March 25, 2022

PARTITION TO K EQUAL SUM SUBSETS

Nikhil Kumar Singh
Vrishchik

DURATION
5min

CATEGORIES

Competitive Programming

How-To

TAGS

C++

SHARE

ABOUT THE PROBLEM:

Given a vector of integers and an integer k, we are to find if we can divide the array into k non-empty subsets with equal sums. Return true if we can or otherwise, false.

Example 1:
Input: arr = [5,2,1,2,3,4,3], k = 4
Output: true
Explanation: Possible subsets are- [2,3],[2,3],[5],[1,4]

Example 2:
Input: arr = [5,2,1,2,3,4,1], k = 4
Output: false

PRECOMPUTATION

Before discussing the approaches we should check if it is feasible for the array to be divided into k equal sets or not. For that, compute whether the sum of integers of the array is divisible by k. If not, we cannot divide the array and the size of the array must be equal or greater than k.

For the above input [5,2,1,2,3,4,3], k = 4. Here the length of the array, i.e., n >= k and the sum, i.e., 20 is divisible by 4, so 20%4==0.

APPROACH #1: BACKTRACKING

1. Consider a sum of the current subset as currSum, we are at index i, and we want the sum of each subset equal to reqSum = sum(nums[])/k.

2. We will use a visited[] to keep hold of already used elements.
3. If currSum + nums[i] > reqSum, we can only skip the ith element and go to the next element.
4. If !visited[i] and currSum + nums[i] <= resSum, we have two choices, either to include it or not include it in the current subset.
5. If we include it, we will make visited[i] = true so that the current element cannot be added to any other subset again.
6. If we find that taking this choice gives us true, we will return true.
7. Otherwise, we will exclude nums[i] from currSum by making visited[i] = false and go to the next element.
8. If at any point we find currSum == reqSum, it means we found a subset with the required sum, so call backtrack for k-1 th subset from 0th index.
9. If your k == 0, it means we found k subsets whose sum is reqSum.

Code:

```
1  bool backtrack(vector < int > & arr, int i, int reqSum, int currSum, vector < bool > &
2  if (k == 0)
3    return true;
4  if (reqSum == currSum)
5    return backtrack(arr, 0, reqSum, 0, isVisited, k - 1);
6
7  if (i == arr.size())
8    return false;
9
10 if (!isVisited[i] && arr[i] + currSum <= reqSum) {
11   isVisited[i] = true;
12   if (backtrack(arr, i + 1, reqSum, currSum + arr[i], isVisited, k))
13     return true;
14   isVisited[i] = false;
15 }
16
17 return backtrack(arr, i + 1, reqSum, currSum, isVisited, k);
18 }
19
20 bool canPartitionIntoKSubsets(vector < int > & arr, int k) {
21
22  int sum = 0;
23  for (int ele: arr) sum += ele;
24  if (sum % k != 0) return false;
25
```

```
26  int n = arr.size();
27
28  vector < bool > vis(n, false);
29  return backtrack(arr, 0, sum / k, 0, vis, k);
30 }
```

Time complexity: O(k*2^n), for every subset we traverse the whole array and make two recursive calls almost in each traversal.

APPROACH #2: BIT MASKING

In this approach we check every subset for the sum = sum/k and if the subset has the req sum we increase the count. If the total count is greater than equal to k we return true. We use bit masking to check if each subset contains sum = totalSum/k. Since each element can be taken only once we use a variable, gone, to keep track of the elements chosen.
For example, if the gone variable binary representation is 10110 then we have chosen the second, third, and the fifth element.

Code:

```
1   bool canPartitionIntoKSubsets(vector < int > & nums, int k) {
2
3   int n = nums.size();
4   int t = (1 << n);
5   sort(nums.begin(), nums.end(), greater < int > ());
6
7   int sum = 0;
8   for (int i = 0; i < n; i++) {
9     sum += nums[i];
10  }
11
12  if (sum % k != 0) {
13    return false;
14  }
15
16  sum = sum / k;
17  int cnt = 0;
18  int gone = 0;
19
20  for (int m = 0; m < t; m++) {
```

```
21    int x = 0;
22    if (!(m & gone)) {
23     for (int i = 0; i < n; i++) {
24      if (m & (1 << i)) {
25       x += nums[i];
26
27      }
28     }
29     if (x == sum) {
30      gone = m;
31      cnt++;
32     }
33    }
34   }
35   return cnt >= k;
36  }
```

Time complexity: O((2^n)*n)

RECOMMENDED FOR YOU