494. Target Sum

David Chen

2024 年 12 月 26 日

优化算法分析：LeetCode 494. Target Sum

问题描述

给定一个整数数组 `nums` 和一个整数 `target`，你需要为数组中的每个整数添加一个'+' 或'-' 符号，然后将所有整数连接起来，构建一个表达式。

返回能够构建出目标表达式的不同方法的数量。

示例 1:

```
Input: nums = [1,1,1,1,1], target = 3
Output: 5
Explanation: 有5种方法将符号添加到数组中，使得表达式的值为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
```

原始算法分析

以下是用户提供的 C++ 实现：

```cpp
int findTargetSumWays(vector<int> &nums, int target) {
    int n = nums.size();
    int res = 0;
    int sum = std::accumulate(nums.begin(), nums.end(), 0);
    if ((sum + target) % 2 != 0) return 0;
    if (abs(target) > abs(sum)) return 0;
    unordered_map<int, int> results_times;
    unordered_map<int, int> tmp;
    results_times[0] = 1;
    for (auto i: nums) {
        for (auto &k: results_times) {
            tmp[k.first + i] += k.second;
            tmp[k.first - i] += k.second;
```

```
            }
            results_times.swap(tmp);
            tmp.clear();
        }
        return results_times[target];
}
```

算法思路

1. ** 边界条件检查 **:

- 如果 $(\text{sum} + \text{target})$ 不是偶数，则无法通过划分为两个子集 P 和 N 满足 $P - N = \text{target}$ 和 $P + N = \text{sum}$，直接返回 0。
- 如果 $|\text{target}| > |\text{sum}|$，则无法通过加减符号达到目标，返回 0。

2. ** 动态规划（DP）方法 **:

- 使用两个哈希映射 `results_times` 和 `tmp` 来记录当前可能的所有和及其对应的方式数。
- 对于每个数字，更新可能的和：既可以加上当前数字，也可以减去当前数字。
- 最终返回目标和对应的方式数。

优化建议

原始算法使用了 `unordered_map` 进行动态规划，时间和空间复杂度较高。考虑到题目约束条件（$1 \leq \text{nums.length} \leq 20$，$0 \leq \text{nums}[i] \leq 1000$），我们可以采用更高效的 DP 方法。

1. 转化为子集和问题

目标和问题可以转化为子集和问题。设 P 为所有被赋予'+' 符号的数字的和，N 为被赋予'-' 符号的数字的和。则有：

$$P - N = \text{target}$$

$$P + N = \text{sum}$$

通过联立得：

$$P = \frac{\text{sum} + \text{target}}{2}$$

因此，问题转化为在数组中找到和为 P 的子集的个数。

2. 使用 1D 动态规划优化

采用一维 DP 数组，空间复杂度由 $O(n \times P)$ 降低为 $O(P)$，其中 $P = \frac{\text{sum}+\text{target}}{2}$。

3. 优化后的代码实现

以下是优化后的 C++ 实现：

```
int findTargetSumWays(vector<int> &nums, int target) {
    int sum = accumulate(nums.begin(), nums.end(), 0);

    // 检查边界条件
    if ((sum + target) % 2 != 0 || abs(target) > sum) return 0;

    int P = (sum + target) / 2;

    // 初始化 DP 数组
    vector<int> dp(P + 1, 0);
    dp[0] = 1;

    for (auto num : nums) {
        // 逆序遍历，避免重复计算
        for (int j = P; j >= num; --j) {
            dp[j] += dp[j - num];
        }
    }

    return dp[P];
}
```

优化说明

1. ** 边界条件检查 **:

- 确保 $(\text{sum} + \text{target})$ 为偶数且 $|\text{target}| \leq \text{sum}$，否则返回 0。

2. **DP 数组初始化 **:

- 使用一维数组 `dp`，其中 `dp[j]` 表示当前处理到某个数时，和为 j 的子集的个数。
- 初始时，$\texttt{dp[0]} = 1$，表示和为 0 的子集只有空集。

3. **DP 状态转移 **:

- 对于每个数字 `num`，从后向前遍历 `dp` 数组，更新 `dp[j]`。
- 逆序遍历避免了重复使用同一个数字。

4. ** 时间和空间复杂度 **:

- 时间复杂度：$\mathcal{O}(n \times P)$，其中 n 是数组长度，$P = \frac{\text{sum}+\text{target}}{2}$。
- 空间复杂度：$\mathcal{O}(P)$。

进一步优化

在特定情况下，可以进一步优化空间和时间复杂度：
1. ** 剪枝优化 **：

- 如果数组中存在大量的 0，可以提前统计 0 的数量，因为每个 0 都有两种选择（+0 或 -0），它们不会影响总和，但会成倍增加方式数。

2. ** 使用位运算 **（适用于更小的 P 值）：

- 利用位运算来压缩状态，但由于本题 P 的范围较大（$P \leq 1000 \times 20 = 20000$），此方法可能不适用。

完整优化后的代码

以下是包含剪枝优化的完整代码实现：

```
int findTargetSumWays(vector<int> &nums, int target) {
    int sum = accumulate(nums.begin(), nums.end(), 0);

    // 检查边界条件
    if ((sum + target) % 2 != 0 || abs(target) > sum) return 0;

    int P = (sum + target) / 2;

    // 统计0的数量
    int zeros = 0;
    vector<int> filtered;
    for(auto num : nums){
        if(num == 0){
            zeros++;
        }
        else{
            filtered.push_back(num);
        }
    }

    // 初始化 DP 数组
    vector<int> dp(P + 1, 0);
    dp[0] = 1;

    for (auto num : filtered) {
        // 逆序遍历，避免重复计算
        for (int j = P; j >= num; --j) {
            dp[j] += dp[j - num];
        }
```

```
    }

    // 每个0有两种选择，乘以2的零的数量
    return dp[P] * pow(2, zeros);
}
```

优化解释

1. ** 处理 0 的情况 **:

- 数组中每个 0 都可以选择 +0 或-0，两者对总和无影响，但会使得方式数翻倍。
- 因此，最后的结果需要乘以 2^{zeros}。

2. ** 过滤 0**:

- 将 0 从数组中移除，只处理非零数字，提高 DP 效率。

时间和空间复杂度分析

时间复杂度

$$\mathcal{O}(n \times P)$$

其中，n 是数组长度，$P = \frac{\text{sum}+\text{target}}{2}$。

空间复杂度

$$\mathcal{O}(P)$$

使用一维 DP 数组存储中间结果。

算法优缺点

优点

- **高效性**：将时间和空间复杂度从 $O(n \times 20000)$ 降低到可接受范围，适用于本题的约束条件。
- **简洁性**：代码结构简洁，易于理解和实现。
- **可扩展性**：适用于更大规模的问题，只需调整 DP 数组的大小。

缺点

- **空间限制**：当 P 较大时，DP 数组的空间消耗较高。
- **不适用于非整数问题**：该方法仅适用于整数数组和目标。

总结

通过将目标和问题转化为子集和问题，并采用一维动态规划优化算法，可以显著提升原始算法的效率和性能。特别是在处理包含 0 的数组时，通过统计和适当调整，进一步优化了算法的计算方式。该优化方法在时间和空间复杂度上均表现优异，适用于解决 LeetCode 494. Target Sum 等类似问题。