<https://leetcode.com/problems/target-sum/>

You are given an integer array nums and an integer target.

You want to build an **expression** out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.

* For example, if nums = [2, 1], you can add a '+' before 2 and a '-' before 1 and concatenate them to build the expression "+2-1".

Return the number of different **expressions** that you can build, which evaluates to target.

**Example 1:**

Input: nums = [1,1,1,1,1], target = 3

Output: 5

Explanation: There are 5 ways to assign symbols to make the sum of nums be target 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

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

**Example 2:**

Input: nums = [1], target = 1

Output: 1

**Constraints:**

* 1 <= nums.length <= 20
* 0 <= nums[i] <= 1000
* 0 <= sum(nums[i]) <= 1000
* -1000 <= target <= 1000

**Attempt 1: 2022-12-18**

**Solution 1:  Native DFS (10 min)**

**Style 1: Recursive traversal with global variable**

class Solution {

// Global variable

int count = 0;

public int findTargetSumWays(int[] nums, int target) {

helper(nums, target, 0, 0);

return count;

}

private void helper(int[] nums, int target, int sum, int index) {

if(index == nums.length) {

if(target == sum) {

count++;

}

return;

}

helper(nums, target, sum + nums[index], index + 1);

helper(nums, target, sum - nums[index], index + 1);

}

}

Time Complexity : O(2^N), size of recursion tree will be 2^n. n refers to the size of nums array

Space Complexity : O(N), the depth of the recursion tree can go up to n

**Style 2: Divide and Conquer with return result**

class Solution {

public int findTargetSumWays(int[] nums, int target) {

return helper(nums, target, 0, 0);

}

private int helper(int[] nums, int target, int sum, int index) {

if(index == nums.length) {

if(target == sum) {

return 1;

}

return 0;

}

// Divide

int add\_element = helper(nums, target, sum + nums[index], index + 1);

int subtract\_element = helper(nums, target, sum - nums[index], index + 1);

// Conquer

return add\_element + subtract\_element;

}

}

Time Complexity : O(2^N), size of recursion tree will be 2^n. n refers to the size of nums array

Space Complexity : O(N), the depth of the recursion tree can go up to n

**Solution 2:  DFS with Memoization (Top down DP) (10 min)**

**Style 1: Initialize memo as Integer[][] and check NULL to identify if value stored before**

=============================================================================

Memo quick return after index == nums.length no additional index check needed

=============================================================================

class Solution {

public int findTargetSumWays(int[] nums, int target) {

int totalSum = 0;

for(int num : nums) {

totalSum += num;

}

Integer[][] memo = new Integer[nums.length][2 \* totalSum + 1];

return helper(nums, target, 0, 0, memo, totalSum);

}

private int helper(int[] nums, int target, int sum, int index, Integer[][] memo, int totalSum) {

if(index == nums.length) {

if(target == sum) {

return 1;

}

return 0;

}

// Quick return

// The factor of totalSum has been added as an offset to the sum value to

// map all the sumsumsums possible to positive integer range

if(memo[index][sum + totalSum] != null) {

return memo[index][sum + totalSum];

}

// Divide

int add\_element = helper(nums, target, sum + nums[index], index + 1, memo, totalSum);

int subtract\_element = helper(nums, target, sum - nums[index], index + 1, memo, totalSum);

// Conquer

int cur\_sum = add\_element + subtract\_element;

// Store into memo

memo[index][sum + totalSum] = cur\_sum;

return cur\_sum;

}

}

=============================================================================

Memo quick return before index == nums.length additional index check needed

=============================================================================

class Solution {

public int findTargetSumWays(int[] nums, int target) {

int totalSum = 0;

for(int num : nums) {

totalSum += num;

}

Integer[][] memo = new Integer[nums.length][2 \* totalSum + 1];

return helper(nums, target, 0, 0, memo, totalSum);

}

private int helper(int[] nums, int target, int sum, int index, Integer[][] memo, int totalSum) {

// Quick return

// The factor of totalSum has been added as an offset to the sum value to

// map all the sumsumsums possible to positive integer range

if(index < nums.length && memo[index][sum + totalSum] != null) {

return memo[index][sum + totalSum];

}

if(index == nums.length) {

if(target == sum) {

return 1;

}

return 0;

}

// Divide

int add\_element = helper(nums, target, sum + nums[index], index + 1, memo, totalSum);

int subtract\_element = helper(nums, target, sum - nums[index], index + 1, memo, totalSum);

// Conquer

int cur\_sum = add\_element + subtract\_element;

// Store into memo

memo[index][sum + totalSum] = cur\_sum;

return cur\_sum;

}

}

Time complexity: O(t\*n). The memo array of size O(t\*n) has been filled just once. Here, t refers to the sum of the nums array and n refers to the length of the nums array

Space complexity: O(t\*n). The depth of recursion tree can go up to n. The memo array contains t\*n elements

**Style 2: Initialize memo as int[][] & Integer.MIN\_VALUE and check if value changed to identify if value stored before**

Memo quick return after index == nums.length no additional index check needed

class Solution {

public int findTargetSumWays(int[] nums, int target) {

int totalSum = 0;

for(int num : nums) {

totalSum += num;

}

int[][] memo = new int[nums.length][2 \* totalSum + 1];

for(int[] row : memo) {

Arrays.fill(row, Integer.MIN\_VALUE);

}

return helper(nums, target, 0, 0, memo, totalSum);

}

private int helper(int[] nums, int target, int sum, int index, int[][] memo, int totalSum) {

if(index == nums.length) {

if(target == sum) {

return 1;

}

return 0;

}

// Quick return

// The factor of totalSum has been added as an offset to the sum value to

// map all the sumsumsums possible to positive integer range

if(memo[index][sum + totalSum] != Integer.MIN\_VALUE) {

return memo[index][sum + totalSum];

}

// Divide

int add\_element = helper(nums, target, sum + nums[index], index + 1, memo, totalSum);

int subtract\_element = helper(nums, target, sum - nums[index], index + 1, memo, totalSum);

// Conquer

int cur\_sum = add\_element + subtract\_element;

// Store into memo

memo[index][sum + totalSum] = cur\_sum;

return cur\_sum;

}

}

=============================================================================

Memo quick return before index == nums.length additional index check needed

=============================================================================

class Solution {

public int findTargetSumWays(int[] nums, int target) {

int totalSum = 0;

for(int num : nums) {

totalSum += num;

}

int[][] memo = new int[nums.length][2 \* totalSum + 1];

for(int[] row : memo) {

Arrays.fill(row, Integer.MIN\_VALUE);

}

return helper(nums, target, 0, 0, memo, totalSum);

}

private int helper(int[] nums, int target, int sum, int index, int[][] memo, int totalSum) {

// Quick return

// The factor of totalSum has been added as an offset to the sum value to

// map all the sumsumsums possible to positive integer range

if(index < nums.length && memo[index][sum + totalSum] != Integer.MIN\_VALUE) {

return memo[index][sum + totalSum];

}

if(index == nums.length) {

if(target == sum) {

return 1;

}

return 0;

}

// Divide

int add\_element = helper(nums, target, sum + nums[index], index + 1, memo, totalSum);

int subtract\_element = helper(nums, target, sum - nums[index], index + 1, memo, totalSum);

// Conquer

int cur\_sum = add\_element + subtract\_element;

// Store into memo

memo[index][sum + totalSum] = cur\_sum;

return cur\_sum;

}

}

Time complexity: O(t\*n). The memo array of size O(t\*n) has been filled just once. Here, t refers to the sum of the nums array and n refers to the length of the nums array

Space complexity: O(t\*n). The depth of recursion tree can go up to n. The memo array contains t\*n elements

**Style 3: Initialize memo as HashMap as {key = index + current sum string, value = count}**

class Solution {

public int findTargetSumWays(int[] nums, int target) {

int totalSum = 0;

for(int num : nums) {

totalSum += num;

}

Map<String, Integer> memo = new HashMap<String, Integer>();

return helper(nums, target, 0, 0, memo);

}

private int helper(int[] nums, int target, int sum, int index, Map<String, Integer> memo) {

if(index == nums.length) {

if(target == sum) {

return 1;

}

return 0;

}

// Quick return

if(memo.containsKey(index + "\_" + sum)) {

return memo.get(index + "\_" + sum);

}

// Divide

int add\_element = helper(nums, target, sum + nums[index], index + 1, memo);

int subtract\_element = helper(nums, target, sum - nums[index], index + 1, memo);

// Conquer

int cur\_sum = add\_element + subtract\_element;

// Store into memo

memo.put(index + "\_" + sum, cur\_sum);

return cur\_sum;

}

}

Time complexity: O(t\*n). The memo array of size O(t\*n) has been filled just once. Here, t refers to the sum of the nums array and n refers to the length of the nums array

Space complexity: O(t\*n). The depth of recursion tree can go up to n. The memo array contains t\*n elements

**Solution 3: 2D bottom up DP (60 min)**

**Wrong Solution**

Input [0,0,0,0,0,0,0,0,1]  1

Output 0

Expected 256

class Solution {

public int findTargetSumWays(int[] nums, int S) {

int sum = 0;

for(int i = 0; i < nums.length; i++) {

sum += nums[i];

}

if(sum < S || (sum + S) % 2 == 1) {

return 0;

}

int target = (sum + S) / 2;

return helper(nums, target);

}

private int helper(int[] nums, int target) {

int[][] dp = new int[nums.length][1 + target];

for(int i = 0; i < nums.length; i++) {

dp[i][0] = 1;

}

for(int i = 1; i <= target; i++) {

dp[0][i] += (nums[0] == i ? 1 : 0);

}

for(int i = 1; i < nums.length; i++) {

for(int j = 0; j <= target; j++) {

dp[i][j] = dp[i - 1][j];

if(j >= nums[i - 1]) {

dp[i][j] += dp[i - 1][j - nums[i - 1]];

}

}

}

return dp[nums.length - 1][target];

}

}

Explain why we need 1 more dummy row when creating 2D array ?

Refer to

https://leetcode.com/problems/target-sum/discuss/278526/Java-1D-and-2D-DP-Solution

https://leetcode.com/problems/target-sum/discuss/278526/Java-1D-and-2D-DP-Solution/355606

The problem is that our previous solution (for 2D) works for positive numbers.

However the problem (and the test case you mentioned has non-negative numbers,

which includes 0's). That changes the calculation as shown below :

With old solution, input\sum : [0,0,1]\1, here is the table :

i\s 0 1

[0] 1 0

[0,0] 1 0

[0,0,1] 1 1

Here adding a new 0, doesn't change the number of subsets to achieve 0's.

Also, an empty set can achieve 0. A set with just {0} can achieve sum of

0 in 2 ways : by choosing empty set and by choosing {0}. Similarly if we

have another 0, it should change the number of subsets.

With new solution, input\sum : [0,0,0,1]\1, here is the table :

i\s 0 1

[] 1 0

[0] 2 0

[0,0] 4 0

[0,0,0] 8 0

[0,0,0,1] 8 8

=========================================================================

great catch for the root cause, so if we adding 1 row as [1 + nums.length]

will good for handling 0 (non-negative) presented in array case

i\s 0 1

[] 1 0 -> The dummy row ('1' in [1 + nums.length]) will handle the empty set option

[0] 2 0

Like you said, especially compare to 416. Partition Equal Subset Sum, the condition

there is only positive integer in array, but here we have 0 which enable empty set option.

So just use int[][] dp = new int[1 + nums.length][1 + target] to initialize the dp array

**Correct Solution**

class Solution {

public int findTargetSumWays(int[] nums, int target) {

int sum = 0;

for(int num : nums) {

sum += num;

}

// Additional check 'sum + target < 0' raised by

// Input: nums = [100], target = -200

if(sum < target || sum + target < 0 || (sum + target) % 2 == 1) {

return 0;

}

int newTarget = (sum + target) / 2;

// dp[i][j] means number of ways to get newTarget j with first i elements from nums.

int[][] dp = new int[1 + nums.length][1 + newTarget];

// populate the newTarget=0 columns, as we can always for '0'

// newTarget with an empty set

for(int i = 0; i <= nums.length; i++) {

dp[i][0] = 1;

}

// No need to separately initialize dp[0][i] since any newTarget

// cannot form by 0 element, just merge into formulas, i start

// from 1 since we want keep dp[0][0] = 1, which means 0 element

// to form newTarget = 0 is a reasonable solution

//for(int i = 0; i <= newTarget; i++) {

// dp[0][i] = 0;

//}

for(int i = 1; i <= nums.length; i++) {

for(int j = 0; j <= newTarget; j++) {

// If we can get the newTarget 'j' without the number at index 'i - 1'

dp[i][j] = dp[i - 1][j];

// else if we can find a subset to get the remaining sum

if(j >= nums[i - 1]) {

dp[i][j] += dp[i - 1][j - nums[i - 1]];

}

}

}

return dp[nums.length][newTarget];

}

}

**Explain on how to transform "Target Sum" problem into "Find a subset of nums that need to be positive"**

The original problem statement is equivalent to:

Find a subset of nums that need to be positive, and the rest of them negative, such that the sum is equal to target

Let P be the positive subset and N be the negative subset

For example:

Given nums = [1, 2, 3, 4, 5] and target = 3 then one possible solution is +1-2+3-4+5 = 3

Here positive subset is P = [1, 3, 5] and negative subset is N = [2, 4]

Then let's see how this can be converted to a subset sum problem:

sum(P) - sum(N) = target

sum(P) + sum(N) + sum(P) - sum(N) = target + sum(P) + sum(N)

2 \* sum(P) = target + sum(nums)

So the original problem has been converted to a subset sum problem as follows:

Find a subset P of nums such that sum(P) = (target + sum(nums)) / 2

Note that the above formula has proved that target + sum(nums) must be even

**Solution 4: 1D bottom up DP (60 min)**

class Solution {

public int findTargetSumWays(int[] nums, int target) {

int sum = 0;

for(int num : nums) {

sum += num;

}

if(sum < target || target + sum < 0 || (target + sum) % 2 == 1) {

return 0;

}

int[] dp = new int[(target + sum) / 2 + 1];

dp[0] = 1;

for(int i = 0; i < nums.length; i++) {

for(int j = (target + sum) / 2; j >= 0; j--) {

if(j >= nums[i]) {

dp[j] += dp[j - nums[i]];

}

}

}

return dp[(target + sum) / 2];

}

}

**Refer to**

<https://leetcode.com/problems/target-sum/solutions/245073/Java-solution-in-Chinese/>

**S1:直接递归**

题目中指出，将数组中的每个数前面加上一个符号（+ 或 -），使得它们构成一个运算式，并且其计算结果等于给定的 S ，求符号的添加种类。

那么也就是说，对于每一个数字，我们可以选择加上它或者减去它，于是，可以直接使用递归，进行2^n次运算，求出所有的符号添加方式，然后再求其中结果为 S 的即可：

public static int findTargetSumWays(int[] nums, int S) {

return ways(nums, S, 0, 0);

}

private static int ways(int[] nums, int S, int pos, int cur) {

if (pos == nums.length) {

return cur == S ? 1 : 0;

} else {

return ways(nums, S, pos+1, cur+nums[pos])

+ ways(nums, S, pos+1, cur-nums[pos]);

}

}

**S2:缓存递归**

上述递归很有可能会增加很多不必要的计算，如对于一个数组[1,1,1,1,1,1]，如果前两个数字的符号分别是[-,+]和[+,-]，那么计算到第三个数字的时候，这两种符号添加方式指向了同一种情况，如果不加处理，肯定要造成后面的[1,1,1,1]的重复计算，此时可以考虑给递归函数加一个缓存。

每个递归函数有两个变量：

1. 当前位置
2. 当前的运算结果

所以初步考虑缓存应该是一个二维数组，此时就要判断这两个变量各自的取值范围：

1. 对于当前的位置，肯定是要在 nums 里面，也就是说，它的范围是 0～n-1
2. 由于数组中所有的数字都是正数，那么必然所有符号取 + 结果最大，所有符号取 - 结果最小，题目里面也指出，所有的数字和是不大于 1000 的，所以，范围取上下 1000 即可，也就是 2001

那么，添加了缓存之后就是：

public static int findTargetSumWays(int[] nums, int S) {

int[][] saved = new int[nums.length][2001];

for (int[] row : saved) {

Arrays.fill(row, -1);

}

return ways2(nums, S, 0, 0, saved);

}

private static int ways2(int[] nums, int S, int pos, int cur, int[][] saved) {

if (pos == nums.length) {

return cur == S ? 1 : 0;

} else {

if (saved[pos][cur+1000] >= 0) return saved[pos][cur+1000];

int ways = ways2(nums, S, pos+1, cur+nums[pos], saved)

+ ways2(nums, S, pos+1, cur-nums[pos], saved);

saved[pos][cur+1000] = ways;

return ways;

}

}

1000 是数组中数字和的上限，如果想要缩减一下内存使用量，也可以直接计算出数字和。

**S3: 动态规划**

动态规划也需要一个数组保存记录值，与递归中使用的缓存数组类似。对于一个数组[1,...,n]来说，如果当前待判断位置是 i ，也就是说已经计算出了前面[1,...,i-1]这部分子数组的结果，每个数字有两种符号添加，也就是说总的应该有2^{i-1}种结果（可能会有重合的），那么在这2^{i-1}种结果加上（或者减去）nums[i] 之后，就会得到2^i种新的结果（当然，有可能在前面的2^{i-1}种结果中，存在 x y ，使得 x-nums[i] == y+nums[i] == cur），此时 cur 对应的解应该是 x y 两个数对应的解的和。那么当整个数组计算完成之后，就可以求出数字 S 对应的解。

public static int findTargetSumWays(int[] nums, int S) {

int sum = 0;

for (int num : nums) sum += num;

if (sum < S || ((S + sum) & 1) == 1) return 0;

int[] dp = new int[(sum<<1) + 1];

dp[nums[0] + sum] = 1;

dp[-nums[0] + sum] += 1;

for (int i = 1; i < nums.length; i++) {

int[] next = new int[(sum<<1) + 1];

for (int j = -sum; j <= sum; j++) {

if (dp[j + sum] > 0) {

next[j + sum + nums[i]] += dp[j + sum];

next[j + sum - nums[i]] += dp[j + sum];

}

}

dp = next;

}

return dp[S + sum];

}

如上，第一个循环是遍历数组，第二个循环则是遍历以求出的2^{i-1}种结果，并将其分别加上（减去）nums[i] 以求出下一个位置的2^i种结果，当然，上述结果肯定是存在重合的，因为所有的运算结果都是在 -sum～sum 的范围内，所以当我们需要遍历已求出的2^{i-1}种结果时，由于会有重合，我们并不知道到底有多少种结果，这时的处理办法就是遍历整个可能的结果（从 -sum 到 sum），当这个结果对应的解不为 0 时，就意味着这是一个2^{i-1}个结果中的一个。

所以，本题中首先给执行了dp[nums[0] + sum] = 1;和dp[-nums[0] + sum] += 1;这两个代码，就是为了先求出2^1的结果，然后之后求第二个位置的2^2的结果的时候，才能利用已求出的结果。

另外，计算下一个位置的结果的时候的代码next[j + sum + nums[i]] += dp[j + sum];中使用的是+=符号，就是将所有导向统一结果的解相加

还有，S 与 sum 是否有什么关系？为什么要判断((S + sum) & 1) == 1？

本题中每个数字前都有一个符号，+ 或 - ，如果将整个数组按照数字之前的符号分成两部分，一部分的符号全是 + ，另一部分的符号全是 - ，然后让两部分的和分别是 x y ，那么有关系式：

x + y = sum

x - y = S

于是有x + x = S + sum = 2 \* x，于是得出结论：S + sum 必然是偶数，所以才有了本题开始的判断。

**S4:划分数组 & 动态规划**

上面的这个关系式S + sum = 2 \* x也可以加以利用，得出(S + sum) / 2 = x，也就是说本题可以变为，使 nums 数组的子数组的和等于 S+sum 的一半，求出符合条件的子数组的个数。

于是有：

public static int findTargetSumWays(int[] nums, int S) {

int sum = 0;

for (int num : nums) sum += num;

if (sum < S || ((sum + S) & 1) == 1) return 0;

sum = (sum + S) >> 1;

int[] dp = new int[sum+1];

dp[0] = 1;

for (int num : nums) {

for (int i = sum; i >= num; i--) {

dp[i] += dp[i-num];

}

}

return dp[sum];

}

此时，外层是对数组的循环，内层通过一个 i=num～sum 的循环，求出所有加上当前位置的 num 能到得到 i 的解。外层对数组的循环，每到一个数字 num ，就是指对于子数组[0,...,num]来说，dp[i] 存放着数组[0,...,num]中和为 i 的子数组的个数，那么外层对数组的循环完了之后，dp[i] 存放的就是数组 nums 的和为 i 的子数组的个数，所以，此时 dp[sum] 就是本题的解。

其中，起始位置的dp[0] = 1，是整个计算的开端，由 0+num=num ，得到dp[num] = 1，然后以此类推。