of cables or wiring is desired

```

```

```
const graph = {
0: { 1: 2, 3: 6 },
1: { 0: 2, 2: 3, 3: 8, 4: 5 },
2: { 1: 3, 4: 7 },
3: { 0: 6, 1: 8, 4: 9 },
4: { 1: 5, 2: 7, 3: 9 }
};

console.log(primsAlgorithm(graph));
// Expected: Minimum Spanning Tree edges and total cost
```

Q30: Detect cycles in a linked list (Floyd's Cycle Detection)

Floyd's cycle detection algorithm, also known as the tortoise and hare algorithm, is used to detect cycles in a sequence of values or linked list. Implement a function that detects if a linked list has a cycle

- Use two pointers moving at different speeds
- Detect when the pointers meet, indicating a cycle
- Handle linked lists with no cycles

```
const node1 = { val: 3 };
const node2 = { val: 2 };
const node3 = { val: 0 };
const node4 = { val: -4 };

node1.next = node2;
node2.next = node3;
node3.next = node4;
node4.next = node2; // Creates a cycle

console.log(hasCycle(node1)); // Expected: true
```

Q31: Encode characters based on frequency for data compression (Huffman Encoding)

Huffman Encoding is a compression algorithm that assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters. Your task is to implement Huffman Encoding to compress a given string

- Build a frequency map of characters
- Construct a priority queue (min-heap) of nodes
- Build the Huffman tree and generate codes
- Encode and decode strings using the generated codes

By effectively encoding data based on character frequencies, Huffman Encoding significantly reduces the size of data, making it ideal for file compression. Implementing this algorithm will deepen your understanding of greedy algorithms and priority queues

```
const input = "this is an example for huffman encoding";
const { encodedStr, huffmanTree } = huffmanEncoding(input);
console.log("Encoded String:", encodedStr);
const decodedStr = huffmanDecoding(encodedStr, huffmanTree);
```

```
console.log("Decoded String:", decodedStr);
// Expected Output:
// Encoded String: [compressed binary string]
// Decoded String: this is an example for huffman encoding
```

Q32: Maximize value in a knapsack by allowing fractional items (Fractional Knapsack Problem)

Your task is to implement the fractional knapsack algorithm to maximize the total value for a given weight capacity

- Calculate value-to-weight ratio for each item
- Sort items based on this ratio in decreasing order
- Add items to the knapsack starting from the highest ratio
- Allow fractions of an item if the knapsack cannot accommodate the whole item

```
const items = [
{ value: 60, weight: 10 },
{ value: 100, weight: 20 },
{ value: 120, weight: 30 }
];
const capacity = 50;
console.log(fractionalKnapsack(items, capacity)); // Expected: 240
```

Q33: Count the number of vowels in a string

Counting vowels in a string involves iterating through the string and incrementing a counter when a vowel is encountered

Your task is to write a function that counts the number of vowels in a given string

- Handle both uppercase and lowercase letters
- Consider vowels: a, e, i, o, u
- Optimize by using a set for vowel lookup

Using a set for vowel lookup improves the performance by allowing constant-time checks. This method ensures the function remains efficient even with longer strings

```
console.log(countVowels('Hello World')); // Expected: 3
```

Q34: Generate a pattern where characters are repeated in steps (Steps String Pattern)

Generating a steps pattern involves creating a series of lines where each line contains a repeated character, and the number of repetitions increases with each step. Your task is to write a function that generates a steps pattern with a given number of steps

- Use loops to control the number of lines and characters
- Adjust the pattern to increment or decrement as required
- Handle edge cases like zero or negative steps

```
generateStepsPattern(5);
/*
Expected output:
#
##
###
```

```
####
#####
*/
```

Q35: Create a centered pyramid pattern with strings (Pyramid String Pattern)

Creating a pyramid pattern involves printing lines of characters in such a way that they form a pyramid shape when centered

Your task is to write a function that prints a centered pyramid pattern of a given height

- Calculate the number of spaces and characters per line
- Use nested loops for lines and characters
- Ensure proper alignment for the pyramid shape

```
generatePyramidPattern(5);
/*
Expected output:
#
###
#####
#######
#########
*/
```

Q36: Traverse a matrix in a spiral order (Spiral Matrix)

Traversing a matrix in spiral order involves visiting elements starting from the outermost layer towards the innermost layer, moving in a spiral pattern. Your task is to write a function that returns the elements of a matrix in spiral order

- Use boundaries to keep track of traversed rows and columns
- Adjust the traversal direction at each boundary
- Collect the elements in the order they are visited

Handling different sizes of matrices and ensuring that all elements are included requires careful boundary management. This algorithm is useful in image processing and matrix manipulation tasks

```
const matrix = [
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
];
console.log(spiralOrder(matrix)); // Expected output: [1,2,3,6,9,8,7,4,5]
```

Q37: Find the minimum spanning tree of a graph using edge weights (Kruskal's Algorithm)

Kruskal's algorithm finds a minimum spanning tree for a connected weighted graph by adding edges in increasing order of weight, ensuring no cycles are formed. Your task is to implement Kruskal's algorithm to find the minimum spanning tree of a given graph

- Sort all edges in non-decreasing order of their weight
- Use a Union-Find data structure to detect cycles
- Include edges that do not form a cycle until all vertices are connected

Kruskal's algorithm is efficient for sparse graphs and helps in network design by minimizing the total cost of connecting all nodes. Understanding Union-Find is crucial for detecting cycles efficiently

```
const vertices = ['A', 'B', 'C', 'D', 'E', 'F'];
const edges = [
{ from: 'A', to: 'B', weight: 4 },
{ from: 'A', to: 'F', weight: 2 },
{ from: 'B', to: 'C', weight: 6 },
{ from: 'B', to: 'F', weight: 5 },
{ from: 'C', to: 'D', weight: 3 },
{ from: 'C', to: 'F', weight: 1 },
{ from: 'D', to: 'E', weight: 2 },
{ from: 'E', to: 'F', weight: 4 },
];
console.log(kruskalsAlgorithm(vertices, edges));
// Expected output: Minimum Spanning Tree edges and total cost
```

Q38: Find the maximum sum of any subarray (Kadane's Algorithm)

Kadane's Algorithm efficiently computes the maximum sum of a contiguous subarray within a one-dimensional array of numbers. Your task is to implement Kadane's Algorithm to find the maximum subarray sum

- Initialize variables to track current and maximum sums
- Iterate through the array, updating sums based on the current element
- Handle cases where all numbers are negative

```
const nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4];
console.log(maxSubArray(nums)); // Expected output: 6 (subarray: [4, -1, 2, 1])
```

Q39: Identify the most frequently occurring character in a string (Maximum Frequency Character)

Finding the maximum frequency character involves counting the occurrences of each character and identifying the one with the highest count

Your task is to write a function that returns the character that appears most frequently in a given string

- Use a hash map to store character counts
- Iterate through the string to update counts
- Handle ties by returning the first occurring character

```
console.log(maxFrequencyChar('javascript')); // Expected output: 'a'
```

Q40: Transform the casing of strings to title case, where the first letter of each word is capitalized

Title casing a string involves capitalizing the first letter of each word and making the rest of the letters lowercase. Your task is to write a function that converts a given string to title case

- Split the string into words
- Capitalize the first letter of each word
- Handle edge cases like multiple spaces or punctuation

Ensure that the function correctly handles strings with irregular spacing and punctuation, converting each word appropriately

```
console.log(toTitleCase('hello world')); // Expected output: 'Hello World'
```

Q41: Use a hash map to find two numbers that add up to a target (Two Sum Problem)

The Two Sum problem involves finding two numbers in an array that add up to a specific target sum. Your task is to implement a function that finds the indices of two numbers that add up to a target sum

- Use a hash map to store numbers and their indices
- Iterate through the array, checking if the complement exists
- Return the indices of the two numbers

```
const nums = [2, 7, 11, 15];
const target = 9;
console.log(twoSum(nums, target)); // Expected output: [0, 1]
```

Q42: Implement a debounce function in JavaScript

Debouncing is a technique used to ensure that a function is only executed after a specified period has elapsed since it was last invoked. This is particularly useful for optimizing performance-intensive tasks like handling window resize or scroll events

Your task is to implement a debounce function in JavaScript that delays the execution of a given function until after a specified wait time has passed since it was last called

- Create a debounce function that accepts a function and a delay
- Use a timer to track the elapsed time
- Ensure the function executes after the delay only if it hasn't been called again
- Maintain the correct context and arguments when invoking the function

```
// Usage Example:
const handleResize = debounce(() => {
 console.log('Window resized');
}, 300);

window.addEventListener('resize', handleResize);
```

Q43: Implement a function to find all permutations of a string in JavaScript

Generating all permutations of a string involves creating every possible arrangement of its characters. Your task is to implement a function that returns all permutations of a given string

Consider handling strings with duplicate characters and optimizing to avoid generating duplicate permutations

- Handle strings with unique and duplicate characters
- Implement recursive backtracking to explore all arrangements
- Optimize to avoid unnecessary computations

```
// Usage Example:
console.log(getPermutations('ABC'));
// Expected output: ['ABC', 'ACB', 'BAC', 'BCA', 'CAB', 'CBA']

console.log(getPermutations('AAB'));
// Expected output: ['AAB', 'ABA', 'BAA']
```

Q44: Implement a closure in JavaScript that maintains private state

Closures allow functions to have access to variables from an enclosing scope, enabling the creation of private state that cannot be accessed from outside the function

Your task is to implement a function that creates a counter with private state. The counter should have methods to increment, decrement, and retrieve its current value

- Understand how lexical scoping works
- Create private variables that are not accessible globally
- Ensure the closure retains access to the private state after the outer function has executed

```
// Usage Example:
const counter = createCounter();
console.log(counter.increment()); // Expected output: 1
console.log(counter.increment()); // Expected output: 2
console.log(counter.decrement()); // Expected output: 1
console.log(counter.value());     // Expected output: 1
```

Q45: Check if a string is a palindrome

A palindrome is a string that reads the same backward as forward. To check for a palindrome, compare the string to its reversed version

Your task is to write a function that determines whether a given string is a palindrome, considering only alphanumeric characters and ignoring cases

- Normalize the string by removing non-alphanumeric characters
- Convert the string to lowercase
- Compare the string to its reversed version
- Return true if it's a palindrome, false otherwise

```
console.log(isPalindrome('A man, a plan, a canal: Panama')); // Expected output: true
console.log(isPalindrome('Hello')); // Expected output: false
```

Q46: Implement a function to deep clone an object

Deep cloning creates a complete copy of an object, including all nested objects, ensuring that no references are shared between the original and the clone. Your task is to implement a function that performs a deep clone of a given object

- Handle primitive data types and objects
- Copy nested objects and arrays recursively
- Maintain the integrity of functions and special objects
- Avoid issues with circular references

```
const original = { a: 1, b: { c: 2 } };
const clone = deepClone(original);
clone.b.c = 3;
console.log(original.b.c); // Expected output: 2
```

Q47: Flatten a nested array

Flattening an array involves converting a nested array structure into a single-level array. Your task is to write a function that flattens an array of any depth

- Handle arrays nested at any depth
- Maintain the order of elements
- Ensure non-array elements are preserved
- Avoid using built-in methods like Array.prototype.flat()

```
const nested = [1, [2, [3, 4], 5], 6];
console.log(flattenArray(nested));
// Expected output: [1, 2, 3, 4, 5, 6]
```

Q48: Remove duplicates from an array in JavaScript

Removing duplicates from an array ensures that each element is unique, which is essential for various data processing tasks. Your task is to implement a function that removes duplicate values from an array

- Maintain the original order of elements
- Handle different data types within the array

By utilizing ES6 features like Set, you can remove duplicates efficiently while keeping your code concise

```
// Usage Example:
const arrayWithDuplicates = [1, 2, 2, 3, 4, 4, 5];
console.log(removeDuplicates(arrayWithDuplicates)); // Expected output: [1, 2, 3, 4, 5]
```

Q49: Implement a memoize function in JavaScript

Memoization is an optimization technique that caches the results of expensive function calls and returns the cached result when the same inputs occur again. Your task is to write a function that memoizes another function

- Cache results based on function arguments
- Handle functions with multiple arguments
- Maintain the correct context when invoking the original function
- Ensure the cache doesn't grow indefinitely

```
const slowFunction = (num) => {
// Simulate a slow computation
 for(let i = 0; i < 1e9; i++) {}
 return num * num;
};

const fastFunction = memoize(slowFunction);

console.time('First Call');
console.log(fastFunction(5)); // Computation happens
console.timeEnd('First Call');

console.time('Second Call');
console.log(fastFunction(5)); // Cached result
console.timeEnd('Second Call');
```

```
// Expected output:
// [After some delay]
// 25
// First Call: [some time]ms
// 25
// Second Call: [much less time]ms
```

Q50: Implement a function to get unique elements from an array

Extracting unique elements from an array can be done using various methods such as using a `Set`, filtering, or using an object to track occurrences. Your task is to implement a function that returns a new array containing only the unique elements from the original array

- Use modern JavaScript features like `Set` or `Array.prototype.filter()`
- Ensure the original array is not mutated
- Handle arrays containing primitive data types
- Maintain the order of elements as they first appear

```
const numbers = [1, 2, 2, 3, 4, 4, 5];
console.log(getUniqueElements(numbers)); // Expected output: [1, 2, 3, 4, 5]
```

Q51: Implement a function to deep compare two objects

Deep comparison checks whether two objects have the same properties and values, including nested objects and arrays. Your task is to write a function that performs a deep equality check between two objects

- Compare primitive values directly
- Recursively compare nested objects and arrays
- Ensure both objects have the same set of keys
- Handle edge cases like null and undefined

```
const objA = { a: 1, b: { c: 2 } };
const objB = { a: 1, b: { c: 2 } };
const objC = { a: 1, b: { c: 3 } };

console.log(deepEqual(objA, objB)); // Expected output: true
console.log(deepEqual(objA, objC)); // Expected output: false
```

Q52: Implement a function to find the intersection of two arrays

Finding the intersection of two arrays involves identifying elements that are present in both arrays. Your task is to write a function that returns a new array containing the common elements

- Use efficient data structures like `Set` for lookups
- Handle arrays containing primitive data types
- Ensure the result contains unique elements
- Maintain the order based on the first array

```
const array1 = [1, 2, 3, 4, 5];
const array2 = [3, 4, 5, 6, 7];
console.log(intersection(array1, array2)); // Expected output: [3, 4, 5]
```

Q53: Implement a function to find all anagrams of a word within a list of words

Finding all anagrams of a word involves identifying words that contain the same letters in a different order as the target word. Your task is to write a function that returns all anagrams of a given word from a list of words

- Ensure case-insensitive comparison
- Handle duplicate words in the list
- Maintain the original order of words
- Optimize for performance with large datasets

By sorting the characters of both the target word and each word in the list, you can easily compare them to find anagrams. This method relies on the fact that anagrams will result in the same sorted string

```
const word = 'listen';
const words = ['enlist', 'google', 'inlets', 'banana'];
console.log(findAnagrams(word, words)); // Expected output: ['enlist', 'inlets']
```

Q54: Implement prototype-based inheritance in JavaScript without using classes

JavaScript uses prototype-based inheritance, allowing objects to inherit properties and methods from other objects without the need for classes

Your task is to implement prototype-based inheritance using constructor functions or factory functions. Create a `Person` object and an `Employee` object that inherits from `Person`

- Create constructor or factory functions for object creation
- Set up the prototype chain correctly
- Ensure that inherited methods are accessible to instances

By correctly setting up the prototypes, you can enable inheritance of properties and methods between objects

```
// Usage Example:
const person = createPerson('John');
person.greet(); // Expected output: "Hello, my name is John"

const employee = createEmployee('Alice', 'Developer');
employee.greet(); // Expected output: "Hello, my name is Alice"
employee.work();  // Expected output: "Alice is working as a Developer"
```

Q55: Implement a function to rotate an array to the right by `k` steps

Rotating an array to the right by `k` steps moves each element to the right by `k` positions, with elements wrapping around to the beginning

Your task is to implement a function that rotates an array to the right by `k` steps

- Handle cases where `k` is larger than the array length
- Ensure the rotation is performed in-place
- Maintain the relative order of elements after rotation

This approach reverses the entire array, then reverses the first `k` elements and the remaining elements separately to achieve the desired rotation

```
let nums = [1,2,3,4,5,6,7];
rotate(nums, 3);
console.log(nums); // Expected output: [5,6,7,1,2,3,4]
```

Q56: Implement a function to count the number of islands in a 2D grid

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. Your task is to implement a function that counts the number of islands in a given 2D grid

- Use a graph traversal algorithm like Depth-First Search (DFS) or Breadth-First Search (BFS)
- Efficiently explore connected lands
- Mark visited positions to avoid revisiting

```
const grid = [
["1","1","1","1","0"],
["1","1","0","1","0"],
["1","1","0","0","0"],
["0","0","0","0","0"]
];
console.log(numIslands(grid)); // Expected output: 1
```

Q57: Implement a function to find the longest consecutive sequence in an array

The longest consecutive sequence is the longest span of integers that appear in the array with no gaps. Your task is to implement a function that finds the length of the longest consecutive elements sequence

- Optimize for O(n) time complexity
- Handle unsorted arrays
- Ensure the sequence is consecutive without duplicates

This solution uses a `Set` to achieve constant-time lookups and efficiently finds the longest streak by only starting from the beginning of potential sequences

```
console.log(longestConsecutive([100,4,200,1,3,2])); // Expected output: 4 (sequence: [1,2,3,4])
```

Q58: Implement a function to find the first unique character in a string

Finding the first unique character involves identifying the first character in a string that appears only once. Your task is to implement a function that returns the index of the first non-repeating character in a given string. If no such character exists, return -1

- Efficiently count character occurrences
- Determine the order of characters
- Handle cases where there is no unique character

```
console.log(firstUniqueChar('leetcode')); // Expected output: 0 ('l')
```

Q59: Implement a function to fetch data using Promises and async/await in JavaScript

Fetching data asynchronously is essential for modern web applications. Promises and `async/await` syntax provide a clean and manageable way to handle asynchronous operations in JavaScript

Your task is to implement a function that fetches data from a given URL using both Promises and `async/await`. The function should handle successful data retrieval and errors that may occur during the fetch process

- Use the Fetch API to make HTTP requests
- Handle responses and parse JSON data
- Implement error handling using `try...catch` blocks

- Utilize arrow functions and modern JavaScript features

```
// Usage Example with Promises:
fetchData('https://jsonplaceholder.typicode.com/posts/1')
.then(data => console.log('Data with Promises:', data))
.catch(error => console.error('Error with Promises:', error));

// Usage Example with async/await:
(async () => {
 try {
 const data = await fetchDataAsync('https://jsonplaceholder.typicode.com/posts/1');
 console.log('Data with async/await:', data);
} catch (error) {
 console.error('Error with async/await:', error);
}
})();
```

Q60: Implement a function to find the maximum depth of a binary tree

Finding the maximum depth of a binary tree involves determining the longest path from the root node down to the farthest leaf node. Your task is to implement a function that calculates the maximum depth of a given binary tree

- Use a recursive approach to explore each subtree
- Handle empty trees correctly by returning 0
- Ensure that each node is only visited once

```
class TreeNode {
 constructor(val, left = null, right = null) {
 this.val = val;
 this.left = left;
 this.right = right;
}
}

// Usage Example:
const root = new TreeNode(1,
 new TreeNode(2,
 new TreeNode(4),
 new TreeNode(5)
),
 new TreeNode(3)
);

console.log(maxDepth(root)); // Expected output: 3
```

Q61: Implement a function to serialize and deserialize a binary tree

Serialization converts a binary tree into a string representation, and deserialization reconstructs the binary tree from the string

Consider how to handle null nodes to ensure the tree structure is accurately preserved during serialization and deserialization

- Use pre-order traversal for serialization
- Represent null nodes with a placeholder (e.g., '#')
- Split the serialized string appropriately during deserialization

This approach ensures that both the structure and the values of the tree nodes are preserved, allowing for accurate reconstruction of the original tree

```
class TreeNode {
 constructor(val, left = null, right = null) {
 this.val = val;
 this.left = left;
 this.right = right;
}
}

const root = new TreeNode(1,
 new TreeNode(2),
 new TreeNode(3, new TreeNode(4), new TreeNode(5))
);

const serialized = serialize(root);
console.log('Serialized:', serialized);
// Expected output: "1,2,#,#,3,4,#,#,5,#,#"
```

Q62: Implement a function to find the maximum product of three numbers in an array

Finding the maximum product of three numbers in an array involves identifying the combination of three numbers that produces the highest possible product. Your task is to implement a function that finds the maximum product of any three numbers in a given array of integers

Consider both positive and negative numbers in the array, as negative numbers can impact the maximum product when multiplied together

- Sort the array to access elements easily
- Consider the product of the three largest numbers
- Consider the product of the two smallest numbers (possibly negative) and the largest number

```
// Usage Example:
console.log(maximumProduct([1,2,3]));          // Expected output: 6
console.log(maximumProduct([1,2,3,4]));        // Expected output: 24
console.log(maximumProduct([-10,-10,5,2]));    // Expected output: 500
```

Q63: Implement a function to find the longest common prefix among an array of strings

Finding the longest common prefix involves identifying the shared starting substring among all strings in an array. Your task is to implement a function that returns the longest common prefix string among an array of strings

If there is no common prefix, the function should return an empty string

- Handle cases with empty arrays
- Consider varying string lengths

By comparing characters of the first and last strings after sorting, you can efficiently find the longest common prefix

```
// Usage Example:
console.log(longestCommonPrefix(['flower','flow','flight'])); // Expected output: 'fl'
console.log(longestCommonPrefix(['dog','racecar','car']));    // Expected output: ''
```

Q64: Implement a regular expression to validate email addresses in JavaScript

Validating email addresses is a common requirement in web development. Your task is to implement a function that uses a regular expression to check if a given string is a valid email address

- Match the local part before the '@' symbol
- Validate the domain part after the '@' symbol
- Handle subdomains and different domain extensions
- Ensure that special characters are handled appropriately

```
// Usage Example:
console.log(isValidEmail('test@example.com'));                 // Expected output: true
console.log(isValidEmail('invalid-email@.com'));               // Expected output: false
console.log(isValidEmail('user.name+tag+sorting@example.com')); // Expected output: true
```

Q65: Merge two sorted arrays in JavaScript

Your task is to implement a function that merges two sorted arrays into one sorted array

Ensure that the merged array maintains the sorted order, and consider how to handle arrays of different lengths

- Maintain the sorted order of the merged array
- Handle arrays of different lengths

```
// Usage Example:
const array1 = [1, 3, 5];
const array2 = [2, 4, 6];
console.log(mergeSortedArrays(array1, array2)); // Expected output: [1, 2, 3, 4, 5, 6]
```

Q66: Implement a function to find the longest substring with at most two distinct characters

Finding the longest substring with at most two distinct characters involves maintaining a substring where no more than two unique characters are present

Your task is to implement a function that returns the length of the longest such substring in a given string

- Use a sliding window approach to efficiently traverse the string
- Keep track of character counts within the window
- Adjust the window when more than two distinct characters are present

This method ensures optimal time complexity by avoiding unnecessary re-computation

```
// Usage Example:
console.log(lengthOfLongestSubstringTwoDistinct('eceba'));   // Expected output: 3 ('ece')
console.log(lengthOfLongestSubstringTwoDistinct('ccaabbb')); // Expected output: 5 ('aabbb')
```

Q67: Find the most common integer in a list, returning the smallest in case of a tie in JavaScript

Identifying the most common integer in a list helps in understanding data distributions and frequency analysis. Your task is to implement a function that finds the most common integer (mode) in an array of integers

If multiple integers occur the same maximum number of times, the function should return the smallest one among them

- Handle lists with multiple modes (ties)
- Return the smallest integer among tied candidates

```
// Usage Example:
const numbers = [4, 1, 2, 2, 3, 3];
console.log(mostCommonNumber(numbers)); // Expected output: 2
```

Q68: Implement a function to flatten a deeply nested object in JavaScript

Flattening a deeply nested object transforms it into a single-level object with dot-separated keys representing the original structure. Your task is to implement a function that flattens a nested object

- Handle nested objects and arrays
- Use recursion to traverse all levels of nesting
- Generate keys that accurately represent the original structure
- Avoid modifying the original object

```
// Usage Example:
const nestedObj = {
 a: {
 b: {
 c: 1,
},
 d: [2, 3],
},
 e: 4,
};

console.log(flattenObject(nestedObj));
// Expected output:
// {
//   'a.b.c': 1,
//   'a.d.0': 2,
//   'a.d.1': 3,
//   'e': 4
// }
```

Q69: Implement a function to check if a number is prime

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself

Your task is to implement a function that checks whether a given number is prime

- Handle edge cases where n is less than or equal to 1
- Check divisibility up to the square root of n

- Optimize by checking only necessary factors

This function uses the 6k ± 1 optimization to efficiently check for factors

```
// Usage Example:
console.log(isPrime(11)); // Expected output: true
console.log(isPrime(15)); // Expected output: false
```

Q70: Reverse a singly linked list in JavaScript

Reversing a linked list is a fundamental operation that involves rearranging the pointers of the nodes so that the list is traversed in the opposite direction. Your task is to implement a function that reverses a singly linked list

- Iterate through the linked list only once
- Reverse the links between the nodes
- Handle edge cases like empty lists or single-node lists

```
// Usage Example:
const list = createLinkedList([1, 2, 3, 4, 5]);
console.log(printList(list)); // Expected output: 1 -> 2 -> 3 -> 4 -> 5 -> null
const reversedList = reverseLinkedList(list);
console.log(printList(reversedList)); // Expected output: 5 -> 4 -> 3 -> 2 -> 1 -> null
```

Q71: Calculate the factorial of a number using recursion in JavaScript

Calculating the factorial of a number using recursion demonstrates the power of self-referential functions in programming. Your task is to implement a recursive function that calculates the factorial of a non-negative integer

- Handle edge cases like zero and negative numbers
- Use recursion effectively
- Ensure that the function returns accurate results

```
// Usage Example:
console.log(factorial(5)); // Expected output: 120
console.log(factorial(0)); // Expected output: 1
```