Easy: Given the list of array return array in which each element is the product of other element except ith element (try to do it without division operation)

 input: [1,2,3,4]

output:[24,12,8,6]

public class ProductExceptSelf {

public static int[] productExceptSelf(int[] nums) {

int n = nums.length;

int[] result = new int[n];

// Prefix products from left to right

int prefix = 1;

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

result[i] = prefix;

prefix \*= nums[i];

}

// Suffix products from right to left

int suffix = 1;

for (int i = n - 1; i >= 0; i--) {

result[i] \*= suffix;

suffix \*= nums[i];

}

return result;

}

public static void main(String[] args) {

int[] nums = {1, 2, 3, 4};

int[] product = productExceptSelf(nums);

System.out.println(Arrays.toString(product)); // Output: [24, 12, 8, 6]

}

}

Medium: Given an array list return all possible permutations Input: nums = [1,4,3]  Output: [[1,4,3],[1,3,4],[4,1,3],[4,3,1],[3,1,4],[3,4,1]]

from collections import deque

def permute(nums):

"""

This function generates all possible permutations of a list.

Args:

nums: A list of numbers.

Returns:

A list of lists, where each sublist is a permutation of the original list.

"""

if len(nums) == 1:

return [nums]

permutations = []

for i, num in enumerate(nums):

remaining\_nums = nums[:i] + nums[i+1:]

for sub\_permutation in permute(remaining\_nums):

permutations.append([num] + sub\_permutation)

return permutations

# Example usage

nums = [1, 4, 3]

permutations = permute(nums)

print(permutations) # Output: [[1, 4, 3], [1, 3, 4], [4, 1, 3], [4, 3, 1], [3, 1, 4], [3, 4, 1]]

Hard: Return all the clubbed words  Input: words =["mat","mate","matbellmates","bell","bellmatesbell","butterribbon","butter","ribbon"] Output: ["matbellmates","bellmatesbell","butterribbon"]

def find\_clubbed\_words(words):

"""

This function finds all words that can be formed by concatenating other words in the given list.

Args:

words: A list of strings.

Returns:

A list of strings that can be formed by clubbing other words.

"""

# Build a trie to efficiently store and search for prefixes of words.

trie = {}

for word in words:

node = trie

for char in word:

if char not in node:

node[char] = {}

node = node[char]

node["#"] = True # Mark the end of a word.

# Find all clubbed words by traversing the trie and checking for ending markers at intermediate nodes.

clubbed\_words = []

def dfs(node, current\_word):

if "#" in node:

clubbed\_words.append(current\_word)

for char, child in node.items():

if char != "#":

dfs(child, current\_word + char)

dfs(trie, "")

return clubbed\_words

# Example usage

words = ["mat", "mate", "matbellmates", "bell", "bellmatesbell", "butterribbon", "butter", "ribbon"]

clubbed\_words = find\_clubbed\_words(words)

print(clubbed\_words) # Output: ["matbellmates", "bellmatesbell", "butterribbon"]