DSA SOLUTION SHEET

1. Number of Good Pairs

Brute Force

- **C++:**

```cpp
int numIdenticalPairs(vector<int>& nums) {
    int count = 0;
    for(int i = 0; i < nums.size(); i++) {
        for(int j = i + 1; j < nums.size(); j++) {
            if(nums[i] == nums[j]) count++;
        }
    }
    return count;
}
```

- **Java:**

```java
public int numIdenticalPairs(int[] nums) {
    int count = 0;
    for(int i = 0; i < nums.length; i++) {
        for(int j = i + 1; j < nums.length; j++) {
            if(nums[i] == nums[j]) count++;
        }
    }
    return count;
}
```

Optimized Solution

- **C++:**

```
int numIdenticalPairs(vector<int>& nums) {
    unordered_map<int, int> freq;
    int count = 0;
    for(int num : nums) {
        count += freq[num]++;
    }
    return count;
}
```

- **Java:**

```
public int numIdenticalPairs(int[] nums) {
    Map<Integer, Integer> freq = new HashMap<>();
    int count = 0;
    for (int num : nums) {
        count += freq.getOrDefault(num, 0);
        freq.put(num, freq.getOrDefault(num, 0) + 1);
    }
    return count;
}
```

2. How Many Numbers Are Smaller Than the Current Number

Brute Force

- **C++:**

```
vector<int> smallerNumbersThanCurrent(vector<int>& nums) {
    vector<int> result(nums.size());
    for(int i = 0; i < nums.size(); i++) {
        int count = 0;
        for(int j = 0; j < nums.size(); j++) {
            if(nums[j] < nums[i]) count++;
```

```
        }
        result[i] = count;
    }
    return result;
}
```

- **Java:**

```
public int[] smallerNumbersThanCurrent(int[] nums) {
    int[] result = new int[nums.length];
    for(int i = 0; i < nums.length; i++) {
        int count = 0;
        for(int j = 0; j < nums.length; j++) {
            if(nums[j] < nums[i]) count++;
        }
        result[i] = count;
    }
    return result;
}
```

Optimized Solution

- **C++:**

```
vector<int> smallerNumbersThanCurrent(vector<int>& nums) {
    vector<int> sorted = nums, result(nums.size());
    sort(sorted.begin(), sorted.end());
    unordered_map<int, int> rank;
    for (int i = 0; i < sorted.size(); i++) {
        if (rank.find(sorted[i]) == rank.end()) rank[sorted[i]] = i;
    }
    for (int i = 0; i < nums.size(); i++) {
        result[i] = rank[nums[i]];
    }
```

```
    return result;
}
```

- **Java:**

```java
public int[] smallerNumbersThanCurrent(int[] nums) {
    int[] sorted = nums.clone();
    Arrays.sort(sorted);
    Map<Integer, Integer> rank = new HashMap<>();
    for (int i = 0; i < sorted.length; i++) {
        rank.putIfAbsent(sorted[i], i);
    }
    int[] result = new int[nums.length];
    for (int i = 0; i < nums.length; i++) {
        result[i] = rank.get(nums[i]);
    }
    return result;
}
```

3. Two Sum

Brute Force

- **C++:**

```cpp
vector<int> twoSum(vector<int>& nums, int target) {
    for(int i = 0; i < nums.size(); i++) {
        for(int j = i + 1; j < nums.size(); j++) {
            if(nums[i] + nums[j] == target) {
                return {i, j};
            }
        }
    }
    return {};
```

```
}
```

- **Java:**

```
public int[] twoSum(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[]{};
}
```

Optimized Solution

- **C++:**

```
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> map;
    for(int i = 0; i < nums.size(); i++) {
        int complement = target - nums[i];
        if(map.find(complement) != map.end()) {
            return {map[complement], i};
        }
        map[nums[i]] = i;
    }
    return {};
}
```

- **Java:**

```
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[]{};
}
```

4. Remove Duplicates from Sorted Array

Brute Force

- **C++:**

```
int removeDuplicates(vector<int>& nums) {
    if(nums.empty()) return 0;
    int index = 1;
    for(int i = 1; i < nums.size(); i++) {
        if(nums[i] != nums[i-1]) {
            nums[index++] = nums[i];
        }
    }
    return index;
}
```

- **Java:**

```
public int removeDuplicates(int[] nums) {
    if(nums.length == 0) return 0;
    int index = 1;
    for(int i = 1; i < nums.length; i++) {
        if(nums[i] != nums[i-1]) {
```

```
            nums[index++] = nums[i];
        }
    }
    return index;
}
```

5. Missing Number

Brute Force

- **C++:**

```
int missingNumber(vector<int>& nums) {
    int n = nums.size();
    for (int i = 0; i <= n; i++) {
        if (find(nums.begin(), nums.end(), i) == nums.end()) {
            return i;
        }
    }
    return -1;
}
```

- **Java:**

```
public int missingNumber(int[] nums) {
    int n = nums.length;
    for (int i = 0; i <= n; i++) {
        boolean found = false;
        for (int num : nums) {
            if (num == i) {
                found = true;
                break;
            }
        }
```

```
        if (!found) return i;
    }
    return -1;
}
```

Optimized Solution

- **C++:**

```
int missingNumber(vector<int>& nums) {
    int n = nums.size();
    int sum = n * (n + 1) / 2;
    for (int num : nums) {
        sum -= num;
    }
    return sum;
}
```

- **Java:**

```
public int missingNumber(int[] nums) {
    int n = nums.length;
    int sum = n * (n + 1) / 2;
    for (int num : nums) {
        sum -= num;
    }
    return sum;
}
```

6. Majority Element

Brute Force

- **C++:**

```
int majorityElement(vector<int>& nums) {
    for (int i = 0; i < nums.size(); i++) {
```

```
        int count = 0;
        for (int j = 0; j < nums.size(); j++) {
            if (nums[j] == nums[i]) count++;
        }
        if (count > nums.size() / 2) return nums[i];
    }
    return -1;
}
```

- **Java:**

```
public int majorityElement(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        int count = 0;
        for (int j = 0; j < nums.length; j++) {
            if (nums[j] == nums[i]) count++;
        }
        if (count > nums.length / 2) return nums[i];
    }
    return -1;
}
```

Optimized Solution (Boyer-Moore Voting Algorithm)

- **C++:**

```
int majorityElement(vector<int>& nums) {
    int count = 0, candidate = 0;
    for (int num : nums) {
        if (count == 0) candidate = num;
        count += (num == candidate) ? 1 : -1;
    }
    return candidate;
```

```
}
```

- **Java:**

```java
public int majorityElement(int[] nums) {
    int count = 0, candidate = 0;
    for (int num : nums) {
        if (count == 0) candidate = num;
        count += (num == candidate) ? 1 : -1;
    }
    return candidate;
}
```

7. Sort Colors

Brute Force (Counting Sort)

- **C++:**

```cpp
void sortColors(vector<int>& nums) {
    int count[3] = {0, 0, 0};
    for (int num : nums) count[num]++;
    int index = 0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < count[i]; j++) {
            nums[index++] = i;
        }
    }
}
```

- **Java:**

```java
public void sortColors(int[] nums) {
    int[] count = new int[3];
    for (int num : nums) count[num]++;
    int index = 0;
```

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < count[i]; j++) {
            nums[index++] = i;
        }
    }
}
```

Optimized Solution (Dutch National Flag Algorithm)

- **C++:**

```cpp
void sortColors(vector<int>& nums) {
    int low = 0, mid = 0, high = nums.size() - 1;
    while (mid <= high) {
        if (nums[mid] == 0) swap(nums[low++], nums[mid++]);
        else if (nums[mid] == 1) mid++;
        else swap(nums[mid], nums[high--]);
    }
}
```

- **Java:**

```java
public void sortColors(int[] nums) {
    int low = 0, mid = 0, high = nums.length - 1;
    while (mid <= high) {
        if (nums[mid] == 0) {
            swap(nums, low++, mid++);
        } else if (nums[mid] == 1) {
            mid++;
        } else {
            swap(nums, mid, high--);
        }
```

```
    }
}

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

8. Maximum Subarray

Brute Force

- **C++:**

```
int maxSubArray(vector<int>& nums) {
    int maxSum = INT_MIN;
    for (int i = 0; i < nums.size(); i++) {
        int currentSum = 0;
        for (int j = i; j < nums.size(); j++) {
            currentSum += nums[j];
            maxSum = max(maxSum, currentSum);
        }
    }
    return maxSum;
}
```

- **Java:**

```
public int maxSubArray(int[] nums) {
    int maxSum = Integer.MIN_VALUE;
    for (int i = 0; i < nums.length; i++) {
        int currentSum = 0;
        for (int j = i; j < nums.length; j++) {
```

```
            currentSum += nums[j];
            maxSum = Math.max(maxSum, currentSum);
        }
    }
    return maxSum;
}
```

Optimized Solution (Kadane's Algorithm)

- **C++:**

```
int maxSubArray(vector<int>& nums) {
    int maxSum = nums[0], currentSum = nums[0];
    for (int i = 1; i < nums.size(); i++) {
        currentSum = max(nums[i], currentSum + nums[i]);
        maxSum = max(maxSum, currentSum);
    }
    return maxSum;
}
```

- **Java:**

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

9. Set Matrix Zeroes

Brute Force

- **C++:**

```
void setZeroes(vector<vector<int>>& matrix) {
    int m = matrix.size(), n = matrix[0].size();
    vector<int> row(m, 1), col(n, 1);

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == 0) {
                row[i] = 0;
                col[j] = 0;
            }
        }
    }

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (row[i] == 0 || col[j] == 0) {
                matrix[i][j] = 0;
            }
        }
    }
}
```

- **Java:**

```
public void setZeroes(int[][] matrix) {
    int m = matrix.length, n = matrix[0].length;
    boolean[] row = new boolean[m], col = new boolean[n];

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == 0) {
```

```
                row[i] = true;
                col[j] = true;
            }
        }
    }

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (row[i] || col[j]) {
                matrix[i][j] = 0;
            }
        }
    }
}
```

Optimized Solution (Using First Row and Column as Markers)

- **C++:**

```
void setZeroes(vector<vector<int>>& matrix) {
    int m = matrix.size(), n = matrix[0].size();
    bool firstRow = false, firstCol = false;

    for (int i = 0; i < m; i++) if (matrix[i][0] == 0) firstCol = true;
    for (int j = 0; j < n; j++) if (matrix[0][j] == 0) firstRow = true;

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (matrix[i][j] == 0) {
                matrix[i][0] = 0;
                matrix[0][j] = 0;
            }
```

```
        }
    }

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (matrix[i][0] == 0 || matrix[0][j] == 0) matrix[i][j] = 0;
        }
    }

    if (firstRow) for (int j = 0; j < n; j++) matrix[0][j] = 0;
    if (firstCol) for (int i = 0; i < m; i++) matrix[i][0] = 0;
}
```

- **Java:**

```
public void setZeroes(int[][] matrix) {
    int m = matrix.length, n = matrix[0].length;
    boolean firstRow = false, firstCol = false;

    for (int i = 0; i < m; i++) if (matrix[i][0] == 0) firstCol = true;
    for (int j = 0; j < n; j++) if (matrix[0][j] == 0) firstRow = true;

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (matrix[i][j] == 0) {
                matrix[i][0] = 0;
                matrix[0][j] = 0;
            }
        }
    }
```

```
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (matrix[i][0] == 0 || matrix[0][j] == 0) matrix[i][j] = 0;
        }
    }

    if (firstRow) for (int j = 0; j < n; j++) matrix[0][j] = 0;
    if (firstCol) for (int i = 0; i < m; i++) matrix[i][0] = 0;
}
```

10. Container With Most Water

Brute Force

- **C++:**

```
int maxArea(vector<int>& height) {
    int maxArea = 0;
    for (int i = 0; i < height.size(); i++) {
        for (int j = i + 1; j < height.size(); j++) {
            int area = min(height[i], height[j]) * (j - i);
            maxArea = max(maxArea, area);
        }
    }
    return maxArea;
}
```

- **Java:**

```
public int maxArea(int[] height) {
    int maxArea = 0;
    for (int i = 0; i < height.length; i++) {
```

```
        for (int j = i + 1; j < height.length; j++) {
            int area = Math.min(height[i], height[j]) * (j - i);
            maxArea = Math.max(maxArea, area);
        }
    }
    return maxArea;
}
```

Optimized Solution (Two-pointer approach)

- **C++:**

```
int maxArea(vector<int>& height) {
    int left = 0, right = height.size() - 1;
    int maxArea = 0;
    while (left < right) {
        int area = min(height[left], height[right]) * (right - left);
        maxArea = max(maxArea, area);
        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }
    return maxArea;
}
```

- **Java:**

```
public int maxArea(int[] height) {
```

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

11. Search a 2D Matrix

Brute Force

- **C++:**

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    for (int i = 0; i < matrix.size(); i++) {
        for (int j = 0; j < matrix[0].size(); j++) {
            if (matrix[i][j] == target) return true;
        }
    }
    return false;
}
```

- **Java:**

```
public boolean searchMatrix(int[][] matrix, int target) {
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[0].length; j++) {
```

```
            if (matrix[i][j] == target) return true;
        }
    }
    return false;
}
```

Optimized Solution (Binary Search)

- **C++:**

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    if (matrix.empty()) return false;
    int m = matrix.size(), n = matrix[0].size();
    int left = 0, right = m * n - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        int midVal = matrix[mid / n][mid % n];
        if (midVal == target) return true;
        else if (midVal < target) left = mid + 1;
        else right = mid - 1;
    }
    return false;
}
```

- **Java:**

```
public boolean searchMatrix(int[][] matrix, int target) {
    if (matrix.length == 0) return false;
    int m = matrix.length, n = matrix[0].length;
```

```
    int left = 0, right = m * n - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        int midVal = matrix[mid / n][mid % n];
        if (midVal == target) return true;
        else if (midVal < target) left = mid + 1;
        else right = mid - 1;
    }
    return false;
}
```

12. Reverse Pairs

Brute Force

- **C++:**

```
int reversePairs(vector<int>& nums) {
    int count = 0;
    for (int i = 0; i < nums.size(); i++) {
        for (int j = i + 1; j < nums.size(); j++) {
            if (nums[i] > 2LL * nums[j]) count++;
        }
    }
    return count;
}
```

- **Java:**

```
public int reversePairs(int[] nums) {
    int count = 0;
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
```

```
            if (nums[i] > 2L * nums[j]) count++;
        }
    }
    return count;
}
```

Optimized Solution (Merge Sort)

- **C++:**

```
int mergeAndCount(vector<int>& nums, int left, int mid, int right) {
    int count = 0, j = mid + 1;
    for (int i = left; i <= mid; i++) {
        while (j <= right && nums[i] > 2LL * nums[j]) j++;
        count += (j - mid - 1);
    }

    vector<int> temp;
    int i = left, k = mid + 1;
    while (i <= mid && k <= right) {
        if (nums[i] <= nums[k]) temp.push_back(nums[i++]);
        else temp.push_back(nums[k++]);
    }

    while (i <= mid) temp.push_back(nums[i++]);
    while (k <= right) temp.push_back(nums[k++]);

    for (i = left; i <= right; i++) nums[i] = temp[i - left];

    return count;
```

```cpp
}

int mergeSort(vector<int>& nums, int left, int right) {
    if (left >= right) return 0;
    int mid = left + (right - left) / 2;
    int count = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);
    count += mergeAndCount(nums, left, mid, right);
    return count;
}

int reversePairs(vector<int>& nums) {
    return mergeSort(nums, 0, nums.size() - 1);
}
```

- **Java:**

```java
public class Solution {
    public int reversePairs(int[] nums) {
        return mergeSort(nums, 0, nums.length - 1);
    }

    private int mergeSort(int[] nums, int left, int right) {
        if (left >= right) return 0;
        int mid = left + (right - left) / 2;
        int count = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);
        count += mergeAndCount(nums, left, mid, right);
        return count;
    }

    private int mergeAndCount(int[] nums, int left, int mid, int right) {
        int count = 0, j = mid + 1;
```

```java
        for (int i = left; i <= mid; i++) {
            while (j <= right && nums[i] > 2L * nums[j]) j++;
            count += (j - mid - 1);
        }

        List<Integer> temp = new ArrayList<>();
        int i = left, k = mid + 1;
        while (i <= mid && k <= right) {
            if (nums[i] <= nums[k]) temp.add(nums[i++]);
            else temp.add(nums[k++]);
        }

        while (i <= mid) temp.add(nums[i++]);
        while (k <= right) temp.add(nums[k++]);

        for (i = left; i <= right; i++) nums[i] = temp.get(i - left);

        return count;
    }
}
```

SLIDING WINDOW –

1. Minimum Size Subarray Sum

C++:

```
int minSubArrayLen(int target, vector<int>& nums) {
    int left = 0, sum = 0, minLen = INT_MAX;
    for (int right = 0; right < nums.size(); right++) {
        sum += nums[right];
        while (sum >= target) {
            minLen = min(minLen, right - left + 1);
            sum -= nums[left++];
        }
    }
    return minLen == INT_MAX ? 0 : minLen;
}
```

Java:

```
public int minSubArrayLen(int target, int[] nums) {
    int left = 0, sum = 0, minLen = Integer.MAX_VALUE;
    for (int right = 0; right < nums.length; right++) {
        sum += nums[right];
        while (sum >= target) {
            minLen = Math.min(minLen, right - left + 1);
            sum -= nums[left++];
        }
    }
```

```
    return minLen == Integer.MAX_VALUE ? 0 : minLen;
}
```

2. Max Consecutive Ones III

C++:

```cpp
int longestOnes(vector<int>& nums, int k) {
    int left = 0, maxLen = 0, zeroCount = 0;
    for (int right = 0; right < nums.size(); right++) {
        if (nums[right] == 0) zeroCount++;
        while (zeroCount > k) {
            if (nums[left++] == 0) zeroCount--;
        }
        maxLen = max(maxLen, right - left + 1);
    }
    return maxLen;
}
```

Java:

```java
public int longestOnes(int[] nums, int k) {
    int left = 0, maxLen = 0, zeroCount = 0;
    for (int right = 0; right < nums.length; right++) {
        if (nums[right] == 0) zeroCount++;
        while (zeroCount > k) {
            if (nums[left++] == 0) zeroCount--;
        }
        maxLen = Math.max(maxLen, right - left + 1);
    }
    return maxLen;
}
```

3. Minimum Operations to Reduce X to Zero

C++:

```
int minOperations(vector<int>& nums, int x) {
    int totalSum = accumulate(nums.begin(), nums.end(), 0);
    int target = totalSum - x;
    if (target < 0) return -1;

    int left = 0, currentSum = 0, maxLen = -1;
    for (int right = 0; right < nums.size(); right++) {
        currentSum += nums[right];
        while (currentSum > target && left <= right) {
            currentSum -= nums[left++];
        }
        if (currentSum == target) {
            maxLen = max(maxLen, right - left + 1);
        }
    }
    return maxLen == -1 ? -1 : nums.size() - maxLen;
}
```

Java:

```
public int minOperations(int[] nums, int x) {
    int totalSum = Arrays.stream(nums).sum();
    int target = totalSum - x;
    if (target < 0) return -1;

    int left = 0, currentSum = 0, maxLen = -1;
    for (int right = 0; right < nums.length; right++) {
```

```
        currentSum += nums[right];
        while (currentSum > target && left <= right) {
            currentSum -= nums[left++];
        }
        if (currentSum == target) {
            maxLen = Math.max(maxLen, right - left + 1);
        }
    }
    return maxLen == -1 ? -1 : nums.length - maxLen;
}
```

4. Minimum Window Substring

C++:

```
string minWindow(string s, string t) {
    unordered_map<char, int> charCount;
    for (char c : t) charCount[c]++;

    int left = 0, count = 0, minLen = INT_MAX, start = 0;
    for (int right = 0; right < s.size(); right++) {
        if (--charCount[s[right]] >= 0) count++;

        while (count == t.size()) {
            if (right - left + 1 < minLen) {
                minLen = right - left + 1;
                start = left;
            }
            if (++charCount[s[left++]] > 0) count--;
```

```
        }
    }
    return minLen == INT_MAX ? "" : s.substr(start, minLen);
}
```

Java:

```java
public String minWindow(String s, String t) {
    Map<Character, Integer> charCount = new HashMap<>();
    for (char c : t.toCharArray()) charCount.put(c, charCount.getOrDefault(c, 0) + 1);

    int left = 0, count = 0, minLen = Integer.MAX_VALUE, start = 0;
    for (int right = 0; right < s.length(); right++) {
        char rChar = s.charAt(right);
        if (charCount.containsKey(rChar)) {
            charCount.put(rChar, charCount.get(rChar) - 1);
            if (charCount.get(rChar) >= 0) count++;
        }

        while (count == t.length()) {
            if (right - left + 1 < minLen) {
                minLen = right - left + 1;
                start = left;
            }
            char lChar = s.charAt(left++);
            if (charCount.containsKey(lChar)) {
                charCount.put(lChar, charCount.get(lChar) + 1);
                if (charCount.get(lChar) > 0) count--;
            }
```

```
        }
    }
    return minLen == Integer.MAX_VALUE ? "" : s.substring(start, start + minLen);
}
```

5. Frequency of the Most Frequent Element

C++:

```
int maxFrequency(vector<int>& nums, int k) {
    sort(nums.begin(), nums.end());
    long long total = 0;
    int left = 0, maxFreq = 0;

    for (int right = 0; right < nums.size(); right++) {
        total += nums[right];
        while (nums[right] * (right - left + 1) > total + k) {
            total -= nums[left++];
        }
        maxFreq = max(maxFreq, right - left + 1);
    }
    return maxFreq;
}
```

Java:

```
public int maxFrequency(int[] nums, int k) {
```

```
    Arrays.sort(nums);
    long total = 0;
    int left = 0, maxFreq = 0;

    for (int right = 0; right < nums.length; right++) {
        total += nums[right];
        while (nums[right] * (right - left + 1) > total + k) {
            total -= nums[left++];
        }
        maxFreq = Math.max(maxFreq, right - left + 1);
    }
    return maxFreq;
}
```

TWO POINTERS

1. Sum of Square Numbers

C++:

```
bool judgeSquareSum(int c) {
    long left = 0, right = sqrt(c);
    while (left <= right) {
        long sum = left * left + right * right;
        if (sum == c) return true;
        else if (sum < c) left++;
        else right--;
    }
    return false;
}
```

Java:

```
public boolean judgeSquareSum(int c) {
    long left = 0, right = (long) Math.sqrt(c);
    while (left <= right) {
        long sum = left * left + right * right;
        if (sum == c) return true;
        else if (sum < c) left++;
        else right--;
    }
    return false;
}
```

2. Number of Subsequences That Satisfy the Given Sum Condition

C++:

```
int numSubseq(vector<int>& nums, int target) {
    sort(nums.begin(), nums.end());
```

```
    int mod = 1e9 + 7;
    int left = 0, right = nums.size() - 1, result = 0;
    vector<int> powers(nums.size(), 1);

    for (int i = 1; i < nums.size(); i++) {
        powers[i] = (powers[i - 1] * 2) % mod;
    }

    while (left <= right) {
        if (nums[left] + nums[right] <= target) {
            result = (result + powers[right - left]) % mod;
            left++;
        } else {
            right--;
        }
    }

    return result;
}
```

Java:

```
public int numSubseq(int[] nums, int target) {
    Arrays.sort(nums);
    int mod = (int) 1e9 + 7;
    int left = 0, right = nums.length - 1, result = 0;
```

```
    int[] powers = new int[nums.length];
    powers[0] = 1;

    for (int i = 1; i < nums.length; i++) {
        powers[i] = (powers[i - 1] * 2) % mod;
    }

    while (left <= right) {
        if (nums[left] + nums[right] <= target) {
            result = (result + powers[right - left]) % mod;
            left++;
        } else {
            right--;
        }
    }

    return result;
}
```

3. Minimize Maximum Pair Sum in Array

C++:

```
int minPairSum(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    int left = 0, right = nums.size() - 1, maxPairSum = 0;
```

```
    while (left < right) {
        maxPairSum = max(maxPairSum, nums[left] + nums[right]);
        left++;
        right--;
    }

    return maxPairSum;
}
```

Java:

```
public int minPairSum(int[] nums) {
    Arrays.sort(nums);
    int left = 0, right = nums.length - 1, maxPairSum = 0;

    while (left < right) {
        maxPairSum = Math.max(maxPairSum, nums[left] + nums[right]);
        left++;
        right--;
    }

    return maxPairSum;
}
```

4. Trapping Rain Water

C++:

```
int trap(vector<int>& height) {
    int left = 0, right = height.size() - 1, leftMax = 0, rightMax = 0, trappedWater = 0;

    while (left < right) {
        if (height[left] < height[right]) {
```

```
            if (height[left] >= leftMax) leftMax = height[left];
            else trappedWater += leftMax - height[left];
            left++;
        } else {
            if (height[right] >= rightMax) rightMax = height[right];
            else trappedWater += rightMax - height[right];
            right--;
        }
    }

    return trappedWater;
}
```

Java:

java

Copy code

```java
public int trap(int[] height) {
    int left = 0, right = height.length - 1, leftMax = 0, rightMax = 0, trappedWater = 0;

    while (left < right) {
        if (height[left] < height[right]) {
            if (height[left] >= leftMax) leftMax = height[left];
            else trappedWater += leftMax - height[left];
            left++;
        } else {
            if (height[right] >= rightMax) rightMax = height[right];
            else trappedWater += rightMax - height[right];
            right--;
```

```
        }
    }

    return trappedWater;
}
```

STRINGS

1. Valid Anagram

Brute Force:

Use two frequency arrays to count characters in both strings and then compare.

C++:

```
bool isAnagram(string s, string t) {
    if (s.size() != t.size()) return false;
    vector<int> countS(26, 0), countT(26, 0);
    for (int i = 0; i < s.size(); i++) {
```

```cpp
        countS[s[i] - 'a']++;
        countT[t[i] - 'a']++;
    }
    return countS == countT;
}
```

Java:

```java
public boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) return false;
    int[] countS = new int[26];
    int[] countT = new int[26];
    for (int i = 0; i < s.length(); i++) {
        countS[s.charAt(i) - 'a']++;
        countT[t.charAt(i) - 'a']++;
    }
    return Arrays.equals(countS, countT);
}
```

Optimized:

Use a single frequency array for comparison.

C++:

```cpp
bool isAnagram(string s, string t) {
    if (s.size() != t.size()) return false;
    vector<int> count(26, 0);
    for (int i = 0; i < s.size(); i++) {
        count[s[i] - 'a']++;
        count[t[i] - 'a']--;
    }
```

```
    for (int i : count) {
        if (i != 0) return false;
    }
    return true;
}
```

Java:

```
public boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) return false;
    int[] count = new int[26];
    for (int i = 0; i < s.length(); i++) {
        count[s.charAt(i) - 'a']++;
        count[t.charAt(i) - 'a']--;
    }
    for (int i : count) {
        if (i != 0) return false;
    }
    return true;
}
```

2. Isomorphic Strings

Brute Force:

For each character, check if the mapping between the two strings is consistent.

C++:

```
bool isIsomorphic(string s, string t) {
    unordered_map<char, char> mapStoT, mapTtoS;
    for (int i = 0; i < s.size(); i++) {
        if (mapStoT.count(s[i]) && mapStoT[s[i]] != t[i]) return false;
        if (mapTtoS.count(t[i]) && mapTtoS[t[i]] != s[i]) return false;
        mapStoT[s[i]] = t[i];
```

```
        mapTtoS[t[i]] = s[i];
    }
    return true;
}
```

Java:

```
public boolean isIsomorphic(String s, String t) {
    Map<Character, Character> mapStoT = new HashMap<>();
    Map<Character, Character> mapTtoS = new HashMap<>();
    for (int i = 0; i < s.length(); i++) {
        char c1 = s.charAt(i), c2 = t.charAt(i);
        if (mapStoT.containsKey(c1) && mapStoT.get(c1) != c2 ||
mapTtoS.containsKey(c2) && mapTtoS.get(c2) != c1)
            return false;
        mapStoT.put(c1, c2);
        mapTtoS.put(c2, c1);
    }
    return true;
}
```

Optimized:

Use arrays for mapping instead of HashMap.

C++:

```
bool isIsomorphic(string s, string t) {
    vector<int> mapStoT(256, -1), mapTtoS(256, -1);
    for (int i = 0; i < s.size(); i++) {
        if (mapStoT[s[i]] != mapTtoS[t[i]]) return false;
        mapStoT[s[i]] = mapTtoS[t[i]] = i;
    }
    return true;
}
```

Java:

```java
public boolean isIsomorphic(String s, String t) {
    int[] mapStoT = new int[256], mapTtoS = new int[256];
    Arrays.fill(mapStoT, -1);
    Arrays.fill(mapTtoS, -1);
    for (int i = 0; i < s.length(); i++) {
        if (mapStoT[s.charAt(i)] != mapTtoS[t.charAt(i)]) return false;
        mapStoT[s.charAt(i)] = mapTtoS[t.charAt(i)] = i;
    }
    return true;
}
```

3. Longest Common Prefix

Brute Force:

Compare characters one by one in all strings and stop at the first mismatch.

C++:

```cpp
string longestCommonPrefix(vector<string>& strs) {
    if (strs.empty()) return "";
    string prefix = strs[0];
    for (int i = 1; i < strs.size(); i++) {
        int j = 0;
        while (j < prefix.size() && j < strs[i].size() && prefix[j] == strs[i][j]) {
            j++;
        }
        prefix = prefix.substr(0, j);
```

```
        if (prefix.empty()) return "";
    }
    return prefix;
}
```

Java:

```
public String longestCommonPrefix(String[] strs) {
    if (strs.length == 0) return "";
    String prefix = strs[0];
    for (int i = 1; i < strs.length; i++) {
        int j = 0;
        while (j < prefix.length() && j < strs[i].length() && prefix.charAt(j) ==
strs[i].charAt(j)) {
            j++;
        }
        prefix = prefix.substring(0, j);
        if (prefix.isEmpty()) return "";
    }
    return prefix;
}
```

Optimized:

Compare character by character across all strings.

C++:

```
string longestCommonPrefix(vector<string>& strs) {
    if (strs.empty()) return "";
    for (int i = 0; i < strs[0].size(); i++) {
        char c = strs[0][i];
        for (int j = 1; j < strs.size(); j++) {
            if (i == strs[j].size() || strs[j][i] != c) {
                return strs[0].substr(0, i);
```

```
            }
        }
    }
    return strs[0];
}
```

Java:

```
public String longestCommonPrefix(String[] strs) {
    if (strs.length == 0) return "";
    for (int i = 0; i < strs[0].length(); i++) {
        char c = strs[0].charAt(i);
        for (int j = 1; j < strs.length; j++) {
            if (i == strs[j].length() || strs[j].charAt(i) != c) {
                return strs[0].substring(0, i);
            }
        }
    }
    return strs[0];
}
```

4. Reverse Words in a String

Brute Force:

Split the string by spaces and reverse the array of words.

C++:

```
string reverseWords(string s) {
    stringstream ss(s);
    string word, result;
    vector<string> words;
    while (ss >> word) {
        words.push_back(word);
```

```
    }
    reverse(words.begin(), words.end());
    for (int i = 0; i < words.size(); i++) {
        result += words[i];
        if (i != words.size() - 1) result += " ";
    }
    return result;
}
```

Java:

```
public String reverseWords(String s) {
    String[] words = s.trim().split("\\s+");
    StringBuilder result = new StringBuilder();
    for (int i = words.length - 1; i >= 0; i--) {
        result.append(words[i]);
        if (i != 0) result.append(" ");
    }
    return result.toString();
}
```

Optimized:

Trim spaces, reverse the entire string, and then reverse each word.

C++:

```
string reverseWords(string s) {
    reverse(s.begin(), s.end());
    int start = 0;
    for (int end = 0; end < s.size(); end++) {
        if (s[end] == ' ') {
            reverse(s.begin() + start, s.begin() + end);
            start = end + 1;
        }
```

```
    }
    reverse(s.begin() + start, s.end());
    return s;
}
```

Java:

```
public String reverseWords(String s) {
    StringBuilder sb = new StringBuilder(s.trim());
    sb.reverse();
    String[] words = sb.toString().split(" ");
    sb.setLength(0);
    for (String word : words) {
        sb.append(new StringBuilder(word).reverse().toString()).append(" ");
    }
    return sb.toString().trim();
}
```

5. Group Anagrams

Brute Force:

Sort each word and group them in a map.

C++:

```
vector<vector<string>> groupAnagrams(vector<string>& strs) {
    unordered_map<string, vector<string>> anagrams;
    for (string str : strs) {
        string sortedStr = str;
        sort(sortedStr.begin(), sortedStr.end());
        anagrams[sortedStr].push_back(str);
    }
    vector<vector<string>> result;
    for (auto& pair : anagrams) {
```

```cpp
        result.push_back(pair.second);
    }
    return result;
}
```

Java:

```java
public List<List<String>> groupAnagrams(String[] strs) {
    Map<String, List<String>> anagrams = new HashMap<>();
    for (String str : strs) {
        char[] chars = str.toCharArray();
        Arrays.sort(chars);
        String sortedStr = new String(chars);
        anagrams.computeIfAbsent(sortedStr, k -> new ArrayList<>()).add(str);
    }
    return new ArrayList<>(anagrams.values());
}
```

Optimized:

Use frequency count (character counts) as keys instead of sorting.

C++:

```cpp
vector<vector<string>> groupAnagrams(vector<string>& strs) {
    unordered_map<string, vector<string>> anagrams;
    for (string& str : strs) {
        vector<int> count(26, 0);
        for (char c : str) count[c - 'a']++;
        string key = "";
        for (int i : count) key += to_string(i) + "#";
        anagrams[key].push_back(str);
    }
    vector<vector<string>> result;
    for (auto& pair : anagrams) {
```

```
        result.push_back(pair.second);
    }
    return result;
}
```

Java:

```
public List<List<String>> groupAnagrams(String[] strs) {
    Map<String, List<String>> anagrams = new HashMap<>();
    for (String str : strs) {
        int[] count = new int[26];
        for (char c : str.toCharArray()) {
            count[c - 'a']++;
        }
        StringBuilder sb = new StringBuilder();
        for (int i : count) {
            sb.append(i).append("#");
        }
        String key = sb.toString();
        anagrams.computeIfAbsent(key, k -> new ArrayList<>()).add(str);
    }
    return new ArrayList<>(anagrams.values());
}
```

6. Sum of Beauty of All Substrings

Brute Force:

Generate all substrings and calculate beauty for each substring.

C++:

```
int beautySum(string s) {
    int result = 0;
    for (int i = 0; i < s.size(); i++) {
```

```
        for (int j = i + 1; j <= s.size(); j++) {
            vector<int> freq(26, 0);
            for (int k = i; k < j; k++) freq[s[k] - 'a']++;
            int maxFreq = *max_element(freq.begin(), freq.end());
            int minFreq = INT_MAX;
            for (int f : freq) if (f > 0) minFreq = min(minFreq, f);
            result += maxFreq - minFreq;
        }
    }
    return result;
}
```

Java:

```
public int beautySum(String s) {
    int result = 0;
    for (int i = 0; i < s.length(); i++) {
        for (int j = i + 1; j <= s.length(); j++) {
            int[] freq = new int[26];
            for (int k = i; k < j; k++) freq[s.charAt(k) - 'a']++;
            int maxFreq = Arrays.stream(freq).max().getAsInt();
            int minFreq = Integer.MAX_VALUE;
            for (int f : freq) if (f > 0) minFreq = Math.min(minFreq, f);
            result += maxFreq - minFreq;
        }
    }
    return result;
}
```

Optimized:

Use sliding window and a frequency count array to calculate the beauty efficiently.

C++:

```
int beautySum(string s) {
    int result = 0;
    for (int i = 0; i < s.size(); i++) {
        vector<int> freq(26, 0);
        for (int j = i; j < s.size(); j++) {
            freq[s[j] - 'a']++;
            int maxFreq = *max_element(freq.begin(), freq.end());
            int minFreq = INT_MAX;
            for (int f : freq) if (f > 0) minFreq = min(minFreq, f);
            result += maxFreq - minFreq;
        }
    }
    return result;
}
```

Java:

```
public int beautySum(String s) {
    int result = 0;
    for (int i = 0; i < s.length(); i++) {
        int[] freq = new int[26];
        for (int j = i; j < s.length(); j++) {
            freq[s.charAt(j) - 'a']++;
            int maxFreq = Arrays.stream(freq).max().getAsInt();
            int minFreq = Integer.MAX_VALUE;
            for (int f : freq) if (f > 0) minFreq = Math.min(minFreq, f);
            result += maxFreq - minFreq;
        }
    }
    return result;
}
```

4. Reverse Words in a String

Brute Force:

Split the string by spaces and reverse the array of words.

C++:

```cpp
string reverseWords(string s) {
    stringstream ss(s);
    string word, result;
    vector<string> words;
    while (ss >> word) {
        words.push_back(word);
    }
    reverse(words.begin(), words.end());
    for (int i = 0; i < words.size(); i++) {
        result += words[i];
        if (i != words.size() - 1) result += " ";
    }
    return result;
}
```

Java:

```java
public String reverseWords(String s) {
    String[] words = s.trim().split("\\s+");
    StringBuilder result = new StringBuilder();
    for (int i = words.length - 1; i >= 0; i--) {
        result.append(words[i]);
        if (i != 0) result.append(" ");
    }
    return result.toString();
}
```

Optimized:

Trim spaces, reverse the entire string, and then reverse each word.

C++:

```
string reverseWords(string s) {
    reverse(s.begin(), s.end());
    int start = 0;
    for (int end = 0; end < s.size(); end++) {
        if (s[end] == ' ') {
            reverse(s.begin() + start, s.begin() + end);
            start = end + 1;
        }
    }
    reverse(s.begin() + start, s.end());
    return s;
}
```

Java:

```
public String reverseWords(String s) {
    StringBuilder sb = new StringBuilder(s.trim());
    sb.reverse();
    String[] words = sb.toString().split(" ");
    sb.setLength(0);
    for (String word : words) {
        sb.append(new StringBuilder(word).reverse().toString()).append(" ");
    }
    return sb.toString().trim();
}
```

5. Group Anagrams

Brute Force:

Sort each word and group them in a map.

C++:

```
vector<vector<string>> groupAnagrams(vector<string>& strs) {
    unordered_map<string, vector<string>> anagrams;
    for (string str : strs) {
        string sortedStr = str;
        sort(sortedStr.begin(), sortedStr.end());
        anagrams[sortedStr].push_back(str);
    }
    vector<vector<string>> result;
    for (auto& pair : anagrams) {
        result.push_back(pair.second);
    }
    return result;
}
```

Java:

```
public List<List<String>> groupAnagrams(String[] strs) {
    Map<String, List<String>> anagrams = new HashMap<>();
    for (String str : strs) {
        char[] chars = str.toCharArray();
        Arrays.sort(chars);
        String sortedStr = new String(chars);
        anagrams.computeIfAbsent(sortedStr, k -> new ArrayList<>()).add(str);
    }
    return new ArrayList<>(anagrams.values());
}
```

Optimized:

Use frequency count (character counts) as keys instead of sorting.

C++:

```
vector<vector<string>> groupAnagrams(vector<string>& strs) {
    unordered_map<string, vector<string>> anagrams;
    for (string& str : strs) {
        vector<int> count(26, 0);
        for (char c : str) count[c - 'a']++;
        string key = "";
        for (int i : count) key += to_string(i) + "#";
        anagrams[key].push_back(str);
    }
    vector<vector<string>> result;
    for (auto& pair : anagrams) {
        result.push_back(pair.second);
    }
    return result;
}
```

Java:

```
public List<List<String>> groupAnagrams(String[] strs) {
    Map<String, List<String>> anagrams = new HashMap<>();
    for (String str : strs) {
        int[] count = new int[26];
        for (char c : str.toCharArray()) {
            count[c - 'a']++;
        }
        StringBuilder sb = new StringBuilder();
        for (int i : count) {
            sb.append(i).append("#");
        }
        String key = sb.toString();
        anagrams.computeIfAbsent(key, k -> new ArrayList<>()).add(str);
```

```
    }
    return new ArrayList<>(anagrams.values());
}
```

6. Sum of Beauty of All Substrings

Brute Force:

Generate all substrings and calculate beauty for each substring.

C++:

```
int beautySum(string s) {
    int result = 0;
    for (int i = 0; i < s.size(); i++) {
        for (int j = i + 1; j <= s.size(); j++) {
            vector<int> freq(26, 0);
            for (int k = i; k < j; k++) freq[s[k] - 'a']++;
            int maxFreq = *max_element(freq.begin(), freq.end());
            int minFreq = INT_MAX;
            for (int f : freq) if (f > 0) minFreq = min(minFreq, f);
            result += maxFreq - minFreq;
        }
    }
    return result;
}
```

Java:

```
public int beautySum(String s) {
    int result = 0;
    for (int i = 0; i < s.length(); i++) {
        for (int j = i + 1; j <= s.length(); j++) {
            int[] freq = new int[26];
            for (int k = i; k < j; k++) freq[s.charAt(k) - 'a']++;
```

```
            int maxFreq = Arrays.stream(freq).max().getAsInt();
            int minFreq = Integer.MAX_VALUE;
            for (int f : freq) if (f > 0) minFreq = Math.min(minFreq, f);
            result += maxFreq - minFreq;
        }
    }
    return result;
}
```

Optimized:

Use sliding window and a frequency count array to calculate the beauty efficiently.

C++:

```
int beautySum(string s) {
    int result = 0;
    for (int i = 0; i < s.size(); i++) {
        vector<int> freq(26, 0);
        for (int j = i; j < s.size(); j++) {
            freq[s[j] - 'a']++;
            int maxFreq = *max_element(freq.begin(), freq.end());
            int minFreq = INT_MAX;
            for (int f : freq) if (f > 0) minFreq = min(minFreq, f);
            result += maxFreq - minFreq;
        }
    }
    return result;
}
```

Java:

```
public int beautySum(String s) {
    int result = 0;
    for (int i = 0; i < s.length(); i++) {
```

```
        int[] freq = new int[26];
        for (int j = i; j < s.length(); j++) {
            freq[s.charAt(j) - 'a']++;
            int maxFreq = Arrays.stream(freq).max().getAsInt();
            int minFreq = Integer.MAX_VALUE;
            for (int f : freq) if (f > 0) minFreq = Math.min(minFreq, f);
            result += maxFreq - minFreq;
        }
    }
    return result;
}
```

7. Longest Substring Without Repeating Characters

Brute Force:

Generate all substrings and check if they have unique characters.

C++:

```
int lengthOfLongestSubstring(string s) {
    int maxLength = 0;
    for (int i = 0; i < s.size(); i++) {
        unordered_set<char> seen;
        for (int j = i; j < s.size(); j++) {
            if (seen.count(s[j])) break;
            seen.insert(s[j]);
            maxLength = max(maxLength, j - i + 1);
        }
    }
    return maxLength;
}
```

Java:

```java
public int lengthOfLongestSubstring(String s) {
    int maxLength = 0;
    for (int i = 0; i < s.length(); i++) {
        Set<Character> seen = new HashSet<>();
        for (int j = i; j < s.length(); j++) {
            if (seen.contains(s.charAt(j))) break;
            seen.add(s.charAt(j));
            maxLength = Math.max(maxLength, j - i + 1);
        }
    }
    return maxLength;
}
```

Optimized (continued):

C++:

```cpp
int lengthOfLongestSubstring(string s) {
    unordered_map<char, int> charMap;
    int maxLength = 0, left = 0;
    for (int right = 0; right < s.size(); right++) {
        if (charMap.count(s[right])) left = max(charMap[s[right]] + 1, left);
        charMap[s[right]] = right;
        maxLength = max(maxLength, right - left + 1);
    }
    return maxLength;
}
```

Java:

```java
public int lengthOfLongestSubstring(String s) {
    Map<Character, Integer> charMap = new HashMap<>();
    int maxLength = 0, left = 0;
    for (int right = 0; right < s.length(); right++) {
        if (charMap.containsKey(s.charAt(right))) {
            left = Math.max(charMap.get(s.charAt(right)) + 1, left);
        }
        charMap.put(s.charAt(right), right);
        maxLength = Math.max(maxLength, right - left + 1);
    }
    return maxLength;
}
```

8. Longest Palindromic Substring

Brute Force:

Generate all substrings and check if they are palindromes.

C++:

```cpp
string longestPalindrome(string s) {
    int maxLength = 1, start = 0;
    for (int i = 0; i < s.size(); i++) {
        for (int j = i; j < s.size(); j++) {
            bool isPalindrome = true;
            for (int k = 0; k < (j - i + 1) / 2; k++) {
                if (s[i + k] != s[j - k]) isPalindrome = false;
            }
            if (isPalindrome && (j - i + 1) > maxLength) {
                start = i;
                maxLength = j - i + 1;
```

```
            }
        }
    }
    return s.substr(start, maxLength);
}
```

Java:

```
public String longestPalindrome(String s) {
    int maxLength = 1, start = 0;
    for (int i = 0; i < s.length(); i++) {
        for (int j = i; j < s.length(); j++) {
            boolean isPalindrome = true;
            for (int k = 0; k < (j - i + 1) / 2; k++) {
                if (s.charAt(i + k) != s.charAt(j - k)) {
                    isPalindrome = false;
                }
            }
            if (isPalindrome && (j - i + 1) > maxLength) {
                start = i;
                maxLength = j - i + 1;
            }
        }
    }
    return s.substring(start, start + maxLength);
}
```

Optimized:

Use expand-around-center technique to find the longest palindrome.

C++:

```
string longestPalindrome(string s) {
    int maxLength = 0, start = 0;
    for (int i = 0; i < s.size(); i++) {
        int len1 = expandAroundCenter(s, i, i);
        int len2 = expandAroundCenter(s, i, i + 1);
        int len = max(len1, len2);
        if (len > maxLength) {
            start = i - (len - 1) / 2;
            maxLength = len;
        }
    }
    return s.substr(start, maxLength);
}

int expandAroundCenter(string& s, int left, int right) {
    while (left >= 0 && right < s.size() && s[left] == s[right]) {
        left--;
        right++;
    }
    return right - left - 1;
}
```

Java:

```
public String longestPalindrome(String s) {
    int maxLength = 0, start = 0;
    for (int i = 0; i < s.length(); i++) {
        int len1 = expandAroundCenter(s, i, i);
        int len2 = expandAroundCenter(s, i, i + 1);
        int len = Math.max(len1, len2);
```

```
        if (len > maxLength) {
            start = i - (len - 1) / 2;
            maxLength = len;
        }
    }
    return s.substring(start, start + maxLength);
}

private int expandAroundCenter(String s, int left, int right) {
    while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
        left--;
        right++;
    }
    return right - left - 1;
}
```

String to Integer (atoi)

Brute Force:

Parse the string character by character, handling signs and edge cases manually.

C++:

```
int myAtoi(string s) {
    int index = 0, sign = 1, result = 0;
    while (index < s.size() && s[index] == ' ') index++; // Skip whitespaces
    if (index < s.size() && (s[index] == '-' || s[index] == '+')) {
        sign = s[index] == '-' ? -1 : 1;
        index++;
    }
    while (index < s.size() && isdigit(s[index])) {
```

```
        int digit = s[index] - '0';
        if (result > (INT_MAX - digit) / 10) return sign == 1 ? INT_MAX : INT_MIN; // Handle overflow
        result = result * 10 + digit;
        index++;
    }
    return result * sign;
}
```

Java:

```java
public int myAtoi(String s) {
    int index = 0, sign = 1, result = 0;
    while (index < s.length() && s.charAt(index) == ' ') index++; // Skip whitespaces
    if (index < s.length() && (s.charAt(index) == '-' || s.charAt(index) == '+')) {
        sign = s.charAt(index) == '-' ? -1 : 1;
        index++;
    }
    while (index < s.length() && Character.isDigit(s.charAt(index))) {
        int digit = s.charAt(index) - '0';
        if (result > (Integer.MAX_VALUE - digit) / 10) return sign == 1 ? Integer.MAX_VALUE : Integer.MIN_VALUE; // Handle overflow
        result = result * 10 + digit;
        index++;
    }
    return result * sign;
}
```

Optimized:

The brute force method is already optimized as it processes the string in a single pass.

10. Minimum Window Substring

Brute Force:

Generate all substrings and check if they contain all characters of the target string.

C++:

```
string minWindow(string s, string t) {
    unordered_map<char, int> targetFreq;
    for (char c : t) targetFreq[c]++;
    int minLen = INT_MAX, start = 0;
    string result = "";

    for (int i = 0; i < s.size(); i++) {
        unordered_map<char, int> windowFreq;
        for (int j = i; j < s.size(); j++) {
            windowFreq[s[j]]++;
            bool valid = true;
            for (auto& [charT, freqT] : targetFreq) {
                if (windowFreq[charT] < freqT) {
                    valid = false;
                    break;
                }
            }
            if (valid && (j - i + 1) < minLen) {
                minLen = j - i + 1;
                result = s.substr(i, minLen);
            }
        }
    }
    return result;
}
```

Java:

```java
public String minWindow(String s, String t) {
    Map<Character, Integer> targetFreq = new HashMap<>();
    for (char c : t.toCharArray()) targetFreq.put(c, targetFreq.getOrDefault(c, 0) + 1);

    int minLen = Integer.MAX_VALUE, start = 0;
    String result = "";

    for (int i = 0; i < s.length(); i++) {
        Map<Character, Integer> windowFreq = new HashMap<>();
        for (int j = i; j < s.length(); j++) {
            windowFreq.put(s.charAt(j), windowFreq.getOrDefault(s.charAt(j), 0) + 1);
            boolean valid = true;
            for (Map.Entry<Character, Integer> entry : targetFreq.entrySet()) {
                if (windowFreq.getOrDefault(entry.getKey(), 0) < entry.getValue()) {
                    valid = false;
                    break;
                }
            }
            if (valid && (j - i + 1) < minLen) {
                minLen = j - i + 1;
                result = s.substring(i, i + minLen);
            }
        }
    }
    return result;
}
```

Optimized (continued):

Use the sliding window technique to shrink the window once all characters from t are found in the current window of s.

C++:

```
string minWindow(string s, string t) {
    unordered_map<char, int> targetFreq, windowFreq;
    for (char c : t) targetFreq[c]++;

    int left = 0, matched = 0, minLen = INT_MAX, start = 0;
    for (int right = 0; right < s.size(); right++) {
        char c = s[right];
        windowFreq[c]++;

        if (targetFreq.count(c) && windowFreq[c] == targetFreq[c]) matched++;

        while (matched == targetFreq.size()) {
            if (right - left + 1 < minLen) {
                minLen = right - left + 1;
                start = left;
            }

            char leftChar = s[left];
            windowFreq[leftChar]--;
            if (targetFreq.count(leftChar) && windowFreq[leftChar] < targetFreq[leftChar])
matched--;
            left++;
        }
    }

    return minLen == INT_MAX ? "" : s.substr(start, minLen);
}
```

Java:

```java
public String minWindow(String s, String t) {
    Map<Character, Integer> targetFreq = new HashMap<>();
    for (char c : t.toCharArray()) targetFreq.put(c, targetFreq.getOrDefault(c, 0) + 1);

    Map<Character, Integer> windowFreq = new HashMap<>();
    int left = 0, matched = 0, minLen = Integer.MAX_VALUE, start = 0;

    for (int right = 0; right < s.length(); right++) {
        char c = s.charAt(right);
        windowFreq.put(c, windowFreq.getOrDefault(c, 0) + 1);

        if (targetFreq.containsKey(c) && windowFreq.get(c).equals(targetFreq.get(c)))
matched++;

        while (matched == targetFreq.size()) {
            if (right - left + 1 < minLen) {
                minLen = right - left + 1;
                start = left;
            }

            char leftChar = s.charAt(left);
            windowFreq.put(leftChar, windowFreq.get(leftChar) - 1);
            if (targetFreq.containsKey(leftChar) && windowFreq.get(leftChar) <
targetFreq.get(leftChar)) {
                matched--;
            }
            left++;
        }
    }
```

```
    return minLen == Integer.MAX_VALUE ? "" : s.substring(start, start + minLen);
}
```

LINKED LIST

1. Palindrome Linked List

Brute Force:

Convert the linked list into an array and check if it's a palindrome.

C++:

```
bool isPalindrome(ListNode* head) {
    vector<int> vals;
    while (head) {
        vals.push_back(head->val);
```

```cpp
        head = head->next;
    }
    int left = 0, right = vals.size() - 1;
    while (left < right) {
        if (vals[left] != vals[right]) return false;
        left++;
        right--;
    }
    return true;
}
```

Java:

```java
public boolean isPalindrome(ListNode head) {
    List<Integer> vals = new ArrayList<>();
    while (head != null) {
        vals.add(head.val);
        head = head.next;
    }
    int left = 0, right = vals.size() - 1;
    while (left < right) {
        if (!vals.get(left).equals(vals.get(right))) return false;
        left++;
        right--;
    }
    return true;
}
```

Optimized:

Use two pointers to reverse the second half of the linked list and then compare the two halves.

C++:

```
bool isPalindrome(ListNode* head) {
    if (!head || !head->next) return true;

    ListNode *slow = head, *fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }

    ListNode* secondHalf = reverseList(slow);
    ListNode* firstHalf = head;

    while (secondHalf) {
        if (firstHalf->val != secondHalf->val) return false;
        firstHalf = firstHalf->next;
        secondHalf = secondHalf->next;
    }

    return true;
}

ListNode* reverseList(ListNode* head) {
    ListNode* prev = nullptr;
    while (head) {
        ListNode* nextNode = head->next;
        head->next = prev;
        prev = head;
        head = nextNode;
    }
```

```
    return prev;
}
```

Java:

```
public boolean isPalindrome(ListNode head) {
    if (head == null || head.next == null) return true;

    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    ListNode secondHalf = reverseList(slow);
    ListNode firstHalf = head;

    while (secondHalf != null) {
        if (firstHalf.val != secondHalf.val) return false;
        firstHalf = firstHalf.next;
        secondHalf = secondHalf.next;
    }

    return true;
}

private ListNode reverseList(ListNode head) {
    ListNode prev = null;
    while (head != null) {
        ListNode nextNode = head.next;
        head.next = prev;
```

```
        prev = head;
        head = nextNode;
    }
    return prev;
}
```

2. Intersection of Two Linked Lists

Brute Force:

For each node in the first list, traverse the second list and check for intersection.

C++:

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    while (headA) {
        ListNode* temp = headB;
        while (temp) {
            if (headA == temp) return headA;
            temp = temp->next;
        }
        headA = headA->next;
    }
    return nullptr;
}
```

Java:

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    while (headA != null) {
        ListNode temp = headB;
        while (temp != null) {
            if (headA == temp) return headA;
            temp = temp.next;
        }
```

```
        headA = headA.next;
    }
    return null;
}
```

Optimized:

Use two pointers. Move both pointers through both lists and they will meet at the intersection point.

C++:

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    if (!headA || !headB) return nullptr;
    ListNode *a = headA, *b = headB;

    while (a != b) {
        a = a ? a->next : headB;
        b = b ? b->next : headA;
    }

    return a;
}
```

Java:

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    if (headA == null || headB == null) return null;
    ListNode a = headA, b = headB;

    while (a != b) {
        a = (a != null) ? a.next : headB;
        b = (b != null) ? b.next : headA;
    }
```

```
    return a;
}
```

3. LRU Cache

Brute Force:

Not applicable for brute force, as LRU cache is inherently an optimized structure

Optimized:

C++:

cpp

Copy code

```cpp
class LRUCache {
public:
    LRUCache(int capacity) : capacity(capacity) {}

    int get(int key) {
        if (cache.find(key) == cache.end()) return -1;
        moveToHead(key);
        return cache[key].first;
    }

    void put(int key, int value) {
        if (cache.find(key) != cache.end()) {
            cache[key].first = value;
            moveToHead(key);
        } else {
            if (cache.size() == capacity) {
                int lruKey = keys.back();
                cache.erase(lruKey);
                keys.pop_back();
```

```
            }
            keys.push_front(key);
            cache[key] = {value, keys.begin()};
        }
    }

private:
    int capacity;
    list<int> keys;
    unordered_map<int, pair<int, list<int>::iterator>> cache;

    void moveToHead(int key) {
        keys.erase(cache[key].second);
        keys.push_front(key);
        cache[key].second = keys.begin();
    }
};
```

Java:

java

Copy code

```
class LRUCache {
    private int capacity;
    private Map<Integer, Integer> cache;
    private LinkedHashSet<Integer> order;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.cache = new HashMap<>();
        this.order = new LinkedHashSet<>();
```

```java
    }

    public int get(int key) {
        if (!cache.containsKey(key)) return -1;
        order.remove(key);
        order.add(key);
        return cache.get(key);
    }

    public void put(int key, int value) {
        if (cache.containsKey(key)) {
            order.remove(key);
        } else if (cache.size() == capacity) {
            int oldest = order.iterator().next();
            cache.remove(oldest);
            order.remove(oldest);
        }
        cache.put(key, value);
        order.add(key);
    }
}
```

4. Add Two Numbers

C++:

```cpp
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode();
        ListNode* res = dummy;
```

```
        int total = 0, carry = 0;

        while (l1 || l2 || carry) {
            total = carry;

            if (l1) {
                total += l1->val;
                l1 = l1->next;
            }
            if (l2) {
                total += l2->val;
                l2 = l2->next;
            }

            int num = total % 10;
            carry = total / 10;
            dummy->next = new ListNode(num);
            dummy = dummy->next;
        }

        return res->next;
    }
};
```

Java:

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode();
        ListNode res = dummy;
        int total = 0, carry = 0;
```

```
        while (l1 != null || l2 != null || carry != 0) {
            total = carry;

            if (l1 != null) {
                total += l1.val;
                l1 = l1.next;
            }
            if (l2 != null) {
                total += l2.val;
                l2 = l2.next;
            }

            int num = total % 10;
            carry = total / 10;
            dummy.next = new ListNode(num);
            dummy = dummy.next;
        }

        return res.next;
    }
}
```

5. Rotate List

Brute Force:

We rotate the list one step at a time for k times, moving the last node to the front each time.

C++:

```cpp
ListNode* rotateRight(ListNode* head, int k) {
    if (!head || !head->next || k == 0) return head;

    int len = 0;
    ListNode* temp = head;

    while (temp) {
        len++;
        temp = temp->next;
    }

    k = k % len;
    while (k--) {
        ListNode* prev = nullptr;
        ListNode* curr = head;
        while (curr->next) {
            prev = curr;
            curr = curr->next;
        }
        prev->next = nullptr;
        curr->next = head;
        head = curr;
    }

    return head;
}
```

Java:

```java
public ListNode rotateRight(ListNode head, int k) {
    if (head == null || head.next == null || k == 0) return head;
```

```
    int len = 0;
    ListNode temp = head;

    while (temp != null) {
        len++;
        temp = temp.next;
    }

    k = k % len;
    while (k-- > 0) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr.next != null) {
            prev = curr;
            curr = curr.next;
        }
        prev.next = null;
        curr.next = head;
        head = curr;
    }

    return head;
}
```

Optimized:

First, compute the length of the list. Then, rotate the list by adjusting the pointers directly to form the new list in one pass.

C++:

```
ListNode* rotateRight(ListNode* head, int k) {
```

```
    if (!head || !head->next || k == 0) return head;

    ListNode* temp = head;
    int len = 1;

    while (temp->next) {
        temp = temp->next;
        len++;
    }

    temp->next = head;
    k = len - k % len;

    while (k--) temp = temp->next;

    head = temp->next;
    temp->next = nullptr;

    return head;
}
```

Java:

```
public ListNode rotateRight(ListNode head, int k) {
    if (head == null || head.next == null || k == 0) return head;

    ListNode temp = head;
    int len = 1;

    while (temp.next != null) {
        temp = temp.next;
```

```
        len++;
    }

    temp.next = head;
    k = len - k % len;

    while (k-- > 0) temp = temp.next;

    head = temp.next;
    temp.next = null;

    return head;
}
```

6. Reorder List

Brute Force:

Extract all nodes into a list, reorder them using two pointers, and reconstruct the list.

C++:

```
void reorderList(ListNode* head) {
    if (!head || !head->next) return;

    vector<ListNode*> nodes;
    ListNode* temp = head;

    while (temp) {
        nodes.push_back(temp);
        temp = temp->next;
    }
```

```cpp
    int i = 0, j = nodes.size() - 1;
    while (i < j) {
        nodes[i]->next = nodes[j];
        i++;
        if (i == j) break;
        nodes[j]->next = nodes[i];
        j--;
    }

    nodes[i]->next = nullptr;
}
```

Java:

```java
public void reorderList(ListNode head) {
    if (head == null || head.next == null) return;

    List<ListNode> nodes = new ArrayList<>();
    ListNode temp = head;

    while (temp != null) {
        nodes.add(temp);
        temp = temp.next;
    }

    int i = 0, j = nodes.size() - 1;
    while (i < j) {
        nodes.get(i).next = nodes.get(j);
        i++;
        if (i == j) break;
        nodes.get(j).next = nodes.get(i);
```

```
        j--;
    }

    nodes.get(i).next = null;
}
```

Optimized:

Find the middle of the list, reverse the second half, and merge the two halves together.

C++:

```
void reorderList(ListNode* head) {
    if (!head || !head->next) return;

    ListNode* slow = head, *fast = head, *prev = nullptr;

    while (fast && fast->next) {
        prev = slow;
        slow = slow->next;
        fast = fast->next->next;
    }

    prev->next = nullptr;

    ListNode* l2 = reverseList(slow);
    mergeLists(head, l2);
}

ListNode* reverseList(ListNode* head) {
    ListNode* prev = nullptr;
    while (head) {
```

```
        ListNode* nextNode = head->next;
        head->next = prev;
        prev = head;
        head = nextNode;
    }
    return prev;
}

void mergeLists(ListNode* l1, ListNode* l2) {
    while (l1 && l2) {
        ListNode* l1Next = l1->next;
        ListNode* l2Next = l2->next;
        l1->next = l2;
        if (l1Next) l2->next = l1Next;
        l1 = l1Next;
        l2 = l2Next;
    }
}
```

Java:

```
public void reorderList(ListNode head) {
    if (head == null || head.next == null) return;

    ListNode slow = head, fast = head, prev = null;

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

```java
    prev.next = null;

    ListNode l2 = reverseList(slow);
    mergeLists(head, l2);
}

private ListNode reverseList(ListNode head) {
    ListNode prev = null;
    while (head != null) {
        ListNode nextNode = head.next;
        head.next = prev;
        prev = head;
        head = nextNode;
    }
    return prev;
}

private void mergeLists(ListNode l1, ListNode l2) {
    while (l1 != null && l2 != null) {
        ListNode l1Next = l1.next;
        ListNode l2Next = l2.next;
        l1.next = l2;
        if (l1Next != null) l2.next = l1Next;
        l1 = l1Next;
        l2 = l2Next;
    }
}
```

7. Linked List Cycle II

Brute Force:

Use a hash set to track visited nodes. If a node is revisited, it's the cycle's start.

C++:

```
ListNode *detectCycle(ListNode *head) {
    unordered_set<ListNode*> visited;
    while (head) {
        if (visited.count(head)) return head;
        visited.insert(head);
        head = head->next;
    }
    return nullptr;
}
```

Java:

```
public ListNode detectCycle(ListNode head) {
    Set<ListNode> visited = new HashSet<>();
    while (head != null) {
        if (visited.contains(head)) return head;
        visited.add(head);
        head = head.next;
    }
    return null;
}
```

Optimized:

Use two pointers (slow and fast) to detect a cycle. If they meet, reset one pointer to the head and move both pointers one step at a time until they meet again.

C++:

```cpp
ListNode *detectCycle(ListNode *head) {
    if (!head || !head->next) return nullptr;


    ListNode *slow = head, *fast = head;


    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {
            slow = head;
            while (slow != fast) {
                slow = slow->next;
                fast = fast->next;
            }
            return slow;
        }
    }
    return nullptr;
}
```

Java:

```java
public ListNode detectCycle(ListNode head) {
    if (head == null || head.next == null) return null;

    ListNode slow = head, fast = head;

    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) {
            slow = head;
            while (slow != fast) {
                slow = slow.next;
                fast = fast.next;
            }
            return slow;
        }
    }
    return null;
}
```

8. Rearrange a Linked List in Zig-Zag Fashion

Brute Force & Optimized:

Traverse the linked list and for each pair of consecutive nodes, swap them if they are not in zig-zag order (i.e., the first node should be less than the second, then greater than the next, and so on).

C++:

```cpp
void zigZagList(ListNode* head) {
    bool flag = true;  // true means "<" relation expected, false means ">" relation
expected
    ListNode* current = head;
```

```
    while (current && current->next) {
        if (flag) {
            if (current->val > current->next->val)
                swap(current->val, current->next->val);
        } else {
            if (current->val < current->next->val)
                swap(current->val, current->next->val);
        }
        flag = !flag;
        current = current->next;
    }
}
```

Java:

```
public void zigZagList(ListNode head) {
    boolean flag = true;  // true means "<" relation expected, false means ">" relation
expected
    ListNode current = head;

    while (current != null && current.next != null) {
        if (flag) {
            if (current.val > current.next.val) {
                int temp = current.val;
                current.val = current.next.val;
                current.next.val = temp;
            }
        } else {
            if (current.val < current.next.val) {
                int temp = current.val;
```

```
                current.val = current.next.val;
                current.next.val = temp;
            }
        }
        flag = !flag;
        current = current.next;
    }
}
```

9. Flattening a Linked List

Brute Force & Optimized :

C++:

```
// C++ program for flattening a Linked List

#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node *next, *bottom;

    Node(int new_data) {
        data = new_data;
        next = bottom = NULL;
    }
};

// function to flatten the linked list
```

```
Node* flatten(Node* root) {

    vector<int> values;

    // Push values of all nodes into an array
    while (root != NULL) {

        // Push the head node of the sub-linked-list
        values.push_back(root->data);

        // Push all the nodes of the sub-linked-list
        Node* temp = root->bottom;
        while (temp != NULL) {
            values.push_back(temp->data);
            temp = temp->bottom;
        }

        // Move to the next head node
        root = root->next;
    }

    // Sort the node values in ascending order
    sort(values.begin(), values.end());

    // Construct the new flattened linked list
    Node* tail = NULL;
    Node* head = NULL;
    for (int i = 0; i < values.size(); i++) {
        Node* newNode = new Node(values[i]);
```

```
        // If this is the first node of the linked list,
        // make the node as head
        if (head == NULL) {
            head = newNode;
        }
        else {
            tail->bottom = newNode;
        }
        tail = newNode;
    }
    return head;
}

void printList(Node* head) {
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->bottom;
    }
    cout << endl;
}

int main() {

    /* Create a hard-coded linked list:
        5 -> 10 -> 19 -> 28
        |    |     |
        V    V     V
```

```
        7    20    22
        |          |
        V          V
        8          50
        |
        V
        30
    */
    Node* head = new Node(5);
    head->bottom = new Node(7);
    head->bottom->bottom = new Node(8);
    head->bottom->bottom->bottom = new Node(30);

    head->next = new Node(10);
    head->next->bottom = new Node(20);

    head->next->next = new Node(19);
    head->next->next->bottom = new Node(22);
    head->next->next->bottom->bottom = new Node(50);

    head->next->next->next = new Node(28);

    // Function call
    head = flatten(head);

    printList(head);
    return 0;
}
```

Java:

```java
// Java Program for flattening a linked list

import java.util.*;

class Node {
    int data;
    Node next, bottom;

    Node(int newData) {
        data = newData;
        next = bottom = null;
    }
}

public class GFG {

    // Function to flatten the linked list
    static Node flatten(Node root) {
        List<Integer> values = new ArrayList<>();

        // Push values of all nodes into a list
        while (root != null) {
            // Push the head node of the sub-linked-list
            values.add(root.data);

            // Push all the nodes of the sub-linked-list
            Node temp = root.bottom;
            while (temp != null) {
                values.add(temp.data);
```

```
            temp = temp.bottom;
        }

        // Move to the next head node
        root = root.next;
    }

    // Sort the node values in ascending order
    Collections.sort(values);

    // Construct the new flattened linked list
    Node tail = null;
    Node head = null;
    for (int value : values) {
        Node newNode = new Node(value);

        // If this is the first node of the linked list,
        // make the node as head
        if (head == null)
            head = newNode;
        else
            tail.bottom = newNode;
        tail = newNode;
    }

    return head;
}

// Function to print the linked list
```

```
static void printList(Node head) {
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data + " ");
        temp = temp.bottom;
    }
    System.out.println();
}

public static void main(String[] args) {

    // Create a hard-coded linked list:
    // 5 -> 10 -> 19 -> 28
    // |    |     |
    // V    V     V
    // 7    20    22
    // |          |
    // V          V
    // 8          50
    // |
    // V
    // 30

    Node head = new Node(5);
    head.bottom = new Node(7);
    head.bottom.bottom = new Node(8);
    head.bottom.bottom.bottom = new Node(30);

    head.next = new Node(10);
```

```java
        head.next.bottom = new Node(20);

        head.next.next = new Node(19);
        head.next.next.bottom = new Node(22);
        head.next.next.bottom.bottom = new Node(50);

        head.next.next.next = new Node(28);

        // Function call
        head = flatten(head);

        printList(head);
    }
}
```

10. Reverse Nodes in k-Group

Brute Force & Optimized:

First, count the nodes, then reverse every k nodes by traversing the list multiple times.

C++:

```cpp
ListNode* reverseKGroup(ListNode* head, int k) {
    int count = 0;
    ListNode* current = head;

    while (current) {
```

```
        count++;
        current = current->next;
    }

    if (count < k) return head;

    ListNode* prev = nullptr;
    ListNode* next = nullptr;
    current = head;
    int i = 0;

    while (current && i < k) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
        i++;
    }

    if (next) head->next = reverseKGroup(next, k);

    return prev;
}
```

Java:

```
public ListNode reverseKGroup(ListNode head, int k) {
    int count = 0;
    ListNode current = head;

    while (current != null) {
```

```
        count++;
        current = current.next;
    }

    if (count < k) return head;

    ListNode prev = null;
    ListNode next = null;
    current = head;
    int i = 0;

    while (current != null && i < k) {
        next = current.next;
        current.next = prev;
        prev = current;
        current = next;
        i++;
    }

    if (next != null) head.next = reverseKGroup(next, k);

    return prev;
}
```

STACK AND QUEUE

1. Implement Stack using Queues

C++:

```
class MyStack {
    queue<int> q1, q2;

public:
    MyStack() {}

    void push(int x) {
        q1.push(x);
    }

    int pop() {
        while (q1.size() > 1) {
            q2.push(q1.front());
            q1.pop();
        }
        int topElem = q1.front();
        q1.pop();
        swap(q1, q2);
        return topElem;
    }

    int top() {
        while (q1.size() > 1) {
            q2.push(q1.front());
            q1.pop();
        }
        int topElem = q1.front();
```

```
        q2.push(topElem);  // keep the last element in the stack
        q1.pop();
        swap(q1, q2);
        return topElem;
    }

    bool empty() {
        return q1.empty();
    }
};
```

Java:

```
class MyStack {
    Queue<Integer> q1 = new LinkedList<>();
    Queue<Integer> q2 = new LinkedList<>();

    public MyStack() {}

    public void push(int x) {
        q1.add(x);
    }

    public int pop() {
        while (q1.size() > 1) {
            q2.add(q1.poll());
        }
        int topElem = q1.poll();
        Queue<Integer> temp = q1;
        q1 = q2;
        q2 = temp;
```

```
        return topElem;
    }

    public int top() {
        while (q1.size() > 1) {
            q2.add(q1.poll());
        }
        int topElem = q1.poll();
        q2.add(topElem);  // keep the last element in the stack
        Queue<Integer> temp = q1;
        q1 = q2;
        q2 = temp;
        return topElem;
    }

    public boolean empty() {
        return q1.isEmpty();
    }
}
```

2. **Next Greater Element I**

Brute Force:

For each element in nums1, scan through nums2 to find the next greater element.

C++:

```
vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
    vector<int> res;
    for (int i = 0; i < nums1.size(); i++) {
        bool found = false;
        int nextGreater = -1;
```

```
        for (int j = 0; j < nums2.size(); j++) {
            if (nums2[j] == nums1[i]) {
                found = true;
            }
            if (found && nums2[j] > nums1[i]) {
                nextGreater = nums2[j];
                break;
            }
        }
        res.push_back(nextGreater);
    }
    return res;
}
```

Java:

```
public int[] nextGreaterElement(int[] nums1, int[] nums2) {
    int[] res = new int[nums1.length];
    for (int i = 0; i < nums1.length; i++) {
        int nextGreater = -1;
        boolean found = false;
        for (int j = 0; j < nums2.length; j++) {
            if (nums2[j] == nums1[i]) found = true;
            if (found && nums2[j] > nums1[i]) {
                nextGreater = nums2[j];
                break;
            }
        }
        res[i] = nextGreater;
    }
```

```
    return res;
}
```

Optimized:

Use a stack to track the next greater element in nums2 and store the result in a map for nums1.

C++:

```
vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
    unordered_map<int, int> map;
    stack<int> s;
    for (int n : nums2) {
        while (!s.empty() && s.top() < n) {
            map[s.top()] = n;
            s.pop();
        }
        s.push(n);
    }
    vector<int> res;
    for (int n : nums1) {
        res.push_back(map.count(n) ? map[n] : -1);
    }
    return res;
}
```

Java:

```
public int[] nextGreaterElement(int[] nums1, int[] nums2) {
    Map<Integer, Integer> map = new HashMap<>();
    Stack<Integer> stack = new Stack<>();
    for (int n : nums2) {
        while (!stack.isEmpty() && stack.peek() < n) {
            map.put(stack.pop(), n);
```

```
        }
        stack.push(n);
    }
    int[] res = new int[nums1.length];
    for (int i = 0; i < nums1.length; i++) {
        res[i] = map.getOrDefault(nums1[i], -1);
    }
    return res;
}
```

3. Valid Parentheses

Brute Force & Optimized :

For each closing bracket, scan backward to find its corresponding opening bracket, validating each one.

C++:

```
bool isValid(string s) {
    stack<char> stack;
    for (char c : s) {
        if (c == '(' || c == '{' || c == '[') {
            stack.push(c);
        } else {
            if (stack.empty()) return false;
            char top = stack.top();
            if ((c == ')' && top == '(') ||
                (c == '}' && top == '{') ||
                (c == ']' && top == '[')) {
```

```
                stack.pop();
            } else {
                return false;
            }
        }
    }
    return stack.empty();
}
```

Java:

```
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()) return false;
            char top = stack.pop();
            if ((c == ')' && top != '(') ||
                (c == '}' && top != '{') ||
                (c == ']' && top != '[')) {
                return false;
            }
        }
    }
    return stack.isEmpty();
}
```

4. Next Greater Element II

Brute Force:

Loop through the array twice to handle the circular nature and find the next greater element for each number.

C++:

```
vector<int> nextGreaterElements(vector<int>& nums) {
    int n = nums.size();
    vector<int> res(n, -1);
    for (int i = 0; i < n; ++i) {
        for (int j = 1; j < n; ++j) {
            if (nums[(i + j) % n] > nums[i]) {
                res[i] = nums[(i + j) % n];
                break;
            }
        }
    }
    return res;
}
```

Java:

```
public int[] nextGreaterElements(int[] nums) {
    int n = nums.length;
    int[] res = new int[n];
    Arrays.fill(res, -1);
    for (int i = 0; i < n; i++) {
        for (int j = 1; j < n; j++) {
            if (nums[(i + j) % n] > nums[i]) {
                res[i] = nums[(i + j) % n];
```

```
            break;
        }
    }
}
return res;
}
```

Optimized:

Use a **monotonic stack** to keep track of indices of the array elements, handling the circular nature by iterating twice through the array.

C++:

```cpp
vector<int> nextGreaterElements(vector<int>& nums) {
    int n = nums.size();
    vector<int> res(n, -1);
    stack<int> s;

    for (int i = 0; i < 2 * n; ++i) {
        while (!s.empty() && nums[s.top()] < nums[i % n]) {
            res[s.top()] = nums[i % n];
            s.pop();
        }
        if (i < n) {
            s.push(i);
        }
    }
    return res;
}
```

Java:

```java
public int[] nextGreaterElements(int[] nums) {
    int n = nums.length;
    int[] res = new int[n];
    Arrays.fill(res, -1);
    Stack<Integer> stack = new Stack<>();

    for (int i = 0; i < 2 * n; i++) {
        while (!stack.isEmpty() && nums[stack.peek()] < nums[i % n]) {
            res[stack.pop()] = nums[i % n];
        }
        if (i < n) {
            stack.push(i);
        }
    }
    return res;
}
```

5. Asteroid Collision

Brute Force:

Compare each asteroid with the next in the array to simulate the collisions, updating the list accordingly.

C++:

```cpp
vector<int> asteroidCollision(vector<int>& asteroids) {
    vector<int> res;
    for (int i = 0; i < asteroids.size(); i++) {
        if (res.empty() || asteroids[i] > 0) {
            res.push_back(asteroids[i]);
        } else {
```

```
            while (!res.empty() && res.back() > 0 && res.back() < abs(asteroids[i])) {
                res.pop_back();
            }
            if (!res.empty() && res.back() == abs(asteroids[i])) {
                res.pop_back();
            } else if (res.empty() || res.back() < 0) {
                res.push_back(asteroids[i]);
            }
        }
    }
    return res;
}
```

Java:

```
public int[] asteroidCollision(int[] asteroids) {
    List<Integer> res = new ArrayList<>();
    for (int i = 0; i < asteroids.length; i++) {
        if (res.isEmpty() || asteroids[i] > 0) {
            res.add(asteroids[i]);
        } else {
            while (!res.isEmpty() && res.get(res.size() - 1) > 0 && res.get(res.size() - 1) <
Math.abs(asteroids[i])) {
                res.remove(res.size() - 1);
            }
            if (!res.isEmpty() && res.get(res.size() - 1) == Math.abs(asteroids[i])) {
                res.remove(res.size() - 1);
            } else if (res.isEmpty() || res.get(res.size() - 1) < 0) {
                res.add(asteroids[i]);
            }
        }
```

```
    }
    return res.stream().mapToInt(i -> i).toArray();
}
```

Optimized:

Use a stack to handle the collisions. Push positive asteroids and resolve collisions with negative asteroids by comparing their magnitudes.

C++:

```
vector<int> asteroidCollision(vector<int>& asteroids) {
    stack<int> s;
    for (int a : asteroids) {
        bool alive = true;
        while (!s.empty() && a < 0 && s.top() > 0) {
            if (s.top() < -a) {
                s.pop();
            } else if (s.top() == -a) {
                s.pop();
                alive = false;
                break;
            } else {
                alive = false;
                break;
            }
        }
        if (alive) {
            s.push(a);
        }
```

```
    }
    vector<int> res(s.size());
    for (int i = s.size() - 1; i >= 0; --i) {
        res[i] = s.top();
        s.pop();
    }
    return res;
}
```

Java:

```
public int[] asteroidCollision(int[] asteroids) {
    Stack<Integer> stack = new Stack<>();
    for (int a : asteroids) {
        boolean alive = true;
        while (!stack.isEmpty() && a < 0 && stack.peek() > 0) {
            if (stack.peek() < -a) {
                stack.pop();
            } else if (stack.peek() == -a) {
                stack.pop();
                alive = false;
                break;
            } else {
                alive = false;
                break;
            }
        }
        if (alive) {
            stack.push(a);
        }
    }
```

```
    int[] res = new int[stack.size()];
    for (int i = stack.size() - 1; i >= 0; i--) {
        res[i] = stack.pop();
    }
    return res;
}
```

6. Remove K Digits

Brute Force:

Generate all possible combinations by removing k digits and compare to find the smallest number.

C++:

```
string removeKdigits(string num, int k) {
    if (k == num.size()) return "0";
    string result = num;

    while (k > 0) {
        int n = result.size();
        int i = 0;
        while (i + 1 < n && result[i] <= result[i + 1]) i++;
        result.erase(i, 1);
        k--;
    }

    int start = 0;
    while (start < result.size() && result[start] == '0') start++;

    result = result.substr(start);
    return result.empty() ? "0" : result;
```

```
}
```

Java:

```java
public String removeKdigits(String num, int k) {
    if (k == num.length()) return "0";
    StringBuilder result = new StringBuilder(num);

    while (k > 0) {
        int n = result.length();
        int i = 0;
        while (i + 1 < n && result.charAt(i) <= result.charAt(i + 1)) i++;
        result.deleteCharAt(i);
        k--;
    }

    int start = 0;
    while (start < result.length() && result.charAt(start) == '0') start++;

    String finalResult = result.substring(start);
    return finalResult.isEmpty() ? "0" : finalResult;
}
```

Optimized:

Use a stack to maintain the smallest possible number by removing k digits.

C++:

```cpp
string removeKdigits(string num, int k) {
    string result = "";
    for (char c : num) {
        while (!result.empty() && result.back() > c && k > 0) {
```

```
            result.pop_back();
            k--;
        }
        result.push_back(c);
    }

    while (k-- > 0) result.pop_back();

    int i = 0;
    while (i < result.size() && result[i] == '0') i++;

    result = result.substr(i);
    return result.empty() ? "0" : result;
}
```

Java:

```
public String removeKdigits(String num, int k) {
    Stack<Character> stack = new Stack<>();
    for (char c : num.toCharArray()) {
        while (!stack.isEmpty() && stack.peek() > c && k > 0) {
            stack.pop();
            k--;
        }
        stack.push(c);
    }

    while (k-- > 0) stack.pop();

    StringBuilder result = new StringBuilder();
    for (char c : stack) result.append(c);
```

```
    while (result.length() > 0 && result.charAt(0) == '0') result.deleteCharAt(0);

    return result.length() == 0 ? "0" : result.toString();
}
```

7. Sliding Window Maximum

Brute Force & Optimized :

For each window of size k, find the maximum value by scanning through the window.

C++:

```
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    vector<int> result;
    for (int i = 0; i <= nums.size() - k; ++i) {
        int maxVal = *max_element(nums.begin() + i, nums.begin() + i + k);
        result.push_back(maxVal);
    }
    return result;
}
```

Java:

java

Copy code

```
public int[] maxSlidingWindow(int[] nums, int k) {
    int n = nums.length;
    int[] result = new int[n - k + 1];
    for (int i = 0; i <= n - k; i++) {
        int maxVal = Integer.MIN_VALUE;
        for (int j = i; j < i + k; j++) {
            maxVal = Math.max(maxVal, nums[j]);
        }
```

```
        result[i] = maxVal;
    }
    return result;
}
```

Recursion And Backtracking

1. Fibonacci Number

C++:

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Java:

```
public int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

2. Pow(x, n)

C++:

```
double myPow(double x, int n) {
    if (n == 0) return 1;
    if (n < 0) return 1 / myPow(x, -n);
    return x * myPow(x, n - 1);
}
```

Java:

```java
public double myPow(double x, int n) {
    if (n == 0) return 1;
    if (n < 0) return 1 / myPow(x, -n);
    return x * myPow(x, n - 1);
}
```

3. Power of Three

C++:

```cpp
bool isPowerOfThree(int n) {
    if (n == 1) return true;
    if (n == 0 || n % 3 != 0) return false;
    return isPowerOfThree(n / 3);
}
```

Java:

```java
public boolean isPowerOfThree(int n) {
    if (n == 1) return true;
    if (n == 0 || n % 3 != 0) return false;
    return isPowerOfThree(n / 3);
}
```

4. Combination Sum

C++:

```cpp
void combinationSumUtil(vector<int>& candidates, int target, vector<int>& curr, int start, vector<vector<int>>& res) {
    if (target == 0) {
        res.push_back(curr);
        return;
    }
```

```
    if (target < 0) return;
    for (int i = start; i < candidates.size(); i++) {
        curr.push_back(candidates[i]);
        combinationSumUtil(candidates, target - candidates[i], curr, i, res);
        curr.pop_back();
    }
}

vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
    vector<vector<int>> res;
    vector<int> curr;
    combinationSumUtil(candidates, target, curr, 0, res);
    return res;
}
```

Java:

```
private void combinationSumUtil(int[] candidates, int target, List<Integer> curr, int
start, List<List<Integer>> res) {
    if (target == 0) {
        res.add(new ArrayList<>(curr));
        return;
    }
    if (target < 0) return;
    for (int i = start; i < candidates.length; i++) {
        curr.add(candidates[i]);
        combinationSumUtil(candidates, target - candidates[i], curr, i, res);
        curr.remove(curr.size() - 1);
    }
}
```

```java
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> res = new ArrayList<>();
    combinationSumUtil(candidates, target, new ArrayList<>(), 0, res);
    return res;
}
```

5. Subsets

C++:

```cpp
void subsetsUtil(vector<int>& nums, vector<int>& curr, int index,
vector<vector<int>>& res) {
    res.push_back(curr);
    for (int i = index; i < nums.size(); i++) {
        curr.push_back(nums[i]);
        subsetsUtil(nums, curr, i + 1, res);
        curr.pop_back();
    }
}

vector<vector<int>> subsets(vector<int>& nums) {
    vector<vector<int>> res;
    vector<int> curr;
    subsetsUtil(nums, curr, 0, res);
    return res;
}
```

Java:

```java
private void subsetsUtil(int[] nums, List<Integer> curr, int index, List<List<Integer>>
res) {
    res.add(new ArrayList<>(curr));
    for (int i = index; i < nums.length; i++) {
```

```java
        curr.add(nums[i]);
        subsetsUtil(nums, curr, i + 1, res);
        curr.remove(curr.size() - 1);
    }
}

public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> res = new ArrayList<>();
    subsetsUtil(nums, new ArrayList<>(), 0, res);
    return res;
}
```

6. Coin Change

C++:

```cpp
int coinChangeUtil(vector<int>& coins, int amount) {
    if (amount == 0) return 0;
    int res = INT_MAX;
    for (int coin : coins) {
        if (amount >= coin) {
            int sub_res = coinChangeUtil(coins, amount - coin);
            if (sub_res != INT_MAX) res = min(res, sub_res + 1);
        }
    }
    return res;
}

int coinChange(vector<int>& coins, int amount) {
    int result = coinChangeUtil(coins, amount);
    return result == INT_MAX ? -1 : result;
```

```
}
```

Java:

```java
private int coinChangeUtil(int[] coins, int amount) {
    if (amount == 0) return 0;
    int res = Integer.MAX_VALUE;
    for (int coin : coins) {
        if (amount >= coin) {
            int sub_res = coinChangeUtil(coins, amount - coin);
            if (sub_res != Integer.MAX_VALUE) res = Math.min(res, sub_res + 1);
        }
    }
    return res;
}

public int coinChange(int[] coins, int amount) {
    int result = coinChangeUtil(coins, amount);
    return result == Integer.MAX_VALUE ? -1 : result;
}
```

7. Elimination Game

C++:

```cpp
int lastRemaining(int n) {
    if (n == 1) return 1;
    return 2 * (n / 2 - lastRemaining(n / 2) + 1);
}
```

Java:

```
public int lastRemaining(int n) {
    if (n == 1) return 1;
    return 2 * (n / 2 - lastRemaining(n / 2) + 1);
}
```

8. Rat in a Maze Problem - I

C++:

```
bool isSafe(int x, int y, vector<vector<int>>& maze, vector<vector<int>>& visited, int n) {
    return (x >= 0 && x < n && y >= 0 && y < n && maze[x][y] == 1 && visited[x][y] == 0);
}

bool solveMazeUtil(vector<vector<int>>& maze, vector<vector<int>>& visited, int x, int y, int n) {
    if (x == n - 1 && y == n - 1) {
        visited[x][y] = 1;
        return true;
    }

    if (isSafe(x, y, maze, visited, n)) {
        visited[x][y] = 1;

        if (solveMazeUtil(maze, visited, x + 1, y, n)) return true;
        if (solveMazeUtil(maze, visited, x, y + 1, n)) return true;

        visited[x][y] = 0;  // Backtrack
    }
    return false;
}
```

```
vector<vector<int>> solveMaze(vector<vector<int>>& maze, int n) {
    vector<vector<int>> visited(n, vector<int>(n, 0));
    if (solveMazeUtil(maze, visited, 0, 0, n)) {
        return visited;
    }
    return {};  // No solution
}
```

Java:

```
private boolean isSafe(int x, int y, int[][] maze, int[][] visited, int n) {
    return (x >= 0 && x < n && y >= 0 && y < n && maze[x][y] == 1 && visited[x][y] ==
0);
}

private boolean solveMazeUtil(int[][] maze, int[][] visited, int x, int y, int n) {
    if (x == n - 1 && y == n - 1) {
        visited[x][y] = 1;
        return true;
    }

    if (isSafe(x, y, maze, visited, n)) {
        visited[x][y] = 1;

        if (solveMazeUtil(maze, visited, x + 1, y, n)) return true;
        if (solveMazeUtil(maze, visited, x, y + 1, n)) return true;

        visited[x][y] = 0;  // Backtrack
    }
    return false;
```

```
}

public int[][] solveMaze(int[][] maze, int n) {
    int[][] visited = new int[n][n];
    if (solveMazeUtil(maze, visited, 0, 0, n)) {
        return visited;
    }
    return new int[0][0];  // No solution
}
```

9. Word Break

C++:

```
bool wordBreakUtil(string s, unordered_set<string>& wordDict) {
    if (s.empty())return true;
    for (int i = 1; i <= s.length(); i++) {
        string prefix = s.substr(0, i);
        if (wordDict.find(prefix) != wordDict.end() && wordBreakUtil(s.substr(i),
wordDict)) {
            return true;
        }
    }
    return false;
}

bool wordBreak(string s, vector<string>& wordDict) {
    unordered_set<string> dict(wordDict.begin(), wordDict.end());
    return wordBreakUtil(s, dict);
}
```

Java:

```java
private boolean wordBreakUtil(String s, Set<String> wordDict) {
    if (s.isEmpty()) return true;
    for (int i = 1; i <= s.length(); i++) {
        String prefix = s.substring(0, i);
        if (wordDict.contains(prefix) && wordBreakUtil(s.substring(i), wordDict)) {
            return true;
        }
    }
    return false;
}

public boolean wordBreak(String s, List<String> wordDict) {
    Set<String> dict = new HashSet<>(wordDict);
    return wordBreakUtil(s, dict);
}
```

10. N-Queens

C++:

```cpp
bool isSafe(vector<string>& board, int row, int col, int n) {
    for (int i = 0; i < row; i++) {
        if (board[i][col] == 'Q') return false;
        if (col - (row - i) >= 0 && board[i][col - (row - i)] == 'Q') return false;
        if (col + (row - i) < n && board[i][col + (row - i)] == 'Q') return false;
    }
    return true;
}

void solveNQueensUtil(int n, int row, vector<string>& board,
vector<vector<string>>& res) {
```

```
    if (row == n) {
        res.push_back(board);
        return;
    }
    for (int col = 0; col < n; col++) {
        if (isSafe(board, row, col, n)) {
            board[row][col] = 'Q';
            solveNQueensUtil(n, row + 1, board, res);
            board[row][col] = '.';
        }
    }
}

vector<vector<string>> solveNQueens(int n) {
    vector<vector<string>> res;
    vector<string> board(n, string(n, '.'));
    solveNQueensUtil(n, 0, board, res);
    return res;
}
```

Java:

```
private boolean isSafe(List<String> board, int row, int col, int n) {
    for (int i = 0; i < row; i++) {
        if (board.get(i).charAt(col) == 'Q') return false;
        if (col - (row - i) >= 0 && board.get(i).charAt(col - (row - i)) == 'Q') return false;
        if (col + (row - i) < n && board.get(i).charAt(col + (row - i)) == 'Q') return false;
    }
    return true;
}
```

```java
private void solveNQueensUtil(int n, int row, List<String> board, List<List<String>> res) {
    if (row == n) {
        res.add(new ArrayList<>(board));
        return;
    }
    for (int col = 0; col < n; col++) {
        if (isSafe(board, row, col, n)) {
            StringBuilder sb = new StringBuilder(board.get(row));
            sb.setCharAt(col, 'Q');
            board.set(row, sb.toString());
            solveNQueensUtil(n, row + 1, board, res);
            sb.setCharAt(col, '.');
            board.set(row, sb.toString());
        }
    }
}

public List<List<String>> solveNQueens(int n) {
    List<List<String>> res = new ArrayList<>();
    List<String> board = new ArrayList<>(Collections.nCopies(n, ".".repeat(n)));
    solveNQueensUtil(n, 0, board, res);
    return res;
}
```

11. Sudoku Solver

C++:

```cpp
bool isValid(vector<vector<char>>& board, int row, int col, char c) {
```

```
    for (int i = 0; i < 9; i++) {
        if (board[i][col] == c || board[row][i] == c || board[3 * (row / 3) + i / 3][3 * (col / 3)
+ i % 3] == c) {
            return false;
        }
    }
    return true;
}

bool solveSudokuUtil(vector<vector<char>>& board) {
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (board[i][j] == '.') {
                for (char c = '1'; c <= '9'; c++) {
                    if (isValid(board, i, j, c)) {
                        board[i][j] = c;
                        if (solveSudokuUtil(board)) return true;
                        board[i][j] = '.';
                    }
                }
                return false;
            }
        }
    }
    return true;
}

void solveSudoku(vector<vector<char>>& board) {
    solveSudokuUtil(board);
```

}

Java:

```
private boolean isValid(char[][] board, int row, int col, char c) {
    for (int i = 0; i < 9; i++) {
        if (board[i][col] == c || board[row][i] == c || board[3 * (row / 3) + i / 3][3 * (col / 3)
+ i % 3] == c) {
            return false;
        }
    }
    return true;
}


private boolean solveSudokuUtil(char[][] board) {
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (board[i][j] == '.') {
                for (char c = '1'; c <= '9'; c++) {
                    if (isValid(board, i, j, c)) {
                        board[i][j] = c;
                        if (solveSudokuUtil(board)) return true;
                        board[i][j] = '.';
                    }
                }
                return false;
            }
        }
    }
    return true;
}
```

```java
public void solveSudoku(char[][] board) {
    solveSudokuUtil(board);
}
```

12. Tug-of-War

C++:

```cpp
void tugOfWarUtil(vector<int>& arr, vector<int>& currSet1, vector<int>& currSet2, int idx, int n, int& minDiff, vector<int>& resSet1, vector<int>& resSet2) {
    if (idx == n) {
        int sum1 = accumulate(currSet1.begin(), currSet1.end(), 0);
        int sum2 = accumulate(currSet2.begin(), currSet2.end(), 0);
        if (abs(sum1 - sum2) < minDiff) {
            minDiff = abs(sum1 - sum2);
            resSet1 = currSet1;
            resSet2 = currSet2;
        }
        return;
    }

    if (currSet1.size() < (n + 1) / 2) {
        currSet1.push_back(arr[idx]);
        tugOfWarUtil(arr, currSet1, currSet2, idx + 1, n, minDiff, resSet1, resSet2);
        currSet1.pop_back();
    }

    if (currSet2.size() < (n + 1) / 2) {
        currSet2.push_back(arr[idx]);
        tugOfWarUtil(arr, currSet1, currSet2, idx + 1, n, minDiff, resSet1, resSet2);
```

```cpp
        currSet2.pop_back();
    }
}

vector<vector<int>> tugOfWar(vector<int>& arr) {
    int n = arr.size();
    int minDiff = INT_MAX;
    vector<int> currSet1, currSet2, resSet1, resSet2;
    tugOfWarUtil(arr, currSet1, currSet2, 0, n, minDiff, resSet1, resSet2);
    return {resSet1, resSet2};
}
```

Java:

```java
private void tugOfWarUtil(int[] arr, List<Integer> currSet1, List<Integer> currSet2, int idx, int n, int[] minDiff, List<Integer> resSet1, List<Integer> resSet2) {
    if (idx == n) {
        int sum1 = currSet1.stream().mapToInt(Integer::intValue).sum();
        int sum2 = currSet2.stream().mapToInt(Integer::intValue).sum();
        if (Math.abs(sum1 - sum2) < minDiff[0]) {
            minDiff[0] = Math.abs(sum1 - sum2);
            resSet1.clear();
            resSet1.addAll(currSet1);
            resSet2.clear();
            resSet2.addAll(currSet2);
        }
        return;
    }

    if (currSet1.size() < (n + 1) / 2) {
        currSet1.add(arr[idx]);
```

```java
            tugOfWarUtil(arr, currSet1, currSet2, idx + 1, n, minDiff, resSet1, resSet2);
            currSet1.remove(currSet1.size() - 1);
        }

        if (currSet2.size() < (n + 1) / 2) {
            currSet2.add(arr[idx]);
            tugOfWarUtil(arr, currSet1, currSet2, idx + 1, n, minDiff, resSet1, resSet2);
            currSet2.remove(currSet2.size() - 1);
        }
    }

    public List<List<Integer>> tugOfWar(int[] arr) {
        int n = arr.length;
        int[] minDiff = {Integer.MAX_VALUE};
        List<Integer> currSet1 = new ArrayList<>();
        List<Integer> currSet2 = new ArrayList<>();
        List<Integer> resSet1 = new ArrayList<>();
        List<Integer> resSet2 = new ArrayList<>();
        tugOfWarUtil(arr, currSet1, currSet2, 0, n, minDiff, resSet1, resSet2);
        return Arrays.asList(resSet1, resSet2);
    }
```

DYANMIC PROGRAMMING

Fibonacci Number

C++:

```
class Solution {
public:
    static vector<int> dp;

    Solution() {
        if (dp.empty()) {
            dp.resize(31, -1);
        }
    }

    int fib(int n) {
        if (n <= 1) {
```

```
            return n;
        }

        // Temporary variables to store values of fib(n-1) & fib(n-2)
        int first, second;

        if (dp[n - 1] != -1) {
            first = dp[n - 1];
        } else {
            first = fib(n - 1);
        }

        if (dp[n - 2] != -1) {
            second = dp[n - 2];
        } else {
            second = fib(n - 2);
        }

        // Memoization
        return dp[n] = first + second;
    }
};
```

JAVA:

```
class Solution {
    public static int[] dp = new int[31];
    static {
        Arrays.fill(dp, -1);
    }
```

```
    public  int fib(int n) {
        if (n <= 1) {
            return n;
        }

        // Temporary variables to store values of fib(n-1) & fib(n-2)
        int first, second;

        if (dp[n - 1] != -1) {
            first = dp[n - 1];
        } else {
            first = fib(n - 1);
        }

        if (dp[n - 2] != -1) {
            second = dp[n - 2];
        } else {
            second = fib(n - 2);
        }

        // Memoization
        return dp[n] = first + second;
    }
}
```

Climbing Stairs

C++:

JAVA:

Counting Bits

C++:

```
class Solution {
public:
    int climbStairs(int n) {
        if (n <= 3) return n;

        int prev1 = 3;
        int prev2 = 2;
        int cur = 0;

        for (int i = 3; i < n; i++) {
            cur = prev1 + prev2;
            prev2 = prev1;
            prev1 = cur;
        }

        return cur;
    }
};
```

JAVA:

```
class Solution {
    public int climbStairs(int n) {
        if (n <= 3) return n;

        int prev1 = 3;
        int prev2 = 2;
```

```
        int cur = 0;

        for (int i = 3; i < n; i++) {
            cur = prev1 + prev2;
            prev2 = prev1;
            prev1 = cur;
        }

        return cur;
    }
}
```

Cherry Pickup II

C++:

```
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    int dp[70][70][70]; // Maximum grid size assumed as per constraints

    int cherryPickup(vector<vector<int>>& grid) {
        // Initialize the dp array with -1
        for (int i = 0; i < 70; ++i) {
            for (int j = 0; j < 70; ++j) {
                for (int k = 0; k < 70; ++k) {
                    dp[i][j][k] = -1;
                }
```

```
        }
    }
    return rec(grid, 0, 0, grid[0].size() - 1);
  }

  int rec(vector<vector<int>>& grid, int level, int c1, int c2) {
    // Base case and pruning code
    if (level >= grid.size() || c1 >= grid[0].size() || c2 >= grid[0].size() || c1 < 0 || c2 <
0) {
      return 0;
    }

    if (dp[level][c1][c2] != -1) {
      return dp[level][c1][c2];
    }

    int maxCherries = INT_MIN;
    for (int di = -1; di <= 1; di++) {
      for (int dj = -1; dj <= 1; dj++) {
        int cherries = 0;
        if (c1 == c2) {
          cherries = grid[level][c1]; // If both are in the same column
        } else {
          cherries = grid[level][c1] + grid[level][c2]; // Collect cherries from both
        }
        maxCherries = max(maxCherries, cherries + rec(grid, level + 1, c1 + di, c2
+ dj));
      }
    }
    return dp[level][c1][c2] = maxCherries; // Store the result in dp
```

```
    }
};
```

JAVA:

```
class Solution {
    int[][][] dp;
    public int cherryPickup(int[][] grid) {
        dp=new int[grid.length][grid[0].length][grid[0].length];
        for(int i=0;i<dp.length;i++){
            for(int j=0;j<dp[i].length;j++){
                for(int k=0;k<dp[i][j].length;k++){
                    dp[i][j][k]=-1;
                }
            }
        }
        return rec(grid,0,0,grid[0].length-1);
    }
    public int rec(int[][] grid,int level,int c1,int c2){
        // base case and pruning code
        if(level>=grid.length || c1>=grid[0].length || c2>=grid[0].length || c1<0 || c2<0){
            return 0;
        }

        if(dp[level][c1][c2]!=-1)
        return dp[level][c1][c2];

        int max=Integer.MIN_VALUE;
        for(int di=-1;di<=1;di++){
            for(int dj=-1;dj<=1;dj++){
```

```
            int cherry=0;
            if(c1==c2){
                cherry=grid[level][c1];
            }else{
                cherry=grid[level][c1]+grid[level][c2];
            }
            max=Math.max(max,cherry+rec(grid,level+1,c1+di,c2+dj));
        }
    }
    return dp[level][c1][c2]=max;
  }
}
```

Count of subsets with sum equal to X

C++:

```
#include <bits/stdc++.h>
using namespace std;

void printBool(int n, int len) {
    while (n) {
        if (n & 1)
            cout << 1;
        else
            cout << 0;

        n >>= 1;
        len--;
    }
```

```
    while (len) {
        cout << 0;
        len--;
    }
    cout << endl;
}

void printSubsetsCount(int set[], int n, int val) {
    int sum;
    int count = 0;
    for (int i = 0; i < (1 << n); i++) {
        sum = 0;
        for (int j = 0; j < n; j++)
            if ((i & (1 << j)) > 0) {
                sum += set[j];
            }
        if (sum == val) {
            count++;
        }
    }
    if (count == 0) {
        cout << ("No subset is found") << endl;
    } else {
        cout << count << endl;
    }
}

int main() {
```

```
    int set[] = { 1, 2, 3, 4, 5 };
    printSubsetsCount(set, 5, 9);
}
```

JAVA:

```
import java.io.*;

class GFG {

    static void printBool(int n, int len) {
        while (n > 0) {
            if ((n & 1) == 1)
                System.out.print(1);
            else
                System.out.print(0);

            n >>= 1;
            len--;
        }

        while (len > 0) {
            System.out.print(0);
            len--;
        }
        System.out.println();
    }

    static void printSubsetsCount(int set[], int n, int val) {
```

```java
        int sum;
        int count = 0;
        for (int i = 0; i < (1 << n); i++) {
            sum = 0;
            for (int j = 0; j < n; j++)
                if ((i & (1 << j)) > 0) {
                    sum += set[j];
                }
            if (sum == val) {
                count++;
            }
        }
        if (count == 0) {
            System.out.println("No subset is found");
        } else {
            System.out.println(count);
        }
    }

    public static void main(String[] args) {
        int set[] = {1, 2, 3, 4, 5};
        printSubsetsCount(set, 5, 9);
    }
}
```

Longest Common Subsequence

C++:

```
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        vector<int> dp(text1.length(), 0);
        int longest = 0;

        for (char c : text2) {
            int curLength = 0;
            for (int i = 0; i < dp.size(); i++) {
                if (curLength < dp[i]) {
                    curLength = dp[i];
                } else if (c == text1[i]) {
                    dp[i] = curLength + 1;
                    longest = max(longest, curLength + 1);
                }
            }
        }

        return longest;
    }
};
```

JAVA:

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        int[] dp = new int[text1.length()];
        int longest = 0;

        for (char c : text2.toCharArray()) {
            int curLength = 0;
            for (int i = 0; i < dp.length; i++) {
                if (curLength < dp[i]) {
                    curLength = dp[i];
                } else if (c == text1.charAt(i)) {
                    dp[i] = curLength + 1;
                    longest = Math.max(longest, curLength + 1);
                }
            }
        }

        return longest;
    }
}
```

Longest Palindromic Subsequence

C++:

```
class Solution {
public:
    int longestPalindromeSubseq(string s) {
        // Get the length of the input string
        int n = s.length();
        // Initialize a 2D array to store the length of the longest palindromic
subsequence
        vector<vector<int>> dp(n, vector<int>(n, 0));

        // Iterate over the substrings in reverse order to fill in the dp table bottom-up
        for (int i = n - 1; i >= 0; i--) {
            // Base case: the longest palindromic subsequence of a single character is 1
            dp[i][i] = 1;
            for (int j = i + 1; j < n; j++) {
                // If the two characters match, the longest palindromic subsequence can
be extended by two
                if (s[i] == s[j]) {
                    dp[i][j] = 2 + dp[i + 1][j - 1];
                } else {
                    // Otherwise, we take the maximum of the two possible substrings
                    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
                }
            }
        }

        // The length of the longest palindromic subsequence is in the top-right corner of
the dp table
```

```
        return dp[0][n - 1];
    }
};
```

JAVA:

```
class Solution {
    public int longestPalindromeSubseq(String s) {
        // Get the length of the input string
        int n = s.length();
        // Initialize a 2D array to store the length of the longest palindromic subsequence
        int[][] dp = new int[n][n];

        // Iterate over the substrings in reverse order to fill in the dp table bottom-up
        for (int i = n-1; i >= 0; i--) {
            // Base case: the longest palindromic subsequence of a single character is 1
            dp[i][i] = 1;
            for (int j = i+1; j < n; j++) {
                // If the two characters match, the longest palindromic subsequence can be extended by two
                if (s.charAt(i) == s.charAt(j)) {
                    dp[i][j] = 2 + dp[i+1][j-1];
                } else {
                    // Otherwise, we take the maximum of the two possible substrings
                    dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1]);
                }
            }
        }

        // The length of the longest palindromic subsequence is in the top-right corner of the dp table
```

```
        return dp[0][n-1];
    }
}
```

Minimum number of deletions and insertions

C++:

```
#include <bits/stdc++.h>
using namespace std;

// Function to calculate the length of the Longest Common Subsequence
int lcs(string s1, string s2) {
    int n = s1.size();
    int m = s2.size();

    // Create two arrays to store the previous and current rows of DP values
    vector<int> prev(m + 1, 0), cur(m + 1, 0);

    // Base Case is covered as we have initialized the prev and cur to 0.

    for (int ind1 = 1; ind1 <= n; ind1++) {
        for (int ind2 = 1; ind2 <= m; ind2++) {
            if (s1[ind1 - 1] == s2[ind2 - 1])
                cur[ind2] = 1 + prev[ind2 - 1];
            else
                cur[ind2] = max(prev[ind2], cur[ind2 - 1]);
        }
        // Update the prev array with the current values
        prev = cur;
    }
```

```
    // The value at prev[m] contains the length of the LCS
    return prev[m];
}

// Function to calculate the minimum operations required to convert str1 to str2
int canYouMake(string str1, string str2) {
    int n = str1.size();
    int m = str2.size();

    // Calculate the length of the longest common subsequence between str1 and str2
    int k = lcs(str1, str2);

    // Calculate the minimum operations required to convert str1 to str2
    return (n - k) + (m - k);
}

int main() {
    string str1 = "abcd";
    string str2 = "anc";

    // Call the canYouMake function and print the result
    cout << "The Minimum operations required to convert str1 to str2: " <<
canYouMake(str1, str2);
    return 0;
}
```

JAVA:

```java
import java.util.*;

class TUF {
    // Function to find the length of the Longest Common Subsequence (LCS)
    static int lcs(String s1, String s2) {
        int n = s1.length();
        int m = s2.length();

        // Create two arrays to store the LCS lengths
        int[] prev = new int[m + 1];
        int[] cur = new int[m + 1];

        // Base Case: Initialized to 0, as no characters matched yet.

        for (int ind1 = 1; ind1 <= n; ind1++) {
            for (int ind2 = 1; ind2 <= m; ind2++) {
                if (s1.charAt(ind1 - 1) == s2.charAt(ind2 - 1))
                    cur[ind2] = 1 + prev[ind2 - 1];
                else
                    cur[ind2] = Math.max(prev[ind2], cur[ind2 - 1]);
            }

            // Update prev array to store the current values
            prev = cur.clone();
        }

        return prev[m];
    }
```

```
    // Function to find the minimum operations required to convert str1 to str2
    static int canYouMake(String str1, String str2) {
        int n = str1.length();
        int m = str2.length();

        // Find the length of the LCS between str1 and str2
        int k = lcs(str1, str2);

        // The minimum operations required is the sum of the lengths of str1 and str2 minus twice the length of LCS
        return (n - k) + (m - k);
    }

    public static void main(String args[]) {
        String str1 = "abcd";
        String str2 = "anc";
        System.out.println("The Minimum operations required to convert str1 to str2: " +
                canYouMake(str1, str2));
    }
}
```

Best Time to Buy and Sell Stock II

C++:

```
class Solution {
```

```
public:
 int solve(vector<int>&prices,int currIndex, int left,int hold,vector<vector<int>>&dp){
        if(currIndex>=prices.size() || left<=0) return 0;
        if(dp[currIndex][hold] != -1) return dp[currIndex][hold];
        if(hold==1){
            int skip = solve(prices,currIndex+1,left,hold,dp);
            int sell = solve(prices,currIndex+1,left-1,0,dp) + prices[currIndex];
            return dp[currIndex][hold]=max(sell,skip);
        }else{
            int buy =  solve(prices,currIndex+1,left,1,dp)-prices[currIndex];
            int skip = solve(prices,currIndex+1,left,0,dp);
            return dp[currIndex][hold]=max(buy ,skip);
        }
    }
    int maxProfit(vector<int>& prices) {

        int k =prices.size();
        int n = prices.size();
        vector<vector<int>>dp(n+1,vector<int>(2,-1));

        return solve(prices,0,k,0,dp);
    }
};
```

JAVA:

```
class Solution {
    public int solve(int[] prices, int currIndex, int left, int hold, int[][] dp) {
        if (currIndex >= prices.length || left <= 0) return 0;
```

```java
        if (dp[currIndex][hold] != -1) return dp[currIndex][hold];

        if (hold == 1) {
            int skip = solve(prices, currIndex + 1, left, hold, dp);
            int sell = solve(prices, currIndex + 1, left - 1, 0, dp) + prices[currIndex];
            return dp[currIndex][hold] = Math.max(sell, skip);
        } else {
            int buy = solve(prices, currIndex + 1, left, 1, dp) - prices[currIndex];
            int skip = solve(prices, currIndex + 1, left, 0, dp);
            return dp[currIndex][hold] = Math.max(buy, skip);
        }
    }

    public int maxProfit(int[] prices) {
        int k = prices.length;
        int n = prices.length;
        int[][] dp = new int[n + 1][2];

        // Initialize dp array with -1
        for (int i = 0; i <= n; i++) {
            dp[i][0] = -1;
            dp[i][1] = -1;
        }

        return solve(prices, 0, k, 0, dp);
    }
}
```

Burst Balloons

C++:

```
class Solution {
public:
    int maxCoins(vector<int>& nums) {
        int n = nums.size();
        nums.insert(nums.begin(), 1);
        nums.push_back(1);
        vector<vector<int>> dp(n+2, vector<int>(n+2, 0));

        for(int i = n; i>=1; i--){
            for(int j = 1; j<=n; j++){
                if(i>j) continue;
                int maxi = INT_MIN;

                for(int ind = i; ind<=j; ind++){
                    int coins = nums[i-1]*nums[ind]*nums[j+1];
                    int remCoins = dp[i][ind-1]+dp[ind+1][j];
                    maxi = max(coins+remCoins, maxi);
                }
                dp[i][j] = maxi;
            }
        }

        return dp[1][n];
    }
};
```

JAVA:

```java
class Solution {
    public int maxCoins(int[] nums) {
        int n = nums.length;
        int[] newNums = new int[n + 2];
        System.arraycopy(nums, 0, newNums, 1, n);
        newNums[0] = 1;
        newNums[n + 1] = 1;

        int[][] dp = new int[n + 2][n + 2];

        for (int i = n; i >= 1; i--) {
            for (int j = 1; j <= n; j++) {
                if (i > j) continue;
                int maxi = Integer.MIN_VALUE;

                for (int ind = i; ind <= j; ind++) {
                    int coins = newNums[i - 1] * newNums[ind] * newNums[j + 1];
                    int remCoins = dp[i][ind - 1] + dp[ind + 1][j];
                    maxi = Math.max(coins + remCoins, maxi);
                }
                dp[i][j] = maxi;
            }
        }

        return dp[1][n];
    }
}
```

BINARY TREE AND BST-

1. Binary Tree Preorder Traversal

Brute Force (C++):

```
void preorder(TreeNode* root, vector<int>& result) {
    if (root == nullptr) return;
    result.push_back(root->val);
```

```
    preorder(root->left, result);
    preorder(root->right, result);
}

vector<int> preorderTraversal(TreeNode* root) {
    vector<int> result;
    preorder(root, result);
    return result;
}
```

Brute Force (Java):

```
void preorder(TreeNode root, List<Integer> result) {
    if (root == null) return;
    result.add(root.val);
    preorder(root.left, result);
    preorder(root.right, result);
}

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

Optimized (C++):

```
vector<int> preorderTraversal(TreeNode* root) {
    vector<int> result;
    stack<TreeNode*> s;
    if (root) s.push(root);

    while (!s.empty()) {
```

```
        TreeNode* node = s.top();
        s.pop();
        result.push_back(node->val);
        if (node->right) s.push(node->right);
        if (node->left) s.push(node->left);
    }
    return result;
}
```

Optimized (Java):

```
public List<Integer> preorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();
    if (root != null) 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;
}
```

2. Binary Tree Inorder Traversal

Brute Force (C++):

```
void inorder(TreeNode* root, vector<int>& result) {
    if (root == nullptr) return;
    inorder(root->left, result);
    result.push_back(root->val);
```

```
    inorder(root->right, result);
}

vector<int> inorderTraversal(TreeNode* root) {
    vector<int> result;
    inorder(root, result);
    return result;
}
```

Brute Force (Java):

```
void inorder(TreeNode root, List<Integer> result) {
    if (root == null) return;
    inorder(root.left, result);
    result.add(root.val);
    inorder(root.right, result);
}

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

Optimized (C++):

```
vector<int> inorderTraversal(TreeNode* root) {
    vector<int> result;
    stack<TreeNode*> s;
    TreeNode* current = root;

    while (current != nullptr || !s.empty()) {
        while (current != nullptr) {
```

```
            s.push(current);
            current = current->left;
        }
        current = s.top();
        s.pop();
        result.push_back(current->val);
        current = current->right;
    }
    return result;
}
```

Optimized (Java):

```java
public List<Integer> inorderTraversal(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;
}
```

3. Binary Tree Level Order Traversal

Brute Force (C++):

```
vector<vector<int>> levelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (root == nullptr) return result;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int size = q.size();
        vector<int> currentLevel;

        for (int i = 0; i < size; i++) {
            TreeNode* node = q.front();
            q.pop();
            currentLevel.push_back(node->val);
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
        result.push_back(currentLevel);
    }
    return result;
}
```

Brute Force (Java):

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) return result;

    Queue<TreeNode> queue = new LinkedList<>();
```

```java
    queue.offer(root);

    while (!queue.isEmpty()) {
        int size = queue.size();
        List<Integer> currentLevel = new ArrayList<>();

        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            currentLevel.add(node.val);
            if (node.left != null) queue.offer(node.left);
            if (node.right != null) queue.offer(node.right);
        }
        result.add(currentLevel);
    }
    return result;
}
```

4. Vertical Order Traversal of a Binary Tree

Brute Force (C++):

```cpp
vector<vector<int>> verticalOrder(TreeNode* root) {
    map<int, vector<int>> nodes;
    queue<pair<TreeNode*, int>> q;
    if (root) q.push({root, 0});

    while (!q.empty()) {
        auto p = q.front();
        q.pop();
        TreeNode* node = p.first;
        int column = p.second;
        nodes[column].push_back(node->val);
```

```
        if (node->left) q.push({node->left, column - 1});
        if (node->right) q.push({node->right, column + 1});
    }

    vector<vector<int>> result;
    for (auto& [key, value] : nodes) {
        result.push_back(value);
    }
    return result;
}
```

Brute Force (Java):

```
public List<List<Integer>> verticalOrder(TreeNode root) {
    Map<Integer, List<Integer>> map = new HashMap<>();
    Queue<Pair<TreeNode, Integer>> queue = new LinkedList<>();
    if (root != null) queue.offer(new Pair<>(root, 0));

    while (!queue.isEmpty()) {
        Pair<TreeNode, Integer> p = queue.poll();
        TreeNode node = p.getKey();
        int column = p.getValue();

        map.computeIfAbsent(column, k -> new ArrayList<>()).add(node.val);

        if (node.left != null) queue.offer(new Pair<>(node.left, column - 1));
        if (node.right != null) queue.offer(new Pair<>(node.right, column + 1));
    }

    List<List<Integer>> result = new ArrayList<>();
```

```
    List<Integer> keys = new ArrayList<>(map.keySet());
    Collections.sort(keys);

    for (int key : keys) {
        result.add(map.get(key));
    }
    return result;
}
```

5. Top View of Binary Tree

Brute Force (C++):

```
vector<int> topView(TreeNode* root) {
    vector<int> result;
    if (!root) return result;

    map<int, int> topNodes;
    queue<pair<TreeNode*, int>> q;
    q.push({root, 0});

    while (!q.empty()) {
        auto p = q.front();
        q.pop();
        TreeNode* node = p.first;
        int hd = p.second;
```

```cpp
        if (topNodes.find(hd) == topNodes.end()) {
            topNodes[hd] = node->val;
        }

        if (node->left) q.push({node->left, hd - 1});
        if (node->right) q.push({node->right, hd + 1});
    }

    for (auto& [key, value] : topNodes) {
        result.push_back(value);
    }
    return result;
}
```

Brute Force (Java):

```java
public List<Integer> topView(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) return result;

    Map<Integer, Integer> topNodes = new TreeMap<>();
    Queue<Pair<TreeNode, Integer>> queue = new LinkedList<>();
    queue.offer(new Pair<>(root, 0));

    while (!queue.isEmpty()) {
        Pair<TreeNode, Integer> pair = queue.poll();
        TreeNode node = pair.getKey();
        int hd = pair.getValue();

        if (!topNodes.containsKey(hd)) {
```

```
            topNodes.put(hd, node.val);
        }

        if (node.left != null) queue.offer(new Pair<>(node.left, hd - 1));
        if (node.right != null) queue.offer(new Pair<>(node.right, hd + 1));
    }

    result.addAll(topNodes.values());
    return result;
}
```

6. Binary Tree Right Side View

Brute Force (C++):

```
vector<int> rightSideView(TreeNode* root) {
    vector<int> result;
    if (!root) return result;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int size = q.size();
        for (int i = 0; i < size; i++) {
            TreeNode* node = q.front();
            q.pop();
            if (i == size - 1) result.push_back(node->val);
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
```

```
        }
    }
    return result;
}
```

Brute Force (Java):

```
public List<Integer> rightSideView(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) return result;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            if (i == size - 1) result.add(node.val);
            if (node.left != null) queue.offer(node.left);
            if (node.right != null) queue.offer(node.right);
        }
    }
    return result;
}
```

7. Left View of Binary Tree

Brute Force (C++):

```
vector<int> leftView(TreeNode* root) {
    vector<int> result;
    if (!root) return result;
```

```
    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int size = q.size();
        for (int i = 0; i < size; i++) {
            TreeNode* node = q.front();
            q.pop();
            if (i == 0) result.push_back(node->val);
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }
    return result;
}
```

Brute Force (Java):

```
public List<Integer> leftView(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) return result;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            if (i == 0) result.add(node.val);
            if (node.left != null) queue.offer(node.left);
```

```
            if (node.right != null) queue.offer(node.right);
        }
    }
    return result;
}
```

8. Bottom View of Binary Tree

Brute Force (C++):

```
vector<int> bottomView(TreeNode* root) {
    vector<int> result;
    if (!root) return result;

    map<int, int> bottomNodes;
    queue<pair<TreeNode*, int>> q;
    q.push({root, 0});

    while (!q.empty()) {
        auto p = q.front();
        q.pop();
        TreeNode* node = p.first;
        int hd = p.second;

        bottomNodes[hd] = node->val;

        if (node->left) q.push({node->left, hd - 1});
        if (node->right) q.push({node->right, hd + 1});
```

```
    }

    for (auto& [key, value] : bottomNodes) {
        result.push_back(value);
    }
    return result;
}
```

Brute Force (Java):

```
public List<Integer> bottomView(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) return result;

    Map<Integer, Integer> bottomNodes = new TreeMap<>();
    Queue<Pair<TreeNode, Integer>> queue = new LinkedList<>();
    queue.offer(new Pair<>(root, 0));

    while (!queue.isEmpty()) {
        Pair<TreeNode, Integer> pair = queue.poll();
        TreeNode node = pair.getKey();
        int hd = pair.getValue();

        bottomNodes.put(hd, node.val);

        if (node.left != null) queue.offer(new Pair<>(node.left, hd - 1));
        if (node.right != null) queue.offer(new Pair<>(node.right, hd + 1));
    }

    result.addAll(bottomNodes.values());
```

```
    return result;
}
```

9. Burning Tree

Brute Force (C++):

```
int minTimeToBurn(TreeNode* root, int target) {
    if (!root) return -1;
    if (root->val == target) return 0;

    int left = minTimeToBurn(root->left, target);
    int right = minTimeToBurn(root->right, target);

    if (left != -1) {
        // If burning from left
        // Do something
    }
    if (right != -1) {
        // If burning from right
        // Do something
    }
    return -1; // or return the time needed to burn
}
```

Brute Force (Java):

```
public int minTimeToBurn(TreeNode root, int target) {
    if (root == null) return -1;
```

```
    if (root.val == target) return 0;

    int left = minTimeToBurn(root.left, target);
    int right = minTimeToBurn(root.right, target);

    if (left != -1) {
        // If burning from left
        // Do something
    }
    if (right != -1) {
        // If burning from right
        // Do something
    }
    return -1; // or return the time needed to burn
}
```

10. Kth Smallest Element in a BST

Brute Force (C++):

```
void inorder(TreeNode* root, vector<int>& result) {
    if (!root) return;
    inorder(root->left, result);
    result.push_back(root->val);
    inorder(root->right, result);
}

int kthSmallest(TreeNode* root, int k) {
    vector<int> result;
    inorder(root, result);
    return result[k - 1];
}
```

Brute Force (Java):

```
void inorder(TreeNode root, List<Integer> result) {
    if (root == null) return;
    inorder(root.left, result);
    result.add(root.val);
    inorder(root.right, result);
}

public int kthSmallest(TreeNode root, int k) {
    List<Integer> result = new ArrayList<>();
    inorder(root, result);
    return result.get(k - 1);
}
```

Optimized (C++):

```
int kthSmallest(TreeNode* root, int& k) {
    if (!root) return -1;

    int left = kthSmallest(root->left, k);
    if (k == 0) return left;

    k--;
    if (k == 0) return root->val;

    return kthSmallest(root->right, k);
}

int kthSmallest(TreeNode* root, int k) {
    return kthSmallest(root, k);
}
```

Optimized (Java):

```java
private int kthSmallest(TreeNode root, int[] k) {
    if (root == null) return -1;

    int left = kthSmallest(root.left, k);
    if (k[0] == 0) return left;

    k[0]--;
    if (k[0] == 0) return root.val;

    return kthSmallest(root.right, k);
}

public int kthSmallest(TreeNode root, int k) {
    return kthSmallest(root, new int[]{k});
}
```

11. Largest BST in a Binary Tree

Brute Force (C++):

```cpp
bool isBST(TreeNode* root, int min, int max) {
    if (!root) return true;
    if (root->val <= min || root->val >= max) return false;
    return isBST(root->left, min, root->val) && isBST(root->right, root->val, max);
}

int largestBSTSubtree(TreeNode* root) {
    if (!root) return 0;

    if (isBST(root, INT_MIN, INT_MAX)) {
```

```
        return countNodes(root);
    }

    return max(largestBSTSubtree(root->left), largestBSTSubtree(root->right));
}
```

Brute Force (Java):

```
public boolean isBST(TreeNode root, int min, int max) {
    if (root == null) return true;
    if (root.val <= min || root.val >= max) return false;
    return isBST(root.left, min, root.val) && isBST(root.right, root.val, max);
}

public int largestBSTSubtree(TreeNode root) {
    if (root == null) return 0;

    if (isBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE)) {
        return countNodes(root);
    }

    return Math.max(largestBSTSubtree(root.left), largestBSTSubtree(root.right));
}
```

Optimized (C++):

```
struct Info {
    int size;
    int min;
    int max;
    bool isBST;
};
```

```
Info largestBSTHelper(TreeNode* root) {
    if (!root) return {0, INT_MAX, INT_MIN, true};

    Info left = largestBSTHelper(root->left);
    Info right = largestBSTHelper(root->right);

    if (left.isBST && right.isBST && root->val > left.max && root->val < right.min) {
        return {left.size + right.size + 1, min(root->val, left.min), max(root->val,
right.max), true};
    }

    return {max(left.size, right.size), 0, 0, false};
}

int largestBSTSubtree(TreeNode* root) {
    return largestBSTHelper(root).size;
}
```

Optimized (Java):

```
class Info {
    int size;
    int min;
    int max;
    boolean isBST;
}

public Info largestBSTHelper(TreeNode root) {
    if (root == null) return new Info(0, Integer.MAX_VALUE, Integer.MIN_VALUE, true);

    Info left = largestBSTHelper(root.left);
```

```
    Info right = largestBSTHelper(root.right);

    if (left.isBST && right.isBST && root.val > left.max && root.val < right.min) {
        return new Info(left.size + right.size + 1, Math.min(root.val, left.min),
Math.max(root.val, right.max), true);
    }

    return new Info(Math.max(left.size, right.size), 0, 0, false);
}

public int largestBSTSubtree(TreeNode root) {
    return largestBSTHelper(root).size;
}
```

12. Binary Tree Maximum Path Sum

Brute Force (C++):

```
int maxPathSumHelper(TreeNode* root, int& maxSum) {
    if (!root) return 0;

    int left = max(maxPathSumHelper(root->left, maxSum), 0);
    int right = max(maxPathSumHelper(root->right, maxSum), 0);

    maxSum = max(maxSum, left + right + root->val);
    return max(left, right) + root->val;
}

int maxPathSum(TreeNode* root) {
    int maxSum = INT_MIN;
    maxPathSumHelper(root, maxSum);
```

```
    return maxSum;
}
```

Brute Force (Java):

```
public int maxPathSumHelper(TreeNode root, int[] maxSum) {
    if (root == null) return 0;

    int left = Math.max(maxPathSumHelper(root.left, maxSum), 0);
    int right = Math.max(maxPathSumHelper(root.right, maxSum), 0);

    maxSum[0] = Math.max(maxSum[0], left + right + root.val);
    return Math.max(left, right) + root.val;
}

public int maxPathSum(TreeNode root) {
    int[] maxSum = new int[]{Integer.MIN_VALUE};
    maxPathSumHelper(root, maxSum);
    return maxSum[0];
}
```

13. Serialize and Deserialize Binary Tree

Brute Force (C++):

```
string serialize(TreeNode* root) {
    if (!root) return "#";
    return to_string(root->val) + "," + serialize(root->left) + "," + serialize(root->right);
}

TreeNode* deserializeHelper(istringstream& iss) {
    string val;
    if (!getline(iss, val, ',')) return nullptr;
    if (val == "#") return nullptr;
```

```
    TreeNode* node = new TreeNode(stoi(val));
    node->left = deserializeHelper(iss);
    node->right = deserializeHelper(iss);
    return node;
}

TreeNode* deserialize(string data) {
    istringstream iss(data);
    return deserializeHelper(iss);
}
```

Brute Force (Java):

```
public String serialize(TreeNode root) {
    if (root == null) return "#,";
    return root.val + "," + serialize(root.left) + serialize(root.right);
}

public TreeNode deserializeHelper(Queue<String> nodes) {
    String val = nodes.poll();
    if (val.equals("#")) return null;

    TreeNode node = new TreeNode(Integer.parseInt(val));
    node.left = deserializeHelper(nodes);
    node.right = deserializeHelper(nodes);
    return node;
}

public TreeNode deserialize(String data) {
    Queue<String> nodes = new LinkedList<>(Arrays.asList(data.split(",")));
```

```
    return deserializeHelper(nodes);
}
```

14. Lowest Common Ancestor of a Binary Search Tree

Brute Force (C++):

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (!root) return nullptr;
    if (root == p || root == q) return root;

    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    TreeNode* right = lowestCommonAncestor(root->right, p, q);

    if (left && right) return root;
    return left ? left : right;
}
```

Brute Force (Java):

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
    if (root == null) return null;
    if (root == p || root == q) return root;

    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);

    if (left != null && right != null) return root;
    return left != null ? left : right;
}
```

Optimized (C++):

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
```

```cpp
    while (root) {
        if (root->val < p->val && root->val < q->val) {
            root = root->right;
        } else if (root->val > p->val && root->val > q->val) {
            root = root->left;
        } else {
            return root;
        }
    }
    return nullptr;
}
```

Optimized (Java):

```java
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
    while (root != null) {
        if (root.val < p.val && root.val < q.val) {
            root = root.right;
        } else if (root.val > p.val && root.val > q.val) {
            root = root.left;
        } else {
            return root;
        }
    }
    return null;
}
```

15. Two Sum IV - Input is a BST

Brute Force (C++):

```cpp
void inorder(TreeNode* root, vector<int>& vals) {
    if (!root) return;
    inorder(root->left, vals);
    vals.push_back(root->val);
    inorder(root->right, vals);
}

bool findTarget(TreeNode* root, int k) {
    vector<int> vals;
    inorder(root, vals);
    unordered_set<int> seen;

    for (int val : vals) {
        if (seen.count(k - val)) return true;
        seen.insert(val);
    }
    return false;
}
```

Brute Force (Java):

```java
public void inorder(TreeNode root, List<Integer> vals) {
    if (root == null) return;
    inorder(root.left, vals);
    vals.add(root.val);
    inorder(root.right, vals);
}

public boolean findTarget(TreeNode root, int k) {
    List<Integer> vals = new ArrayList<>();
    inorder(root, vals);
```

```
    Set<Integer> seen = new HashSet<>();

    for (int val : vals) {
        if (seen.contains(k - val)) return true;
        seen.add(val);
    }
    return false;
}
```

Optimized (C++):

```
bool findTarget(TreeNode* root, int k) {
    unordered_set<int> seen;
    stack<TreeNode*> stk;
    TreeNode* curr = root;

    while (curr || !stk.empty()) {
        while (curr) {
            stk.push(curr);
            curr = curr->left;
        }
        curr = stk.top();
        stk.pop();

        if (seen.count(k - curr->val)) return true;
        seen.insert(curr->val);
        curr = curr->right;
    }
    return false;
}
```

Optimized (Java):

```java
public boolean findTarget(TreeNode root, int k) {
    Set<Integer> seen = new HashSet<>();
    Stack<TreeNode> stk = new Stack<>();
    TreeNode curr = root;

    while (curr != null || !stk.isEmpty()) {
        while (curr != null) {
            stk.push(curr);
            curr = curr.left;
        }
        curr = stk.pop();

        if (seen.contains(k - curr.val)) return true;
        seen.add(curr.val);
        curr = curr.right;
    }
    return false;
}
```

GRAPH

1. DFS of Graph

Brute Force (C++):

```cpp
void dfs(int node, vector<bool>& visited, vector<vector<int>>& graph) {
    visited[node] = true; // Mark the current node as visited
```

```cpp
    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor, visited, graph); // Recur for all unvisited neighbors
        }
    }
}

void depthFirstSearch(vector<vector<int>>& graph) {
    int n = graph.size();
    vector<bool> visited(n, false); // Initialize visited array
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(i, visited, graph); // Call DFS for each unvisited node
        }
    }
}
```

Brute Force (Java):

```java
public void dfs(int node, boolean[] visited, List<List<Integer>> graph) {
    visited[node] = true; // Mark the current node as visited
    for (int neighbor : graph.get(node)) {
        if (!visited[neighbor]) {
            dfs(neighbor, visited, graph); // Recur for all unvisited neighbors
        }
    }
}

public void depthFirstSearch(List<List<Integer>> graph) {
    int n = graph.size();
    boolean[] visited = new boolean[n]; // Initialize visited array
```

```
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(i, visited, graph); // Call DFS for each unvisited node
        }
    }
}
```

Optimized (C++): The optimized version utilizes an iterative approach to avoid stack overflow issues with deep recursions.

```
void depthFirstSearch(vector<vector<int>>& graph) {
    int n = graph.size();
    vector<bool> visited(n, false); // Initialize visited array
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            stack<int> stk; // Create a stack for iterative DFS
            stk.push(i);
            while (!stk.empty()) {
                int node = stk.top();
                stk.pop();
                if (!visited[node]) {
                    visited[node] = true; // Mark the current node as visited
                    for (int neighbor : graph[node]) {
                        if (!visited[neighbor]) {
                            stk.push(neighbor); // Push unvisited neighbors onto the stack
                        }
                    }
                }
            }
        }
    }
```

```
}
```

Optimized (Java):

```
public void depthFirstSearch(List<List<Integer>> graph) {
    int n = graph.size();
    boolean[] visited = new boolean[n]; // Initialize visited array
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            Stack<Integer> stk = new Stack<>(); // Create a stack for iterative DFS
            stk.push(i);
            while (!stk.isEmpty()) {
                int node = stk.pop();
                if (!visited[node]) {
                    visited[node] = true; // Mark the current node as visited
                    for (int neighbor : graph.get(node)) {
                        if (!visited[neighbor]) {
                            stk.push(neighbor); // Push unvisited neighbors onto the stack
                        }
                    }
                }
            }
        }
    }
}
```

2. BFS of Graph

Brute Force (C++):

```
void bfs(int start, vector<bool>& visited, vector<vector<int>>& graph) {
    queue<int> q; // Create a queue for BFS
    q.push(start);
```

```
    visited[start] = true; // Mark the start node as visited

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
                q.push(neighbor); // Add unvisited neighbors to the queue
                visited[neighbor] = true; // Mark them as visited
            }
        }
    }
}

void breadthFirstSearch(vector<vector<int>>& graph) {
    int n = graph.size();
    vector<bool> visited(n, false); // Initialize visited array
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            bfs(i, visited, graph); // Call BFS for each unvisited node
        }
    }
}
```

Brute Force (Java):

```
public void bfs(int start, boolean[] visited, List<List<Integer>> graph) {
    Queue<Integer> q = new LinkedList<>(); // Create a queue for BFS
    q.add(start);
    visited[start] = true; // Mark the start node as visited
```

```
    while (!q.isEmpty()) {
        int node = q.poll();
        for (int neighbor : graph.get(node)) {
            if (!visited[neighbor]) {
                q.add(neighbor); // Add unvisited neighbors to the queue
                visited[neighbor] = true; // Mark them as visited
            }
        }
    }
}

public void breadthFirstSearch(List<List<Integer>> graph) {
    int n = graph.size();
    boolean[] visited = new boolean[n]; // Initialize visited array
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            bfs(i, visited, graph); // Call BFS for each unvisited node
        }
    }
}
```

Optimized (C++): The optimized approach utilizes a queue for BFS, which inherently provides efficiency.

```
void breadthFirstSearch(vector<vector<int>>& graph) {
    int n = graph.size();
    vector<bool> visited(n, false); // Initialize visited array
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            queue<int> q; // Create a queue for BFS
            q.push(i);
```

```
            visited[i] = true; // Mark the start node as visited

            while (!q.empty()) {
                int node = q.front();
                q.pop();
                for (int neighbor : graph[node]) {
                    if (!visited[neighbor]) {
                        q.push(neighbor); // Add unvisited neighbors to the queue
                        visited[neighbor] = true; // Mark them as visited
                    }
                }
            }
        }
    }
}
```

Optimized (Java): This approach utilizes a queue for efficient breadth-first traversal.

```
public void breadthFirstSearch(List<List<Integer>> graph) {
    int n = graph.size();
    boolean[] visited = new boolean[n]; // Initialize visited array
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            Queue<Integer> q = new LinkedList<>(); // Create a queue for BFS
            q.add(i);
            visited[i] = true; // Mark the start node as visited

            while (!q.isEmpty()) {
```

```java
            int node = q.poll();
            for (int neighbor : graph.get(node)) {
                if (!visited[neighbor]) {
                    q.add(neighbor); // Add unvisited neighbors to the queue
                    visited[neighbor] = true; // Mark them as visited
                }
            }
        }
    }
}
}
```

3. Number of Triangles

Brute Force (C++): This approach checks all combinations of three vertices to count triangles.

```cpp
int countTriangles(vector<vector<int>>& graph) {
    int count = 0;
    int n = graph.size();
    for (int i = 0; i < n; i++) {
        for (int j : graph[i]) {
            for (int k : graph[j]) {
                if (i == k) count++; // Check if i, j, k form a triangle
            }
        }
    }
    return count / 3; // Each triangle is counted three times
}
```

Brute Force (Java):

```java
public int countTriangles(List<List<Integer>> graph) {
```

```
    int count = 0;
    int n = graph.size();
    for (int i = 0; i < n; i++) {
        for (int j : graph.get(i)) {
            for (int k : graph.get(j)) {
                if (i == k) count++; // Check if i, j, k form a triangle
            }
        }
    }
    return count / 3; // Each triangle is counted three times
}
```

Optimized (C++): This approach uses adjacency sets to reduce the time complexity for checking edges.

```
int countTriangles(vector<vector<int>>& graph) {
    int count = 0;
    int n = graph.size();
    vector<unordered_set<int>> adj(n);
    for (int i = 0; i < n; i++) {
        for (int neighbor : graph[i]) {
            adj[i].insert(neighbor);
        }
    }
    for (int i = 0; i < n; i++) {
        for (int j : graph[i]) {
            for (int k : graph[j]) {
                if (i != j && i != k && j != k && adj[i].count(k)) {
                    count++; // Check if i, j, k form a triangle
                }
            }
```

```
        }
    }
    return count / 3; // Each triangle is counted three times
}
```

Optimized (Java): This version uses sets for efficient edge existence checks.

```
public int countTriangles(List<List<Integer>> graph) {
    int count = 0;
    int n = graph.size();
    Set<Integer>[] adj = new HashSet[n];
    for (int i = 0; i < n; i++) {
        adj[i] = new HashSet<>();
        for (int neighbor : graph.get(i)) {
            adj[i].add(neighbor);
        }
    }
    for (int i = 0; i < n; i++) {
        for (int j : graph.get(i)) {
            for (int k : graph.get(j)) {
                if (i != j && i != k && j != k && adj[i].contains(k)) {
                    count++; // Check if i, j, k form a triangle
                }
            }
        }
    }
    return count / 3; // Each triangle is counted three times
}
```

4. Rotting Oranges

Brute Force (C++): This approach uses BFS to propagate the rotting effect over time.

```
int orangesRotting(vector<vector<int>>& grid) {
    int rows = grid.size(), cols = grid[0].size();
    queue<pair<int, int>> q;
    int freshCount = 0;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (grid[i][j] == 2) {
                q.push({i, j}); // Add rotten oranges to the queue
            } else if (grid[i][j] == 1) {
                freshCount++; // Count fresh oranges
            }
        }
    }

    int minutes = 0;
    while (!q.empty() && freshCount > 0) {
        int size = q.size();
        for (int i = 0; i < size; i++) {
            auto [x, y] = q.front(); q.pop();
            for (auto& dir : vector<pair<int, int>>{{1,0}, {-1,0}, {0,1}, {0,-1}}) {
                int nx = x + dir.first, ny = y + dir.second;
                if (nx >= 0 && nx < rows && ny >= 0 && ny < cols && grid[nx][ny] == 1) {
                    grid[nx][ny] = 2; // Rot the fresh orange
                    q.push({nx, ny}); // Add it to the queue
                    freshCount--;
                }
```

```
            }
        }
        minutes++; // Increment minute count after processing one layer
    }

    return freshCount == 0 ? minutes : -1; // Return -1 if there are still fresh oranges
}
```

Brute Force (Java): This version uses a similar BFS approach.

```
public int orangesRotting(int[][] grid) {
    int rows = grid.length, cols = grid[0].length;
    Queue<int[]> q = new LinkedList<>();
    int freshCount = 0;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (grid[i][j] == 2) {
                q.add(new int[]{i, j}); // Add rotten oranges to the queue
            } else if (grid[i][j] == 1) {
                freshCount++; // Count fresh oranges
            }
        }
    }

    int minutes = 0;
    while (!q.isEmpty() && freshCount > 0) {
        int size = q.size();
        for (int i = 0; i < size; i++) {
            int[] orange = q.poll();
            for (int[] dir : new int[][]{{1,0}, {-1,0}, {0,1}, {0,-1}}) {
```

```
            int nx = orange[0] + dir[0], ny = orange[1] + dir[1];
            if (nx >= 0 && nx < rows && ny >= 0 && ny < cols && grid[nx][ny] == 1) {
                grid[nx][ny] = 2; // Rot the fresh orange
                q.add(new int[]{nx, ny}); // Add it to the queue
                freshCount--;
            }
        }
    }
    minutes++; // Increment minute count after processing one layer
  }

  return freshCount == 0 ? minutes : -1; // Return -1 if there are still fresh oranges
}
```

Optimized (C++): The optimized approach uses the same BFS but may utilize additional space for direction vectors.

```
int orangesRotting(vector<vector<int>>& grid) {
  int rows = grid.size(), cols = grid[0].size();
  queue<pair<int, int>> q;
  int freshCount = 0;

  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      if (grid[i][j] == 2) {
        q.push({i, j}); // Add rotten oranges to the queue
      } else if (grid[i][j] == 1) {
        freshCount++; // Count fresh oranges
      }
    }
  }
```

```
    int minutes = 0;
    while (!q.empty() && freshCount > 0) {
        int size = q.size();
        for (int i = 0; i < size; i++) {
            auto [x, y] = q.front(); q.pop();
            for (auto& dir : vector<pair<int, int>>{{1,0}, {-1,0}, {0,1}, {0,-1}}) {
                int nx = x + dir.first, ny = y + dir.second;
                if (nx >= 0 && nx < rows && ny >= 0 && ny < cols && grid[nx][ny] == 1) {
                    grid[nx][ny] = 2; // Rot the fresh orange
                    q.push({nx, ny}); // Add it to the queue
                    freshCount--;
                }
            }
        }
        minutes++; // Increment minute count after processing one layer
    }

    return freshCount == 0 ? minutes : -1; // Return -1 if there are still fresh oranges
}
```

Optimized (Java):

```
public int orangesRotting(int[][] grid) {
    int rows = grid.length, cols = grid[0].length;
    Queue<int[]> q = new LinkedList<>();
    int freshCount = 0;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (grid[i][j] == 2) {
```

```
            q.add(new int[]{i, j}); // Add rotten oranges to the queue
        } else if (grid[i][j] == 1) {
            freshCount++; // Count fresh oranges
        }
    }
}

int minutes = 0;
while (!q.isEmpty() && freshCount > 0) {
    int size = q.size();
    for (int i = 0; i < size; i++) {
        int[] orange = q.poll();
        for (int[] dir : new int[][]{{1,0}, {-1,0}, {0,1}, {0,-1}}) {
            int nx = orange[0] + dir[0], ny = orange[1] + dir[1];
            if (nx >= 0 && nx < rows && ny >= 0 && ny < cols && grid[nx][ny] == 1) {
                grid[nx][ny] = 2; // Rot the fresh orange
                q.add(new int[]{nx, ny}); // Add it to the queue
                freshCount--;
            }
        }
    }
    minutes++; // Increment minute count after processing one layer
}

return freshCount == 0 ? minutes : -1; // Return -1 if there are still fresh oranges
}
```

5. Course Schedule

Brute Force (C++): This approach attempts to use BFS to check for cycles and see if all courses can be completed.

```
bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
    vector<int> inDegree(numCourses, 0);
    vector<vector<int>> adj(numCourses);

    for (auto& p : prerequisites) {
        adj[p[1]].push_back(p[0]);
        inDegree[p[0]]++; // Count incoming edges for each course
    }

    queue<int> q;
    for (int i = 0; i < numCourses; i++) {
        if (inDegree[i] == 0) {
            q.push(i); // Add courses with no prerequisites to the queue
        }
    }

    int count = 0;
    while (!q.empty()) {
        int course = q.front(); q.pop();
        count++; // Increment completed courses count
        for (int neighbor : adj[course]) {
            inDegree[neighbor]--; // Decrease in-degree
            if (inDegree[neighbor] == 0) {
                q.push(neighbor); // Add to queue if no more prerequisites
            }
        }
    }
```

```java
    return count == numCourses; // Check if all courses can be completed
}
```

Brute Force (Java):

```java
public boolean canFinish(int numCourses, int[][] prerequisites) {
    int[] inDegree = new int[numCourses];
    List<List<Integer>> adj = new ArrayList<>();
    for (int i = 0; i < numCourses; i++) {
        adj.add(new ArrayList<>());
    }

    for (int[] p : prerequisites) {
        adj.get(p[1]).add(p[0]);
        inDegree[p[0]]++; // Count incoming edges for each course
    }

    Queue<Integer> q = new LinkedList<>();
    for (int i = 0; i < numCourses; i++) {
        if (inDegree[i] == 0) {
            q.add(i); // Add courses with no prerequisites to the queue
        }
    }

    int count = 0;
    while (!q.isEmpty()) {
        int course = q.poll();
        count++; // Increment completed courses count
        for (int neighbor : adj.get(course)) {
            inDegree[neighbor]--; // Decrease in-degree
```

```
            if (inDegree[neighbor] == 0) {
                q.add(neighbor); // Add to queue if no more prerequisites
            }
        }
    }

    return count == numCourses; // Check if all courses can be completed
}
```

Optimized (C++): This version also uses BFS and is efficient.

```
bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
    vector<int> inDegree(numCourses, 0);
    vector<vector<int>> adj(numCourses);

    for (auto& p : prerequisites) {
        adj[p[1]].push_back(p[0]);
        inDegree[p[0]]++; // Count incoming edges for each course
    }

    queue<int> q;
    for (int i = 0; i < numCourses; i++) {
        if (inDegree[i] == 0) {
            q.push(i); // Add courses with no prerequisites to the queue
        }
    }

    int count = 0;
    while (!q.empty()) {
        int course = q.front(); q.pop();
        count++; // Increment completed courses count
```

```
        for (int neighbor : adj[course]) {
            inDegree[neighbor]--; // Decrease in-degree
            if (inDegree[neighbor] == 0) {
                q.push(neighbor); // Add to queue if no more prerequisites
            }
        }
    }

    return count == numCourses; // Check if all courses can be completed
}
```

Optimized (Java): This version is similarly efficient in Java.

```
public boolean canFinish(int numCourses, int[][] prerequisites) {
    int[] inDegree = new int[numCourses];
    List<List<Integer>> adj = new ArrayList<>();
    for (int i = 0; i < numCourses; i++) {
        adj.add(new ArrayList<>());
    }

    for (int[] p : prerequisites) {
        adj.get(p[1]).add(p[0]);
        inDegree[p[0]]++; // Count incoming edges for each course
    }

    Queue<Integer> q = new LinkedList<>();
    for (int i = 0; i < numCourses; i++) {
        if (inDegree[i] == 0) {
            q.add(i); // Add courses with no prerequisites to the queue
        }
    }
```

```
    int count = 0;
    while (!q.isEmpty()) {
        int course = q.poll();
        count++; // Increment completed courses count
        for (int neighbor : adj.get(course)) {
            inDegree[neighbor]--; // Decrease in-degree
            if (inDegree[neighbor] == 0) {
                q.add(neighbor); // Add to queue if no more prerequisites
            }
        }
    }

    return count == numCourses; // Check if all courses can be completed
}
```

6. Course Schedule II

Brute Force (C++): This approach utilizes a modified BFS to find a valid course order.

```
vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
    vector<int> inDegree(numCourses, 0);
    vector<vector<int>> adj(numCourses);

    for (auto& p : prerequisites) {
        adj[p[1]].push_back(p[0]);
        inDegree[p[0]]++; // Count incoming edges for each course
    }

    queue<int> q;
```

```
    for (int i = 0; i < numCourses; i++) {
        if (inDegree[i] == 0) {
            q.push(i); // Add courses with no prerequisites to the queue
        }
    }

    vector<int> order;
    while (!q.empty()) {
        int course = q.front(); q.pop();
        order.push_back(course); // Add the course to the order
        for (int neighbor : adj[course]) {
            inDegree[neighbor]--; // Decrease in-degree
            if (inDegree[neighbor] == 0) {
                q.push(neighbor); // Add to queue if no more prerequisites
            }
        }
    }

    return order.size() == numCourses ? order : vector<int>(); // Return order if valid
}
```

Brute Force (Java):

```
public int[] findOrder(int numCourses, int[][] prerequisites) {
    int[] inDegree = new int[numCourses];
    List<List<Integer>> adj = new ArrayList<>();
    for (int i = 0; i < numCourses; i++) {
        adj.add(new ArrayList<>());
    }

    for (int[] p : prerequisites) {
```

```java
        adj.get(p[1]).add(p[0]);
        inDegree[p[0]]++; // Count incoming edges for each course
    }

    Queue<Integer> q = new LinkedList<>();
    for (int i = 0; i < numCourses; i++) {
        if (inDegree[i] == 0) {
            q.add(i); // Add courses with no prerequisites to the queue
        }
    }

    List<Integer> order = new ArrayList<>();
    while (!q.isEmpty()) {
        int course = q.poll();
        order.add(course); // Add the course to the order
        for (int neighbor : adj.get(course)) {
            inDegree[neighbor]--; // Decrease in-degree
            if (inDegree[neighbor] == 0) {
                q.add(neighbor); // Add to queue if no more prerequisites
            }
        }
    }

    if (order.size() == numCourses) {
        return order.stream().mapToInt(i -> i).toArray(); // Convert to int array
    }
    return new int[0]; // Return empty array if no valid order
}
```

Optimized (C++):

```
}

vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
    vector<int> inDegree(numCourses, 0);
    vector<vector<int>> adj(numCourses);

    for (auto& p : prerequisites) {
        adj[p[1]].push_back(p[0]);
        inDegree[p[0]]++; // Count incoming edges for each course
    }

    queue<int> q;
    for (int i = 0; i < numCourses; i++) {
        if (inDegree[i] == 0) {
            q.push(i); // Add courses with no prerequisites to the queue
        }
    }

    vector<int> order;
    while (!q.empty()) {
        int course = q.front(); q.pop();
        order.push_back(course); // Add the course to the order
        for (int neighbor : adj[course]) {
            inDegree[neighbor]--; // Decrease in-degree
            if (inDegree[neighbor] == 0) {
                q.push(neighbor); // Add to queue if no more prerequisites
            }
        }
    }

    return order.size() == numCourses ? order : vector<int>(); // Return order if valid
```

```
}
```

Optimized (Java):

```
public int[] findOrder(int numCourses, int[][] prerequisites) {
    int[] inDegree = new int[numCourses];
    List<List<Integer>> adj = new ArrayList<>();
    for (int i = 0; i < numCourses; i++) {
        adj.add(new ArrayList<>());
    }

    for (int[] p : prerequisites) {
        adj.get(p[1]).add(p[0]);
        inDegree[p[0]]++; // Count incoming edges for each course
    }

    Queue<Integer> q = new LinkedList<>();
    for (int i = 0; i < numCourses; i++) {
        if (inDegree[i] == 0) {
            q.add(i); // Add courses with no prerequisites to the queue
        }
    }

    List<Integer> order = new ArrayList<>();
    while (!q.isEmpty()) {
        int course = q.poll();
        order.add(course); // Add the course to the order
        for (int neighbor : adj.get(course)) {
```

```
            inDegree[neighbor]--; // Decrease in-degree
            if (inDegree[neighbor] == 0) {
                q.add(neighbor); // Add to queue if no more prerequisites
            }
        }
    }

    if (order.size() == numCourses) {
        return order.stream().mapToInt(i -> i).toArray(); // Convert to int array
    }
    return new int[0]; // Return empty array if no valid order
}
```

BIT MANIPULATION

Count set bits in an integer

C++:

Here’s the C++ program with the comments removed:

```cpp
#include <bits/stdc++.h>
using namespace std;

unsigned int countSetBits(unsigned int n)
{
	unsigned int count = 0;
	while (n) {
		count += n & 1;
		n >>= 1;
	}
	return count;
}

int main()
{
	int i = 9;
	cout << countSetBits(i);
	return 0;
}
```

JAVA:

```
import java.io.*;

class countSetBits {
	static int countSetBits(int n)
	{
		int count = 0;
		while (n > 0) {
			count += n & 1;
			n >>= 1;
		}
		return count;
	}

	public static void main(String args[])
	{
		int i = 9;
		System.out.println(countSetBits(i));
	}
}
```

Power of Two

C++:

```
class Solution {
public:
    bool isPowerOfTwo(int n) {
        int x =1;
```

```
        while(x<=n){
            if(x==n)return true;
            if(x > INT_MAX /2)break;
            x = x<<1;
        }
        return false;
    }
};
```

JAVA:

```
public class Solution {
    public boolean isPowerOfTwo(int n) {
        int x = 1;
        while (x <= n) {
            if (x == n) return true;
            if (x > Integer.MAX_VALUE / 2) break;
            x = x << 1;
        }
        return false;
    }
}
```

Single Number

C++:

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
       int ans=0;
       for(auto i:nums){
        ans^=i;
```

```
        }
        return ans;
    }
};
```

JAVA:

```java
import java.util.List;

class Solution {
    public int singleNumber(List<Integer> nums) {
        int ans = 0;
        for (int num : nums) {
            ans ^= num;
        }
        return ans;
    }
}
```

Find position of the only set bit

C++:

Here’s the C++ program without comments:

````
```cpp
#include <bits/stdc++.h>
using namespace std;

int isPowerOfTwo(unsigned n) {
    return n && (!(n & (n - 1)));
}
````
````

```
int findPosition(unsigned n) {
    if (!isPowerOfTwo(n))
        return -1;

    unsigned i = 1, pos = 1;

    while (!(i & n)) {
        i = i << 1;
        ++pos;
    }

    return pos;
}

int main(void) {
    int n = 16;
    int pos = findPosition(n);
    (pos == -1)
        ? cout << "n = " << n << ", Invalid number" << endl
        : cout << "n = " << n << ", Position " << pos << endl;

    n = 12;
    pos = findPosition(n);
    (pos == -1)
        ? cout << "n = " << n << ", Invalid number" << endl
        : cout << "n = " << n << ", Position " << pos << endl;

    n = 128;
```

```
    pos = findPosition(n);
    (pos == -1)
        ? cout << "n = " << n << ", Invalid number" << endl
        : cout << "n = " << n << ", Position " << pos << endl;

    return 0;
}
```

JAVA:

Here’s the Java program without comments:

```java
class GFG {

    static boolean isPowerOfTwo(int n) {
        return (n > 0 && ((n & (n - 1)) == 0)) ? true : false;
    }

    static int findPosition(int n) {
        if (!isPowerOfTwo(n))
            return -1;

        int i = 1, pos = 1;

        while ((i & n) == 0) {
            i = i << 1;
            ++pos;
        }
```

```
        return pos;
    }

    public static void main(String[] args) {
        int n = 16;
        int pos = findPosition(n);
        if (pos == -1)
            System.out.println("n = " + n + ", Invalid number");
        else
            System.out.println("n = " + n + ", Position " + pos);

        n = 12;
        pos = findPosition(n);
        if (pos == -1)
            System.out.println("n = " + n + ", Invalid number");
        else
            System.out.println("n = " + n + ", Position " + pos);
        n = 128;
        pos = findPosition(n);
        if (pos == -1)
            System.out.println("n = " + n + ", Invalid number");
        else
            System.out.println("n = " + n + ", Position " + pos);
    }
}
```

Count number of bits to be flipped to convert A to B

C++:

```
#include <iostream>
```

```
using namespace std;

class Solution {
public:
    int countBitsToFlip(int A, int B) {
        int xorValue = A ^ B; // XOR to find differing bits
        int count = 0;

        // Count set bits in xorValue
        while (xorValue) {
            count += xorValue & 1; // Increment count if last bit is set
            xorValue >>= 1; // Right shift to check next bit
        }

        return count;
    }
};

int main() {
    Solution sol;
    int A = 10, B = 20;
    cout << "Number of bits to flip to convert " << A << " to " << B << ": " <<
sol.countBitsToFlip(A, B) << endl;
    return 0;
}
```

JAVA:

```
public class Solution {
    public int countBitsToFlip(int A, int B) {
        int xorValue = A ^ B; // XOR to find differing bits
```

```
        int count = 0;

        // Count set bits in xorValue
        while (xorValue != 0) {
            count += (xorValue & 1); // Increment count if last bit is set
            xorValue >>= 1; // Right shift to check next bit
        }

        return count;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        int A = 10, B = 20;
        System.out.println("Number of bits to flip to convert " + A + " to " + B + ": " +
sol.countBitsToFlip(A, B));
    }
}
```

Find XOR of numbers from L to R.

C++:

```
#include <vector>
#include <climits>
```

```
#include <iostream>

using namespace std;

int xorTill(int n){
    // Check if n is
    // congruent to 1 modulo 4
    if(n%4 == 1){
        return 1;
    }
    // Check if n is congruent
    // to 2 modulo 4
    else if(n%4 == 2){
        return n+1;
    }
    // Check if n is
    // congruent to 3 modulo 4
    else if(n%4 == 3){
        return 0;
    }
    // Return condition
    // when n is a multiple
    else{
        return n;
    }
}

// Function to compute XOR of
// numbers in the range [L, R]
```

```
int xorInRange(int L, int R){
   // Compute XOR of numbers from 1 to L-1
   // and 1 to R using the xorTill function
   int xorTillL = xorTill(L-1);
   int xorTillR = xorTill(R);
   // Compute XOR of the range from L to R
   return xorTillL ^ xorTillR;
}


int main() {
   int L = 3;
   int R = 19;
   int ans = xorInRange(L, R);
   cout << "XOR of of Numbers from " << L;
   cout << " to " << R << ": "<< ans << endl;
   return 0;
}
```

JAVA:

```
public class Main {
   // Function to compute XOR of all numbers
   // from 1 to n based on observed pattern
   public static int xorTill(int n) {
      // Check if n is
      // congruent to 1 modulo 4
```

```
    if (n % 4 == 1) {
        return 1;
    }
    // Check if n is congruent
    // to 2 modulo 4
    else if (n % 4 == 2) {
        return n + 1;
    }
    // Check if n is
    // congruent to 3 modulo 4
    else if (n % 4 == 3) {
        return 0;
    }
    // Return condition
    // when n is a multiple
    else {
        return n;
    }
}

// Function to compute XOR of numbers in the range [L, R]
public static int xorInRange(int L, int R) {
    // Compute XOR of numbers from 1 to L-1
    // and 1 to R using the xorTill function
    int xorTillL = xorTill(L - 1);
    int xorTillR = xorTill(R);
    // Compute XOR of the range from L to R
    return xorTillL ^ xorTillR;
}
```

```java
    public static void main(String[] args) {
        int L = 3;
        int R = 19;
        int ans = xorInRange(L, R);
        System.out.println("XOR of of Numbers from " + L + " to " + R + ": " + ans);
    }
}
```

Calculate square of a number without using *, / and pow()

C++:

```cpp
// Square of a number using bitwise operators
#include <bits/stdc++.h>
using namespace std;

int square(int n)
{
        // Base case
        if (n == 0)
                return 0;

        // Handle negative number
        if (n < 0)
                n = -n;

        // Get floor(n/2) using right shift
        int x = n >> 1;
```

```
        // If n is odd
        if (n & 1)
                return ((square(x) << 2) + (x << 2) + 1);
        else // If n is even
                return (square(x) << 2);
}

// Driver Code
int main()
{
        // Function calls
        for (int n = 1; n <= 5; n++)
                cout << "n = " << n << ", n^2 = " << square(n)
                        << endl;
        return 0;
}
```

JAVA:

```
// Square of a number using
// bitwise operators
class GFG {
        static int square(int n)
        {

                // Base case
                if (n == 0)
                        return 0;

                // Handle negative number
                if (n < 0)
```

```
                n = -n;

            // Get floor(n/2) using
            // right shift
            int x = n >> 1;

            // If n is odd
            ;
            if (n % 2 != 0)
                return ((square(x) << 2) + (x << 2) + 1);
            else // If n is even
                return (square(x) << 2);
        }

        // Driver code
        public static void main(String args[])
        {
            // Function calls
            for (int n = 1; n <= 5; n++)
                System.out.println("n = " + n
                                   + " n^2 = " + square(n));
        }
    }
```

Divide two integers without using multiplication, division and mod operator

C++:

```
// C++ implementation to Divide two
// integers using Bit Manipulation
```

```
#include <iostream>
#include <limits.h>
using namespace std;

// Function to divide a by b and
// return floor value it
long long divide(long long a, long long b) {

    // Handle overflow
    if(a == INT_MIN && b == -1)
        return INT_MAX;

    // The sign will be negative only if sign of
    // divisor and dividend are different
    int sign = ((a < 0) ^ (b < 0)) ? -1 : 1;

    // remove sign of operands
    a = abs(a);
    b = abs(b);

    // Initialize the quotient
    long long quotient = 0;

    // Iterate from most significant bit to
    // least significant bit
    for (int i = 31; i >= 0; --i) {

        // Check if (divisor << i) <= dividend
```

```
        if ((b << i) <= a) {
            a -= (b << i);
            quotient |= (1LL << i);
        }
    }

    return sign * quotient;
}

int main() {
    long long a = 43, b = -8;
    cout << divide(a, b);

    return 0;
}
```

JAVA:

```
// Java implementation to Divide two
// integers using Bit Manipulation

class GfG {

    // Function to divide a by b and
    // return floor value it
    static long divide(long a, long b) {

        // Handle overflow
        if (a == Integer.MIN_VALUE && b == -1)
            return Integer.MAX_VALUE;
```

```
    // The sign will be negative only if sign of
    // divisor and dividend are different
    int sign = ((a < 0) ^ (b < 0)) ? -1 : 1;

    // remove sign of operands
    a = Math.abs(a);
    b = Math.abs(b);

    // Initialize the quotient
    long quotient = 0;

    // Iterate from most significant bit to
    // least significant bit
    for (int i = 31; i >= 0; --i) {

        // Check if (divisor << i) <= dividend
        if ((b << i) <= a) {
            a -= (b << i);
            quotient |= (1L << i);
        }
    }

    return sign * quotient;
}

public static void main(String[] args) {
    long a = 43, b = -8;
    System.out.println(divide(a, b));
```

```
    }
}
```

Subsets

C++:

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        int n=nums.size();
        int Mask=1<<n;
        vector<vector<int>> powerSet(Mask);
        for (unsigned m=0; m<Mask; m++){
            powerSet[m].reserve(popcount(m));
            for(int i=0; i<n; i++){
                if (m& 1<<i)
                    powerSet[m].push_back(nums[i]);
            }
        }
        return powerSet;
    }
};
```

JAVA:

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

class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        int n = nums.length;
        int mask = 1 << n; // Total number of subsets
```

```java
    List<List<Integer>> powerSet = new ArrayList<>(mask);

    // Initialize the power set with empty lists
    for (int i = 0; i < mask; i++) {
        powerSet.add(new ArrayList<>());
    }

    for (int m = 0; m < mask; m++) {
        // Generate the subset corresponding to the current mask
        for (int i = 0; i < n; i++) {
            if ((m & (1 << i)) != 0) { // Check if the i-th bit is set
                powerSet.get(m).add(nums[i]);
            }
        }
    }

    return powerSet;
}

public static void main(String[] args) {
    Solution sol = new Solution();
    int[] nums = {1, 2, 3};
    List<List<Integer>> result = sol.subsets(nums);

    // Print the result
    for (List<Integer> subset : result) {
        System.out.println(subset);
    }
}
```

```
}
```

Count Primes

C++:

```
class Solution {
public:
    int countPrimes(int n) {
        vector<bool> prime(n+1,true);
        for(int i=2;i*i<=n;i++){
            if(prime[i]){
                for(long long j=(long long)i*i;j<=n;j+=i)
                prime[j]=false;
            }
        }
        int cnt=0;
        for(int i=2;i<n;i++){
            if(prime[i]==true) cnt++;
        }
        return cnt;
```

```
    }
};
```

JAVA:

```
class Solution {
    public int countPrimes(int n) {
        if (n <= 2) return 0; // There are no primes less than 2

        boolean[] prime = new boolean[n]; // Array to track prime status
        for (int i = 2; i < n; i++) {
            prime[i] = true; // Initialize all entries as true
        }

        for (int i = 2; i * i < n; i++) {
            if (prime[i]) {
                // Start marking multiples of i as false
                for (long j = (long) i * i; j < n; j += i) {
                    prime[(int) j] = false;
                }
            }
        }

        int count = 0;
        for (int i = 2; i < n; i++) {
            if (prime[i]) {
                count++; // Count the number of primes
            }
        }
```

```
        return count; // Return the count of primes
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        int n = 10; // Example input
        int result = sol.countPrimes(n);
        System.out.println("Number of primes less than " + n + ": " + result);
    }
}
```

BINARY SEARCH

Binary Search

C++:

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int low = 0;
        int high = nums.size() - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < target)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return -1;
    }
```

```
};
```

JAVA:

```java
class Solution {
    public int search(int[] nums, int target) {

        int left = 0, right = nums.length - 1;

        while (left <= right) {

            int mid = left + (right - left) / 2;

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

Search Insert Position

C++:

```cpp
class Solution {
public:
```

```cpp
    int searchInsert(vector<int>& nums, int target) {
        int left = 0;
        int right = nums.size() - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

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

        return left;
    }
};
```

JAVA:

```java
class Solution {
    public int searchInsert(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
```

```
            return mid;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return left;
  }
}
```

Sqrt(x) USING BS
C++:

```
class Solution {
public:
  int mySqrt(int x) {
    // For special cases when x is 0 or 1, return x.
    if (x == 0 || x == 1)
        return x;

    // Initialize the search range for the square root.
    int start = 1;
    int end = x;
    int mid = -1;

    // Perform binary search to find the square root of x.
    while (start <= end) {
        // Calculate the middle point using "start + (end - start) / 2" to avoid integer
overflow.
```

```
            mid = start + (end - start) / 2;

            // Convert mid to long to handle large values without overflow.
            long long square = static_cast<long long>(mid) * mid;

            // If the square of the middle value is greater than x, move the "end" to the left
(mid - 1).
            if (square > x)
                end = mid - 1;
            else if (square == x)
                // If the square of the middle value is equal to x, we found the square root.
                return mid;
            else
                // If the square of the middle value is less than x, move the "start" to the
right (mid + 1).
                start = mid + 1;
        }

        // The loop ends when "start" becomes greater than "end", and "end" is the
integer value of the square root.
        // However, since we might have been using integer division in the calculations,
        // we round down the value of "end" to the nearest integer to get the correct
square root.
        return static_cast<int>(std::round(end));
    }
};
```

JAVA:

```
class Solution {
    public int mySqrt(int x) {
        // For special cases when x is 0 or 1, return x.
```

```
    if (x == 0 || x == 1)
        return x;

    // Initialize the search range for the square root.
    int start = 1;
    int end = x;
    int mid = -1;

    // Perform binary search to find the square root of x.
    while (start <= end) {
        // Calculate the middle point using "start + (end - start) / 2" to avoid integer overflow.
        mid = start + (end - start) / 2;

        // If the square of the middle value is greater than x, move the "end" to the left (mid - 1).
        if ((long) mid * mid > (long) x)
            end = mid - 1;
        else if (mid * mid == x)
            // If the square of the middle value is equal to x, we found the square root.
            return mid;
        else
            // If the square of the middle value is less than x, move the "start" to the right (mid + 1).
            start = mid + 1;
    }

    // The loop ends when "start" becomes greater than "end", and "end" is the integer value of the square root.
    // However, since we might have been using integer division in the calculations,
```

```
        // we round down the value of "end" to the nearest integer to get the correct
square root.
        return Math.round(end);
    }
}
```

Search in Rotated Sorted Array
C++:

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0;
        int right = nums.size() - 1;

        while (left <= right) {
            int mid = (left + right) / 2;

            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] >= nums[left]) {
                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;
    }
};
```

JAVA:

```
class Solution {
    public int search(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;

        while (left <= right) {
            int mid = (left + right) / 2;

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

```
            }
        }
    }

    return -1;
  }
}
```

Find First and Last Position of Element in Sorted Array
C++:

```
class Solution {
public:
  vector<int> searchRange(vector<int>& nums, int target) {
    vector<int> result = {-1, -1};
    int left = binarySearch(nums, target, true);
    int right = binarySearch(nums, target, false);
    result[0] = left;
    result[1] = right;
    return result;
  }

  int binarySearch(vector<int>& nums, int target, bool isSearchingLeft) {
    int left = 0;
    int right = nums.size() - 1;
    int idx = -1;

    while (left <= right) {
      int mid = left + (right - left) / 2;
```

```
            if (nums[mid] > target) {
                right = mid - 1;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                idx = mid;
                if (isSearchingLeft) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }
        }

        return idx;
    }
};
```

JAVA:

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        int[] result = {-1, -1};
        int left = binarySearch(nums, target, true);
        int right = binarySearch(nums, target, false);
        result[0] = left;
        result[1] = right;
        return result;
    }

    private int binarySearch(int[] nums, int target, boolean isSearchingLeft) {
```

```
        int left = 0;
        int right = nums.length - 1;
        int idx = -1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

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

        return idx;
    }

}
```

Find Minimum in Rotated Sorted Array
C++:

```
class Solution
```

```
{
public:
    int findMin(vector<int> &nums)
    {
       // Initialize the result with the first element of the array
        int res = nums[0];

        // Initialize left and right pointers for binary search
        int l = 0;
        int r = nums.size() - 1;

         // Perform binary search
        while (l <= r)
        {
            // Check if the subarray is already sorted
            if (nums[l] < nums[r])
            {
                res = min(res, nums[l]);
                break;
            }

            // Compute the midpoint
            int mid = l + (r - l) / 2;
            res = min(res, nums[mid]);

            // Determine if the midpoint is in the left sorted portion
            if (nums[mid] >= nums[l])
            {
                l = mid + 1; // try to move closer to right sorted array
```

```
            }
            else
            {
                r = mid - 1;
            }
        }

        return res;// Return the minimum value found
    }
};
```

JAVA:

```
class Solution {
    public int findMin(int[] nums) {
        // Initialize the result with the first element of the array
        int res = nums[0];

        // Initialize left and right pointers for binary search
        int l = 0;
        int r = nums.length - 1;

        // Perform binary search
        while (l <= r) {
            // Check if the subarray is already sorted
            if (nums[l] < nums[r]) {
                res = Math.min(res, nums[l]);
                break;
            }

            // Compute the midpoint
```

```
            int mid = l + (r - l) / 2;
            res = Math.min(res, nums[mid]);

            // Determine if the midpoint is in the left sorted portion
            if (nums[mid] >= nums[l]) {
                l = mid + 1; // Move closer to the right sorted array
            } else {
                r = mid - 1;
            }
        }

        return res; // Return the minimum value found
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        int[] nums = {3, 4, 5, 1, 2}; // Example input
        int result = sol.findMin(nums);
        System.out.println("The minimum value in the rotated sorted array is: " + result);
    }
}
```

Find Peak Element
C++:

```
class Solution {
    public int findPeakElement(int[] arr) {
        int start = 0;
        int end = arr.length - 1;
```

```
        while(start < end){
            int mid = start + (end - start)/2;
            if(arr[mid] > arr[mid+1]){
                end = mid;
            }
            else{
                start = mid +1;
            }
        }
        return start;

    }
}
```

JAVA:

```java
class Solution {
    public int findPeakElement(int[] arr) {
        int start = 0;
        int end = arr.length - 1;

        while (start < end) {
            int mid = start + (end - start) / 2; // Calculate the mid index

            // Compare the middle element with its next element
            if (arr[mid] > arr[mid + 1]) {
                end = mid; // Move to the left side (including mid)
            } else {
                start = mid + 1; // Move to the right side (excluding mid)
            }
        }
```

```
        // When start == end, we've found the peak element
        return start;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        int[] arr = {1, 2, 3, 1}; // Example input
        int peakIndex = sol.findPeakElement(arr);
        System.out.println("Peak Element is at index: " + peakIndex + ", Value: " +
arr[peakIndex]);
    }
}
```

Search a 2D Matrix

C++:

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int top = 0;
        int bot = matrix.size() - 1;

        while (top <= bot) {
            int mid = (top + bot) / 2;

            if (matrix[mid][0] < target && matrix[mid][matrix[mid].size() - 1] > target) {
                break;
            } else if (matrix[mid][0] > target) {
```

```
            bot = mid - 1;
        } else {
            top = mid + 1;
        }
    }

    int row = (top + bot) / 2;

    int left = 0;
    int right = matrix[row].size() - 1;

    while (left <= right) {
        int mid = (left + right) / 2;

        if (matrix[row][mid] == target) {
            return true;
        } else if (matrix[row][mid] > target) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return false;
  }
};
```

JAVA:

```
class Solution {
  public boolean searchMatrix(int[][] matrix, int target) {
```

```
int top = 0;
int bot = matrix.length - 1;

while (top <= bot) {
    int mid = (top + bot) / 2;

    if (matrix[mid][0] < target && matrix[mid][matrix[mid].length - 1] > target) {
        break;
    } else if (matrix[mid][0] > target) {
        bot = mid - 1;
    } else {
        top = mid + 1;
    }
}

int row = (top + bot) / 2;

int left = 0;
int right = matrix[row].length - 1;

while (left <= right) {
    int mid = (left + right) / 2;

    if (matrix[row][mid] == target) {
        return true;
    } else if (matrix[row][mid] > target) {
        right = mid - 1;
    } else {
        left = mid + 1;
```

```
            }
        }

        return false;
    }
}
```

Find a Peak Element II

C++:

```
class Solution {
public:
    vector<int> findPeakGrid(vector<vector<int>>& mat) {
        int n = mat.size();
        int m = mat[0].size();
        int low = 0, high = m-1;
        while(low <= high){
            int maxRow = 0;
            int midCol = (low+high) >> 1;
            for(int row=0; row<n; row++){
                if(mat[row][midCol] > mat[maxRow][midCol]){
                    maxRow = row;
                }
            }
            int currElement =  mat[maxRow][midCol];
            int leftElement = midCol == 0 ? -1 : mat[maxRow][midCol-1];
            int rightElement = midCol == m-1 ? -1 : mat[maxRow][midCol+1];
            if(currElement > leftElement && currElement > rightElement){
```

```
            return {maxRow, midCol};
        }
        else if(currElement < leftElement){
            high = midCol - 1;
        }
        else{
            low = midCol + 1;
        }
    }
    return {-1, -1};
  }
};
```

JAVA:

```
class Solution {
  public int[] findPeakGrid(int[][] mat) {
    int n = mat.length;
    int m = mat[0].length;
    int low = 0, high = m - 1;

    while (low <= high) {
        int maxRow = 0;
        int midCol = (low + high) / 2;

        for (int row = 0; row < n; row++) {
            if (mat[row][midCol] > mat[maxRow][midCol]) {
                maxRow = row;
            }
        }
```

```
            int currElement = mat[maxRow][midCol];
            int leftElement = midCol == 0 ? Integer.MIN_VALUE : mat[maxRow][midCol - 1];
            int rightElement = midCol == m - 1 ? Integer.MIN_VALUE : mat[maxRow][midCol + 1];

            if (currElement > leftElement && currElement > rightElement) {
                return new int[] {maxRow, midCol};
            } else if (currElement < leftElement) {
                high = midCol - 1;
            } else {
                low = midCol + 1;
            }
        }

        return new int[] {-1, -1};
    }
}
```

Capacity To Ship Packages Within D Days

C++:

```
class Solution {
public:
    bool check(const vector<int> &v, int x, int d) {
        int cnt = 1, curr = 0;
        for(int i = 0; i < v.size(); ++i) {
            if(v[i] > x)    return 0;
            else if(curr + v[i] <= x) curr += v[i];
```

```
        else    ++cnt, curr = v[i];
    }
    return cnt <= d;
  }
  int shipWithinDays(vector<int>& weights, int days) {
    int lo = 1, hi = INT_MAX, id = -1;
    while(hi >= lo) {
        int mid = hi - (hi - lo) / 2;
        if(check(weights, mid, days))   {
            hi = mid - 1;
            id = mid;
        }
        else    lo = mid + 1;
    }
    return id;
  }
};
```

JAVA:

```
class Solution {
  public boolean check(int[] v, int x, int d) {
    int cnt = 1, curr = 0;
    for (int i = 0; i < v.length; ++i) {
        if (v[i] > x) return false;
        else if (curr + v[i] <= x) curr += v[i];
        else {
            ++cnt;
            curr = v[i];
        }
    }
```

```
        return cnt <= d;
    }

    public int shipWithinDays(int[] weights, int days) {
        int lo = 1, hi = Integer.MAX_VALUE, id = -1;
        while (hi >= lo) {
            int mid = hi - (hi - lo) / 2;
            if (check(weights, mid, days)) {
                hi = mid - 1;
                id = mid;
            } else {
                lo = mid + 1;
            }
        }
        return id;
    }
}
```

Painter's Partition Problem

C++:

```
// CPP program for The painter's partition problem
#include <climits>
#include <iostream>
using namespace std;

// function to calculate sum between two indices
// in array
```

```
int sum(int arr[], int from, int to)
{
	int total = 0;
	for (int i = from; i <= to; i++)
		total += arr[i];
	return total;
}

// for n boards and k partitions
int partition(int arr[], int n, int k)
{
	// base cases
	if (k == 1) // one partition
		return sum(arr, 0, n - 1);
	if (n == 1) // one board
		return arr[0];

	int best = INT_MAX;

	// find minimum of all possible maximum
	// k-1 partitions to the left of arr[i],
	// with i elements, put k-1 th divider
	// between arr[i-1] & arr[i] to get k-th
	// partition
	for (int i = 1; i <= n; i++)
		best = min(best, max(partition(arr, i, k - 1),
								sum(arr, i, n - 1)));

	return best;
```

```
}

int main()
{
	int arr[] = { 10, 20, 60, 50, 30, 40 };
	int n = sizeof(arr) / sizeof(arr[0]);
	int k = 3;
	cout << partition(arr, n, k) << endl;

	return 0;
}
```

JAVA:

```
// Java Program for The painter's partition problem
import java.io.*;
import java.util.*;

class GFG {
	// function to calculate sum between two indices
	// in array
	static int sum(int arr[], int from, int to)
	{
		int total = 0;
		for (int i = from; i <= to; i++)
			total += arr[i];
		return total;
	}

	// for n boards and k partitions
	static int partition(int arr[], int n, int k)
```

```
{
	// base cases
	if (k == 1) // one partition
		return sum(arr, 0, n - 1);
	if (n == 1) // one board
		return arr[0];

	int best = Integer.MAX_VALUE;

	// find minimum of all possible maximum
	// k-1 partitions to the left of arr[i],
	// with i elements, put k-1 th divider
	// between arr[i-1] & arr[i] to get k-th
	// partition
	for (int i = 1; i <= n; i++)
		best = Math.min(
			best, Math.max(partition(arr, i, k - 1),
						sum(arr, i, n - 1)));

	return best;
}

// Driver code
public static void main(String args[])
{
	int arr[] = { 10, 20, 60, 50, 30, 40 };

	// Calculate size of array.
	int n = arr.length;
```

```java
            int k = 3;
            System.out.println(partition(arr, n, k));
        }
}
```

Minimize Max Distance to Gas Station

C++:

```cpp
#include <bits/stdc++.h>
using namespace std;

int numberOfGasStationsRequired(long double dist, vector<int> &arr) {
    int n = arr.size(); // size of the array
    int cnt = 0;
    for (int i = 1; i < n; i++) {
        int numberInBetween = ((arr[i] - arr[i - 1]) / dist);
        if ((arr[i] - arr[i - 1]) == (dist * numberInBetween)) {
            numberInBetween--;
        }
        cnt += numberInBetween;
    }
    return cnt;
}

long double minimiseMaxDistance(vector<int> &arr, int k) {
    int n = arr.size(); // size of the array
    long double low = 0;
    long double high = 0;
```

```
    //Find the maximum distance:
    for (int i = 0; i < n - 1; i++) {
        high = max(high, (long double)(arr[i + 1] - arr[i]));
    }

    //Apply Binary search:
    long double diff = 1e-6 ;
    while (high - low > diff) {
        long double mid = (low + high) / (2.0);
        int cnt = numberOfGasStationsRequired(mid, arr);
        if (cnt > k) {
            low = mid;
        }
        else {
            high = mid;
        }
    }
    return high;
}

int main()
{
    vector<int> arr = {1, 2, 3, 4, 5};
    int k = 4;
    long double ans = minimiseMaxDistance(arr, k);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

JAVA:

```java
import java.util.*;

public class tUf {
    public static int numberOfGasStationsRequired(double dist, int[] arr) {
        int n = arr.length; // size of the array
        int cnt = 0;
        for (int i = 1; i < n; i++) {
            int numberInBetween = (int)((arr[i] - arr[i - 1]) / dist);
            if ((arr[i] - arr[i - 1]) == (dist * numberInBetween)) {
                numberInBetween--;
            }
            cnt += numberInBetween;
        }
        return cnt;
    }

    public static double minimiseMaxDistance(int[] arr, int k) {
        int n = arr.length; // size of the array
        double low = 0;
        double high = 0;

        //Find the maximum distance:
        for (int i = 0; i < n - 1; i++) {
            high = Math.max(high, (double)(arr[i + 1] - arr[i]));
        }

        //Apply Binary search:
        double diff = 1e-6 ;
```

```
        while (high - low > diff) {
            double mid = (low + high) / (2.0);
            int cnt = numberOfGasStationsRequired(mid, arr);
            if (cnt > k) {
                low = mid;
            } else {
                high = mid;
            }
        }
        return high;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int k = 4;
        double ans = minimiseMaxDistance(arr, k);
        System.out.println("The answer is: " + ans);
    }
}
```

Allocate Books
C++:

```
#include <bits/stdc++.h>
using namespace std;
```

```
int countStudents(vector<int> &arr, int pages) {
    int n = arr.size(); //size of array.
    int students = 1;
    long long pagesStudent = 0;
    for (int i = 0; i < n; i++) {
        if (pagesStudent + arr[i] <= pages) {
            //add pages to current student
            pagesStudent += arr[i];
        }
        else {
            //add pages to next student
            students++;
            pagesStudent = arr[i];
        }
    }
    return students;
}

int findPages(vector<int>& arr, int n, int m) {
    //book allocation impossible:
    if (m > n) return -1;

    int low = *max_element(arr.begin(), arr.end());
    int high = accumulate(arr.begin(), arr.end(), 0);
    while (low <= high) {
        int mid = (low + high) / 2;
        int students = countStudents(arr, mid);
        if (students > m) {
            low = mid + 1;
```

```
        }
        else {
            high = mid - 1;
        }
    }
    return low;
}

int main()
{
    vector<int> arr = {25, 46, 28, 49, 24};
    int n = 5;
    int m = 4;
    int ans = findPages(arr, n, m);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

JAVA:

```
import java.util.*;

public class Main {
    public static int countStudents(ArrayList<Integer> arr, int pages) {
        int n = arr.size(); // size of array
        int students = 1;
        long pagesStudent = 0;
```

```
    for (int i = 0; i < n; i++) {
        if (pagesStudent + arr.get(i) <= pages) {
            // add pages to current student
            pagesStudent += arr.get(i);
        } else {
            // add pages to next student
            students++;
            pagesStudent = arr.get(i);
        }
    }
    return students;
}

public static int findPages(ArrayList<Integer> arr, int n, int m) {
    // book allocation impossible
    if (m > n)
        return -1;

    int low = Collections.max(arr);
    int high = arr.stream().mapToInt(Integer::intValue).sum();
    while (low <= high) {
        int mid = (low + high) / 2;
        int students = countStudents(arr, mid);
        if (students > m) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
```

```java
        return low;
    }

    public static void main(String[] args) {
        ArrayList<Integer> arr = new ArrayList<>(Arrays.asList(25, 46, 28, 49, 24));
        int n = 5;
        int m = 4;
        int ans = findPages(arr, n, m);
        System.out.println("The answer is: " + ans);
    }
}
```

Aggressive Cows

C++:

```cpp
#include <bits/stdc++.h>
using namespace std;

bool canWePlace(vector<int> &stalls, int dist, int cows) {
    int n = stalls.size(); //size of array
    int cntCows = 1; //no. of cows placed
    int last = stalls[0]; //position of last placed cow.
    for (int i = 1; i < n; i++) {
        if (stalls[i] - last >= dist) {
            cntCows++; //place next cow.
            last = stalls[i]; //update the last location.
        }
```

```
        if (cntCows >= cows) return true;
    }
    return false;
}
int aggressiveCows(vector<int> &stalls, int k) {
    int n = stalls.size(); //size of array
    //sort the stalls[]:
    sort(stalls.begin(), stalls.end());

    int low = 1, high = stalls[n - 1] - stalls[0];
    //apply binary search:
    while (low <= high) {
        int mid = (low + high) / 2;
        if (canWePlace(stalls, mid, k) == true) {
            low = mid + 1;
        }
        else high = mid - 1;
    }
    return high;
}

int main()
{
    vector<int> stalls = {0, 3, 4, 7, 10, 9};
    int k = 4;
    int ans = aggressiveCows(stalls, k);
    cout << "The maximum possible minimum distance is: " << ans << "\n";
    return 0;
}
```

JAVA:

```
import java.util.*;

public class tUf {
    public static boolean canWePlace(int[] stalls, int dist, int cows) {
        int n = stalls.length; //size of array
        int cntCows = 1; //no. of cows placed
        int last = stalls[0]; //position of last placed cow.
        for (int i = 1; i < n; i++) {
            if (stalls[i] - last >= dist) {
                cntCows++; //place next cow.
                last = stalls[i]; //update the last location.
            }
            if (cntCows >= cows) return true;
        }
        return false;
    }
    public static int aggressiveCows(int[] stalls, int k) {
        int n = stalls.length; //size of array
        //sort the stalls[]:
        Arrays.sort(stalls);

        int low = 1, high = stalls[n - 1] - stalls[0];
        //apply binary search:
        while (low <= high) {
            int mid = (low + high) / 2;
            if (canWePlace(stalls, mid, k) == true) {
                low = mid + 1;
            } else high = mid - 1;
```

```java
        }
        return high;
    }
    public static void main(String[] args) {
        int[] stalls = {0, 3, 4, 7, 10, 9};
        int k = 4;
        int ans = aggressiveCows(stalls, k);
        System.out.println("The maximum possible minimum distance is: " + ans);
    }
}
```