JAVA

INTERVIEW QUESTIONS

1. **What is Java? Explain its features.**
2. **What are the main principles of Object-Oriented Programming (OOP)?**
3. **Differentiate between JDK, JRE, and JVM.**
4. **Explain the concept of platform independence in Java.**
5. **What is the significance of the `main` method in Java?**
6. **How does Java achieve memory management?**
7. **What are constructors in Java? How are they different from methods?**
8. **Explain method overloading and method overriding with examples.**
9. **What is inheritance in Java? Discuss its types.**
10. **Define polymorphism and its types in Java.**
11. **What is an interface in Java, and how does it differ from an abstract class?**
12. **Describe the access modifiers in Java.**
13. **What is encapsulation? How is it implemented in Java?**
14. **Explain the concept of packages in Java.**
15. **What are static variables and methods? Provide examples.**
16. **Discuss the lifecycle of a thread in Java.**
17. **What is exception handling? How is it implemented in Java?**
18. **Differentiate between `throw` and `throws` keywords.**
19. **What are checked and unchecked exceptions? Give examples.**
20. **Explain the concept of synchronization in Java.**
21. **What is the Java Collections Framework? Name its main interfaces.**
22. **Differentiate between `ArrayList` and `LinkedList`.**
23. **What is a `HashMap`? How does it work internally?**
24. **Explain the significance of the `equals()` and `hashCode()` methods.**
25. **What is the difference between `Comparable` and `Comparator` interfaces?**
26. **Describe the Java Memory Model (JMM).**
27. **What is garbage collection in Java? How does it work?**
28. **Explain the concept of Java annotations.**
29. **What are lambda expressions? Provide a use case.**
30. **Discuss the Stream API in Java.**

31. **What is the purpose of the `Optional` class?**
32. **Explain the `try-with-resources` statement.**
33. **What is the difference between `final`, `finally`, and `finalize()`?**
34. **How does the `volatile` keyword affect thread behavior?**
35. **What are design patterns? Name a few commonly used ones in Java.**
36. **Explain the Singleton design pattern and its implementation.**
37. **What is JDBC? How is it used in Java applications?**
38. **Discuss the differences between `Statement` and `PreparedStatement`.**
39. **What is the purpose of the `transient` keyword?**
40. **Explain serialization and deserialization in Java.**
41. **What are inner classes? Differentiate between static and non-static inner classes.**
42. **Describe the use of the `synchronized` keyword.**
43. **What is the difference between `String`, `StringBuilder`, and `StringBuffer`?**
44. **Explain the concept of immutability in Java.**
45. **How does Java handle memory leaks?**
46. **What are functional interfaces? Provide examples.**
47. **Discuss the role of the `default` keyword in interfaces.**
48. **What is the `enum` type in Java? How is it used?**
49. **Explain the concept of reflection in Java.**
50. **What are modules in Java? Discuss their significance.**

ANSWERS

1. What is Java? Explain its features.

Java is a high-level, object-oriented programming language developed by Sun Microsystems (now Oracle) in 1995.
Key features:

- **Platform Independent**: Write Once, Run Anywhere (WORA).
- **Object-Oriented**: Follows OOP principles like encapsulation and inheritance.
- **Robust**: Strong memory management and exception handling.
- **Multithreaded**: Supports concurrent execution of threads.
- **Secure**: No explicit pointers and runs in a virtual machine.

2. What are the main principles of Object-Oriented Programming (OOP)?

1. **Encapsulation**: Wrapping data and methods in a single unit (class).
2. **Abstraction**: Hiding implementation details and showing only the functionality.
3. **Inheritance**: Allowing a class to inherit properties and methods from another class.
4. **Polymorphism**: Using a single interface to represent different forms (overloading and overriding).

3. Differentiate between JDK, JRE, and JVM.

- **JDK (Java Development Kit)**: Provides tools for development (compiler, debugger).
- **JRE (Java Runtime Environment)**: Includes libraries and JVM for running Java applications.
- **JVM (Java Virtual Machine)**: Converts bytecode into machine code and executes it.

4. Explain the concept of platform independence in Java.

Java programs are compiled into **bytecode**, which is platform-independent. Bytecode is executed by the JVM, which is platform-specific, ensuring the same Java program runs on any OS with a compatible JVM.

5. What is the significance of the `main` method in Java?

The `main` method is the entry point of a Java application. Its signature is:

```
java
CopyEdit
public static void main(String[] args)
```

- **`public`**: Accessible globally.
- **`static`**: Allows the JVM to call it without object instantiation.
- **`void`**: Returns no value.
- **`String[] args`**: Accepts command-line arguments.

6. How does Java achieve memory management?

Java uses **automatic garbage collection** to manage memory. Objects are allocated in the heap memory, and when they are no longer referenced, the garbage collector deallocates them.

7. What are constructors in Java? How are they different from methods?

- **Constructors**: Special methods to initialize objects.
  - Name matches the class.
  - No return type.
- **Difference from methods**: Methods perform actions; constructors initialize objects.

8. Explain method overloading and method overriding with examples.

- **Overloading**: Same method name, different parameters (compile-time polymorphism).

```
java
CopyEdit
class Example {
    void display(int a) { }
    void display(String b) { }
}
```

- **Overriding**: Subclass provides a new implementation for a method in the superclass (runtime polymorphism).

```
java
CopyEdit
class Parent {
    void display() { }
}
class Child extends Parent {
    @Override
    void display() { }
}
```

9. What is inheritance in Java? Discuss its types.

Inheritance allows a class to acquire the properties and methods of another class using the `extends` keyword. Types:

1. **Single**: One class inherits from another.
2. **Multilevel**: A chain of inheritance.
3. **Hierarchical**: Multiple classes inherit from one superclass.
4. **Multiple (via interfaces)**: A class implements multiple interfaces.

10. Define polymorphism and its types in Java.

Polymorphism allows methods to perform different tasks based on the object. Types:

1. **Compile-time (Method Overloading)**.
2. **Runtime (Method Overriding)**.

11. What is an interface in Java, and how does it differ from an abstract class?

- **Interface**: A collection of abstract methods and static constants.
  - Can have default and static methods (since Java 8).

- A class can implement multiple interfaces.

Difference:

- Abstract class can have both abstract and concrete methods; an interface has abstract methods by default (Java 7 and below).
- A class extends one abstract class but can implement multiple interfaces.

12. Describe the access modifiers in Java.

- **Public**: Accessible everywhere.
- **Protected**: Accessible within the same package and subclasses.
- **Default**: Accessible within the same package only.
- **Private**: Accessible within the same class only.

13. What is encapsulation? How is it implemented in Java?

Encapsulation is bundling data (variables) and methods into a single unit (class). It's implemented using:

1. Private access modifiers for fields.
2. Public getter and setter methods for access.

14. Explain the concept of packages in Java.

Packages are namespaces used to group related classes and interfaces. They help avoid name conflicts and improve organization.

15. What are static variables and methods? Provide examples.

- **Static Variable**: Belongs to the class, shared by all objects.
- **Static Method**: Can be called without creating an object of the class.

```
java
CopyEdit
class Example {
    static int count = 0;  // Static variable
    static void display() {  // Static method
        System.out.println("Count: " + count);
    }
•
}
```

16. Discuss the lifecycle of a thread in Java.

1. **New**: Thread is created.
2. **Runnable**: Thread is ready to run.
3. **Running**: Thread is executing.
4. **Blocked/Waiting**: Thread is waiting for a resource.
5. **Terminated**: Thread execution is complete.

17. What is exception handling? How is it implemented in Java?

Exception handling manages runtime errors using `try`, `catch`, `throw`, `throws`, and `finally`.

18. Differentiate between `throw` and `throws` keywords.

- **`throw`**: Used to explicitly throw an exception.
- **`throws`**: Declares exceptions a method might throw.

19. What are checked and unchecked exceptions?

- **Checked**: Checked at compile-time (e.g., `IOException`).
- **Unchecked**: Occur at runtime (e.g., `NullPointerException`).

20. Explain the concept of synchronization in Java.

Synchronization prevents thread interference by allowing only one thread to access a critical section at a time, using the `synchronized` keyword.

21. What is the Java Collections Framework?

A unified architecture for storing and manipulating groups of objects, including interfaces like `List`, `Set`, and `Map`.

22. Differentiate between `ArrayList` and `LinkedList`.

- **ArrayList**: Backed by a dynamic array, faster for indexing.
- **LinkedList**: Backed by a doubly-linked list, better for insertions/deletions.

23. What is a `HashMap`? How does it work internally?

`HashMap` stores key-value pairs using a hash table. Keys are hashed to determine the index, and collisions are handled using linked lists or trees.

24. Explain the significance of the `equals()` and `hashCode()` methods.

- **`equals()`**: Checks logical equality.
- **`hashCode()`**: Provides a unique hash for an object, used in hash-based collections like `HashMap`.

25. What is the difference between `Comparable` and `Comparator`?

- **Comparable**: Used to define natural ordering.
- **Comparator**: Defines custom ordering.

26. Describe the Java Memory Model (JMM).

Defines how threads interact through memory, ensuring visibility and ordering of variable accesses.

27. What is garbage collection in Java? How does it work?

Garbage collection automatically deallocates memory for objects no longer in use, reclaiming memory in the heap.

28. Explain the concept of Java annotations.

Annotations provide metadata about code, such as `@Override`, `@Deprecated`, and custom annotations.

29. What are lambda expressions? Provide a use case.

Lambda expressions provide a concise way to implement functional interfaces.
Example:

```
java
CopyEdit
List<Integer> list = Arrays.asList(1, 2, 3);
list.forEach(n -> System.out.println(n));
```

30. Discuss the Stream API in Java.

The Stream API processes collections of objects in a functional style, supporting operations like `filter`, `map`, and `reduce`.

31. What is the purpose of the `Optional` class?

`Optional` prevents `NullPointerException` by representing optional values.

32. Explain the `try-with-resources` statement.

Manages resources (like files) automatically, ensuring they are closed after use.
Example:

```
java
CopyEdit
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    // Read file
}
```

33. What is the difference between `final`, `finally`, and `finalize()`?

- **`final`**: Prevents modification of variables, methods, or classes.
- **`finally`**: Ensures execution of code after a `try-catch`.
- **`finalize()`**: Called by the garbage collector before destroying an object.

34. How does the `volatile` keyword affect thread behavior?

Ensures visibility of changes to a variable across threads, preventing caching.

35. What are design patterns?

Design patterns are reusable solutions to common software design problems. Examples: Singleton, Factory, Observer.

36. Explain the Singleton design pattern.

Restricts a class to one instance and provides a global access point to it.

```
java
CopyEdit
class Singleton {
    private static Singleton instance;
    private Singleton() { }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

37. What is JDBC? How is it used?

JDBC (Java Database Connectivity) is an API for connecting to databases.
Steps:

1. Load driver.
2. Establish connection.
3. Execute SQL queries.
4. Close connection.

38. Discuss the differences between `Statement` and `PreparedStatement`.

- **Statement**: Used for static queries.
- **PreparedStatement**: Precompiled and supports dynamic queries.

39. What is the purpose of the `transient` keyword?

Excludes fields from serialization.

40. Explain serialization and deserialization.

- **Serialization**: Converts an object to a byte stream.
- **Deserialization**: Converts a byte stream back to an object.

41. What are inner classes?

Classes defined within another class. Types: static, non-static, local, and anonymous.

42. Describe the use of the `synchronized` keyword.

Locks a block/method to allow only one thread access at a time.

43. What is the difference between `String`, `StringBuilder`, and `StringBuffer`?

- **String**: Immutable.
- **StringBuilder**: Mutable, non-thread-safe.
- **StringBuffer**: Mutable, thread-safe.

44. Explain the concept of immutability in Java.

Immutable objects cannot be modified after creation, e.g., `String`.

45. How does Java handle memory leaks?

Java uses garbage collection but memory leaks can occur if references to unused objects are maintained.

46. What are functional interfaces?

Interfaces with a single abstract method, e.g., `Runnable`.

47. Discuss the role of the `default` keyword in interfaces.

Allows adding methods to interfaces without breaking existing implementations.

48. What is the `enum` type in Java?

Used to define a set of named constants.
Example:

```
java
CopyEdit
enum Day { MONDAY, TUESDAY }
```

49. Explain the concept of reflection in Java.

Allows inspection and modification of classes, methods, and fields at runtime.

50. What are modules in Java?

Introduced in Java 9, modules allow better packaging, encapsulation, and dependency management.

CODING QUESTIONS

1. **Two Sum: Given an array of integers, find two numbers that add up to a specific target.**
2. **Reverse a String**: Write a function to reverse a string without using built-in functions.
3. **Palindrome Check**: Determine if a given string is a palindrome.
4. **Merge Two Sorted Lists**: Merge two sorted linked lists and return it as a new sorted list.
5. **Longest Substring Without Repeating Characters**: Find the length of the longest substring without repeating characters.
6. **Valid Parentheses**: Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
7. **Search in Rotated Sorted Array**: Search for a target value in a rotated sorted array.
8. **Container With Most Water**: Given n non-negative integers, find two lines that together with the x-axis form a container, such that the container contains the most water.
9. **3Sum**: Find all unique triplets in the array which gives the sum of zero.
10. **Remove Nth Node From End of List**: Remove the n-th node from the end of a linked list and return its head.
11. **Maximum Subarray**: Find the contiguous subarray with the largest sum.
12. **Climbing Stairs**: You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?
13. **Set Matrix Zeroes**: Given a m x n matrix, if an element is 0, set its entire row and column to 0.
14. **Group Anagrams**: Given an array of strings, group anagrams together.
15. **Merge Intervals**: Given a collection of intervals, merge all overlapping intervals.
16. **Linked List Cycle**: Given a linked list, determine if it has a cycle in it.
17. **Implement Stack using Queues**: Implement a last-in-first-out (LIFO) stack using only two queues.
18. **Minimum Window Substring**: Given two strings s and t, find the minimum window in s which will contain all the characters in t.
19. **Word Search**: Given a 2D board and a word, find if the word exists in the grid.
20. **Longest Increasing Subsequence**: Find the length of the longest increasing subsequence in an array.
21. **Decode Ways**: A message containing letters from A-Z is encoded to numbers using 'A' -> 1, 'B' -> 2, ..., 'Z' -> 26. Given an encoded message, determine the total number of ways to decode it.
22. **Coin Change**: Given coins of different denominations and a total amount of money, find the fewest number of coins needed to make up that amount.
23. **House Robber**: Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

24. **Binary Tree Inorder Traversal**: Given a binary tree, return the inorder traversal of its nodes' values.
25. **Validate Binary Search Tree**: Determine if a given binary tree is a valid binary search tree.
26. **Lowest Common Ancestor of a Binary Tree**: Given a binary tree, find the lowest common ancestor of two given nodes in the tree.
27. **Serialize and Deserialize Binary Tree**: Design an algorithm to serialize and deserialize a binary tree.
28. **Kth Smallest Element in a BST**: Find the kth smallest element in a binary search tree.
29. **Number of Islands**: Given a 2D grid of '1's (land) and '0's (water), count the number of islands.
30. **Course Schedule**: There are a total of numCourses you have to take, labeled from 0 to numCourses-1. Some courses may have prerequisites. Determine if you can finish all courses.
31. **Implement Trie (Prefix Tree)**: Implement a trie with insert, search, and startsWith methods.
32. **Add and Search Word - Data structure design**: Design a data structure that supports the addition of words and the search for a word in a dictionary.
33. **Word Ladder**: Given two words (beginWord and endWord), and a dictionary's word list, find the length of the shortest transformation sequence from beginWord to endWord.
34. **Find Median from Data Stream**: The median is the middle value in an ordered integer list. Write a program that finds the median of input data stream.
35. **Sliding Window Maximum**: Given an array and an integer k, find the maximum for each sliding window of size k.
36. **Longest Consecutive Sequence**: Given an unsorted array of integers, find the length of the longest consecutive elements sequence.
37. **Graph Valid Tree**: Given n nodes labeled from 0 to n-1 and a list of undirected edges, determine if these edges form a valid tree.
38. Number of Connected Components in an Undirected Graph

PROGRAMS

1. Two Sum

```
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
            return new int[] { map.get(complement), i };
        }
        map.put(nums[i], i);
    }
    return new int[] {};
}
```

2. Reverse a String

```
public String reverseString(String s) {
    return new StringBuilder(s).reverse().toString();
}
```

3. Palindrome Check

```
public boolean isPalindrome(String s) {
    int left = 0, right = s.length() - 1;
    while (left < right) {
        if (s.charAt(left++) != s.charAt(right--)) return false;
    }
    return true;
}
```

4. Merge Two Sorted Lists

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    if (l1 == null) return l2;
    if (l2 == null) return l1;
    if (l1.val < l2.val) {
        l1.next = mergeTwoLists(l1.next, l2);
        return l1;
    } else {
        l2.next = mergeTwoLists(l1, l2.next);
        return l2;
    }
}
```

5. Longest Substring Without Repeating Characters

```
public int lengthOfLongestSubstring(String s) {
    Set<Character> set = new HashSet<>();
    int left = 0, maxLen = 0;
    for (int right = 0; right < s.length(); right++) {
        while (set.contains(s.charAt(right))) {
```

```
            set.remove(s.charAt(left++));
        }
        set.add(s.charAt(right));
        maxLen = Math.max(maxLen, right - left + 1);
    }
    return maxLen;
}
```

6. Valid Parentheses

```
public boolean isValid(String s) {
    Stack<Character> stack = new Stack<>();
    for (char c : s.toCharArray()) {
        if (c == '(' || c == '{' || c == '[') {
            stack.push(c);
        } else if (!stack.isEmpty() &&
                   ((c == ')' && stack.peek() == '(') ||
                    (c == '}' && stack.peek() == '{') ||
                    (c == ']' && stack.peek() == '['))) {
            stack.pop();
        } else {
            return false;
        }
    }
    return stack.isEmpty();
}
```

7. Search in Rotated Sorted Array

```
public int search(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = (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;
}
```

8. Container With Most Water

```
public int maxArea(int[] height) {
    int left = 0, right = height.length - 1, max = 0;
    while (left < right) {
        max = Math.max(max, Math.min(height[left], height[right]) * (right
- left));
        if (height[left] < height[right]) left++;
        else right--;
    }
    return max;
}
```

9. 3Sum

```
public List<List<Integer>> threeSum(int[] nums) {
    Arrays.sort(nums);
    List<List<Integer>> result = new ArrayList<>();
    for (int i = 0; i < nums.length - 2; i++) {
        if (i > 0 && nums[i] == nums[i - 1]) continue;
        int left = i + 1, right = nums.length - 1;
        while (left < right) {
            int sum = nums[i] + nums[left] + nums[right];
            if (sum == 0) {
                result.add(Arrays.asList(nums[i], nums[left++],
nums[right--]));
                while (left < right && nums[left] == nums[left - 1]) left+
+;
                while (left < right && nums[right] == nums[right + 1])
right--;
            } else if (sum < 0) left++;
            else right--;
        }
    }
    return result;
}
```

10. Remove Nth Node From End of List

```
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode slow = dummy, fast = dummy;
    for (int 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;
}
```

11. Maximum Subarray

```
public int maxSubArray(int[] nums) {
    int max = nums[0], currentSum = nums[0];
    for (int i = 1; i < nums.length; i++) {
        currentSum = Math.max(nums[i], currentSum + nums[i]);
        max = Math.max(max, currentSum);
    }
    return max;
}
```

12. Climbing Stairs

```
public int climbStairs(int n) {
    if (n <= 2) return n;
    int first = 1, second = 2;
    for (int i = 3; i <= n; i++) {
        int third = first + second;
```

```
        first = second;
        second = third;
    }
    return second;
}
```

13. Set Matrix Zeroes

```
public void setZeroes(int[][] matrix) {
    boolean firstRow = false, firstCol = false;
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[0].length; j++) {
            if (matrix[i][j] == 0) {
                if (i == 0) firstRow = true;
                if (j == 0) firstCol = true;
                matrix[i][0] = 0;
                matrix[0][j] = 0;
            }
        }
    }
    for (int i = 1; i < matrix.length; i++) {
        for (int j = 1; j < matrix[0].length; j++) {
            if (matrix[i][0] == 0 || matrix[0][j] == 0) matrix[i][j] = 0;
        }
    }
    if (firstRow) Arrays.fill(matrix[0], 0);
    if (firstCol) for (int i = 0; i < matrix.length; i++) matrix[i][0] = 0;
}
```

14. Group Anagrams

```
public List<List<String>> groupAnagrams(String[] strs) {
    Map<String, List<String>> map = new HashMap<>();
    for (String s : strs) {
        char[] chars = s.toCharArray();
        Arrays.sort(chars);
        String key = new String(chars);
        map.putIfAbsent(key, new ArrayList<>());
        map.get(key).add(s);
    }
    return new ArrayList<>(map.values());
}
```

15. Merge Intervals

```
public int[][] merge(int[][] intervals) {
    Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
    List<int[]> merged = new ArrayList<>();
    for (int[] interval : intervals) {
        if (merged.isEmpty() || merged.get(merged.size() - 1)[1] <
interval[0]) {
            merged.add(interval);
        } else {
            merged.get(merged.size() - 1)[1] =
Math.max(merged.get(merged.size() - 1)[1], interval[1]);
        }
    }
    return merged.toArray(new int[merged.size()][]);
}
```

16. Linked List Cycle

```
public boolean hasCycle(ListNode head) {
    if (head == null || head.next == null) return false;
    ListNode slow = head, fast = head.next;
    while (slow != fast) {
        if (fast == null || fast.next == null) return false;
        slow = slow.next;
        fast = fast.next.next;
    }
    return true;
}
```

17. Implement Stack using Queues

```
class MyStack {
    Queue<Integer> queue = new LinkedList<>();

    public void push(int x) {
        queue.add(x);
        for (int i = 1; i < queue.size(); i++) {
            queue.add(queue.poll());
        }
    }

    public int pop() {
        return queue.poll();
    }

    public int top() {
        return queue.peek();
    }

    public boolean empty() {
        return queue.isEmpty();
    }
}
```

18. Minimum Window Substring

```
public String minWindow(String s, String t) {
    if (s.length() < t.length()) return "";
    Map<Character, Integer> map = new HashMap<>();
    for (char c : t.toCharArray()) map.put(c, map.getOrDefault(c, 0) + 1);
    int left = 0, count = 0, minLen = Integer.MAX_VALUE, start = 0;
    for (int right = 0; right < s.length(); right++) {
        char c = s.charAt(right);
        if (map.containsKey(c)) {
            map.put(c, map.get(c) - 1);
            if (map.get(c) >= 0) count++;
        }
        while (count == t.length()) {
            if (right - left + 1 < minLen) {
                minLen = right - left + 1;
                start = left;
            }
            char lc = s.charAt(left++);
            if (map.containsKey(lc)) {
```

```
                map.put(lc, map.get(lc) + 1);
                if (map.get(lc) > 0) count--;
            }
        }
    }
    return minLen == Integer.MAX_VALUE ? "" : s.substring(start, start +
minLen);
}
```

19. Word Search

```
public boolean exist(char[][] board, String word) {
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            if (dfs(board, word, i, j, 0)) return true;
        }
    }
    return false;
}

private boolean dfs(char[][] board, String word, int i, int j, int index) {
    if (index == word.length()) return true;
    if (i < 0 || j < 0 || i >= board.length || j >= board[0].length ||
board[i][j] != word.charAt(index)) return false;
    char temp = board[i][j];
    board[i][j] = '#';
    boolean found = dfs(board, word, i + 1, j, index + 1) ||
                    dfs(board, word, i - 1, j, index + 1) ||
                    dfs(board, word, i, j + 1, index + 1) ||
                    dfs(board, word, i, j - 1, index + 1);
    board[i][j] = temp;
    return found;
}
```

29. Number of Islands

```
public int numIslands(char[][] grid) {
    int count = 0;
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[0].length; j++) {
            if (grid[i][j] == '1') {
                count++;
                dfs(grid, i, j);
            }
        }
    }
    return count;
}

private void dfs(char[][] grid, int i, int j) {
    if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length ||
grid[i][j] == '0') return;
    grid[i][j] = '0';
    dfs(grid, i + 1, j);
    dfs(grid, i - 1, j);
    dfs(grid, i, j + 1);
    dfs(grid, i, j - 1);
```

```
}
```

30. Course Schedule

```
public boolean canFinish(int numCourses, int[][] prerequisites) {
    List<List<Integer>> graph = new ArrayList<>();
    for (int i = 0; i < numCourses; i++) graph.add(new ArrayList<>());
    int[] inDegree = new int[numCourses];
    for (int[] prereq : prerequisites) {
        graph.get(prereq[1]).add(prereq[0]);
        inDegree[prereq[0]]++;
    }
    Queue<Integer> queue = new LinkedList<>();
    for (int i = 0; i < numCourses; i++) if (inDegree[i] == 0)
queue.add(i);
    int count = 0;
    while (!queue.isEmpty()) {
        int course = queue.poll();
        count++;
        for (int next : graph.get(course)) {
            if (--inDegree[next] == 0) queue.add(next);
        }
    }
    return count == numCourses;
}
```

31. Implement Trie (Prefix Tree)

```
class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            if (!node.containsKey(c)) node.put(c, new TrieNode());
            node = node.get(c);
        }
        node.setEnd();
    }

    public boolean search(String word) {
        TrieNode node = searchPrefix(word);
        return node != null && node.isEnd();
    }

    public boolean startsWith(String prefix) {
        return searchPrefix(prefix) != null;
    }

    private TrieNode searchPrefix(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            if (node.containsKey(c)) node = node.get(c);
```

```
            else return null;
        }
        return node;
    }
}

class TrieNode {
    private TrieNode[] links;
    private final int R = 26;
    private boolean isEnd;

    public TrieNode() {
        links = new TrieNode[R];
    }

    public boolean containsKey(char ch) {
        return links[ch - 'a'] != null;
    }

    public TrieNode get(char ch) {
        return links[ch - 'a'];
    }

    public void put(char ch, TrieNode node) {
        links[ch - 'a'] = node;
    }

    public void setEnd() {
        isEnd = true;
    }

    public boolean isEnd() {
        return isEnd;
    }
}
```

32. Add and Search Word - Data Structure Design

```
class WordDictionary {
    private TrieNode root;

    public WordDictionary() {
        root = new TrieNode();
    }

    public void addWord(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            if (!node.containsKey(c)) node.put(c, new TrieNode());
            node = node.get(c);
        }
        node.setEnd();
    }

    public boolean search(String word) {
        return search(word, 0, root);
    }

    private boolean search(String word, int index, TrieNode node) {
```

```
        if (index == word.length()) return node.isEnd();
        char c = word.charAt(index);
        if (c == '.') {
            for (char ch = 'a'; ch <= 'z'; ch++) {
                if (node.containsKey(ch) && search(word, index + 1,
node.get(ch))) return true;
            }
            return false;
        } else {
            return node.containsKey(c) && search(word, index + 1,
node.get(c));
        }
    }
}
```

33. Word Ladder

```
public int ladderLength(String beginWord, String endWord, List<String>
wordList) {
    Set<String> wordSet = new HashSet<>(wordList);
    if (!wordSet.contains(endWord)) return 0;
    Queue<String> queue = new LinkedList<>();
    queue.add(beginWord);
    int steps = 1;
    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            String word = queue.poll();
            if (word.equals(endWord)) return steps;
            for (int j = 0; j < word.length(); j++) {
                char[] chars = word.toCharArray();
                for (char c = 'a'; c <= 'z'; c++) {
                    chars[j] = c;
                    String newWord = new String(chars);
                    if (wordSet.contains(newWord)) {
                        queue.add(newWord);
                        wordSet.remove(newWord);
                    }
                }
            }
        }
        steps++;
    }
    return 0;
}
```

34. Find Median from Data Stream

```
class MedianFinder {
    private PriorityQueue<Integer> small = new
PriorityQueue<>(Collections.reverseOrder());
    private PriorityQueue<Integer> large = new PriorityQueue<>();

    public void addNum(int num) {
        small.add(num);
        large.add(small.poll());
        if (small.size() < large.size()) small.add(large.poll());
```

```
    }

    public double findMedian() {
        if (small.size() > large.size()) return small.peek();
        return (small.peek() + large.peek()) / 2.0;
    }
}
```

35. Sliding Window Maximum

```
public int[] maxSlidingWindow(int[] nums, int k) {
    Deque<Integer> deque = new ArrayDeque<>();
    int[] result = new int[nums.length - k + 1];
    for (int i = 0; i < nums.length; i++) {
        if (!deque.isEmpty() && deque.peek() == i - k) deque.poll();
        while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i])
deque.pollLast();
        deque.offer(i);
        if (i >= k - 1) result[i - k + 1] = nums[deque.peek()];
    }
    return result;
}
```

36. Longest Consecutive Sequence

```
public int longestConsecutive(int[] nums) {
    Set<Integer> set = new HashSet<>();
    for (int num : nums) set.add(num);
    int maxStreak = 0;
    for (int num : nums) {
        if (!set.contains(num - 1)) {
            int currentNum = num;
            int streak = 1;
            while (set.contains(currentNum + 1)) {
                currentNum++;
                streak++;
            }
            maxStreak = Math.max(maxStreak, streak);
        }
    }
    return maxStreak;
}
```

37. Graph Valid Tree

```
public boolean validTree(int n, int[][] edges) {
    if (edges.length != n - 1) return false;
    UnionFind uf = new UnionFind(n);
    for (int[] edge : edges) {
        if (!uf.union(edge[0], edge[1])) return false;
    }
    return true;
}

class UnionFind {
```

```
    private int[] parent;
    public UnionFind(int n) {
        parent = new int[n];
        for (int i = 0; i < n; i++) parent[i] = i;
    }
    public int find(int x) {
        if (parent[x] != x) parent[x] = find(parent[x]);
        return parent[x];
    }
    public boolean union(int x, int y) {
        int rootX = find(x), rootY = find(y);
        if (rootX == rootY) return false;
        parent[rootX] = rootY;
        return true;
    }
}
```

38. Number of Connected Components in an Undirected Graph

```
public int countComponents(int n, int[][] edges) {
    UnionFind uf = new UnionFind(n);
    for (int[] edge : edges) uf.union(edge[0], edge[1]);
    Set<Integer> uniqueParents = new HashSet<>();
    for (int i = 0; i < n; i++) uniqueParents.add(uf.find(i));
    return uniqueParents.size();
}
```