Partition to K Equal Sum Subsets

Given an array of integers `nums` and a positive integer `k`, find whether it's 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's possible to divide it into 4 subsets (5), (1, 4), (2,3), (2,3) with
 equal sums.
```

Note:

- `1 .`
- `0 .`

Solution 1

Update: This question has been changed after the contest. It added the special restriction `0 < nums[i] < 10000`. My solution here is without that consideration.

Assume `sum` is the sum of `nums[]`. The dfs process is to find a subset of `nums[]` which sum equals to `sum/k`. We use an array `visited[]` to record which element in `nums[]` is used. Each time when we get a `cur_sum = sum/k`, we will start from position `0` in `nums[]` to look up the elements that are not used yet and find another `cur_sum = sum/k`.

An corner case is when `sum = 0`, my method is to use `cur_num` to record the number of elements in the current subset. Only if `cur_sum = sum/k && cur_num >0`, we can start another look up process.

Some test cases may need to be added in:
`nums = {-1,1,0,0}, k = 4`
`nums = {-1,1}, k = 1`
`nums = {-1,1}, k = 2`
`nums = {-1,1,0}, k = 2`
...

Java version:

```
    public boolean canPartitionKSubsets(int[] nums, int k) {
        int sum = 0;
        for(int num:nums)sum += num;
        if(k <= 0 || sum%k != 0)return false;
        int[] visited = new int[nums.length];
        return canPartition(nums, visited, 0, k, 0, 0, sum/k);
    }

    public boolean canPartition(int[] nums, int[] visited, int start_index, int k, i
nt cur_sum, int cur_num, int target){
        if(k==1)return true;
        if(cur_sum == target && cur_num>0)return canPartition(nums, visited, 0, k-1,
0, 0, target);
        for(int i = start_index; i<nums.length; i++){
            if(visited[i] == 0){
                visited[i] = 1;
                if(canPartition(nums, visited, i+1, k, cur_sum + nums[i], cur_num++
, target))return true;
                visited[i] = 0;
            }
        }
        return false;
    }
```

C++ version:

```
    bool canPartitionKSubsets(vector<int>& nums, int k) {
        int sum = 0;
        for(int num:nums)sum+=num;
        if(k <= 0 || sum%k != 0)return false;
        vector<int> visited(nums.size(), 0);
        return canPartition(nums, visited, 0, k, 0, 0, sum/k);
    }

    bool canPartition(vector<int>& nums, vector<int>& visited, int start_index, int
k, int cur_sum, int cur_num, int target){
        if(k==1)return true;
        if(cur_sum == target && cur_num >0 )return canPartition(nums, visited, 0, k
-1, 0, 0, target);
        for(int i = start_index; i<nums.size(); i++){
            if(!visited[i]){
                visited[i] = 1;
                if(canPartition(nums, visited, i+1, k, cur_sum + nums[i], cur_num++
, target))return true;
                visited[i] = 0;
            }
        }
        return false;
    }
```

written by Vincent Cai original link here

Solution 2

It's a very classical question.
Pay attention to the note:
`1 <= k <= len(nums) <= 16`
`0 < nums[i] < 10000`

Ref: http://www.geeksforgeeks.org/partition-set-k-subsets-equal-sum/

```
class Solution {
public:

    //  Method returns true if nums can be partitioned into K subsets
    // with equal sum
    bool canPartitionKSubsets(vector<int>& nums, int K)
    {
        int N = nums.size();
        //  If K is 1, then complete array will be our answer
        if (K == 1) return true;

        //  If total number of partitions are more than N, then
        // division is not possible
        if (N < K) return false;

        // if array sum is not divisible by K then we can't divide
        // array into K partitions
        int sum = 0;
        for (int i = 0; i < N; i++) sum += nums[i];
        if (sum % K != 0) return false;

        //  the sum of each subset should be subset (= sum / K)
        int subset = sum / K;
        int subsetSum[K];
        bool taken[N];

        //  Initialize sum of each subset from 0
        for (int i = 0; i < K; i++) subsetSum[i] = 0;

        //  mark all elements as not taken
        for (int i = 0; i < N; i++) taken[i] = false;

        // initialize first subsubset sum as last element of
        // array and mark that as taken
        subsetSum[0] = nums[N - 1];
        taken[N - 1] = true;

        //  call recursive method to check K-substitution condition
        return canPartitionKSubsets(nums, subsetSum, taken, subset, K, N, 0, N - 1)
;
    }

    // Recursive Utility method to check K equal sum
    // subsetition of array
    /**
        array           - given input array
        subsetSum array   - sum to store each subset of the array
        taken             - boolean array to check whether element
```

```
        taken           - boolean array to check whether element
                          is taken into sum partition or not
        K               - number of partitions needed
        N               - total number of element in array
        curIdx          - current subsetSum index
        limitIdx        - lastIdx from where array element should be taken
    */
    bool canPartitionKSubsets(vector<int>& nums, int subsetSum[], bool taken[], int
subset, int K, int N, int curIdx, int limitIdx) {
        if (subsetSum[curIdx] == subset) {
            /*  current index (K - 2) represents (K - 1) subsets of equal
                sum last partition will already remain with sum 'subset'*/
            if (curIdx == K - 2) return true;

            //  recursive call for next subsetition
            return canPartitionKSubsets(nums, subsetSum, taken, subset,
                                        K, N, curIdx + 1, N - 1);
        }

        //  start from limitIdx and include elements into current partition
        for (int i = limitIdx; i >= 0; i--) {
            //  if already taken, continue
            if (taken[i]) continue;
            int tmp = subsetSum[curIdx] + nums[i];

            // if temp is less than subset then only include the element
            // and call recursively
            if (tmp <= subset) {
                //  mark the element and include into current partition sum
                taken[i] = true;
                subsetSum[curIdx] += nums[i];
                bool nxt = canPartitionKSubsets(nums, subsetSum, taken, subset, K,
N, curIdx, i - 1);
                // after recursive call unmark the element and remove from
                // subsetition sum
                taken[i] = false;
                subsetSum[curIdx] -= nums[i];
                if (nxt) return true;
            }
        }
        return false;
    }

};
```

written by lee215 original link here

Solution 3

dfs: ~90%

Instead of remember all sums, this dfs only remember one sum

```
class Solution(object):
    def canPartitionKSubsets(self, nums, k):
        if k==1: return True

        self.n=len(nums)
        if k>self.n: return False

        total=sum(nums)
        if total%k: return False

        self.target=total/k
        visit=[0]*self.n

        nums.sort(reverse=True)
        def dfs(k,ind,sum,cnt):
            if k==1: return True
            if sum==self.target and cnt>0:
                return dfs(k-1,0,0,0)
            for i in range(ind,self.n):
                if not visit[i] and sum+nums[i]<=self.target:
                    visit[i]=1
                    if dfs(k,i+1,sum+nums[i],cnt+1):
                        return True
                    visit[i]=0
            return False

        return dfs(k,0,0,0)
```

written by weidairpi original link here