473. Matchsticks to Square

Medium · Bit Manipulation · Array · Dynamic Programming · Backtracking · Bitmask · Leetcode Link

Problem Description

In this problem, you are given a set of matchsticks, each with a different length. The objective is to determine whether these matchsticks can be arranged to form a perfect square without breaking any of them. This means that each matchstick must be used exactly once and all must contribute to the shape of the square.

To imagine this more concretely, think of it as a puzzle where we must create four equal-length sides (as only a square has equal sides) with the given set of matchsticks. If we can successfully do this, the function should return `true`, otherwise, it should return `false`.

Intuition

The solution to this problem relies on a depth-first search (DFS) algorithm, which attempts to build the square by adding matchsticks one by one to each of the four sides until all matchsticks have been placed. The key insights behind this approach include:

1. **The Total Length of Matchsticks**: Before attempting to place matchsticks, we calculate the total length of all matchsticks by summing them. For a square to be possible, this total must be divisible by 4 equally (since a square has four equal sides). If it is not divisible, we know right away that we can't make a square, so we return `false`.

2. **Avoiding Useless Work**: Two optimizations help us reduce the computation time. First, we sort the matchsticks in descending order, beginning with the longest. This helps because if the longest matchstick is longer than the side of the square, we know it's impossible to form a square and can return `false` immediately. Second, when trying to place a matchstick on a side that is equal in length to a side we just attempted, we skip that attempt since the order of matchsticks on the sides doesn't matter.

3. **Recursive DFS**: Using a recursive function `dfs(u)`, we try to place the `u`th matchstick on each side. We only proceed with the recursion if adding the matchstick to the current side does not exceed the expected length of a side (which is the total length of the matchsticks divided by 4). If, at any point, a side becomes too long, we backtrack by removing the matchstick from that side (`edges[i] -= matchsticks[u]`) and trying other options. If we can place the last matchstick without exceeding the side length, we know a square is possible and return `true`.

By carefully placing each matchstick using DFS and backtracking when necessary, the algorithm systematically explores all possible placements until it either succeeds in forming a square or exhausts all options and returns `false`.

Solution Approach

The implementation of this solution is a classic example of utilizing **Depth-First Search (DFS)** alongside **backtracking** to explore all possible combinations of placing matchsticks on four sides of a square. Here's a step-by-step explanation of the different parts of the code:

1. **Preparation Step**: We begin by adding up the total length of all matchsticks and trying to divide it by 4 to get the target length for each side of the square (`x`). If there's a remainder (`mod`), or if the longest matchstick is longer than the target side length, we cannot form a square and return `false`.

2. **Sorting**: The matchsticks are sorted in descending order because placing longer matchsticks earlier can help us reach the conclusion faster. This is because if a long matchstick doesn't fit, we don't need to consider combinations of smaller ones that would also not fit.

3. **Depth-First Search (DFS)**: The recursive function `dfs(u)` tries to place the `u`th matchstick in turn on each of the four sides. For this, we utilize an array `edges` of length 4, initialized with zeroes, to keep track of the current length of each side as we add matchsticks.

4. **Branching and Pruning**:

   - The DFS explores different branches, each representing a different combination of matchstick placements.
   - While placing matchsticks, we avoid redundant work by skipping a side if it is of the same length as the previous one (`i > 0 and edges[i - 1] == edges[i]`), because different permutations of sides of the same current length don't change the overall problem.
   - We ensure that adding a matchstick does not make any side longer than the target (`edges[i] <= x`).

5. **Backtracking**:

   - After trying to add a matchstick to a side, if adding the next matchstick results in exceeding the target length or if all matchsticks up to the last are placed without forming a square, the function backtracks. This means undoing the last matchstick addition (`edges[i] -= matchsticks[u]`) and trying a different placement.

6. **Completion**: If all matchsticks are placed, `dfs(u)` will eventually reach past the last matchstick indicating that all matchsticks have been used to form a square, and will return `true`.

The entry point is calling `dfs(0)`, indicating we start with the first matchstick. The process will recurse, branching, and backtracking until a solution is found or all possibilities are exhausted. The final outcome will be the return of either `true` or `false` from the `makesquare` function indicating the possibility of forming a square with the given matchsticks.

Example Walkthrough

To illustrate the solution approach, let's use a small example. Suppose we have the set of matchsticks with lengths `[3, 3, 3, 3, 4]`.

1. **Preparation Step**: We add up the lengths of all matchsticks. Here, the sum is `3+3+3+3+4=16`. If we divide 16 by 4, the target length for each side of the square (x) is 4, and since there's no remainder, we can proceed. The longest matchstick is exactly 4 units long, which matches our target side length, so it's possible to form a square.

2. **Sorting**: We sort the matchsticks in descending order to get `[4, 3, 3, 3, 3]`.

3. **Depth-First Search (DFS)**: The recursive function `dfs(u)` will try to add the `u`th matchstick to one of the sides.

4. **Branching and Pruning**: We start with `dfs(0)` and attempt to place the matchstick of length 4 on a side. Since the array `edges` starts as `[0, 0, 0, 0]`, after placing the first matchstick, it becomes `[4, 0, 0, 0]`.

   - Next, we call `dfs(1)` to place the matchstick of length 3. As per our pruning rule, we start with `edges[1]` because `edges[0]` is already 4, which is the target length for a side. Now `edges` becomes `[4, 3, 0, 0]`.

   - Then `dfs(2)` is called, and we place another 3 length matchstick on `edges[2]`. `edges` now is `[4, 3, 3, 0]`.

   - With `dfs(3)`, we can't place the next 3 on `edges[1]` because it would exceed our target. We put it in `edges[2]` to get `[4, 3, 3, 3]`. However, this is a side with the same length as the previous side (`edges[1]`), so we should skip this step and place it in `edges[3]` instead, resulting in `[4, 3, 3, 3]`.

   - Finally, we have the last matchstick with `dfs(4)`. We can't add it to `edges[1]`, `edges[2]`, or `edges[3]` as they're already at 3, but we can add it to `edges[1]` to fill the side up to the target of 4, resulting in `[4, 4, 3, 3]`.

5. **Backtracking**: If at any point a move had been invalid, we would have removed the last matchstick tried, and attempted another configuration. However, in this example, all moves were valid.

6. **Completion**: All matchsticks have been placed and the `edges` array reads `[4, 4, 3, 3]`. The target length for each side was 4, and `dfs(4)` will exit successfully, showing that these matchsticks can indeed form a perfect square, returning `true`.

As such, we can conclude that `[3, 3, 3, 3, 4]` can form a square based on the described DFS and backtracking solution approach.

Python Solution

```python
from typing import List

class Solution:
    def makesquare(self, matchsticks: List[int]) -> bool:
        def backtrack(index):
            # If we have considered all matchsticks, we're done and can potentially form a square.
            if index == len(matchsticks):
                return True

            for edge in range(4):
                # If the current edge and the previous one are the same, skip to avoid redundant work.
                if edge > 0 and edges[edge - 1] == edges[edge]:
                    continue
                # Try to add the current matchstick to the current edge.
                edges[edge] += matchsticks[index]
                # If adding the current matchstick didn't exceed the target length and the remaining matchsticks can be placed, ret
                if edges[edge] <= target_per_edge and backtrack(index + 1):
                    return True
                # Otherwise, backtrack by removing the matchstick from the current edge.
                edges[edge] -= matchsticks[index]

            # If no combination worked, return False.
            return False

        # Calculate the total length of all matchsticks and the target length per edge.
        total_length, remainder = divmod(sum(matchsticks), 4)
        # If the total length of matchsticks is not divisible by 4 or the longest matchstick is longer
        # than the target per edge, we cannot possibly form a square.
        if remainder or total_length < max(matchsticks):
            return False

        # Initialize the array representing the four edges of the potential square.
        edges = [0] * 4
        # Sort the matchsticks in descending order to improve efficiency.
        matchsticks.sort(reverse=True)
        # Start the backtracking process from the first matchstick.
        return backtrack(0)

# Example usage:
# sol = Solution()
# result = sol.makesquare([1,1,2,2,2])
# print(result)  # Output: True, since the matchsticks can form a square

```

Java Solution

```java
class Solution {
    public boolean makesquare(int[] matchsticks) {
        int sum = 0; // This will hold the total length of all matchsticks
        int maxStickLength = 0; // This will hold the length of the longest matchstick

        // Calculate the total sum of matchsticks' lengths and find the longest matchstick
        for (int length : matchsticks) {
            sum += length;
            maxStickLength = Math.max(maxStickLength, length);
        }

        // Each side of the square should be sum / 4
        int sideLength = sum / 4;
        int remainder = sum % 4; // Remainder should be zero for a perfect square

        // If remainder is not zero or any matchstick is longer than the calculated side length, we can't form a square
        if (remainder != 0 || sideLength < maxStickLength) {
            return false;
        }

        // Sort the array in ascending order to optimize the dfs function later
        Arrays.sort(matchsticks);

        // This will keep track of the lengths of the four sides of the square
        int[] sides = new int[4];

        // Start the depth-first search from the last (and largest due to sorting) matchstick
        return dfs(matchsticks.length - 1, sideLength, matchsticks, sides);
    }

    private boolean dfs(int index, int targetSideLength, int[] matchsticks, int[] sides) {
        // If we have considered all matchsticks, check if all sides are equal to target side length
        if (index < 0) {
            return true;
        }

        // Try to place current matchstick on each side and see if it helps in making a square
        for (int i = 0; i < 4; ++i) {
            // This check ensures we do not place matchsticks redundantly when previous side is the same as the current one
            if (i > 0 && sides[i - 1] == sides[i]) {
                continue;
            }

            // Add the current matchstick to the i-th side
            sides[i] += matchsticks[index];

            // If the side does not exceed the target side length, recursively continue to place the next matchstick
            if (sides[i] <= targetSideLength && dfs(index - 1, targetSideLength, matchsticks, sides)) {
                // If dfs returns true, this means all matchsticks can be placed to form a square
                return true;
            }

            // If adding the matchstick to the i-th side doesn't lead to a solution, remove it
            sides[i] -= matchsticks[index];
        }

        // If none of the sides can accommodate the current matchstick, return false
        return false;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

using namespace std;

class Solution {
public:
    // Function to check whether it's possible to make a square with the matchsticks.
    bool makesquare(vector<int>& matchsticks) {
        int totalLength = 0, maxLength = 0;

        // Calculate the total length of matchsticks and the length of the longest matchstick.
        for (int stick : matchsticks) {
            totalLength += stick;
            maxLength = max(maxLength, stick);
        }

        // The sum of all matchsticks lengths should be divisible by 4 to make a square,
        // and no single matchstick should be longer than the side length of the square.
        int sideLength = totalLength / 4;
        int remainder = totalLength % 4;
        if (remainder != 0 || sideLength < maxLength) return false;

        // Sort the matchsticks in decreasing order to optimize the DFS process.
        sort(matchsticks.begin(), matchsticks.end(), greater<int>());

        // Create an array to store the lengths of the 4 sides of the square.
        vector<int> sides(4, 0);

        // Start the DFS search to see if a square can be formed.
        return dfs(0, sideLength, matchsticks, sides);
    }

private:
    // Helper function to perform DFS (depth-first search).
    bool dfs(int index, int sideLength, vector<int>& matchsticks, vector<int>& sides) {
        // Base case: if all matchsticks are used.
        if (index == matchsticks.size()) return true;

        // Try to place the current matchstick on each side of the square.
        for (int i = 0; i < 4; ++i) {
            // Skip trying the same length as previous side to reduce repetition.
            if (i > 0 && sides[i - 1] == sides[i]) continue;

            sides[i] += matchsticks[index];

            // Recur only if the current side does not exceed the intended side length
            // and if the rest of the matchsticks can be placed to make a square.
            if (sides[i] <= sideLength && dfs(index + 1, sideLength, matchsticks, sides)) return true;

            // This is backtracking - if adding the current matchstick doesn't lead to a solution,
            // remove it from the current side and try the next side.
            sides[i] -= matchsticks[index];
        }

        // Return false if placing the current matchstick on any side
        // doesn't lead to a solution (doesn't make a square).
        return false;
    }
};

```

Typescript Solution

```typescript
const totalLength = (sticks: number[]): number => sticks.reduce((acc, length) => acc + length, 0);

// Function to perform depth-first search to find if it's possible to construct a square
const dfs = (index: number, sideLength: number, matchsticks: number[], sides: number[]): boolean => {
    // Base case: all matchsticks are used, true if all sides are equal to side length
    if (index === matchsticks.length) {
        return sides.every(side => side === sideLength);
    }

    for (let i = 0; i < 4; ++i) {
        // Skip trying the same length as previous side to reduce repetition
        if (i > 0 && sides[i - 1] === sides[i]) continue;

        // Place the current matchstick on the i-th side of the square if it doesn't exceed the side length
        if (sides[i] + matchsticks[index] <= sideLength) {
            sides[i] += matchsticks[index];
            // Recursively try to place the rest of the matchsticks
            if (dfs(index + 1, sideLength, matchsticks, sides)) return true;
            // Backtrack if unable to form a square with current placement
            sides[i] -= matchsticks[index];
        }
    }

    // Couldn't place the current matchstick in any side to form a square
    return false;
}

// Function to check whether it's possible to form a square with the matchsticks
const makesquare = (matchsticks: number[]): boolean => {
    const sumOfLength = totalLength(matchsticks);
    // The sum of all matchsticks lengths should be divisible by 4 to form a square
    if (sumOfLength % 4 !== 0) return false;

    const sideLength = sumOfLength / 4;
    const maxLength = Math.max(...matchsticks);
    // No single matchstick should be longer than the side length of the square
    if (sideLength < maxLength) return false;

    // Sort matchsticks in decreasing order to help optimize the search
    matchsticks.sort((a, b) => b - a);

    // Initialize sides of the square with zero length
    const sides = [0, 0, 0, 0];

    // Perform DFS to determine if a square can be formed
    return dfs(0, sideLength, matchsticks, sides);
}

// Example of how to use the global function makesquare
const matchsticks = [1,1,2,2,2];
console.log(makesquare(matchsticks)); // Output should be either true or false based on the input array

```

Time and Space Complexity

The provided code aims to determine if an array of integers (`matchsticks`) can be used to form a square. The core of this approach is to use Depth-First Search (DFS) to try to place each matchstick on one of the four sides of the square until all matchsticks are placed or no valid solution can be found.

Time Complexity:

The time complexity of the given algorithm is primarily dependent on the DFS search. In the worst-case scenario, DFS would explore all possible placements of matchsticks across the four sides. This exponential behavior essentially looks at `4^n` combinations, where `n` is the number of matchsticks. However, some optimizations are present:

1. The matchsticks are sorted in descending order, which can lead to earlier termination in certain branches of the search tree when the matchstick is too long to fit on any side.
2. The algorithm also skips over sides with the same current length to avoid redundant placements.

Considering these optimizations, the upper bound on the time complexity still remains exponential, close to O(4^n), due to the nature of the problem which is essentially a NP-Complete problem akin to the Subset Sum or Partition problem. However, the actual performance can be somewhat better than O(4^n) on average due to the early pruning of the search space.

Space Complexity:

The space complexity of the algorithm is mainly due to the recursive call stack of the DFS function and the storage of the sides of the square. In the worst-case scenario, DFS recursion could go as deep as `n` levels where `n` is the number of matchsticks. Each level of recursion uses additional space for the execution context, leading to a space complexity of O(n). The `edges` array contains 4 integers, so it uses O(1) space.

Combining both factors, the overall space complexity is O(n), mainly due to the call stack depth in the recursive DFS search.