LeetCode P494 解题总结（DP）

题目

给定一个非负整数数组，a1, a2, ..., an，和一个目标数，S。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 -中选择一个符号添加在前面。
https://leetcode-cn.com/problems/target-sum/

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

示例 1:
输入: nums: [1, 1, 1, 1, 1], S: 3
输出: 5
解释:

-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

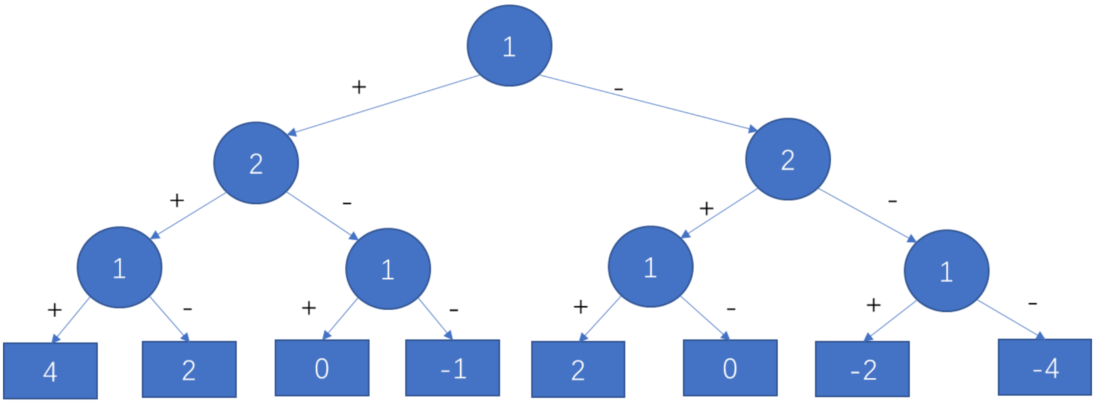
一共有 5 种方法让最终目标和为 3。

注意:

1. 数组非空，且长度不会超过 20。
2. 初始的数组的和不会超过 1000。
3. 保证返回的最终结果能被 32 位整数存下。

解法一（回溯）:

回溯，回溯的方式比较简单，数组中每个元素可以选正号或负号两种，按照 2 叉树的形式组织，以{1,2,1}输入为例：



每个内部节点代表当前数组的元素，内部节点的左子树代表当前节点选择为正号时的决策子树，右子树代表选为负号的决策子树，叶子节点代表当前的决策路径的最终结果。问题即为求解：有多少个叶子节点上的值等于给定的值。

通过回溯可以将叶子遍历出来，但是可以看到以上的解空间非常大，算法的复杂度即为叶子节点的数量，因为需要将叶子节点遍历完成。叶子节点的数量为2^n，其中 n 为二叉树高度-1，即为输入数组的长度。算法时间复杂度为$O(2^n)$。题目的输入已经做了限制，长度为 20，则叶子数量最多为$2^{20} = 4096$个。回溯也可以解决，但是不是最优解。

回溯的代码如下：

```
    private int count;
    private void backtrace(int index, int sum) {
        if (index == nums.length) {
            if (sum == target) count++;
            return;
        }

        sum += nums[index];
        backtrace(index+1, sum);
        sum -= nums[index];

        sum -= nums[index];
        backtrace(index+1, sum);
        sum += nums[index];
    }
```

解法二（动态规划）

此问题实际上与背包问题类似，取不同的符号就是背包取舍的问题，可以考虑用动态规划的方法解决。

动态规划的递归性质：

假设数组为a_1，a_2,⋯ a_n，需要判断整个数组加完符号后等于S的方案的个数。则为以下两种

情况之和:
1、 a_n取正号，那么方案个数等于a_1，a_2,⋯a_{n-1}加符号后等于$S - a_n$的方案个数
2、 a_n取负号，那么方案个数等于a_1，a_2,⋯a_{n-1}加符号后等于 $S + a_n$的方案个数
存在递归性质。而且a_1，a_2,⋯a_{n-1}的计算结果存在可重用的可能。

状态方程：
$f(i,s)$表示从数组下标$0\sim i$的数组元素加符号后等于 s 的方案个数，则:

$$f(i,s) \;=\; f(i-1, s-a_i) \;+\; f(i-1, s+a_i)$$

动态规划的套路，申请$n \times S$二维数组 dp（n 为数组长度，S 为输入值），通过数组从下到上进行递推，获取 dp[n-1][S]的值即为所求结果，但是以上的$s - a_i$可能存在负数，而且输入的 S 本身也可能为负数。需要对数组进行处理，把所有的下标都偏移为非负下标。

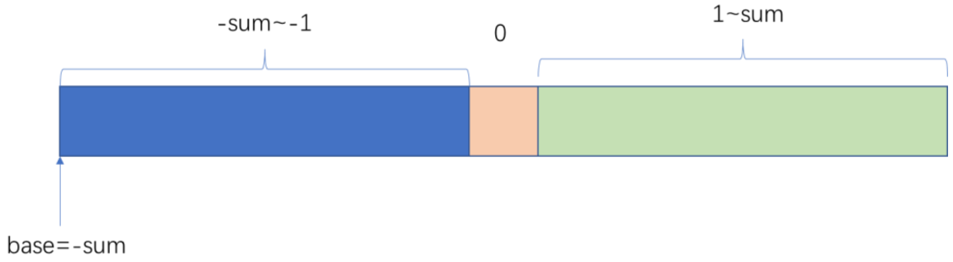
数组下标处理主要针对可能存在负数的维，题目已经有约束条件，说明数组中所有数组和不超过 1000，可以对数组进行简单偏移处理，而且题目说明了输入数组中为非负整数，证明其上下界都可以求解到:

$$[-Sum,\ Sum],\ Sum \;=\; \sum a_i$$

当符号全部取正时达到上界，全部取负号时达到下界。申请的数组维数变为:

$$n \times (2 \times Sum + 1)$$

其中加 1 是因为多了 0 的额外数据，数组第二维区间如下图所示:



数组下标（$index$）和实际值（$value$）之间通过 base 进行转化:

$$index \;=\; value - base$$
$$value \;=\; base \;+\; index$$

边界条件:
对于数组中的第一个元素a_i，可以构成值为a_i和$-a_i$的唯一的一个方案，所以:

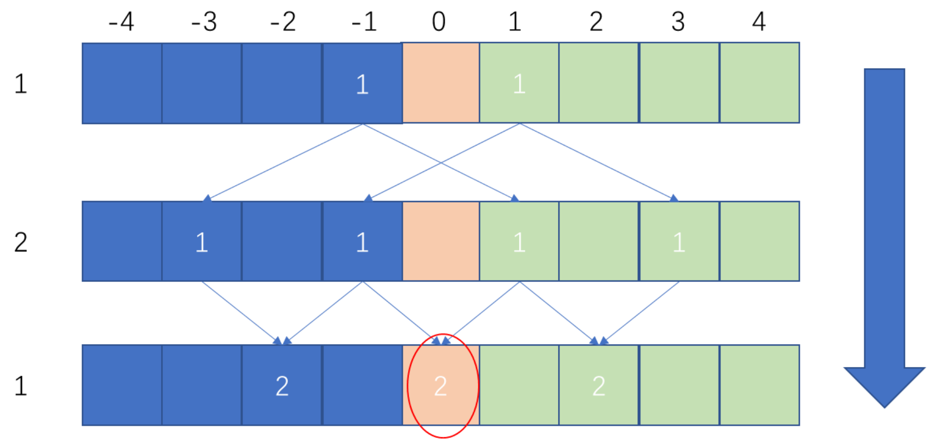
$$f(0, a_i) = 1$$
$$f(0, -a_i) = 1$$

注意，如果a_i为 0，则:

$$f(0,0) = 2$$

因为 0 加正号和负号结果都是一样的，有 2 种方案。其中 f 的结果经过数组下标处理后存储到二维数组中。

状态变化：
同样以{1,2,1}为例进行说明，求等于 0 的加符号方案数。算法示意图如下:



初始化:

$$f(0,-1) = 1$$
$$f(0,1) = 1$$

第一轮迭代:

$$f(1,-3) = f(0,-5) + f(0,-1) = 0 + 1 = 1$$
$$f(1,-1) = f(0,-3) + f(0,1) = 0 + 1 = 1$$
$$f(1,1) = f(0,-1) + f(0,3) = 1 + 0 = 1$$
$$f(1,3) = f(0,1) + f(0,5) = 1 + 0 = 1$$

第二轮迭代:

$$f(2,-2) = f(1,-3) + f(1,-1) = 1 + 1 = 2$$
$$f(2,0) = f(1,-1) + f(1,1) = 1 + 1 = 2$$
$$f(2,2) = f(1,1) + f(1,3) = 1 + 1 = 2$$

返回$f(2,0) = 2$

代码:

```
private int[][] dp;
private int base;

private int dp1(int[] nums, int target) {
    int maxSum = 0;
    for (int n : nums) {
        maxSum += n;
    }

    if (Math.abs(target) > maxSum ) {
        return 0;
    }
    this.nums = nums;
    dp = new int[nums.length][2*(maxSum + 1) + 1];
    base = Math.abs(maxSum);
    int v0 = nums[0];
    if (v0 == 0) {
```

```
			addDp(0, 0, 2);
		} else {
			addDp(0, v0, 1);
			addDp(0, -v0, 1);
		}
		System.out.println("dp table(update row#0):");
		printDP();
		for (int i = 1; i < nums.length; i++) {
			for (int j=0; j < dp[0].length; j++) {
				int val = offsetVal(j);
				int v1 = getDp(i-1, val-nums[i]);
				int v2 = getDp(i-1, val+nums[i]);
				int tmp = v1 + v2;
				dp[i][j] += tmp;
			}
			System.out.println("dp table(update row#"+i+"):");
			printDP();
		}
		return getDp(nums.length-1, target);
	}

	private int offsetVal(int v) {
		return v - base;
	}

	private int getDp(int index, int target) {
		int col = base + target;
		if (col >= dp[0].length || col < 0) {
			return 0;
		}
		return dp[index][col];
	}

	private void addDp(int index, int target, int val) {
		int col = base + target;
		if (col >= dp[0].length || col < 0) {
			return;
		}
		dp[index][col] += val;
	}

	private void printDP() {
		for (int i = 0; i < dp.length; i++) {
			System.out.format("%2d:", nums[i]);
```

```
            for (int j = 0; j < dp[0].length; j++) {
                System.out.format("%4d", dp[i][j]);
            }
            System.out.println();
        }
    }
```

算法复杂度：

从代码可以看出，算法经过两次循环，时间复杂度和空间复杂度为$O(kN)$，k为数组总和，N为数组长度。代码可进一步优化为一维数组。空间复杂度可优化到$O(k)$，相比直接回溯，有较大收益。