# 题目

给定一个整数数组 nums 和一个正整数 k，找出是否有可能把这个数组分成 k 个非空子集，其总和都相等。

示例 1：

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

输出： True

说明： 有可能将其分成 4 个子集（5），（1,4），（2,3），（2,3）等于总和。

示例 2:

输入: nums = [1,2,3,4], k = 3

输出: false

提示：

1 <= k <= len(nums) <= 16

0 < nums[i] < 10000

每个元素的频率在 [1,4] 范围内

# 分析

## 方法一：状态压缩+动态规划

要解决将整数数组分成k个总和相等的非空子集的问题，我们可以利用状态压缩+动态规划的方法。这种方法通过二进制表示数组元素的使用状态，高效地探索所有可能的子集划分方式。

解题思路：

1、可行性检查：

- 首先计算数组总和所有元素的总和total，若total不能被k整除，则直接返回false（无法分成k个等和子集）。

- 目标子集和为target = total / k，若数组中存在元素大于target，则返回false（单个元素无法放入任何子集）。

2、状态压缩与动态规划：

- 状态表示：用一个整数mask表示数组元素的使用状态（二进制位i为1表示nums[i]已被使用，为0表示未使用）。

- DP定义：dp[mask]表示在状态mask下，当前子集已积累的和（若mask对应状态不可行，则dp[mask]为-1）。

- 状态转移：

- 初始状态dp[0] = 0（未使用任何元素时，当前积累和为0）。

- 对于每个状态mask，若dp[mask] != -1（状态可行），则尝试添加未使用的元素nums[i]：

- 若添加后积累和dp[mask] + nums[i] <= target，则新状态mask | (1 << i)的积累和为dp[mask] + nums[i]。

- 若添加后积累和恰好等于target，则新状态的积累和重置为0（开始一个新的子集）。

- 最终判断：若dp[(1 << n) - 1] == 0（所有元素都被使用，且最后一个子集的和恰好为target），则返回true。

代码：

class Solution {

public:

bool canPartitionKSubsets(vector<int>& nums, int k) {

int n = nums.size();

int total = accumulate(nums.begin(), nums.end(), 0);

// 可行性检查：总和必须能被k整除

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

int target = total / k;

// 可行性检查：单个元素不能超过目标和

for (int num : nums) {

if (num > target) return false;

}

// 状态压缩：mask表示元素使用状态，dp[mask]表示当前子集已积累的和

int max\_mask = 1 << n;

vector<int> dp(max\_mask, -1);

dp[0] = 0; // 初始状态：未使用任何元素，积累和为0

for (int mask = 0; mask < max\_mask; ++mask) {

if (dp[mask] == -1) continue; // 跳过不可行状态

// 尝试添加未使用的元素

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

if (!(mask & (1 << i))) { // 元素i未被使用

int new\_sum = dp[mask] + nums[i];

if (new\_sum > target) continue; // 超过目标和，不可行

// 新状态：使用元素i后的mask

int new\_mask = mask | (1 << i);

// 若新和等于target，重置为0（开始新子集），否则保留新和

dp[new\_mask] = (new\_sum == target) ? 0 : new\_sum;

}

}

}

// 所有元素都被使用，且最后一个子集恰好装满

return dp[max\_mask - 1] == 0;

}

};

解释：

1、可行性检查：

- 若数组总和不能被k整除，或存在元素大于target，直接返回false，减少无效计算。

2、状态压缩：

- 用mask的二进制表示元素使用状态（例如mask=0b101表示第0和第2个元素已被使用）。

- dp[mask]记录当前状态下已积累的和，-1表示该状态不可行。

3、状态转移：

- 对每个可行状态mask，遍历所有未使用的元素，计算添加后的新和：

若新和不超过target，则更新新状态new\_mask的dp值（等于新和，或重置为0如果新和等于target）。

4、最终判断：

- 当mask为(1 << n) - 1时，所有元素都被使用。若此时dp[mask] == 0，说明所有k个子集的和都恰好为target，返回true。

复杂度分析：

- 时间复杂度：O(n \* 2^n)，其中n是数组长度。共2^n个状态，每个状态遍历n个元素检查是否使用。

- 空间复杂度：O(2^n)，用于存储dp数组（大小为2^n）。

该方法利用状态压缩将子集划分问题转化为动态规划问题，高效处理了n ≤ 16的约束场景。