Problem Statement

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`

Solution

We solve Target Sum with dynamic programming.

We have some numbers. The problem states we must make a choice for each number: give it a positive or negative sign. Our goal is to sum these signed numbers to a target (and then count the number of times we reach our goal).

First, let's consider an alternative view of the problem to make it easier to solve. We will start with the numbers all added together, and view our choice as between keeping the sign for a number positive or swapping the sign to negative.[1] The goal now becomes: "starting with the numbers all added together, flip signs until we get to the original target." Or equivalently, let the new target, a "bank" we are subtracting from, be set to the sum of the numbers minus the old target: we reach our goal when this bank reaches 0.[2]

We can now define our DP recurrence. Let target_sum_ways(i, bank) be the number of distinct ways to get to bank with a summed nums[0..i) by flipping signs in that nums[0..i). The recurrence follows:

$$\begin{aligned}
\text{target_sum_ways}(0, 0) &= 1 \\
\text{target_sum_ways}(0, *) &= 0 \\
\text{target_sum_ways}(i, \text{bank}) &= \text{sum}\begin{cases}\text{target_sum_ways}(i-1, \text{bank}) \\ \text{target_sum_ways}(i-1, \text{bank}'), \text{ or } 0 \text{ if bank}' < 0\end{cases} \\
\text{where bank}' &= \text{bank} - 2 \cdot \text{nums}[i-1]
\end{aligned}$$

Notice we have our two cases: don't flip sign (keep bank) or do flip sign (going from a positive to a negative is like subtracting that number from bank twice, but as defined, we can't do this if our bank goes below 0).[3]

To solve, just use target_sum_ways with i = `nums.len()` and bank as described above. A solution in Rust, with memory-optimized DP,[4] follows:

```
fn find_target_sum_ways(nums: Vec<i32>, target: i32) -> i32 {
    let original_bank = nums.iter().sum::<i32>() - target;
    let mut target_sum_ways = vec![0; original_bank as usize + 1];
    target_sum_ways[0] = 1;
    for i in 1..=nums.len() {
        for bank in (0..target_sum_ways.len()).rev() {
            let bank_if_sign_swap = bank as i32 - 2 * nums[i - 1];
            if bank_if_sign_swap >= 0 {
                target_sum_ways[bank] +=
                  target_sum_ways[bank_if_sign_swap as usize];
            }
        }
    }
    target_sum_ways[target_sum_ways.len() - 1]
}
```

[1] The sum of these numbers will never increase, because we only ever stay the same from keeping a number positive or subtract from the sum by swapping a number to a negative. This property will be a surprise tool that will help us later.

[2] Notice that this is now reminiscent of 322. Coin Change: https://leetcode.com/problems/coin-change/description/.

[3] This is the "surprise tool"; We can easily understand which when choices aren't possible because the sum of the numbers will never increase. Determining this is more difficult if we continued with the original problem formulation.

[4] Notice the direction of dependencies. We can just compute row by row. We do need to iterate over the bank dimension in reverse so we don't influence future DP cells.