Revision Time

Let's break down each topic and provide Brute Force, Optimized, and Most Optimized (if applicable) solutions in Java, along with explanations.

Arrays & Strings

1. Two Sum

Problem: Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

Brute Force:

Iterate through all possible pairs of numbers and check if their sum equals the target.

Java

```java
class Solution {
    public int[] twoSumBruteForce(int[] nums, int target) {
        for (int i = 0; i < nums.length; i++) {
            for (int j = i + 1; j < nums.length; j++) {
                if (nums[i] + nums[j] == target) {
                    return new int[]{i, j};
                }
            }
        }
        return new int[]{}; // No solution found
    }
}
```

- **Time Complexity:** O(N2)
- **Space Complexity:** O(1)

Optimized:

Use a HashMap to store numbers and their indices. For each number, check if `target - current_number` exists in the map.

Java

```java
import java.util.HashMap;
import java.util.Map;
```

```java
class Solution {
    public int[] twoSumOptimized(int[] nums, int target) {
        Map<Integer, Integer> numMap = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (numMap.containsKey(complement)) {
                return new int[]{numMap.get(complement), i};
            }
            numMap.put(nums[i], i);
        }
        return new int[]{}; // No solution found
    }
}
```

- **Time Complexity:** O(N) (average case, due to HashMap operations)
- **Space Complexity:** O(N) (to store elements in the HashMap)

2. Subarray with Given Sum

Problem: Given an array of non-negative integers `arr` and a sum `S`, find a contiguous subarray that adds to `S`.

Brute Force:

Generate all possible subarrays and check their sum.

Java

```java
class Solution {
    public int[] subarraySumBruteForce(int[] arr, int S) {
        for (int i = 0; i < arr.length; i++) {
            int currentSum = 0;
            for (int j = i; j < arr.length; j++) {
                currentSum += arr[j];
                if (currentSum == S) {
                    return new int[]{i + 1, j + 1}; // 1-based indexing
                }
            }
        }
        return new int[]{-1}; // No solution found
    }
}
```

- **Time Complexity:** O(N2)
- **Space Complexity:** O(1)

Optimized (Using Sliding Window for positive numbers):

This is the most optimized solution for arrays with non-negative numbers. It uses a two-pointer (sliding window) approach.

Java

```
class Solution {
    public int[] subarraySumOptimized(int[] arr, int S) {
        int currentSum = 0;
        int start = 0;
        for (int end = 0; end < arr.length; end++) {
            currentSum += arr[end];

            while (currentSum > S && start <= end) {
                currentSum -= arr[start];
                start++;
            }

            if (currentSum == S) {
                return new int[]{start + 1, end + 1}; // 1-based indexing
            }
        }
        return new int[]{-1}; // No solution found
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(1)

Most Optimized (Using HashMap for positive/negative numbers):

If the array can contain negative numbers, the sliding window approach won't work directly. A HashMap approach is needed.

Java

```
import java.util.HashMap;
import java.util.Map;

class Solution {
    public int[] subarraySumWithNegatives(int[] arr, int S) {
        Map<Integer, Integer> sumMap = new HashMap<>();
        int currentSum = 0;
        sumMap.put(0, -1); // To handle cases where the subarray starts from index
```

```
0

        for (int i = 0; i < arr.length; i++) {
            currentSum += arr[i];
            if (sumMap.containsKey(currentSum - S)) {
                return new int[]{sumMap.get(currentSum - S) + 2, i + 1}; // 1-based
indexing
            }
            sumMap.put(currentSum, i);
        }
        return new int[]{-1}; // No solution found
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(N)

Sliding Window Problems

Sliding Window is a technique used for problems that involve arrays or strings where we need to find a subarray or substring of a given size or satisfying a certain condition. The "most optimized" solution often **is** a sliding window.

Example: Longest Substring Without Repeating Characters

Problem: Given a string `s`, find the length of the longest substring without repeating characters.

Brute Force:

Generate all possible substrings and check if each substring has repeating characters.

Java

```
import java.util.HashSet;
import java.util.Set;

class Solution {
    public int lengthOfLongestSubstringBruteForce(String s) {
        int maxLength = 0;
        for (int i = 0; i < s.length(); i++) {
            for (int j = i; j < s.length(); j++) {
                if (allUnique(s, i, j)) {
                    maxLength = Math.max(maxLength, j - i + 1);
                }
            }
```

```
        }
        return maxLength;
    }

    private boolean allUnique(String s, int start, int end) {
        Set<Character> charSet = new HashSet<>();
        for (int i = start; i <= end; i++) {
            char currentChar = s.charAt(i);
            if (charSet.contains(currentChar)) {
                return false;
            }
            charSet.add(currentChar);
        }
        return true;
    }
}
```

- **Time Complexity:** O(N3) (generating substrings O(N2), checking uniqueness O(N))
- **Space Complexity:** O(min(N,M)) where M is the size of the character set.

Most Optimized (Sliding Window):

Use a sliding window with a HashMap (or an array for ASCII characters) to keep track of character occurrences.

Java

```
import java.util.HashMap;
import java.util.Map;

class Solution {
    public int lengthOfLongestSubstringOptimized(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }

        Map<Character, Integer> charIndexMap = new HashMap<>();
        int maxLength = 0;
        int left = 0;

        for (int right = 0; right < s.length(); right++) {
            char currentChar = s.charAt(right);
            if (charIndexMap.containsKey(currentChar)) {
                // Move the left pointer past the last occurrence of the repeating
character
```

```
                left = Math.max(left, charIndexMap.get(currentChar) + 1);
            }
            charIndexMap.put(currentChar, right);
            maxLength = Math.max(maxLength, right - left + 1);
        }
        return maxLength;
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(min(N,M)) where M is the size of the character set.

String Reversal, Palindrome Check

1. String Reversal

Problem: Reverse a given string.

Brute Force (Conceptual, not truly brute force for string reversal):

Building a new string character by character from the end of the original.

Java

```
class Solution {
    public String reverseStringBruteForce(String s) {
        if (s == null || s.length() <= 1) {
            return s;
        }
        String reversed = "";
        for (int i = s.length() - 1; i >= 0; i--) {
            reversed += s.charAt(i); // String concatenation creates new String
objects
        }
        return reversed;
    }
}
```

- **Time Complexity:** O(N2) due to repeated string concatenation.
- **Space Complexity:** O(N)

Optimized (Using StringBuilder/char array):

StringBuilder is mutable and more efficient for string manipulations.

Java

```java
class Solution {
    public String reverseStringOptimized(String s) {
        if (s == null || s.length() <= 1) {
            return s;
        }
        StringBuilder sb = new StringBuilder(s);
        return sb.reverse().toString();
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(N)

Most Optimized (In-place for char array):

If allowed to modify a char array directly (as typically seen in interview questions like reversing a char array), it's in-place.

Java

```java
class Solution {
    public void reverseStringInPlace(char[] s) {
        int left = 0;
        int right = s.length - 1;
        while (left < right) {
            char temp = s[left];
            s[left] = s[right];
            s[right] = temp;
            left++;
            right--;
        }
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(1)

2. Palindrome Check

Problem: Check if a given string is a palindrome.

Brute Force (Conceptual):

Reverse the string and compare with the original.

Java

```
class Solution {
    public boolean isPalindromeBruteForce(String s) {
        if (s == null || s.length() <= 1) {
            return true;
        }
        String cleanedS = s.toLowerCase().replaceAll("[^a-z0-9]", ""); // Handle
non-alphanumeric and case
        StringBuilder sb = new StringBuilder(cleanedS);
        String reversedCleanedS = sb.reverse().toString();
        return cleanedS.equals(reversedCleanedS);
    }
}
```

- **Time Complexity:** O(N) (cleaning, reversing, comparing are all O(N))
- **Space Complexity:** O(N)

Optimized (Two Pointers):

Use two pointers, one from the beginning and one from the end, and compare characters. Skip non-alphanumeric characters.

Java

```
class Solution {
    public boolean isPalindromeOptimized(String s) {
        if (s == null || s.length() <= 1) {
            return true;
        }

        int left = 0;
        int right = s.length() - 1;

        while (left < right) {
            char charLeft = s.charAt(left);
            char charRight = s.charAt(right);

            if (!Character.isLetterOrDigit(charLeft)) {
                left++;
            } else if (!Character.isLetterOrDigit(charRight)) {
                right--;
            } else {
                if (Character.toLowerCase(charLeft) !=
Character.toLowerCase(charRight)) {
                    return false;
```

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

- **Time Complexity:** O(N)
- **Space Complexity:** O(1)

Linked Lists

1. Reverse a Linked List

Problem: Reverse a singly linked list.

Brute Force (Conceptual - Not truly brute force for linked lists but rather an inefficient approach):

Create a new linked list and insert elements from the original list at the beginning.

Java

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

class Solution {
    public ListNode reverseListBruteForce(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode newHead = null;
        ListNode current = head;
        while (current != null) {
            ListNode newNode = new ListNode(current.val); // Create new node
            newNode.next = newHead;
            newHead = newNode;
            current = current.next;
        }
        return newHead;
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(N) (for the new list)

Most Optimized (Iterative):

Modify the `next` pointers in-place.

Java

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

class Solution {
    public ListNode reverseListIterative(ListNode head) {
        ListNode prev = null;
        ListNode current = head;
        while (current != null) {
            ListNode nextTemp = current.next; // Store next node
            current.next = prev;              // Reverse current node's pointer
            prev = current;                   // Move prev to current
            current = nextTemp;               // Move current to next
        }
        return prev; // prev is now the new head
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(1)

Optimized (Recursive):

A recursive approach also reverses in-place.

Java

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

class Solution {
```

```
    public ListNode reverseListRecursive(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode newHead = reverseListRecursive(head.next);
        head.next.next = head;
        head.next = null;
        return newHead;
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(N) (due to recursion stack)

2. Detect Loop in a List

Problem: Detect if a linked list has a cycle.

Brute Force (Conceptual - Not truly brute force, but an inefficient approach):

Use a HashSet to store visited nodes. If a node is encountered again, there's a loop.

Java

```
import java.util.HashSet;
import java.util.Set;

class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

class Solution {
    public boolean hasCycleBruteForce(ListNode head) {
        Set<ListNode> visitedNodes = new HashSet<>();
        ListNode current = head;
        while (current != null) {
            if (visitedNodes.contains(current)) {
                return true;
            }
            visitedNodes.add(current);
            current = current.next;
        }
        return false;
```

```
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(N)

Most Optimized (Floyd's Cycle-Finding Algorithm / Tortoise and Hare):

Use two pointers, one slow and one fast. If they meet, there's a cycle.

Java

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

class Solution {
    public boolean hasCycleOptimized(ListNode head) {
        if (head == null || head.next == null) {
            return false;
        }
        ListNode slow = head;
        ListNode fast = head.next; // Fast starts one step ahead to detect
immediately if head points to itself

        while (slow != fast) {
            if (fast == null || fast.next == null) {
                return false; // Reached end, no cycle
            }
            slow = slow.next;
            fast = fast.next.next;
        }
        return true; // Pointers met, cycle detected
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(1)

3. Merge Two Sorted Lists

Problem: Merge two sorted linked lists into a single sorted list.

Brute Force (Conceptual - creating a new list from scratch and sorting):

Combine all elements into an array, sort the array, then build a new linked list. This is overly complex for linked lists. A more direct "brute force" for linked lists might be to repeatedly insert elements from one list into the correct position of the other, but even that is more complex than direct merging.

Let's stick to the common interview solution structure.

Most Optimized (Iterative):

Create a dummy head and iteratively append the smaller element from the two lists.

Java

```java
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

class Solution {
    public ListNode mergeTwoListsIterative(ListNode l1, ListNode l2) {
        ListNode dummyHead = new ListNode(0);
        ListNode current = dummyHead;

        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;
        }

        // Append remaining nodes
        if (l1 != null) {
            current.next = l1;
        } else if (l2 != null) {
            current.next = l2;
        }

        return dummyHead.next;
    }
}
```

- **Time Complexity:** O(M+N) where M and N are lengths of the lists.
- **Space Complexity:** O(1)

Optimized (Recursive):

Java

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

class Solution {
    public ListNode mergeTwoListsRecursive(ListNode l1, ListNode l2) {
        if (l1 == null) return l2;
        if (l2 == null) return l1;

        if (l1.val < l2.val) {
            l1.next = mergeTwoListsRecursive(l1.next, l2);
            return l1;
        } else {
            l2.next = mergeTwoListsRecursive(l1, l2.next);
            return l2;
        }
    }
}
```

- **Time Complexity:** O(M+N)
- **Space Complexity:** O(M+N) (due to recursion stack)

Stacks & Queues

1. Balanced Parentheses

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

Brute Force (Not suitable for a clear brute force definition):

A true "brute force" would involve trying all possible pairings, which quickly becomes combinatorial and impractical. The most straightforward approach is using a stack.

Most Optimized (Using a Stack):

Use a stack to keep track of opening parentheses. When a closing parenthesis is encountered, check if the top of the stack matches the corresponding opening parenthesis.

Java

```
import java.util.Stack;
import java.util.HashMap;
import java.util.Map;

class Solution {
    public boolean isValid(String s) {
        if (s == null || s.length() == 0) {
            return true;
        }

        Stack<Character> stack = new Stack<>();
        Map<Character, Character> map = new HashMap<>();
        map.put(')', '(');
        map.put('}', '{');
        map.put(']', '[');

        for (char c : s.toCharArray()) {
            if (map.containsKey(c)) { // It's a closing parenthesis
                char topElement = stack.empty() ? '#' : stack.pop(); // Use '#' for
empty stack
                if (topElement != map.get(c)) {
                    return false;
                }
            } else { // It's an opening parenthesis
                stack.push(c);
            }
        }
        return stack.empty(); // Stack should be empty if all parentheses are
matched
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(N) (in the worst case, e.g., "(((((((((")

2. Next Greater Element

Problem: Given a circular integer array `nums`, find the next greater element for every element. The next greater element of a number `x` is the first greater number to its traversing order next in the array, which means you could search in a circular way. If it doesn't exist, output -1.

Brute Force:

For each element, iterate through the rest of the array (circularly) to find the next greater element.

Java

```
class Solution {
    public int[] nextGreaterElementsBruteForce(int[] nums) {
        int n = nums.length;
        int[] result = new int[n];

        for (int i = 0; i < n; i++) {
            result[i] = -1; // Default to -1
            for (int j = 1; j < n; j++) { // Start search from next element
                int nextIndex = (i + j) % n;
                if (nums[nextIndex] > nums[i]) {
                    result[i] = nums[nextIndex];
                    break;
                }
            }
        }
        return result;
    }
}
```

- **Time Complexity:** O(N2)
- **Space Complexity:** O(N) (for result array)

Most Optimized (Using a Monotonic Stack):

A monotonic stack (decreasing in this case) efficiently keeps track of potential next greater elements. For circular arrays, we can iterate twice.

Java

```
import java.util.Stack;

class Solution {
    public int[] nextGreaterElementsOptimized(int[] nums) {
        int n = nums.length;
        int[] result = new int[n];
        Stack<Integer> stack = new Stack<>(); // Stores indices

        // Iterate twice to handle circularity
        for (int i = 2 * n - 1; i >= 0; i--) {
```

```
            int currentNum = nums[i % n]; // Get the actual number

            while (!stack.isEmpty() && nums[stack.peek()] <= currentNum) {
                stack.pop();
            }

            if (stack.isEmpty()) {
                result[i % n] = -1;
            } else {
                result[i % n] = nums[stack.peek()];
            }
            stack.push(i % n); // Push the current index
        }
        return result;
    }
}
```

- **Time Complexity:** O(N) (each element is pushed and popped at most once)
- **Space Complexity:** O(N) (for the stack and result array)

3. Implement Queue using Stacks

Problem: Implement a first-in-first-out (FIFO) queue using only two stacks.

Brute Force (Conceptual):

Not applicable for implementation problems, but a basic, inefficient way would be to empty `stack1` to `stack2` for every dequeue/peek operation, and then transfer back, which is essentially the optimized approach but done inefficiently if not careful.

Most Optimized (Using two stacks):

Maintain two stacks: `inStack` for enqueue operations and `outStack` for dequeue/peek operations. When `outStack` is empty, transfer all elements from `inStack` to `outStack`.

Java

```
import java.util.Stack;

class MyQueue {
    private Stack<Integer> inStack;
    private Stack<Integer> outStack;

    public MyQueue() {
        inStack = new Stack<>();
        outStack = new Stack<>();
```

```
    }

    public void push(int x) {
        inStack.push(x);
    }

    public int pop() {
        peek(); // Ensure outStack is populated
        return outStack.pop();
    }

    public int peek() {
        if (outStack.empty()) {
            while (!inStack.empty()) {
                outStack.push(inStack.pop());
            }
        }
        return outStack.peek();
    }

    public boolean empty() {
        return inStack.empty() && outStack.empty();
    }
}
```

- **Time Complexity:**
  - `push`: O(1)
  - `pop`: Amortized O(1) (each element is moved between stacks twice, but overall balanced)
  - `peek`: Amortized O(1)
  - `empty`: O(1)
- **Space Complexity:** O(N) (for the stacks)

Recursion & Backtracking

1. Factorial

Problem: Calculate the factorial of a non-negative integer.

Brute Force (Iterative is simpler for factorial):

Java

```
class Solution {
    public long factorialIterative(int n) {
```

```
        if (n < 0) {
            throw new IllegalArgumentException("Factorial is not defined for
negative numbers.");
        }
        long result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(1)

Most Optimized (Recursive):

Java

```
class Solution {
    public long factorialRecursive(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("Factorial is not defined for
negative numbers.");
        }
        if (n == 0 || n == 1) {
            return 1;
        }
        return n * factorialRecursive(n - 1);
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(N) (due to recursion stack)

2. Fibonacci

Problem: Calculate the nth Fibonacci number.

Brute Force (Recursive without memoization):

This is highly inefficient due to redundant calculations.

Java

```
class Solution {
    public long fibonacciBruteForce(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("Fibonacci is not defined for
negative numbers.");
        }
        if (n <= 1) {
            return n;
        }
        return fibonacciBruteForce(n - 1) + fibonacciBruteForce(n - 2);
    }
}
```

- **Time Complexity:** O(2N)
- **Space Complexity:** O(N) (due to recursion stack)

Optimized (Recursive with Memoization / Top-Down DP):

Java

```
import java.util.HashMap;
import java.util.Map;

class Solution {
    Map<Integer, Long> memo = new HashMap<>();

    public long fibonacciMemoized(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("Fibonacci is not defined for
negative numbers.");
        }
        if (n <= 1) {
            return n;
        }
        if (memo.containsKey(n)) {
            return memo.get(n);
        }
        long result = fibonacciMemoized(n - 1) + fibonacciMemoized(n - 2);
        memo.put(n, result);
        return result;
    }
}
```

- **Time Complexity:** O(N)

- **Space Complexity:** O(N) (for memoization map and recursion stack)

Most Optimized (Iterative / Bottom-Up DP):

Java

```
class Solution {
    public long fibonacciIterative(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("Fibonacci is not defined for
negative numbers.");
        }
        if (n <= 1) {
            return n;
        }
        long a = 0;
        long b = 1;
        for (int i = 2; i <= n; i++) {
            long temp = a + b;
            a = b;
            b = temp;
        }
        return b;
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(1)

3. N-Queens

Problem: Place `n` queens on an `n x n` chessboard such that no two queens attack each other.

Brute Force (Conceptual - trying all permutations):

Try every possible position for each queen on the board and check if it's a valid configuration. This would involve iterating through NN possibilities and then checking each.

Most Optimized (Backtracking):

Place queens one by one column by column. Use helper functions to check for conflicts (row, column, diagonals). Backtrack if a conflict is found.

Java

```java
import java.util.ArrayList;
import java.util.List;

class Solution {
    public List<List<String>> solveNQueens(int n) {
        List<List<String>> solutions = new ArrayList<>();
        char[][] board = new char[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                board[i][j] = '.';
            }
        }
        solve(board, 0, solutions);
        return solutions;
    }

    private void solve(char[][] board, int col, List<List<String>> solutions) {
        if (col == board.length) {
            solutions.add(constructSolution(board));
            return;
        }

        for (int row = 0; row < board.length; row++) {
            if (isValid(board, row, col)) {
                board[row][col] = 'Q';
                solve(board, col + 1, solutions);
                board[row][col] = '.'; // Backtrack
            }
        }
    }

    private boolean isValid(char[][] board, int row, int col) {
        // Check row
        for (int j = 0; j < col; j++) {
            if (board[row][j] == 'Q') {
                return false;
            }
        }

        // Check upper diagonal
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 'Q') {
                return false;
```

```
            }
        }

        // Check lower diagonal
        for (int i = row + 1, j = col - 1; i < board.length && j >= 0; i++, j--) {
            if (board[i][j] == 'Q') {
                return false;
            }
        }
        return true;
    }

    private List<String> constructSolution(char[][] board) {
        List<String> solution = new ArrayList<>();
        for (int i = 0; i < board.length; i++) {
            solution.add(new String(board[i]));
        }
        return solution;
    }
}
```

- **Time Complexity:** Roughly O(N!) in the worst case (pruning reduces it significantly, but still exponential). The actual complexity is closer to O(N!) or O(N3) based on more precise analysis, but for practical purposes, it's considered exponential.
- **Space Complexity:** O(N2) for the board and O(N) for the recursion stack.

4. Subset Sum

Problem: Given a set of non-negative integers and a value `sum`, determine if there is a subset of the given set with sum equal to `sum`.

Brute Force (Recursive - checking all subsets):

For each element, either include it in the subset or exclude it. This generates all 2N subsets.

Java

```
class Solution {
    public boolean isSubsetSumBruteForce(int[] set, int n, int sum) {
        if (sum == 0) {
            return true;
        }
        if (n == 0) {
            return false;
        }
```

```
        // If last element is greater than sum, then ignore it
        if (set[n - 1] > sum) {
            return isSubsetSumBruteForce(set, n - 1, sum);
        }

        // Else, check if sum can be obtained by including or excluding the last
element
        return isSubsetSumBruteForce(set, n - 1, sum) ||
               isSubsetSumBruteForce(set, n - 1, sum - set[n - 1]);
    }
}
```

- **Time Complexity:** O(2N)
- **Space Complexity:** O(N) (due to recursion stack)

Most Optimized (Dynamic Programming):

Use a 2D boolean array `dp[i][j]` where `dp[i][j]` is true if a subset of `set[0...i-1]` sums to `j`.

Java

```
class Solution {
    public boolean isSubsetSumDP(int[] set, int sum) {
        int n = set.length;
        boolean[][] dp = new boolean[n + 1][sum + 1];

        // If sum is 0, then answer is true
        for (int i = 0; i <= n; i++) {
            dp[i][0] = true;
        }

        // If sum is not 0 and set is empty, then answer is false
        for (int i = 1; i <= sum; i++) {
            dp[0][i] = false;
        }

        // Fill the dp table
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum; j++) {
                if (j < set[i - 1]) {
                    dp[i][j] = dp[i - 1][j];
                } else {
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - set[i - 1]];
                }
```

```
            }
        }
        return dp[n][sum];
    }
}
```

- **Time Complexity:** O(N∗Sum)
- **Space Complexity:** O(N∗Sum)

Space Optimized DP (if values are small and distinct):

You can optimize the space to O(Sum) by only considering the previous row.

5. Permutations of a String

Problem: Given a string, return all possible permutations of the string.

Brute Force (Conceptual):

Generate all combinations of characters and then filter to keep only unique permutations. This is generally not how permutations are generated.

Most Optimized (Backtracking):

Use backtracking to swap characters at different positions.

Java

```
import java.util.ArrayList;
import java.util.List;

class Solution {
    public List<String> permuteString(String str) {
        List<String> result = new ArrayList<>();
        if (str == null || str.length() == 0) {
            return result;
        }
        char[] chars = str.toCharArray();
        permute(chars, 0, result);
        return result;
    }

    private void permute(char[] chars, int index, List<String> result) {
        if (index == chars.length - 1) {
            result.add(new String(chars));
            return;
```

```
        }

        for (int i = index; i < chars.length; i++) {
            swap(chars, index, i);
            permute(chars, index + 1, result);
            swap(chars, index, i); // Backtrack: undo the swap
        }
    }

    private void swap(char[] chars, int i, int j) {
        char temp = chars[i];
        chars[i] = chars[j];
        chars[j] = temp;
    }
}
```

- **Time Complexity:** O(N∗N!) (N! permutations, each taking O(N) to build the string)
- **Space Complexity:** O(N) (for the recursion stack and char array)

Searching & Sorting

1. Binary Search and Its Variants

Problem: Search for an element in a sorted array.

Brute Force (Linear Search):

Iterate through the array from start to end.

Java

```
class Solution {
    public int linearSearch(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i;
            }
        }
        return -1;
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(1)

Most Optimized (Binary Search - Iterative):

Works only on sorted arrays. Repeatedly divide the search interval in half.

Java

```
class Solution {
    public int binarySearchIterative(int[] arr, int target) {
        int low = 0;
        int high = arr.length - 1;

        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (arr[mid] == target) {
                return mid;
            } else if (arr[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return -1;
    }
}
```

- **Time Complexity:** O(logN)
- **Space Complexity:** O(1)

Binary Search - Recursive:

Java

```
class Solution {
    public int binarySearchRecursive(int[] arr, int target) {
        return binarySearchRecursiveHelper(arr, target, 0, arr.length - 1);
    }

    private int binarySearchRecursiveHelper(int[] arr, int target, int low, int
high) {
        if (low > high) {
            return -1;
        }

        int mid = low + (high - low) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
```

```
            return binarySearchRecursiveHelper(arr, target, mid + 1, high);
        } else {
            return binarySearchRecursiveHelper(arr, target, low, mid - 1);
        }
    }
}
```

- **Time Complexity:** O(logN)
- **Space Complexity:** O(logN) (due to recursion stack)

Variants (e.g., finding first/last occurrence, smallest greater element): These are solved by small modifications to the basic binary search logic.

2. Quick Sort, Merge Sort

These are widely used comparison-based sorting algorithms.

Quick Sort

Brute Force (Conceptual for sorting - not Quick Sort):

Any O(N2) sort like Bubble Sort, Selection Sort, Insertion Sort.

Java

```
// Example: Bubble Sort as a "brute force" for comparison
class Solution {
    public void bubbleSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
}
```

- **Time Complexity:** O(N2)
- **Space Complexity:** O(1)

Most Optimized (Quick Sort - Average Case):

Divides array into smaller sub-arrays and recursively sorts them.

Java

```
class Solution {
    public void quickSort(int[] arr) {
        if (arr == null || arr.length <= 1) {
            return;
        }
        quickSort(arr, 0, arr.length - 1);
    }

    private void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    private int partition(int[] arr, int low, int high) {
        int pivot = arr[high]; // Choose last element as pivot
        int i = (low - 1); // Index of smaller element

        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }
}
```

- **Time Complexity:** Average O(NlogN), Worst O(N2)
- **Space Complexity:** Average O(logN) (for recursion stack), Worst O(N)

Merge Sort

Brute Force (Conceptual - see Bubble Sort above).

Most Optimized (Merge Sort):

Divides array into two halves, recursively sorts them, and then merges the sorted halves.

Java

```
class Solution {
    public void mergeSort(int[] arr) {
        if (arr == null || arr.length <= 1) {
            return;
        }
        mergeSort(arr, 0, arr.length - 1);
    }

    private void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    private void merge(int[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        int[] L = new int[n1];
        int[] R = new int[n2];

        for (int i = 0; i < n1; ++i) {
            L[i] = arr[left + i];
        }
        for (int j = 0; j < n2; ++j) {
            R[j] = arr[mid + 1 + j];
        }

        int i = 0, j = 0;
        int k = left;
        while (i < n1 && j < n2) {
```

```
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
}
```

- **Time Complexity:** O(NlogN) (guaranteed)
- **Space Complexity:** O(N) (for temporary arrays in merge)

3. Kth Smallest/Largest Element

Problem: Find the Kth smallest/largest element in an unsorted array.

Brute Force:

Sort the array and then pick the Kth element.

Java

```
import java.util.Arrays;

class Solution {
    public int findKthSmallestBruteForce(int[] nums, int k) {
        Arrays.sort(nums); // Uses Dual-Pivot QuickSort for primitives in Java
        return nums[k - 1]; // Kth smallest is at index k-1
    }
}
```

- **Time Complexity:** O(NlogN) (due to sorting)
- **Space Complexity:** O(logN) or O(N) depending on sort implementation

Optimized (Using a Min/Max Heap / Priority Queue):

For Kth smallest, use a Max-Heap of size K. For Kth largest, use a Min-Heap of size K.

Java

```
import java.util.PriorityQueue;
import java.util.Collections; // For max-heap

class Solution {
    public int findKthSmallestOptimized(int[] nums, int k) {
        // Max-heap to store the K smallest elements
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>
(Collections.reverseOrder());

        for (int num : nums) {
            maxHeap.offer(num);
            if (maxHeap.size() > k) {
                maxHeap.poll(); // Remove largest if size exceeds k
            }
        }
        return maxHeap.peek();
    }

    public int findKthLargestOptimized(int[] nums, int k) {
        // Min-heap to store the K largest elements
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int num : nums) {
            minHeap.offer(num);
            if (minHeap.size() > k) {
                minHeap.poll(); // Remove smallest if size exceeds k
            }
        }
        return minHeap.peek();
    }
}
```

- **Time Complexity:** O(NlogK)
- **Space Complexity:** O(K)

Most Optimized (Quickselect - Average Case):

A selection algorithm related to QuickSort. It finds the Kth element in linear time on average.

Java

```
import java.util.Random;

class Solution {
    private Random random = new Random();

    public int findKthSmallestQuickselect(int[] nums, int k) {
        int n = nums.length;
        if (k < 1 || k > n) {
            throw new IllegalArgumentException("Invalid k");
        }
        // Kth smallest is equivalent to (n - k)th largest
        return quickselect(nums, 0, n - 1, n - k);
    }

    private int quickselect(int[] nums, int left, int right, int k_smallest) {
        if (left == right) {
            return nums[left];
        }

        // Randomly pick a pivot index
        int pivotIndex = left + random.nextInt(right - left + 1);
        pivotIndex = partition(nums, left, right, pivotIndex);

        if (k_smallest == pivotIndex) {
            return nums[k_smallest];
        } else if (k_smallest < pivotIndex) {
            return quickselect(nums, left, pivotIndex - 1, k_smallest);
        } else {
            return quickselect(nums, pivotIndex + 1, right, k_smallest);
        }
    }

    private int partition(int[] nums, int left, int right, int pivotIndex) {
        int pivotValue = nums[pivotIndex];
        swap(nums, pivotIndex, right); // Move pivot to end
        int storeIndex = left;

        for (int i = left; i < right; i++) {
            if (nums[i] < pivotValue) {
                swap(nums, storeIndex, i);
```

```
                storeIndex++;
            }
        }
        swap(nums, right, storeIndex); // Move pivot to its final sorted position
        return storeIndex;
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

- **Time Complexity:** Average O(N), Worst O(N2)
- **Space Complexity:** O(logN) (for recursion stack)

Trees & Graphs

1. Inorder/Preorder/Postorder Traversal (Binary Tree)

Problem: Traverse a binary tree in different orders.

Brute Force (Conceptual - Not applicable for traversal):

There isn't a "brute force" for tree traversal itself; the traversal methods are the algorithms.

Most Optimized (Recursive):

These are the standard, most common, and often most intuitive ways to implement tree traversals.

Java

```
import java.util.ArrayList;
import java.util.List;
import java.util.Stack; // For iterative

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

class Solution {
    // Inorder Traversal (Left, Root, Right)
```

```
public List<Integer> inorderTraversalRecursive(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    inorderHelper(root, result);
    return result;
}

private void inorderHelper(TreeNode node, List<Integer> result) {
    if (node == null) {
        return;
    }
    inorderHelper(node.left, result);
    result.add(node.val);
    inorderHelper(node.right, result);
}

// Preorder Traversal (Root, Left, Right)
public List<Integer> preorderTraversalRecursive(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    preorderHelper(root, result);
    return result;
}

private void preorderHelper(TreeNode node, List<Integer> result) {
    if (node == null) {
        return;
    }
    result.add(node.val);
    preorderHelper(node.left, result);
    preorderHelper(node.right, result);
}

// Postorder Traversal (Left, Right, Root)
public List<Integer> postorderTraversalRecursive(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    postorderHelper(root, result);
    return result;
}

private void postorderHelper(TreeNode node, List<Integer> result) {
    if (node == null) {
        return;
    }
    postorderHelper(node.left, result);
```

```
        postorderHelper(node.right, result);
        result.add(node.val);
    }

    // Iterative Inorder Traversal (using Stack)
    public List<Integer> inorderTraversalIterative(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        TreeNode current = root;

        while (current != null || !stack.isEmpty()) {
            while (current != null) {
                stack.push(current);
                current = current.left;
            }
            current = stack.pop();
            result.add(current.val);
            current = current.right;
        }
        return result;
    }

    // Iterative Preorder Traversal (using Stack)
    public List<Integer> preorderTraversalIterative(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) {
            return result;
        }
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);

        while (!stack.isEmpty()) {
            TreeNode node = stack.pop();
            result.add(node.val);
            if (node.right != null) {
                stack.push(node.right);
            }
            if (node.left != null) {
                stack.push(node.left);
            }
        }
        return result;
    }
```

```
    // Iterative Postorder Traversal (using two Stacks or a complex single stack)
    // Using two stacks is more straightforward
    public List<Integer> postorderTraversalIterative(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) {
            return result;
        }
        Stack<TreeNode> s1 = new Stack<>();
        Stack<TreeNode> s2 = new Stack<>();
        s1.push(root);

        while (!s1.isEmpty()) {
            TreeNode node = s1.pop();
            s2.push(node);
            if (node.left != null) {
                s1.push(node.left);
            }
            if (node.right != null) {
                s1.push(node.right);
            }
        }
        while (!s2.isEmpty()) {
            result.add(s2.pop().val);
        }
        return result;
    }
}
```

- **Time Complexity (all traversals):** O(N) where N is the number of nodes.
- **Space Complexity (recursive):** O(H) where H is the height of the tree (for recursion stack). In worst case (skewed tree), O(N).
- **Space Complexity (iterative):** O(H) for the stack. In worst case (skewed tree), O(N).

2. BFS, DFS (Graph Traversal)

Problem: Traverse a graph.

Brute Force (Conceptual): Not applicable. BFS and DFS are the fundamental traversal algorithms.

Most Optimized (BFS - Breadth-First Search):

Uses a queue to visit nodes level by level.

Java

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

class Graph {
    private int V; // Number of vertices
    private List<List<Integer>> adj; // Adjacency list

    public Graph(int v) {
        V = v;
        adj = new ArrayList<>(V);
        for (int i = 0; i < V; ++i) {
            adj.add(new ArrayList<>());
        }
    }

    public void addEdge(int v, int w) {
        adj.get(v).add(w);
    }

    public void BFS(int startNode) {
        boolean[] visited = new boolean[V];
        Queue<Integer> queue = new LinkedList<>();

        visited[startNode] = true;
        queue.offer(startNode);

        System.out.print("BFS Traversal starting from " + startNode + ": ");
        while (!queue.isEmpty()) {
            int u = queue.poll();
            System.out.print(u + " ");

            for (int v : adj.get(u)) {
                if (!visited[v]) {
                    visited[v] = true;
                    queue.offer(v);
                }
            }
        }
        System.out.println();
    }
}
```

- **Time Complexity:** O(V+E) where V is number of vertices and E is number of edges.
- **Space Complexity:** O(V) (for queue and visited array)

Most Optimized (DFS - Depth-First Search):

Uses recursion (or a stack) to visit nodes as deep as possible before backtracking.

Java

```
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

class Graph {
    private int V;
    private List<List<Integer>> adj;

    public Graph(int v) {
        V = v;
        adj = new ArrayList<>(V);
        for (int i = 0; i < V; ++i) {
            adj.add(new ArrayList<>());
        }
    }

    public void addEdge(int v, int w) {
        adj.get(v).add(w);
    }

    // Recursive DFS
    public void DFS(int startNode) {
        boolean[] visited = new boolean[V];
        System.out.print("DFS Traversal starting from " + startNode + "
(Recursive): ");
        DFSUtil(startNode, visited);
        System.out.println();
    }

    private void DFSUtil(int u, boolean[] visited) {
        visited[u] = true;
        System.out.print(u + " ");

        for (int v : adj.get(u)) {
            if (!visited[v]) {
```

```
                DFSUtil(v, visited);
            }
        }
    }

    // Iterative DFS (using Stack)
    public void DFSIterative(int startNode) {
        boolean[] visited = new boolean[V];
        Stack<Integer> stack = new Stack<>();

        stack.push(startNode);
        visited[startNode] = true;

        System.out.print("DFS Traversal starting from " + startNode + "
(Iterative): ");
        while (!stack.isEmpty()) {
            int u = stack.pop();
            System.out.print(u + " ");

            // Get all adjacent vertices of the popped vertex u
            // If a adjacent has not been visited, then push it to the stack
            // Note: For consistent order with recursive DFS, iterate in reverse if
adding to stack
            for (int v : adj.get(u)) { // Can iterate in reverse for specific
ordering
                if (!visited[v]) {
                    visited[v] = true;
                    stack.push(v);
                }
            }
        }
        System.out.println();
    }
}
```

- **Time Complexity:** O(V+E)
- **Space Complexity:** O(V) (for recursion stack or explicit stack and visited array)

3. Shortest Path (Dijkstra's, BFS)

BFS for Shortest Path (Unweighted Graphs):

Problem: Find the shortest path in an unweighted graph.

Most Optimized: BFS naturally finds the shortest path in terms of number of edges in an unweighted graph because it explores layer by layer.

Java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

class Graph {
    private int V;
    private List<List<Integer>> adj;

    public Graph(int v) {
        V = v;
        adj = new ArrayList<>(V);
        for (int i = 0; i < V; ++i) {
            adj.add(new ArrayList<>());
        }
    }

    public void addEdge(int u, int v) {
        adj.get(u).add(v);
        adj.get(v).add(u); // For undirected graph
    }

    public int shortestPathBFS(int startNode, int endNode) {
        int[] dist = new int[V];
        Arrays.fill(dist, -1); // -1 indicates unvisited
        Queue<Integer> queue = new LinkedList<>();

        dist[startNode] = 0;
        queue.offer(startNode);

        while (!queue.isEmpty()) {
            int u = queue.poll();

            if (u == endNode) {
                return dist[u]; // Found the shortest path
            }

            for (int v : adj.get(u)) {
```

```
                if (dist[v] == -1) { // If not visited
                    dist[v] = dist[u] + 1;
                    queue.offer(v);
                }
            }
        }
        return -1; // No path found
    }
}
```

- **Time Complexity:** O(V+E)
- **Space Complexity:** O(V)

Dijkstra's Algorithm (Weighted Graphs with Non-Negative Edge Weights):

Problem: Find the shortest paths from a single source to all other vertices in a graph with non-negative edge weights.

Brute Force (Conceptual for shortest path):

Trying all possible paths between two nodes, which is combinatorially explosive.

Most Optimized (Dijkstra's with Priority Queue):

Uses a priority queue to efficiently extract the vertex with the minimum distance.

Java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.PriorityQueue;
import java.util.List;

class Edge {
    int to;
    int weight;

    public Edge(int to, int weight) {
        this.to = to;
        this.weight = weight;
    }
}

class Node implements Comparable<Node> {
    int vertex;
```

```
    int distance;

    public Node(int vertex, int distance) {
        this.vertex = vertex;
        this.distance = distance;
    }

    @Override
    public int compareTo(Node other) {
        return Integer.compare(this.distance, other.distance);
    }
}

class Dijkstra {
    private int V;
    private List<List<Edge>> adj;

    public Dijkstra(int v) {
        V = v;
        adj = new ArrayList<>(V);
        for (int i = 0; i < V; ++i) {
            adj.add(new ArrayList<>());
        }
    }

    public void addEdge(int u, int v, int weight) {
        adj.get(u).add(new Edge(v, weight));
        // For undirected graph, add reverse edge as well:
        // adj.get(v).add(new Edge(u, weight));
    }

    public int[] findShortestPaths(int startNode) {
        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE); // Initialize distances to infinity
        boolean[] visited = new boolean[V]; // Keep track of visited nodes

        PriorityQueue<Node> pq = new PriorityQueue<>();

        dist[startNode] = 0;
        pq.offer(new Node(startNode, 0));

        while (!pq.isEmpty()) {
            int u = pq.poll().vertex;
```

```
            if (visited[u]) {
                continue; // Already processed
            }
            visited[u] = true;

            for (Edge edge : adj.get(u)) {
                int v = edge.to;
                int weight = edge.weight;

                // Relaxation step
                if (!visited[v] && dist[u] != Integer.MAX_VALUE && dist[u] + weight
< dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.offer(new Node(v, dist[v]));
                }
            }
        }
        return dist; // Contains shortest distances from startNode
    }
}
```

- **Time Complexity:** O(ElogV) or O(E+VlogV) with Fibonacci heap, commonly O(ElogV) with binary heap (PriorityQueue in Java).
- **Space Complexity:** O(V+E) (for adjacency list, distance array, and priority queue)

Dynamic Programming

1. Longest Common Subsequence (LCS)

Problem: Given two sequences, find the length of the longest subsequence present in both of them.

Brute Force (Recursive without memoization):

Explore all possible subsequences of both strings. This is highly inefficient.

Java

```
class Solution {
    public int longestCommonSubsequenceBruteForce(String text1, String text2) {
        return lcsHelper(text1, text2, text1.length(), text2.length());
    }

    private int lcsHelper(String s1, String s2, int m, int n) {
        if (m == 0 || n == 0) {
```

```
            return 0;
        }
        if (s1.charAt(m - 1) == s2.charAt(n - 1)) {
            return 1 + lcsHelper(s1, s2, m - 1, n - 1);
        } else {
            return Math.max(lcsHelper(s1, s2, m, n - 1), lcsHelper(s1, s2, m - 1,
n));
        }
    }
}
```

- **Time Complexity:** O(2max(M,N)) (exponential)
- **Space Complexity:** O(max(M,N)) (recursion stack)

Most Optimized (Dynamic Programming - Bottom-Up):

Use a 2D array `dp[i][j]` to store the length of LCS of `text1[0...i-1]` and `text2[0...j-1]`.

Java

```
class Solution {
    public int longestCommonSubsequenceDP(String text1, String text2) {
        int m = text1.length();
        int n = text2.length();

        int[][] dp = new int[m + 1][n + 1];

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

- **Time Complexity:** O(M∗N)
- **Space Complexity:** O(M∗N)

Space Optimized DP (using only two rows):

Can be optimized to O(min(M,N)) space.

2. Knapsack Problem (0/1 Knapsack)

Problem: Given weights and values of N items, put these items in a knapsack of capacity `W` to get the maximum total value in the knapsack. Each item can only be included once (0/1 property).

Brute Force (Recursive without memoization):

For each item, either include it or exclude it.

Java

```
class Solution {
    public int knapsackBruteForce(int W, int[] wt, int[] val, int n) {
        if (n == 0 || W == 0) {
            return 0;
        }

        // If weight of the nth item is more than Knapsack capacity W, then this
item cannot be included
        if (wt[n - 1] > W) {
            return knapsackBruteForce(W, wt, val, n - 1);
        } else {
            // Return the maximum of two cases:
            // 1. nth item included
            // 2. nth item not included
            return Math.max(val[n - 1] + knapsackBruteForce(W - wt[n - 1], wt, val,
n - 1),
                            knapsackBruteForce(W, wt, val, n - 1));
        }
    }
}
```

- **Time Complexity:** O(2N)
- **Space Complexity:** O(N) (recursion stack)

Most Optimized (Dynamic Programming - Bottom-Up):

Use a 2D array `dp[i][w]` to store the maximum value that can be obtained with first `i` items and weight capacity `w`.

Java

```
class Solution {
    public int knapsackDP(int W, int[] wt, int[] val, int n) {
        int[][] dp = new int[n + 1][W + 1];
```

```
        for (int i = 0; i <= n; i++) {
            for (int w = 0; w <= W; w++) {
                if (i == 0 || w == 0) {
                    dp[i][w] = 0;
                } else if (wt[i - 1] <= w) {
                    dp[i][w] = Math.max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i
- 1][w]);
                } else {
                    dp[i][w] = dp[i - 1][w];
                }
            }
        }
        return dp[n][W];
    }
}
```

- **Time Complexity:** O(N∗W)
- **Space Complexity:** O(N∗W)

Space Optimized DP (using one row):

Java

```
class Solution {
    public int knapsackSpaceOptimized(int W, int[] wt, int[] val, int n) {
        int[] dp = new int[W + 1]; // dp[w] stores max value for capacity w

        for (int i = 1; i <= n; i++) {
            for (int w = W; w >= wt[i - 1]; w--) { // Iterate backwards to use
values from current row correctly
                dp[w] = Math.max(dp[w], val[i - 1] + dp[w - wt[i - 1]]);
            }
        }
        return dp[W];
    }
}
```

- **Time Complexity:** O(N∗W)
- **Space Complexity:** O(W)

3. Coin Change

Problem: Given a set of coin denominations and a target amount, find the minimum number of coins needed to make up that amount. If it's not possible, return -1.

Brute Force (Recursive without memoization):

Try all combinations of coins. Highly redundant.

Java

```
class Solution {
    public int coinChangeBruteForce(int[] coins, int amount) {
        if (amount < 0) return -1;
        if (amount == 0) return 0;

        int minCoins = Integer.MAX_VALUE;
        for (int coin : coins) {
            int result = coinChangeBruteForce(coins, amount - coin);
            if (result != -1) {
                minCoins = Math.min(minCoins, 1 + result);
            }
        }
        return (minCoins == Integer.MAX_VALUE) ? -1 : minCoins;
    }
}
```

- **Time Complexity:** O(SN) where S is amount, N is number of coins (very rough estimate, better thought of as exponential related to amount and number of coin types).
- **Space Complexity:** O(Amount) (recursion stack)

Most Optimized (Dynamic Programming - Bottom-Up):

Use a 1D array `dp[i]` to store the minimum number of coins to make amount `i`.

Java

```
import java.util.Arrays;

class Solution {
    public int coinChangeDP(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1); // Initialize with a value greater than any
possible answer
        dp[0] = 0; // 0 coins needed for amount 0

        for (int i = 1; i <= amount; i++) {
            for (int coin : coins) {
                if (coin <= i) {
                    dp[i] = Math.min(dp[i], 1 + dp[i - coin]);
                }
            }
```

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

- **Time Complexity:** O(Amount∗N) where N is the number of coin denominations.
- **Space Complexity:** O(Amount)

Bit Manipulation

1. Check if a Number is Power of 2

Problem: Given an integer `n`, return `true` if it is a power of two. Otherwise, return `false`.

Brute Force (Iterative Division):

Keep dividing by 2 until it's 1.

Java

```
class Solution {
    public boolean isPowerOfTwoBruteForce(int n) {
        if (n <= 0) {
            return false;
        }
        while (n % 2 == 0) {
            n /= 2;
        }
        return n == 1;
    }
}
```

- **Time Complexity:** O(logN)
- **Space Complexity:** O(1)

Most Optimized (Bitwise AND):

A number `n` is a power of 2 if and only if `n > 0` and `(n & (n - 1))` is 0. This works because powers of two have only one bit set (e.g., 8 is `1000`), and `n-1` will have all bits to the right of that set bit flipped (e.g., 7 is `0111`). Their bitwise AND will be 0.

Java

```
class Solution {
    public boolean isPowerOfTwoOptimized(int n) {
        return n > 0 && (n & (n - 1)) == 0;
```

```
    }
}
```

- **Time Complexity:** O(1)
- **Space Complexity:** O(1)

2. Find the Only Non-Repeating Element

Problem: Given a non-empty array of integers, every element appears twice except for one. Find that single one.

Brute Force (Using HashMap/HashSet):

Count occurrences of each number using a HashMap, then find the one with count 1. Or use a HashSet to track seen numbers.

Java

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

class Solution {
    public int singleNumberBruteForce(int[] nums) {
        Map<Integer, Integer> counts = new HashMap<>();
        for (int num : nums) {
            counts.put(num, counts.getOrDefault(num, 0) + 1);
        }
        for (Map.Entry<Integer, Integer> entry : counts.entrySet()) {
            if (entry.getValue() == 1) {
                return entry.getKey();
            }
        }
        return -1; // Should not happen given problem statement
    }

    // Another brute force using HashSet
    public int singleNumberHashSet(int[] nums) {
        Set<Integer> set = new HashSet<>();
        for (int num : nums) {
            if (!set.add(num)) { // If add returns false, it means element already
exists
                set.remove(num); // So remove it
            }
        }
```

```
        return set.iterator().next(); // The remaining element is the single one
    }
}
```

- **Time Complexity:** O(N) (average for HashMap/HashSet operations)
- **Space Complexity:** O(N) (for storing counts or elements)

Most Optimized (Using XOR):

The XOR operator (`^`) has the property that `A ^ A = 0` and `A ^ 0 = A`. If you XOR all elements in the array, all numbers that appear twice will cancel each other out (result in 0), leaving only the single non-repeating element.

Java

```
class Solution {
    public int singleNumberOptimized(int[] nums) {
        int single = 0;
        for (int num : nums) {
            single ^= num;
        }
        return single;
    }
}
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(1)