<https://leetcode.com/problems/partition-to-k-equal-sum-subsets/description/>

Given an integer array nums and an integer k, return true if it is possible to divide this array into k non-empty subsets whose sums are all equal.

**Example 1:**

**Input:** nums = [4,3,2,3,5,2,1], k = 4

**Output:** true

**Explanation:** It is possible to divide it into 4 subsets (5), (1, 4), (2,3), (2,3) with equal sums.

**Example 2:**

**Input:** nums = [1,2,3,4], k = 3

**Output:** false

**Constraints:**

1 <= k <= nums.length <= 16

1 <= nums[i] <= 104

The frequency of each element is in the range [1, 4].

**Attempt 1: 2025-08-27**

**Solution 1: Backtracking + Sorting (10 min)**

class Solution {

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

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

        if(sum % k != 0) {

            return false;

        }

        int target = sum / k;

        Arrays.sort(nums);

        for(int i = 0; i < nums.length / 2; i++) {

            int tmp = nums[i];

            nums[i] = nums[nums.length - 1 - i];

            nums[nums.length - 1 - i] = tmp;

        }

        return helper(nums, target, new int[k], 0);

    }

    private boolean helper(int[] nums, int target, int[] sides, int index) {

        if(index == nums.length) {

            return true;

        }

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

            if(sides[i] + nums[index] > target) {

                continue;

            }

            if(i > 0 && sides[i] == sides[i - 1]) {

                continue;

            }

            sides[i] += nums[index];

            if(helper(nums, target, sides, index + 1)) {

                return true;

            }

            sides[i] -= nums[index];

        }

        return false;

    }

}

Time Complexity: O(k^n) in worst case, but heavily pruned in practice

Space Complexity: O(n) for recursion stack

**Refer to Deepseek**

**Approach**

This problem is similar to the "Matchsticks to Square" problem but generalized to K subsets. We use backtracking with optimization:

**Calculate Target Sum**: Total sum must be divisible by K

**Sort in Descending Order**: Try larger elements first for better pruning

**Backtracking**: Try placing each element in one of the K subsets

**Pruning**: Skip duplicate states and impossible paths

**Solution Code**

import java.util.Arrays;

class Solution {

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

if (nums == null || nums.length < k) {

return false;

}

int total = Arrays.stream(nums).sum();

if (total % k != 0) {

return false;

}

int target = total / k;

Arrays.sort(nums);

reverse(nums);

// If any element is larger than target, impossible

if (nums[0] > target) {

return false;

}

int[] subsets = new int[k];

return backtrack(nums, subsets, 0, target);

}

private boolean backtrack(int[] nums, int[] subsets, int index, int target) {

if (index == nums.length) {

return true;

}

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

// Skip if adding this number would exceed target

if (subsets[i] + nums[index] > target) {

continue;

}

// Skip duplicate states (if previous subsets have same sum)

int j = i;

while (j-- > 0) {

if (subsets[i] == subsets[j]) {

break;

}

}

if (j >= 0) {

continue;

}

subsets[i] += nums[index];

if (backtrack(nums, subsets, index + 1, target)) {

return true;

}

subsets[i] -= nums[index];

// Early termination: if we can't place the number in any subset

if (subsets[i] == 0) {

break;

}

}

return false;

}

private void reverse(int[] arr) {

int left = 0, right = arr.length - 1;

while (left < right) {

int temp = arr[left];

arr[left] = arr[right];

arr[right] = temp;

left++;

right--;

}

}

}

**Optimized Solution with Better Pruning**

import java.util.Arrays;

class Solution {

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

int total = Arrays.stream(nums).sum();

if (total % k != 0) return false;

int target = total / k;

Arrays.sort(nums);

// Reverse to descending order

for (int i = 0; i < nums.length / 2; i++) {

int temp = nums[i];

nums[i] = nums[nums.length - 1 - i];

nums[nums.length - 1 - i] = temp;

}

if (nums[0] > target) return false;

return dfs(nums, new int[k], 0, target);

}

private boolean dfs(int[] nums, int[] subsets, int index, int target) {

if (index == nums.length) {

return true;

}

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

if (subsets[i] + nums[index] > target) continue;

// Skip duplicate combinations

if (i > 0 && subsets[i] == subsets[i - 1]) continue;

subsets[i] += nums[index];

if (dfs(nums, subsets, index + 1, target)) return true;

subsets[i] -= nums[index];

// Early termination: if we can't place the number in any subset

if (subsets[i] == 0) break;

}

return false;

}

}

**Explanation**

**Initial Checks**:

Check if array can be partitioned into K subsets

Calculate target sum (total sum / K)

Sort in descending order for better pruning

**Backtracking**:

Try placing each number in one of the K subsets

Skip subsets where adding the number would exceed target

Skip duplicate subset sums to avoid redundant calculations

Early termination if number can't be placed in any subset

**Optimizations**:

Descending sort: try larger numbers first for better pruning

Duplicate skipping: avoid trying the same subset sum multiple times

Early termination: if a number can't be placed in any empty subset, return false

**Time Complexity**: O(K^N) in worst case, but heavily pruned in practice

**Space Complexity**: O(N) for recursion stack

This approach efficiently explores all possible partitions while pruning impossible paths early, making it feasible for the problem constraints.

**Refer to**

[L473.Matchsticks to Square (Ref.L2397)](note://WEB803bb8fa17c70b8a004a7117d62b41b1)

[L2397.Maximum Rows Covered by Columns (Ref.L473)](note://WEB050bb59c3f216a8c91af09746dbf6efe)