切换主题: 默认主题

题目地址(416. 分割等和子集)

https://leetcode-cn.com/problems/partition-equal-subset-sum/

入选理由

1. 背包换皮题，锻炼大家抽象能力

标签

- 动态规划
- DFS

难度

- 中等

题目描述

给定一个只包含正整数的非空数组。是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

```
给定一个只包含正整数的非空数组。是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

注意:

每个数组中的元素不会超过 100
数组的大小不会超过 200
示例 1:

输入: [1, 5, 11, 5]

输出: true

解释: 数组可以分割成 [1, 5, 5] 和 [11].


示例 2:

输入: [1, 2, 3, 5]

输出: false
```

```
解释：数组不能分割成两个元素和相等的子集.
```

思路

这次是背包的第一个题目，我们讲详细一点，之后就直接提重点信息。如果你还是没懂，建议回头再看讲义或者这篇题解。

抽象能力不管是在工程还是算法中都占据着绝对重要的位置。比如上题我们可以抽象为：

给定一个非空数组，和是 sum，能否找到这样的一个子序列，使其和为 sum/2

我们做过二数和，三数和， 四数和，看到这种类似的题会不会舒适一点，思路更开阔一点。

老司机们看到转化后的题，会立马想到背包问题，这里会提供**深度优先搜索**和**背包**两种解法。

深度优先遍历

我们再来看下题目描述，sum 有两种情况，

1. 如果 sum % 2 === 1, 则肯定无解，因为 sum/2 为小数，而数组全由整数构成，子数组和不可能为小数。

2. 如果 sum % 2 === 0, 需要找到和为 sum/2 的子集。

针对 2，我们要在 nums 里找到满足条件的子集 subNums。 这个过程可以类比为在一个大篮子里面有 N 个球，每个球代表不同的数字，我们用一小篮子去抓取球，使得拿到的球数字和为 sum/2。那么很自然的一个想法就是，对大篮子里面的每一个球，我们考虑取它或者不取它，如果我们足够耐心，最后肯定能穷举所有的情况，判断是否有解。

这是一种**穷举的暴力思维**。

上述思维表述为伪代码如下：

```
令 target = sum / 2, nums 为输入数组, cur 为当前当前要选择的数字的索引
nums 为输入数组，target为当前求和目标，cur为当前判断的数
function dfs(nums, target, cur)
  如果target < 0 或者 cur > nums.length
    return false
  否则
    如果 target = 0, 说明找到答案了，返回true
    否则
      取当前数或者不取，进入递归 dfs(nums, target - nums[cur], cur + 1) || dfs(nums, target, cur + 1)
```

因为对每个数都考虑取不取，所以这里时间复杂度是 O(2 ^ n), 其中 n 是 nums 数组长度，

> 关于时间复杂度大家可以尝试用《算法通关之路》中的递归树法或者是公式法来理解

javascript 实现

```javascript
var canPartition = function (nums) {
  let sum = nums.reduce((acc, num) => acc + num, 0);
  if (sum % 2) {
    return false;
  }
  sum = sum / 2;
  return dfs(nums, sum, 0);
};

function dfs(nums, target, cur) {
  if (target < 0 || cur > nums.length) {
    return false;
  }
  return (
    target === 0 ||
    dfs(nums, target - nums[cur], cur + 1) ||
    dfs(nums, target, cur + 1)
  );
}
```

不出所料，这里是超时了，我们看看有没优化空间。

这里我们使用几个剪枝，关于剪枝我们还会在后面的进阶篇讲解。

1. 如果 nums 中最大值 > sum/2, 那么肯定无解
2. 在搜索过程中，我们对每个数都是取或者不取，并且数组中所有项都为正数。我们设取的数和为 `pickedSum` ，不难得 pickedSum <= sum/2， 同时要求丢弃的数为 `discardSum` 。

我们同时引入这两个约束条件加强剪枝：

优化后的代码如下

```javascript
var canPartition = function (nums) {
  let sum = nums.reduce((acc, num) => acc + num, 0);
  if (sum % 2) {
    return false;
  }
  sum = sum / 2;
  nums = nums.sort((a, b) => b - a);
  if (sum < nums[0]) {
    return false;
  }
  return dfs(nums, sum, sum, 0);
};
```

```
function dfs(nums, pickRemain, discardRemain, cur) {
  if (pickRemain === 0 || discardRemain === 0) {
    return true;
  }

  if (pickRemain < 0 || discardRemain < 0 || cur > nums.length) {
    return false;
  }

  return (
    dfs(nums, pickRemain - nums[cur], discardRemain, cur + 1) ||
    dfs(nums, pickRemain, discardRemain - nums[cur], cur + 1)
  );
}
```

leetcode 是 AC 了（仅仅测试了 JS，其他语言没有测试），但是时间复杂度 O(2 ^ n), 算法时间复杂度很差，我们看看有没更好的。

DP 解法

在用 DFS 是时候，我们是不关心取数的规律的，只要保证接下来要取的数在之前没有被取过即可。那如果我们有规律去安排取数策略的时候会怎么样呢，比如第一次取数安排在第一位，第二位取数安排在第二位，在判断第 i 位是取数的时候，我们是已经知道前 i-1 个数每次是否取的所有组合，记集合 S 为这个子集的和。

再看第 i 位取数的情况, 有两种情况取或者不取：

1. 取的情况，如果 target-nums[i]在集合 S 内，则返回 true，说明前 i 个数能找到和为 target 的序列
2. 不取的情况，如果 target 在集合 S 内，则返回 true，否则返回 false

也就是说，前 i 个数能否构成和为 target 的子集取决为前 i-1 数的情况。

记 F[i, target] 为 nums 数组内前 i 个数能否构成和为 target 的子序列的可能，则状态转移方程为

`F[i, target] = F[i - 1, target] || F[i - 1, target - nums[i]]`

状态转移方程出来了，代码就很好写了，DFS + DP 都可以解，有不清晰的可以参考下 **递归和动态规划**， 这里只提供 DP 解法

伪代码表示

```
n = nums.length
target 为 nums 各数之和
如果target不能被2整除，
  返回false

令dp为n * target 的二维矩阵，并初始为false
遍历0:n, dp[i][0] = true 表示前i个数组成和为0的可能
```

```
遍历 0 到 n
  遍历 0 到 target
    if 当前值j大于nums[i]
      dp[i + 1][j] = dp[i][j-nums[i]] || dp[i][j]
    else
      dp[i+1][j] = dp[i][j]
```

算法时间复杂度 O(n*m), 空间复杂度 O(n*m), m 为 sum(nums) / 2

javascript 实现

```javascript
var canPartition = function (nums) {
  let sum = nums.reduce((acc, num) => acc + num, 0);
  if (sum % 2) {
    return false;
  } else {
    sum = sum / 2;
  }

  const dp = Array.from(nums).map(() =>
    Array.from({ length: sum + 1 }).fill(false)
  );

  for (let i = 0; i < nums.length; i++) {
    dp[i][0] = true;
  }

  for (let i = 0; i < dp.length - 1; i++) {
    for (let j = 0; j < dp[0].length; j++) {
      dp[i + 1][j] =
        j - nums[i] >= 0 ? dp[i][j] || dp[i][j - nums[i]] : dp[i][j];
    }
  }

  return dp[nums.length - 1][sum];
};
```

再看看有没有优化空间，看状态转移方程 `F[i, target] = F[i - 1, target] || F[i - 1, target - nums[i]]` 第 n 行的状态只依赖于第 n-1 行的状态，也就是说我们可以把二维空间压缩成一维

伪代码

```
遍历 0 到 n
  遍历 j 从 target 到 0
    if 当前值j大于nums[i]
      dp[j] = dp[j-nums[i]] || dp[j]
```

```
    else
      dp[j] = dp[j]
```

时间复杂度 $O(n * m)$, 空间复杂度 $O(n)$

javascript 实现

```javascript
var canPartition = function (nums) {
  let sum = nums.reduce((acc, num) => acc + num, 0);
  if (sum % 2) {
    return false;
  }
  sum = sum / 2;
  const dp = Array.from({ length: sum + 1 }).fill(false);
  dp[0] = true;

  for (let i = 0; i < nums.length; i++) {
    for (let j = sum; j > 0; j--) {
      dp[j] = dp[j] || (j - nums[i] >= 0 && dp[j - nums[i]]);
    }
  }

  return dp[sum];
};
```

其实这道题和 **leetcode 518** 是换皮题，它们都可以归属于背包问题

背包问题

背包问题描述

有 N 件物品和一个容量为 V 的背包。放入第 i 件物品耗费的费用是 Ci，得到的 价值是 Wi。求解将哪些物品装入背包可使价值总和最大。

背包问题的特性是，每种物品，我们都可以选择放或者不放。令 F[i, v]表示前 i 件物品放入到容量为 v 的背包的状态。

针对上述背包，F[i, v]表示能得到最大价值，那么状态转移方程为

```
F[i, v] = max{F[i-1, v], F[i-1, v-Ci] + Wi}
```

针对 416. 分割等和子集这题，F[i, v]的状态含义就表示前 i 个数能组成和为 v 的可能，状态转移方程为

```
F[i, v] = F[i-1, v] || F[i-1, v-Ci]
```

再回过头来看下leetcode 518, 原题如下

> 给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

带入背包思想，F[i,v] 表示用前 i 种硬币能兑换金额数为 v 的组合数，状态转移方程为 `F[i, v] = F[i-1, v] + F[i-1, v-Ci]`

javascript 实现

```javascript
/**
 * @param {number} amount
 * @param {number[]} coins
 * @return {number}
 */
var change = function (amount, coins) {
  const dp = Array.from({ length: amount + 1 }).fill(0);
  dp[0] = 1;
  for (let i = 0; i < coins.length; i++) {
    for (let j = 1; j <= amount; j++) {
      dp[j] = dp[j] + (j - coins[i] >= 0 ? dp[j - coins[i]] : 0);
    }
  }
  return dp[amount];
};
```

上一页 下一页