# 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.

## Solution 1

The recursive solution is very slow, because its runtime is exponential.

This problem can be converted to subset sum problem, by adding `nums` sum to target, then doubling every number of `nums`

For example:

```
// adding (1+2+3+4+5) to both sides gives subset sum problem
original:  1 2 3 4 5   target =  3
          +1+2+3+4+5          = 15
sum:       2 4 6 8 10  target = 18

// choosing + sign for 3 is like including 6 in the subset
// choosing - sign for 4 is like excluding 8 in the subset
original: +1-2+3-4+5   target =  3
          +1+2+3+4+5          = 15
sum:       2 0 6 0 10  target = 18
```

Each solution to the subset sum problem should correspond to one solution to the original problem
The numbers with `+` sign are in the subset.
The numbers with `-` sign are not in the subset.
In the above example, the subset `[2, 6, 10]` should correspond to `+1 +3 +5`

Then we can use subset sum solver to solve the problem

Here is a Java solution based on the idea above using two-dimensional `dp` (30 ms)
Here `dp[i][j]` means number of ways to get sum `i` using subsets of first `j` numbers

```
    public int findTargetSumWays(int[] nums, int s) {
        int sum = 0;
        for (int i = 0; i < nums.length; i++) {
            sum += nums[i];
            nums[i] += nums[i];
        }
        return sum < s ? 0 : subsetSum(nums, s + sum);
    }

    public int subsetSum(int[] nums, int s) {
        int n = nums.length, dp[][] = new int[s + 1][n + 1];
        dp[0][0] = 1;
        for (int i = 0; i <= s; i++) {
            for (int j = 1; j <= n; j++) {
                dp[i][j] = dp[i][j - 1];
                if (i - nums[j - 1] >= 0)
                    dp[i][j] += dp[i - nums[j - 1]][j - 1];
            }
        }
        return dp[s][n];
    }
```

Thanks to @BrunoDeNadaiSarnaglia's suggestion on optimization
The optimization is based on the observation that all numbers in the subset sum problem are even
Here we divide both sides by two and we now have the original `nums` back

```
// subset sum
2 4 6 8 10  target = 18
2   6   10
// subset sum divided by two
1 2 3 4 5   target = 9
1   3   5
```

Here is a much faster Java solution using the optimization and one-dimensional `dp` (15 ms)

```
    public int findTargetSumWays(int[] nums, int s) {
        int sum = 0;
        for (int n : nums)
            sum += n;
        return sum < s || (s + sum) % 2 > 0 ? 0 : subsetSum(nums, (s + sum) >>> 1
);
    }

    public int subsetSum(int[] nums, int s) {
        int[] dp = new int[s + 1];
        dp[0] = 1;
        for (int i = 0; i < nums.length; i++)
            for (int j = s; j >= nums[i]; j--)
                dp[j] += dp[j - nums[i]];
        return dp[s];
    }
```

Here is C++ solution (3 ms)

```
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int s) {
        int sum = accumulate(nums.begin(), nums.end(), 0);
        return sum < s || (s + sum) & 1 ? 0 : subsetSum(nums, (s + sum) >> 1);
    }

    int subsetSum(vector<int>& nums, int s) {
        int dp[s + 1] = { 0 };
        dp[0] = 1;
        for (int i = 0; i < nums.size(); i++)
            for (int j = s; j >= nums[i]; j--)
                dp[j] += dp[j - nums[i]];
        return dp[s];
    }
};
```

written by yuxiangmusic original link here

## Solution 2

```java
public class Solution {
    public int findTargetSumWays(int[] nums, int s) {
        int sum = 0;
        for(int i: nums) sum+=i;
        if(s>sum || s<-sum) return 0;
        int[] dp = new int[2*sum+1];
        dp[0+sum] = 1;
        for(int i = 0; i<nums.length; i++){
            int[] next = new int[2*sum+1];
            for(int k = 0; k<2*sum+1; k++){
                if(dp[k]!=0){
                    next[k + nums[i]] += dp[k];
                    next[k - nums[i]] += dp[k];
                }
            }
            dp = next;
        }
        return dp[sum+s];
    }
}
```

Please up vote if it helped, so that more people can see it. ;)

written by chidong original link here

## Solution 3

This is a pretty easy problem. Just do DFS and try both "+" and "-" at every position. Easy version of `Expression Add Operators`
https://leetcode.com/problems/expression-add-operators/

```
public class Solution {
    int result = 0;

    public int findTargetSumWays(int[] nums, int S) {
        if (nums == null || nums.length == 0) return result;
        helper(nums, S, 0, 0);
        return result;
    }

    public void helper(int[] nums, int target, int pos, long eval){
        if (pos == nums.length) {
            if (target == eval) result++;
            return;
        }
        helper(nums, target, pos + 1, eval + nums[pos]);
        helper(nums, target, pos + 1, eval - nums[pos]);
    }
}
```

Optimization: The idea is `If the sum of all elements left is smaller than absolute value of target, there will be no answer following the current path. Thus we can return.`

```
public class Solution {
    int result = 0;

    public int findTargetSumWays(int[] nums, int S) {
        if(nums == null || nums.length == 0) return result;

        int n = nums.length;
        int[] sums = new int[n];
        sums[n - 1] = nums[n - 1];
        for (int i = n - 2; i >= 0; i--)
            sums[i] = sums[i + 1] + nums[i];

        helper(nums, sums, S, 0);
        return result;
    }
    public void helper(int[] nums, int[] sums, int target, int pos){
        if(pos == nums.length){
            if(target == 0) result++;
            return;
        }

        if (sums[pos] < Math.abs(target)) return;

        helper(nums, sums, target + nums[pos], pos + 1);
        helper(nums, sums, target - nums[pos], pos + 1);
    }
}
```

written by shawngao original link here