**Assignment Backtracking**

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:

Output:

Explanation :

arr = [1,3,2,2] k = 2

true

1 + 3 and 2+2 are two subsets with equal sum.

import java.util.Arrays;

public class PartitionKEqualSumSubsets {

public boolean canPartitionKSubsets(int[] arr, int k) {

int sum = Arrays.stream(arr).sum();

if (k <= 0 || sum % k != 0) return false;

int target = sum / k;

boolean[] visited = new boolean[arr.length];

return canPartition(0, arr, visited, k, 0, target);

}

private boolean canPartition(int startIndex, int[] arr, boolean[] visited, int k, int currentSum, int target) {

if (k == 1) return true; // If only one subset left, the rest of the array must sum to the target.

if (currentSum == target) return canPartition(0, arr, visited, k - 1, 0, target);

for (int i = startIndex; i < arr.length; i++) {

if (!visited[i]) {

visited[i] = true;

if (canPartition(i + 1, arr, visited, k, currentSum + arr[i], target)) {

return true;

}

visited[i] = false; // backtrack

}

}

return false;

}

public static void main(String[] args) {

PartitionKEqualSumSubsets solution = new PartitionKEqualSumSubsets();

int[] arr = {1, 3, 2, 2};

int k = 2;

System.out.println(solution.canPartitionKSubsets(arr, k)); // Output: true

}

}

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

Note : The array will only contain non repeating elements.

arr = [1, 2, 3]

[[1,2,3] , [1,3,2] , [2,1,3] , [2,3,1] , [3,1,2] , [3,2,1]]

Input 1

Output1

import java.util.ArrayList;

import java.util.List;

public class Permutations {

public List<List<Integer>> permute(int[] arr) {

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

List<Integer> currentPermutation = new ArrayList<>();

boolean[] used = new boolean[arr.length];

backtrack(arr, result, currentPermutation, used);

return result;

}

private void backtrack(int[] arr, List<List<Integer>> result, List<Integer> currentPermutation, boolean[] used) {

if (currentPermutation.size() == arr.length) {

result.add(new ArrayList<>(currentPermutation));

return;

}

for (int i = 0; i < arr.length; i++) {

if (used[i]) continue;

used[i] = true;

currentPermutation.add(arr[i]);

backtrack(arr, result, currentPermutation, used);

used[i] = false;

currentPermutation.remove(currentPermutation.size() - 1);

}

}

public static void main(String[] args) {

Permutations solution = new Permutations();

int[] arr = {1, 2, 3};

List<List<Integer>> permutations = solution.permute(arr);

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]]

Example 2:

Input

nums = [1,2,3]

output:

[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

public class UniquePermutations {

public List<List<Integer>> permuteUnique(int[] nums) {

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

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

boolean[] used = new boolean[nums.length];

backtrack(nums, result, new ArrayList<>(), used);

return result;

}

private 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 the same element in the current position to avoid duplicates

if (used[i] || (i > 0 && nums[i] == nums[i - 1] && !used[i - 1])) {

continue;

}

used[i] = true;

currentPermutation.add(nums[i]);

backtrack(nums, result, currentPermutation, used);

used[i] = false;

currentPermutation.remove(currentPermutation.size() - 1);

}

}

public static void main(String[] args) {

UniquePermutations solution = new UniquePermutations();

int[] nums1 = {1, 1, 2};

System.out.println(solution.permuteUnique(nums1));

int[] nums2 = {1, 2, 3};

System.out.println(solution.permuteUnique(nums2));

}

}

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

Input :

n = 5 , target = 16

n = 5 , target = 16

Array = [2 3 2 5 4]

Here the target will be equal to 2x2x4 = 16

Output :YES

public class SubsetProduct {

public boolean isProductPossible(int[] arr, int n, int target) {

return isProductPossibleHelper(arr, n, target, 1, 0);

}

private boolean isProductPossibleHelper(int[] arr, int n, int target, int currentProduct, int index) {

// Base case: if the current product equals the target

if (currentProduct == target) {

return true;

}

// Base case: if we've reached the end of the array or if the product exceeds the target

if (index == n || currentProduct > target) {

return false;

}

// Recursive case: include the current element in the product or skip it

// 1. Include the current element in the product

if (isProductPossibleHelper(arr, n, target, currentProduct \* arr[index], index + 1)) {

return true;

}

// 2. Skip the current element

if (isProductPossibleHelper(arr, n, target, currentProduct, index + 1)) {

return true;

}

return false;

}

public static void main(String[] args) {

SubsetProduct solution = new SubsetProduct();

int[] arr = {2, 3, 2, 5, 4};

int target = 16;

if (solution.isProductPossible(arr, arr.length, target)) {

System.out.println("YES");

} else {

System.out.println("NO");

}

}

}

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:4

Output:2

Explanation:

There are two distinct solutions to the 4-queens puzzle as

shown.

Input:1

Output:1

public class NQueens {

public int totalNQueens(int n) {

int[] result = {0};

boolean[] columns = new boolean[n]; // columns being attacked

boolean[] diagonals1 = new boolean[2 \* n - 1]; // \ diagonals being attacked

boolean[] diagonals2 = new boolean[2 \* n - 1]; // / diagonals being attacked

solveNQueens(0, n, columns, diagonals1, diagonals2, result);

return result[0];

}

private void solveNQueens(int row, int n, boolean[] columns, boolean[] diagonals1, boolean[] diagonals2, int[] result) {

if (row == n) {

result[0]++;

return;

}

for (int col = 0; col < n; col++) {

int diag1 = row - col + n - 1;

int diag2 = row + col;

if (columns[col] || diagonals1[diag1] || diagonals2[diag2]) {

continue;

}

// Place the queen

columns[col] = true;

diagonals1[diag1] = true;

diagonals2[diag2] = true;

// Move on to the next row

solveNQueens(row + 1, n, columns, diagonals1, diagonals2, result);

// Backtrack and remove the queen

columns[col] = false;

diagonals1[diag1] = false;

diagonals2[diag2] = false;

}

}

public static void main(String[] args) {

NQueens solution = new NQueens();

int n1 = 4;

System.out.println(solution.totalNQueens(n1)); // Output: 2

int n2 = 1;

System.out.println(solution.totalNQueens(n2)); // Output: 1

}

}