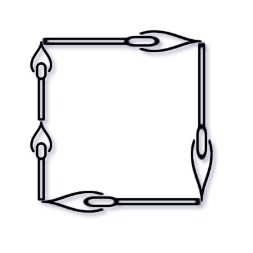
<https://leetcode.com/problems/matchsticks-to-square/description/>

You are given an integer array matchsticks where matchsticks[i] is the length of the ith matchstick. You want to use **all the matchsticks** to make one square. You **should not break** any stick, but you can link them up, and each matchstick must be used **exactly one time**.

Return true if you can make this square and false otherwise.

**Example 1:**



**Input:** matchsticks = [1,1,2,2,2]

**Output:** true

**Explanation:** You can form a square with length 2, one side of the square came two sticks with length 1.

**Example 2:**

**Input:** matchsticks = [3,3,3,3,4]

**Output:** false

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

**Constraints:**

1 <= matchsticks.length <= 15

1 <= matchsticks[i] <= 10^8

**Attempt 1: 2025-08-25**

**Solution 1: Backtracking + Sorting (60 min)**

class Solution {

    public boolean makesquare(int[] matchsticks) {

        if(matchsticks.length < 4) {

            return false;

        }

        int total = Arrays.stream(matchsticks).sum();

        if(total % 4 != 0) {

            return false;

        }

        int target = total / 4;

        Arrays.sort(matchsticks);

        // Reverse to descending order

        for(int i = 0; i < matchsticks.length / 2; i++) {

            int tmp = matchsticks[i];

            matchsticks[i] = matchsticks[matchsticks.length - 1 - i];

            matchsticks[matchsticks.length - 1 - i] = tmp;

        }

        if(matchsticks[0] > target) {

            return false;

        }

        return helper(matchsticks, new int[4], target, 0);

    }

    private boolean helper(int[] matchsticks, int[] sides, int target, int index) {

        if(index == matchsticks.length) {

            // Implicit Guarantee by Constraints:

            // - We've already verified that the total sum of matchsticks equals 4 \* target

            // - We never allow any side to exceed the target (due to the pruning condition)

            // - Therefore, if we successfully place all matchsticks, each side must exactly equal the target

            return true;

        }

        for(int i = 0; i < 4; i++) {

            // Skip if adding this matchstick would exceed target

            if(sides[i] + matchsticks[index] > target) {

                continue;

            }

            // Skip duplicate combinations

            if(i > 0 && sides[i] == sides[i - 1]) {

                continue;

            }

            sides[i] += matchsticks[index];

            if(helper(matchsticks, sides, target, index + 1)) {

                return true;

            }

            sides[i] -= matchsticks[index];

        }

        return false;

    }

}

Time Complexity: O(4^n) in worst case, but heavily pruned in practice

Space Complexity: O(n) for recursion stack

**Refer to Deepseek**

**Approach**

This problem can be solved using **backtracking with optimization**. The key idea is to determine if we can partition the matchsticks into 4 subsets that each sum to the target side length (total sum / 4). We use:

**Early pruning**: Sort matchsticks in descending order to try larger pieces first

**Backtracking**: Try placing each matchstick in one of the 4 sides

**Memoization**: Skip duplicate states and impossible cases

**Solution Code**

import java.util.Arrays;

class Solution {

public boolean makesquare(int[] matchsticks) {

if (matchsticks == null || matchsticks.length < 4) {

return false;

}

int total = 0;

for (int stick : matchsticks) {

total += stick;

}

// Check if total length can be divided into 4 equal sides

if (total % 4 != 0) {

return false;

}

int side = total / 4;

// Sort in descending order to try larger matchsticks first (better pruning)

Arrays.sort(matchsticks);

reverse(matchsticks);

// If any matchstick is longer than side length, impossible

if (matchsticks[0] > side) {

return false;

}

int[] sides = new int[4];

return backtrack(matchsticks, sides, 0, side);

}

private boolean backtrack(int[] matchsticks, int[] sides, int index, int target) {

if (index == matchsticks.length) {

// All matchsticks used, check if all sides are equal to target

}

for (int i = 0; i < 4; i++) {

// Skip if adding this matchstick would exceed target

if (sides[i] + matchsticks[index] > target) {

continue;

}

// Skip duplicate states (if previous sides have same length)

int j = i;

while (j-- > 0) {

if (sides[i] == sides[j]) {

break;

}

}

if (j >= 0) {

continue;

}

sides[i] += matchsticks[index];

if (backtrack(matchsticks, sides, index + 1, target)) {

return true;

}

sides[i] -= matchsticks[index];

}

return false;

}

private void reverse(int[] arr) {

int left = 0, right = arr.length - 1;

while (left < right) {

int temp = arr[left];

arr[left] = arr[right];

arr[right] = temp;

left++;

right--;

}

}

}

**Optimized Solution with Better Pruning**

import java.util.Arrays;

class Solution {

public boolean makesquare(int[] matchsticks) {

if (matchsticks == null || matchsticks.length < 4) return false;

int total = Arrays.stream(matchsticks).sum();

if (total % 4 != 0) return false;

int side = total / 4;

Arrays.sort(matchsticks);

// Reverse to descending order

for (int i = 0; i < matchsticks.length / 2; i++) {

int temp = matchsticks[i];

matchsticks[i] = matchsticks[matchsticks.length - 1 - i];

matchsticks[matchsticks.length - 1 - i] = temp;

}

if (matchsticks[0] > side) return false;

return dfs(matchsticks, new int[4], 0, side);

}

private boolean dfs(int[] matchsticks, int[] sides, int index, int target) {

if (index == matchsticks.length) {

return true;

}

for (int i = 0; i < 4; i++) {

if (sides[i] + matchsticks[index] > target) continue;

// Skip duplicate combinations

if (i > 0 && sides[i] == sides[i - 1]) continue;

sides[i] += matchsticks[index];

if (dfs(matchsticks, sides, index + 1, target)) return true;

sides[i] -= matchsticks[index];

// Early termination: if we can't place the first matchstick anywhere

if (sides[i] == 0) break;

}

return false;

}

}

**Explanation**

**Initial Checks**:

Check if there are at least 4 matchsticks

Check if total length is divisible by 4

Check if any matchstick is longer than the target side length

**Sorting**:

Sort matchsticks in descending order to try larger pieces first

This helps prune impossible paths early in the backtracking

**Backtracking**:

Try placing each matchstick in one of the 4 sides

Skip sides where adding the matchstick would exceed the target

Skip duplicate states (sides with same current length)

If a matchstick can't be placed in any side, backtrack

**Optimizations**:

Early termination when first matchstick can't be placed

Skipping duplicate side lengths to avoid redundant calculations

Pruning when current side + matchstick > target

**Time Complexity**: O(4^n) in worst case, but heavily pruned in practice

**Space Complexity**: O(n) for recursion stack

This approach efficiently explores all possible partitions while pruning impossible paths early, making it feasible for the problem constraints.

**Explanation of DFS Termination Condition in Optimized Solution**

**Key Reasons:**

**Implicit Guarantee by Constraints**:

We've already verified that the total sum of matchsticks equals 4 \* target

We never allow any side to exceed the target (due to the pruning condition)

Therefore, if we successfully place all matchsticks, each side must exactly equal the target

**Mathematical Certainty**:

Let S = total sum of all matchsticks = 4 \* target

Let s₁, s₂, s₃, s₄ be the current sums of the 4 sides

We know s₁ + s₂ + s₃ + s₄ = S = 4 \* target

We also know s₁ ≤ target, s₂ ≤ target, s₃ ≤ target, s₄ ≤ target (due to pruning)

The only possible solution is s₁ = s₂ = s₃ = s₄ = target

**Algorithmic Efficiency**:

Removing the explicit check makes the code slightly faster

The condition index == matchsticks.length is sufficient to indicate success

**Why This Works:**

private boolean dfs(int[] matchsticks, int[] sides, int index, int target) {

if (index == matchsticks.length) {

return true; // All matchsticks placed successfully

}

for (int i = 0; i < 4; i++) {

if (sides[i] + matchsticks[index] > target) continue; // Pruning: never exceed target

// ... rest of the backtracking logic

}

return false;

}

The algorithm ensures that:

We only proceed if adding a matchstick doesn't make any side exceed the target

We use all matchsticks (total sum = 4 \* target)

Therefore, when we use all matchsticks, each side must equal the target

This optimization makes the code cleaner and slightly more efficient while maintaining correctness.

**Refer to**

[L2397.Maximum Rows Covered by Columns (Ref.L473)](note://WEB050bb59c3f216a8c91af09746dbf6efe)