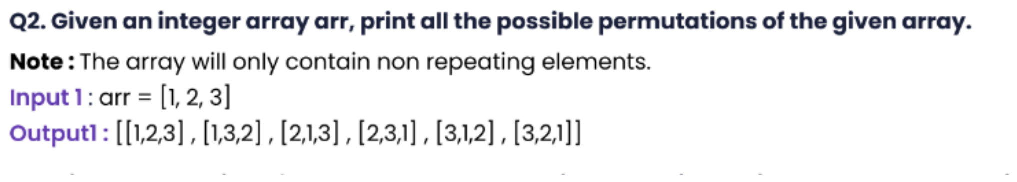
**Q1. Given an integer array arr and an integer k, return true if it is possible to divide the vector into k non-empty subsets with equal sum.**

**Input:** arr = [1,3,2,2] k = 2

**Output:** true

Ans:

```
public class PartitionArrayIntoKSubsets {
    public static boolean canPartitionKSubsets(int[] arr, int k) {
        int sum = 0;
        for (int num : arr) {
            sum += num;
        }

        if (k <= 0 || sum % k != 0) {
            return false; // Invalid input
        }

        int targetSum = sum / k;
        boolean[] visited = new boolean[arr.length];

        return backtrack(arr, k, 0, 0, targetSum, visited);
    }

    private static boolean backtrack(int[] arr, int k, int currentSum, int startIndex, int targetSum,
boolean[] visited) {
        if (k == 0) {
            return true; // All subsets found
        }

        if (currentSum == targetSum) {
            return backtrack(arr, k - 1, 0, 0, targetSum, visited);
        }

        for (int i = startIndex; i < arr.length; i++) {
            if (!visited[i] && currentSum + arr[i] <= targetSum) {
                visited[i] = true;
                if (backtrack(arr, k, currentSum + arr[i], i + 1, targetSum, visited)) {
                    return true;
                }
                visited[i] = false;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        // Example usage
        int[] arr = {4, 3, 2, 3, 5, 2, 1};
        int k = 4;

        boolean result = canPartitionKSubsets(arr, k);
        System.out.println("Can partition into " + k + " subsets: " + result);
    }
}
```



**Q2. Given an integer array arr, print all the possible permutations of the given array.**

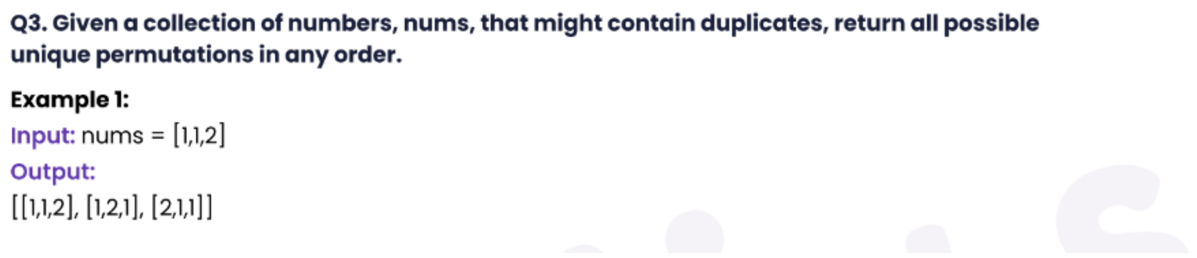
**Note :** The array will only contain non repeating elements.

**Input 1 :** arr = [1, 2, 3]

**Output1 :** [[1,2,3] , [1,3,2] , [2,1,3] , [2,3,1] , [3,1,2] , [3,2,1]]

Ans:

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

public class Permutations {
    public static List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        List<Integer> currentPermutation = new ArrayList<>();
        boolean[] used = new boolean[nums.length];

        backtrack(nums, result, currentPermutation, used);

        return result;
    }

    private static void backtrack(int[] nums, List<List<Integer>> result, List<Integer>
currentPermutation, boolean[] used) {
        if (currentPermutation.size() == nums.length) {
            result.add(new ArrayList<>(currentPermutation));
            return;
        }
        for (int i = 0; i < nums.length; i++) {
            if (!used[i]) {
                // Choose
                used[i] = true;
                currentPermutation.add(nums[i]);

                // Explore
                backtrack(nums, result, currentPermutation, used);

                // Unchoose
                used[i] = false;
                currentPermutation.remove(currentPermutation.size() - 1);
            }
        }
    }

    public static void main(String[] args) {
        // Example usage
        int[] nums = {1, 2, 3};
        List<List<Integer>> permutations = permute(nums);

        System.out.println("All permutations:");
        for (List<Integer> permutation : permutations) {
            System.out.println(permutation);
        }
    }
}
```



**Q3. Given a collection of numbers, nums, that might contain duplicates, return all possible unique permutations in any order.**

**Example 1:**

**Input:** nums = [1,1,2]

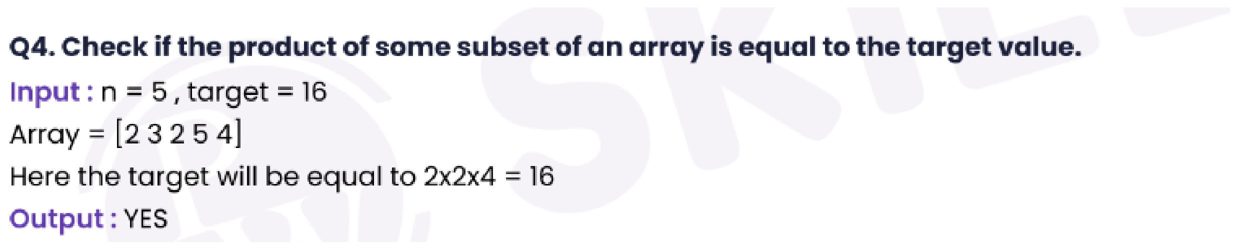
**Output:**

[[1,1,2], [1,2,1], [2,1,1]]

Ans:

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

```
public class UniquePermutations {
    public static List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        List<Integer> currentPermutation = new ArrayList<>();
        boolean[] used = new boolean[nums.length];

        Arrays.sort(nums); // Sort the array to handle duplicates

        backtrack(nums, result, currentPermutation, used);

        return result;
    }

    private static void backtrack(int[] nums, List<List<Integer>> result, List<Integer>
currentPermutation, boolean[] used) {
        if (currentPermutation.size() == nums.length) {
            result.add(new ArrayList<>(currentPermutation));
            return;
        }

        for (int i = 0; i < nums.length; i++) {
            // Skip duplicates at the same level of recursion
            if (used[i] || (i > 0 && nums[i] == nums[i - 1] && !used[i - 1])) {
                continue;
            }

            // Choose
            used[i] = true;
            currentPermutation.add(nums[i]);

            // Explore
            backtrack(nums, result, currentPermutation, used);

            // Unchoose
            used[i] = false;
            currentPermutation.remove(currentPermutation.size() - 1);
        }
    }

    public static void main(String[] args) {
        // Example usage
        int[] nums = {1, 1, 2};
        List<List<Integer>> permutations = permuteUnique(nums);
        System.out.println("All unique permutations:");
        for (List<Integer> permutation : permutations) {
            System.out.println(permutation);
        }
    }
}
```



**Q4. Check if the product of some subset of an array is equal to the target value.**

**Input :** n = 5 , target = 16

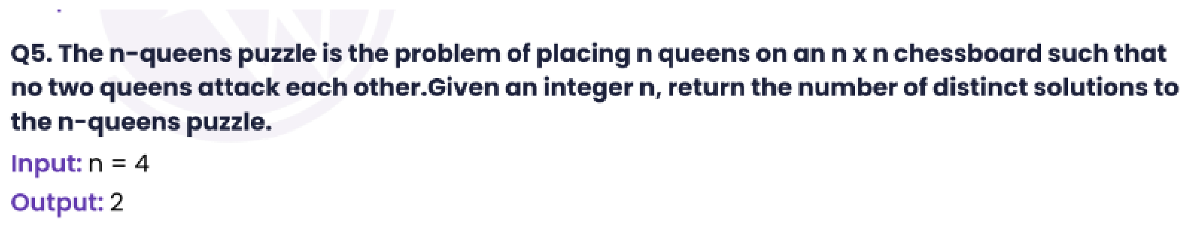
Array = [2 3 2 5 4]

Here the target will be equal to 2x2x4 = 16

**Output :** YES

Ans:

```
public class SubsetProduct {
    public static boolean isSubsetProduct(int[] nums, int target) {
        return backtrack(nums, target, 1, 0);
    }

    private static boolean backtrack(int[] nums, int target, long currentProduct, int index) {
        if (currentProduct == target) {
            return true; // Subset found with the target product
        }

        for (int i = index; i < nums.length; i++) {
            // Choose
            currentProduct *= nums[i];

            // Explore
            if (backtrack(nums, target, currentProduct, i + 1)) {
                return true; // Found a subset with the target product
            }

            // Unchoose
            currentProduct /= nums[i];
        }

        return false;
    }
    public static void main(String[] args) {
        // Example usage
        int[] nums = {2, 3, 5};
        int target = 30;

        boolean result = isSubsetProduct(nums, target);
        System.out.println("Subset with product equal to " + target + ": " + result);
    }
}
```



**Q5. The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.Given an integer n, return the number of distinct solutions to the n-queens puzzle.**

**Input:** n = 4

**Output:** 2

Ans:

```
public class NQueens {
    public static int totalNQueens(int n) {
        int[] placement = new int[n];
        int[] result = new int[1];
        solveNQueens(n, 0, placement, result);
        return result[0];
    }

    private static void solveNQueens(int n, int row, int[] placement, int[] result) {
        if (row == n) {
            // All queens are placed successfully, increment the result count
            result[0]++;
            return;
        }

        for (int col = 0; col < n; col++) {
            if (isValid(row, col, placement)) {
                placement[row] = col;
                solveNQueens(n, row + 1, placement, result);
            }
        }
    }
    private static boolean isValid(int row, int col, int[] placement) {
        for (int i = 0; i < row; i++) {
            // Check if there is a queen in the same column or diagonals
            if (placement[i] == col || Math.abs(placement[i] - col) == Math.abs(i - row)) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        // Example usage
        int n = 4;
        int result = totalNQueens(n);
        System.out.println("Total distinct solutions for " + n + "-Queens puzzle: " + result);
    }
}
```