Matchsticks to Square

Remember the story of Little Match Girl? By now, you know exactly what matchsticks the little match girl has, please find out a way you can make one square by using up all those matchsticks. You should not break any stick, but you can link them up, and each matchstick must be used **exactly** one time.

Your input will be several matchsticks the girl has, represented with their stick length. Your output will either be true or false, to represent whether you could make one square using all the matchsticks the little match girl has.

Example 1:

```
Input: [1,1,2,2,2]
Output: true

Explanation: You can form a square with length 2, one side of the square came two s
ticks with length 1.
```

Example 2:

```
Input: [3,3,3,3,4]
Output: false

Explanation: You cannot find a way to form a square with all the matchsticks.
```

Note:

1. The length sum of the given matchsticks is in the range of `0` to `10^9`.
2. The length of the given matchstick array will not exceed `15`.

Solution 1

According to https://en.wikipedia.org/wiki/Partition_problem, the partition problem (or number partitioning) is the task of deciding whether a given multiset `S` of positive integers can be partitioned into two subsets `S1` and `S2` such that the sum of the numbers in `S1` equals the sum of the numbers in `S2`. The partition problem is `NP-complete`.

When I trying to think how to apply dynamic programming solution of above problem to this one (difference is divid `S` into 4 subsets), I took another look at the constraints of the problem:
The length sum of the given matchsticks is in the range of `0` to `10^9`.
The length of the given matchstick array will not exceed `15`.

Sounds like the input will not be very large... Then why not just do DFS? In fact, DFS solution passed judges.

Anyone solved this problem by using DP? Please let me know :)

```
public class Solution {
    public boolean makesquare(int[] nums) {
     if (nums == null || nums.length < 4) return false;
        int sum = 0;
        for (int num : nums) sum += num;
        if (sum % 4 != 0) return false;

     return dfs(nums, new int[4], 0, sum / 4);
    }

    private boolean dfs(int[] nums, int[] sums, int index, int target) {
     if (index == nums.length) {
         if (sums[0] == target && sums[1] == target && sums[2] == target) {
      return true;
         }
         return false;
     }

     for (int i = 0; i < 4; i++) {
         if (sums[i] + nums[index] > target) continue;
         sums[i] += nums[index];
            if (dfs(nums, sums, index + 1, target)) return true;
         sums[i] -= nums[index];
     }

     return false;
    }
}
```

`Updates on 12/19/2016` Thanks @benjamin19890721 for pointing out a very good optimization: Sorting the input array `DESC` will make the DFS process run much faster. Reason behind this is we always try to put the next matchstick in the first subset. If there is no solution, trying a longer matchstick first will get to negative conclusion earlier. Following is the updated code. Runtime is improved from more than 1000ms to around 40ms. A big improvement.

```
public class Solution {
    public boolean makesquare(int[] nums) {
     if (nums == null || nums.length < 4) return false;
        int sum = 0;
        for (int num : nums) sum += num;
        if (sum % 4 != 0) return false;

        Arrays.sort(nums);
        reverse(nums);

     return dfs(nums, new int[4], 0, sum / 4);
    }

    private boolean dfs(int[] nums, int[] sums, int index, int target) {
     if (index == nums.length) {
         if (sums[0] == target && sums[1] == target && sums[2] == target) {
      return true;
         }
         return false;
     }

     for (int i = 0; i < 4; i++) {
         if (sums[i] + nums[index] > target) continue;
         sums[i] += nums[index];
            if (dfs(nums, sums, index + 1, target)) return true;
         sums[i] -= nums[index];
     }

     return false;
    }

    private void reverse(int[] nums) {
        int i = 0, j = nums.length - 1;
        while (i < j) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
            i++; j--;
        }
    }
}
```

written by shawngao original link here

Solution 2

This is a solution inspired by a friend who doesn't do leetcode. I am just posting his solution with the best explanation I can give. The bitmasking technique may look sophisticated but the idea is actually pretty straightforward because it uses brute force with some optimizations. A bitmask is used as a representation of a subset. For example if nums = {1,1,2,2,2}, then a bitmask = 01100 represents the subset {1,2}.

```
bool makesquare(vector<int>& nums) {
    int n = nums.size();

    long sum = accumulate(nums.begin(), nums.end(), 0l);
    if (sum % 4)
        return false;
    long sideLen = sum / 4;
    // need to solve the problem of partitioning nums into four equal subsets eac
h having
    // sum equal to sideLen
    vector<int> usedMasks;
    // validHalfSubsets[i] == true iff the subset represented by bitmask i
    // has sum == 2*sideLen, AND the subset represented by i can be further parti
tioned into
    // two equal subsets. See below for how it is used.
    vector<bool> validHalfSubsets(1<<n, false);

    // E.g., if n = 5, (1 << 5 - 1) = 11111 represents the whole set
    int all = (1 << n) - 1;
    // go through all possible subsets each represented by a bitmask
    for (int mask = 0; mask <= all; mask++) {
        long subsetSum = 0;
        // calculate the sum of this subset
        for (int i = 0; i < 32; i++) {
     if ((mask >> i) & 1)
  subsetSum += nums[i];
        }
 // if this subset has what we want
 if (subsetSum == sideLen) {
     for (int usedMask : usedMasks) {
     // if this mask and usedMask are mutually exclusive
         if ((usedMask & mask) == 0) {
      // then they form a valid half subset whose sum is 2 * sideLen,
                    // that can be further partitioned into two equal subsets (us
edMask and mask)
      int validHalf = usedMask | mask;
      validHalfSubsets[validHalf] = true;
      // if in the past we concluded that the other half is also a valid
      // half subset, DONE!
      if (validHalfSubsets[all ^ validHalf])
          return true;
         }
            }
     usedMasks.push_back(mask);
        }
    }
    return false;
}
```

written by yangluphil original link here

Solution 3

If the sum isn't divisible by 4, or if there are less than 4 matchsticks, or if one matchstick is greater than the expected sidelength, then it is impossible.

Otherwise, let's see if it is possible to partition the set into 4 equal parts. We do this by bitmask dp. Let the i-th position of mask be 1 if we have not used A[i] yet, and let cur represent the length of an unfilled sidelength we have yet to fill.

Code:

```
if len(A) < 4 or sum(A) % 4 or max(A) > sum(A) / 4:
  return False

T = sum(A) / 4
N = len(A)
A.sort()

memo = {}
def dp(mask, cur = T):
  if (mask, cur) in memo: return memo[mask, cur]
  if mask == 0: return cur == 0
  if cur == 0: return dp(mask, T)

  ans = False
  for bit in xrange(N):
    if mask & (1 << bit):
      if A[bit] > cur:
        break
      if dp(mask ^ (1 << bit), cur - A[bit]):
        ans = True
        break
  memo[mask, cur] = ans
  return ans

return dp(2**N - 1)
```

written by awice original link here