Problem Statement

Given a set of positive numbers, partition the set into two subsets with a minimum difference between their subset sums.

Example 1:

```
Input: {1, 2, 3, 9}
Output: 3
Explanation: We can partition the given set into two subsets where minimum
absolute difference between the sum of numbers is '3'. Following are the
two subsets: {1, 2, 3} and {9}
```

Example 2:

```
Input: {1, 2, 7, 1, 5}
Output: 0
Explanation: We can partition the given set into two subsets where minimum
absolute difference between the sum of number is '0'. Following are the
two subsets: {1, 2, 5} and {7, 1}.
```

Basic Solution

The problem follows the **0/1 Knapsack pattern** and can be converted into a subset sum problem

Let's assume S1 and S2 are the two desired subsets. A basic brute-force solution could be to try adding each element either in S1 or S2, to find the combination that gives the minimum sum difference between thw two sets.

So, our brute force algorithm will look like:

```
for each number 'i':
    add number 'i' to S1 and recursively process the remaining numbers
    add number 'i' to S2 and recursively process the remaining numbers
return the minimum absolute difference of the above two sets
```

Code:

```
let canPartition = (num) => {
    return canPartitionHelper(num, 0, 0, 0);
};

let canPartitionHelper = (num, currentIndex, sum1, sum2) => {
    // base check
    if (currentIndex === num.length) return Math.abs(sum1 - sum2);
```

```
    //recursive call after including the number at the current index in
the first set
    const diff1 = canPartitionHelper(num, currentIndex + 1, sum1 +
num[currentIndex], sum2);

    //recursive call after including the number at the current index in
the second set
    const diff2 = canPartitionHelper(num, currentIndex + 1, sum1, sum2 +
num[currentIndex]);

    return Math.min(diff1, diff2);
};

console.log(`Minimum subset difference is: ---> ${canPartition([1, 2, 3,
9])}`);
console.log(`Minimum subset difference is: ---> ${canPartition([1, 2, 7,
1, 5])}`);
console.log(`Minimum subset difference is: ---> ${canPartition([1, 3, 100,
4])}`);
```

The time complexity of the above algorithm is exponential `O(2^n)`, where 'n' represents the total number. The space complexity is `O(n)` which is used to store the recursion stack.

Top-down Dynamic Programming with Memoization

We can use memoization to overcome the overlapping subproblems.

We will be using a two-dimensional array to store the results of the solved sub-problems. We can uniquely identify a sub-problem from `currentIndex` and `sum1`; as `sum2` will always be the sum of the remaining numbers.

Here is the code:

```
let canPartition = (num) => {
    const dp = [];
    let canPartitionHelper = (num, currentIndex, sum1, sum2) => {
        // base check
        if (currentIndex === num.length) return Math.abs(sum1 - sum2);

        dp[currentIndex] = dp[currentIndex] || [];

        if (typeof dp[currentIndex][sum1] === 'undefined') {
            //recursive call after including the number at the current
index in the first set
            const diff1 = canPartitionHelper(num, currentIndex + 1, sum1 +
num[currentIndex], sum2);

            //recursive call after including the number at the current
index in the second set
            const diff2 = canPartitionHelper(num, currentIndex + 1, sum1,
```

```
    sum2 + num[currentIndex]);

                dp[currentIndex][sum1] = Math.min(diff1, diff2);
            }

            return dp[currentIndex][sum1];
        };

        return canPartitionHelper(num, 0, 0, 0);
    };

    console.log(`Minimum subset difference is: ---> ${canPartition([1, 2, 3,
    9])}`);
    console.log(`Minimum subset difference is: ---> ${canPartition([1, 2, 7,
    1, 5])}`);
    console.log(`Minimum subset difference is: ---> ${canPartition([1, 3, 100,
    4])}`);
```

Bottom-up Dynamic Programming

Let's assume 'S' represents the total sum of all the numbers. So what are we trying to achieve in this problem is to find a subset whose sum is close to 'S/2' as possible, because if we can partition the given set into two subsets of equal sum, we get the minimum difference i.e. 0. This transforms our problem to subset sum, where we try to find a subset whose sum is equal to a given number - 'S/2' in our case. If we can't find such a subset, then we will take the subset which has the sum closest to 'S/2'. This is easily possible since we are calculating all possible sums with every subset.

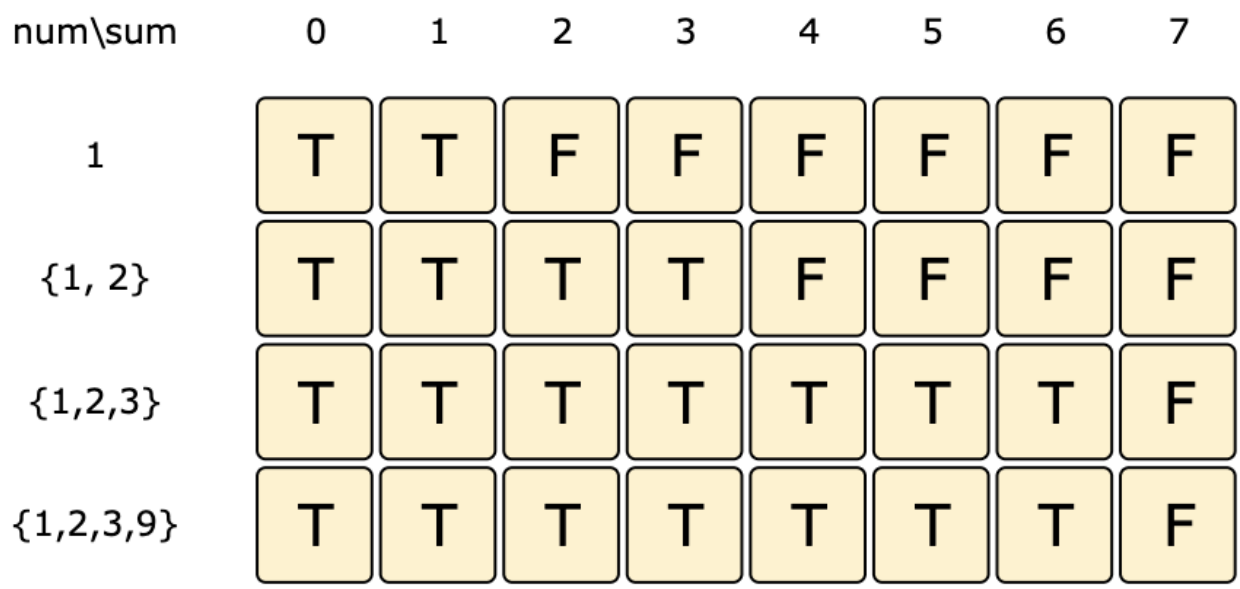
Essentially, we need to calculate all the possible sums up to ‘S/2’ for all numbers. So how do we populate the array `d[TotalNumbers][s/2+1]` in the bottom-up fashion?

For every possible sum ‘s’ (where 0 <= s <= S/2), we have two options:

1. Exclude the number. In this case, we will see if we can get the sum ‘s’ from the subset excluding this number => `dp [index-1][s]`
2. Include the number if its value is not more than ‘s’. In this case, we will see if we can find a subset to get the remaining sum => `dp[index-1][s-num[index]]`

If either of the two above scenarios is true, we can find a subset with a sum equal to ‘s’. We should dig into this before we can learn how to find the closest subset.

Let's draw the table visually, with the example input {1, 2, 3, 9}. Since the total is '15', therefore, we will try to find a subset whose sum is equal to the half of it i.e. 7.



The above visualization tells us that it is not possible to find a subset whose sum is equal to '7'. So what is the closest subset we can find? We can find such a subset if we start moving backward in the last row from the bottom right corner to find the first 'T'. The first 'T' in the above diagram is the sum '6', which means we can find a subset whose sum is equal to '6'. This means the other set will have a sum of '9', and the minimum difference will be '3'.

Code:

```
let canPartition = function (num) {
    const n = num.length;
    let sum = 0;
    for (let i = 0; i < n; i++) sum += num[i];

    const requiredSum = Math.floor(sum / 2);

    const dp = Array(n)
        .fill(false)
        .map(() => Array(requiredSum + 1).fill(false));

    // populate the sum=0 columns, as we can always form '0' sum with an
empty set
    for (let i = 0; i < n; i++) dp[i][0] = true;

    // with only one number, we can form a subset only when the required
sum is equal to that number
    for (let s = 1; s <= requiredSum; s++) {
        dp[0][s] = num[0] === s;
    }

    // prrocess all subsets for all sums
    for (let i = 1; i < n; i++) {
        for (let s = 1; s <= requiredSum; s++) {
            if (dp[i - 1][s]) {
```

```
                dp[i][s] = dp[i - 1][s];
            } else if (s >= num[i]) {
                dp[i][s] = dp[i - 1][s - num[i]];
            }
        }
    }

    let sum1 = 0;
    for (let i = requiredSum; i >= 0; i--) {
        if (dp[n - 1][i]) {
            sum1 = i;
            break;
        }
    }

    const sum2 = sum - sum1;
    return Math.abs(sum2 - sum1);
};

console.log(`Minimum subset difference is: ---> ${canPartition([1, 2, 3,
9])}`);
console.log(`Minimum subset difference is: ---> ${canPartition([1, 2, 7,
1, 5])}`);
console.log(`Minimum subset difference is: ---> ${canPartition([1, 3, 100,
4])}`);
```

This solution has time and space complexity of `O(N*S)` where 'N' represents total numbers and 'S' is the sum of all numbers