Target Sum

You are given a list of non-negative integers, a1, a2, ..., an, and a target, S. Now you have 2 symbols `+` and `-`. For each integer, you should choose one from `+` and `-` as its new symbol.
Find out how many ways to assign symbols to make sum of integers equal to target S.

Example 1:

```
Input: nums is [1, 1, 1, 1, 1], S is 3.
Output: 5
Explanation:

-1+1+1+1+1 = 3
+1-1+1+1+1 = 3
+1+1-1+1+1 = 3
+1+1+1-1+1 = 3
+1+1+1+1-1 = 3

There are 5 ways to assign symbols to make the sum of nums be target 3.
```

Note:

1. The length of the given array is positive and will not exceed 20.
2. The sum of elements in the given array will not exceed 1000.
3. Your output answer is guaranteed to be fitted in a 32-bit integer.

How To Solve ?
We will use **FAST** method of dp to solve this
Steps to solve:

1. **F**irst solution

Think of a simple recursive solution targetSum(nums,i,sum,T)
where we either add nums[i] to sum OR subtract nums[i] from sum
then call recursively
BASE case :
if i == nums.size()
check if sum == T : if yes return 1 , else return 0

Recursive solution code:

```
public class Solution {
    int count = 0;
    public int findTargetSumWays(int[] nums, int S) {
        calculate(nums, 0, 0, S);
        return count;
    }
    public void calculate(int[] nums, int i, int sum, int S) {
        if (i == nums.length) {
            if (sum == S)
                count++;
        } else {
            calculate(nums, i + 1, sum + nums[i], S);
            calculate(nums, i + 1, sum - nums[i], S);
        }
    }
}
```

---

2. **A**nalyze the solution
We observe that at each step we are making two calls :
Hence T(n) = 2T(n-1)
complexity : O(2^n)
Optimal Substructure : Solution to ith index depends on i+1 th index
Overlapping subproblems : same (i,sum) may be called several times as can be seen below:

3. Find **S**ubproblems

In this step we find the memoized solution
here we go :

```
public class Solution {
    int count = 0;
    public int findTargetSumWays(int[] nums, int S) {
        int[][] memo = new int[nums.length][2001];
        for (int[] row: memo)
            Arrays.fill(row, Integer.MIN_VALUE);
        return calculate(nums, 0, 0, S, memo);
    }
    public int calculate(int[] nums, int i, int sum, int S, int[][] memo) {
        if (i == nums.length) {
            if (sum == S)
                return 1;
            else
                return 0;
        } else {
            if (memo[i][sum + 1000] != Integer.MIN_VALUE) {
                return memo[i][sum + 1000];
            }
            int add = calculate(nums, i + 1, sum + nums[i], S, memo);
            int subtract = calculate(nums, i + 1, sum - nums[i], S, memo);
            memo[i][sum + 1000] = add + subtract;
            return memo[i][sum + 1000];
        }
    }
}
```

4. **T**urn the solution

In this step we convert the top-down solution to bottom up solution

```
public class Solution {
    public int findTargetSumWays(int[] nums, int S) {
        int[] dp = new int[2001];
        dp[nums[0] + 1000] = 1;
        dp[-nums[0] + 1000] += 1;
        for (int i = 1; i < nums.length; i++) {
            int[] next = new int[2001];
            for (int sum = -1000; sum <= 1000; sum++) {
                if (dp[sum + 1000] > 0) {
                    next[sum + nums[i] + 1000] += dp[sum + 1000];
                    next[sum - nums[i] + 1000] += dp[sum + 1000];
                }
            }
            dp = next;
        }
        return S > 1000 ? 0 : dp[S + 1000];
    }
}
```