Partition Equal Subset Sum

Given a **non-empty** array containing **only positive integers**, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Note:

1. Each of the array element will not exceed 100.
2. The array size will not exceed 200.

Example 1:

```
Input: [1, 5, 11, 5]

Output: true

Explanation: The array can be partitioned as [1, 5, 5] and [11].
```

Example 2:

```
Input: [1, 2, 3, 5]

Output: false

Explanation: The array cannot be partitioned into equal sum subsets.
```

Solution 1

```java
public class Solution {
    public boolean canPartition(int[] nums) {
        // check edge case
        if (nums == null || nums.length == 0) {
            return true;
        }
        // preprocess
        int volumn = 0;
        for (int num : nums) {
            volumn += num;
        }
        if (volumn % 2 != 0) {
            return false;
        }
        volumn /= 2;
        // dp def
        boolean[] dp = new boolean[volumn + 1];
        // dp init
        dp[0] = true;
        // dp transition
        for (int i = 1; i <= nums.length; i++) {
            for (int j = volumn; j >= nums[i-1]; j--) {
                dp[j] = dp[j] || dp[j - nums[i-1]];
            }
        }
        return dp[volumn];
    }
}
```

written by tao62 original link here

Solution 2

1. DFS solution:

```
class Solution {
public:
    bool backtrack(vector<int>& nums, int start, int target) {
        if (target <= 0) return target == 0;
        for (int i = start; i < nums.size(); i++)
            if (backtrack(nums, i + 1, target - nums[i])) return true;
        return false;
    }

    bool canPartition(vector<int>& nums) {
        int sum = accumulate(nums.begin(), nums.end(), 0);
        return !(sum & 1) && backtrack(nums, 0, sum >> 1);
    }
};
```

2. DFS can't pass the OJ, as more test cases are added. So here comes a DP solution based on @Hermits solution

```
bool canPartition(vector<int>& nums) {
    int sum = accumulate(nums.begin(), nums.end(), 0), target = sum >> 1;
    if (sum & 1) return false;
    vector<int> dp(target + 1, 0);
    dp[0] = 1;
    for(auto num : nums)
        for(int i = target; i >= num; i--)
            dp[i] = dp[i] || dp[i - num];
    return dp[target];
}
```

3. A very fast and cool Bit solution by @alvin-777 solution

```
bool canPartition(vector<int>& nums) {
    bitset<5001> bits(1);
    int sum = accumulate(nums.begin(), nums.end(), 0);
    for (auto n : nums) bits |= bits << n;
    return !(sum & 1) && bits[sum >> 1];
}
```

written by zyoppy008 original link here

Solution 3

```
/*
Standard dfs.
We sum all the numbers in the array, if it's an odd number, retrun false;
otherwise the question is reduced to
    find a subset of numbers, whos sum is total sum / 2.
We can use standard dfs to find if there is one
*/

public class Solution {
    public boolean canPartition(int[] nums) {
        Map<Integer, Integer> map = new HashMap<Integer, Integer>();
        int sum = 0;
        for(int i : nums){
            if(map.containsKey(i)){
                map.put(i, map.get(i) + 1);
            }else{
                map.put(i, 1);
            }
            sum += i;
        }
        if(sum % 2 == 1) return false;
        return helper(map, sum / 2);
    }

    private boolean helper(Map<Integer, Integer> map, int target){
        /*target is achieveable*/
        if(map.containsKey(target) && map.get(target) > 0) return true;
        /*dfs*/
        for(int key : map.keySet()){
            if(key < target && map.get(key) > 0){
                map.put(key, map.get(key) - 1);
                if(helper(map, target - key)) return true;
                map.put(key, map.get(key) + 1);
            }
        }
        return false;
    }
}````
```

written by DonaldTrump original link here