宫水三叶的刷题日记

背包 DP

Author ：宫水三叶
Date ：2021/10/07
QQ Group：703311589
WeChat ：oaoaya

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「背包 DP」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「背包 DP」即可获取最新下载链接。

💡下面介绍使用本合集的最佳使用实践：

学习算法：

1. 打开在线目录（Github 版 & Gitee 版）；
2. 从侧边栏的类别目录找到「背包 DP」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题‘
4. 拿到题号之后，回到本合集进行检索。

维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流 🍭🍭🍭

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **279. 完全平方数** ，难度为 **中等**。

Tag : 「完全背包」、「动态规划」、「背包问题」

给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。

给你一个整数 n ，返回和为 n 的完全平方数的 最少数量 。

完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。

示例 1：

```
输入：n = 12
输出：3
解释：12 = 4 + 4 + 4
```

示例 2：

```
输入：n = 13
输出：2
解释：13 = 4 + 9
```

提示：

- 1 <= n <= 10^4

完全背包（朴素解法）

首先「完全平方数」有无限个，但要凑成的数字是给定的。

因此第一步可以将范围在 $[1, n]$ 内的「完全平方数」预处理出来。

这一步其实就是把所有可能用到的「物品」预处理出来。

从而将问题转换为：**给定了若干个数字，每个数字可以被使用无限次，求凑出目标值 n 所需要用到的是最少数字个数是多少。**

由于题目没有限制我们相同的「完全平方数」只能使用一次，属于「完全背包」模型。

目前我们学过的两类背包问题（01 背包 & 完全背包）的原始状态定义都是两维：

- 第一维 i 代表物品编号
- 第二维 j 代表容量

其中第二维 j 又有「不超过容量 j」和「容量恰好为 j」两种定义，本题要我们求「恰好」凑出 n 所需要的最少个数。

因此我们可以调整我们的「状态定义」：

$f[i][j]$ **为考虑前 i 个数字，凑出数字总和 j 所需要用到的最少数字数量。**

不失一般性的分析 $f[i][j]$，对于第 i 个数字（假设数值为 t），我们有如下选择：

- 选 0 个数字 i，此时有 $f[i][j] = f[i-1][j]$
- 选 1 个数字 i，此时有 $f[i][j] = f[i-1][j-t] + 1$
- 选 2 个数字 i，此时有 $f[i][j] = f[i-1][j-2*t] + 2$
  ...
- 选 k 个数字 i，此时有 $f[i][j] = f[i-1][j-k*t] + k$

因此我们的状态转移方程为：

$$f[i][j] = min(f[i-1][j-k*t]+k), 0 \leqslant k*t \leqslant j$$

当然，能够选择 k 个数字 i 的前提是，剩余的数字 $j-k*t$ 也能够被其他「完全平方数」凑出，即 $f[i-1][j-k*t]$ 为有意义的值。

代码（朴素完全背包问题的复杂度是 $O(n^2 * \sqrt{n})$ 的，有超时风险，让物品下标从 0 开始，单独处理第一个物品的 $P2$ 代码勉强能过）：

```java
class Solution {
    int INF = 0x3f3f3f3f;
    public int numSquares(int n) {
        // 预处理出所有可能用到的「完全平方数」
        List<Integer> list = new ArrayList<>();
        int t = 1;
        while (t * t <= n) {
            list.add(t * t);
            t++;
        }

        // f[i][j] 代表考虑前 i 个物品，凑出 j 所使用到的最小元素个数
        int m = list.size();
        int[][] f = new int[m + 1][n + 1];

        // 当没有任何数时，除了 f[0][0] 为 0（花费 0 个数值凑出 0），其他均为无效值
        Arrays.fill(f[0], INF);
        f[0][0] = 0;

        // 处理剩余数的情况
        for (int i = 1; i <= m ; i++) {
            int x = list.get(i - 1);
            for (int j = 0; j <= n; j++) {
                // 对于不选第 i 个数的情况
                f[i][j] = f[i - 1][j];
                // 对于选 k 次第 i 个数的情况
                for (int k = 1; k * x <= j; k++) {
                    // 能够选择 k 个 x 的前提是剩余的数字 j - k * x 也能被凑出
                    if (f[i - 1][j - k * x] != INF) {
                        f[i][j] = Math.min(f[i][j], f[i - 1][j - k * x] + k);
                    }
                }
            }
        }
        return f[m][n];
    }
}
```

```java
class Solution {
    int INF = -1;
    public int numSquares(int n) {
        // 预处理出所有可能用到的「完全平方数」
        List<Integer> list = new ArrayList<>();
        int idx = 1;
        while (idx * idx <= n) {
            list.add(idx * idx);
            idx++;
        }

        // f[i][j] 代表考虑前 i 个物品，凑出 j 所使用到的最小元素个数
        int len = list.size();
        int[][] f = new int[len][n + 1];

        // 处理第一个数的情况
        for (int j = 0; j <= n; j++) {
            int t = list.get(0);
            int k = j / t;
            if (k * t == j) { // 只有容量为第一个数的整数倍的才能凑出
                f[0][j] = k;
            } else { // 其余则为无效值
                f[0][j] = INF;
            }
        }

        // 处理剩余数的情况
        for (int i = 1; i < len; i++) {
            int t = list.get(i);
            for (int j = 0; j <= n; j++) {
                // 对于不选第 i 个数的情况
                f[i][j] = f[i - 1][j];
                // 对于选 k 次第 i 个数的情况
                for (int k = 1; k * t <= j; k++) {
                    // 能够选择 k 个 t 的前提是剩余的数字 j - k * t 也能被凑出
                    // 使用 0x3f3f3f3f 作为最大值（预留累加空间）可以省去该判断
                    if (f[i - 1][j - k * t] != INF) {
                        f[i][j] = Math.min(f[i][j], f[i - 1][j - k * t] + k);
                    }
                }

            }
        }
        return f[len - 1][n];
    }
}
```

- 时间复杂度：预处理出所有可能用到的数字复杂度为 $O(\sqrt{n})$，共有 $n * \sqrt{n}$ 个状态需要转移，每个状态转移最多遍历 n 次，因此转移完所有状态复杂度为 $O(n^2 * \sqrt{n})$。整体复杂度为 $O(n^2 * \sqrt{n})$。
- 空间复杂度：$O(n * \sqrt{n})$。

完全背包（进阶）

显然朴素版的完全背包进行求解复杂度有点高。

这次我们还是按照同样的思路再进行一次推导，加强对这种「一维空间优化」方式的理解。

从二维的状态转移方程入手进行分析（假设第 i 个数字为 t）：

目标是求 f[i][j] 的最小值
而 f[i - 1][j] 代表不选数字 t，而且这个状态必然能够取到（有效值）
所以问题转换为求下面横线部分的最小值

f[i][j] = min(f[i - 1][j], f[i - 1][j - t] + 1, f[i - 1][j - 2 * t] + 2 ,..., f[i - 1][j - k * t] + k), 0 <= k * t <= j

将 j - t 作为 j 代入上述式子，可得：

f[i][j - t] = min(f[i - 1][j - t], f[i - 1][j - 2 * t] + 1, f[i - 1][j - 3 * t] + 2 ,..., f[i - 1][j - k * t] + (k - 1)), 0 <= k * t <= j

f[i][j] 的画线部分和 f[i][j - t] 的画线部分具有“等差”特性，总是相差 1（相对应的我用了相同颜色进行标注）

因此，我们要求的 f[i][j] 的最小值问题，又转换为：
求 f[i][j - t] + 1 的最小值

也就是 f[i][j] = min(f[i - 1][j], f[i][j - t] + 1)

再进行 i 的维度消除，可得：
f[j] = min(f[j], f[j - t] + 1)

既然我们在更新 f[j] 的时候用到了 f[j - t]，那么我们必须保证使用的 f[j - t] 是已经计算好的，是最终值
所以当我们将状态定义优化至一维的时候，要将 j 的遍历顺序调整为从小到大，以确保我们使用 f[j - t] 更新 f[j] 时，f[j - t] 是最终值

至此，我们得到了最终的状态转移方程：

$$f[j] = min(f[j], f[j - t] + 1)$$

同时，预处理「物品」的逻辑也能直接下放到转移过程去做。

代码：

```java
class Solution {
    public int numSquares(int n) {
        int[] f = new int[n + 1];
        Arrays.fill(f, 0x3f3f3f3f);
        f[0] = 0;
        for (int t = 1; t * t <= n; t++) {
            int x = t * t;
            for (int j = x; j <= n; j++) {
                f[j] = Math.min(f[j], f[j - x] + 1);
            }
        }
        return f[n];
    }
}
```

- 时间复杂度：共有 $n * \sqrt{n}$ 个状态需要转移，复杂度为 $O(n * \sqrt{n})$。
- 空间复杂度：$O(n)$。

> **🍭 更多精彩内容，欢迎关注：**公众号** / Github / LeetCode / **知乎** **

题目描述

这是 LeetCode 上的 **322. 零钱兑换** ，难度为 **中等**。

Tag : 「完全背包」、「动态规划」、「背包问题」

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

你可以认为每种硬币的数量是无限的。

示例 1：

```
输入：coins = [1, 2, 5], amount = 11
输出：3
解释：11 = 5 + 5 + 1
```

示例 2：

```
输入：coins = [2], amount = 3
输出：-1
```

示例 3：

```
输入：coins = [1], amount = 0
输出：0
```

示例 4：

```
输入：coins = [1], amount = 1
输出：1
```

示例 5：

```
输入：coins = [1], amount = 2
输出：2
```

提示：

- 1 <= coins.length <= 12
- 1 <= coins[i] <= 2^{31} - 1
- 0 <= amount <= 10^4

完全背包（朴素解法）

硬币相当于我们的物品，每种硬币可以选择「无限次」，我们应该很自然的想到「完全背包」。

如果不能，那么从现在开始就要培养这样的习惯：

当看到题目是给定一些「物品」，让我们从中进行选择，以达到「最大价值」或者「特定价值」时，我们应该联想到「背包问题」。

这本质上其实是一个组合问题：被选物品之间不需要满足特定关系，只需要选择物品，以达到「全局最优」或者「特定状态」即可。

再根据物品的选择次数限制来判断是何种背包问题。

本题每种硬币可以被选择「无限次」，我们可以直接套用「完全背包」的状态定义进行微调：

定义 $f[i][j]$ 为考虑前 i 件物品，凑成总和为 j 所需要的最少硬币数量。

为了方便初始化，我们一般让 $f[0][x]$ 代表不考虑任何物品的情况。

因此我们有显而易见的初始化条件：$f[0][0] = 0$，其余 $f[0][x] = INF$。

代表当没有任何硬币的时候，存在凑成总和为 0 的方案，方案所使用的硬币为 0；凑成其他总和的方案不存在。

由于我们要求的是「最少」硬币数量，因此我们不希望「无效值」参与转移，因此可设 $INF = INT\_MAX$。

当「状态定义」与「基本初始化」有了之后，我们不失一般性的考虑 $f[i][j]$ 该如何转移。

对于第 i 个硬币我们有两种决策方案：

- 不使用该硬币：

$$f[i-1][j]$$

- 使用该硬币，由于每个硬币可以被选择多次（容量允许的情况下），因此最优解应当是所有方案中的最小值：

$$min(f[i-1][j-k*coin]+k)$$

代码：

```java
class Solution {
    int INF = Integer.MAX_VALUE;
    public int coinChange(int[] cs, int cnt) {
        int n = cs.length;
        int[][] f = new int[n + 1][cnt + 1];

        // 初始化（没有任何硬币的情况）：只有 f[0][0] = 0；其余情况均为无效值。
        // 这是由「状态定义」决定的，当不考虑任何硬币的时候，只能凑出总和为 0 的方案，所使用的硬币数量为
        for (int i = 1; i <= cnt; i++) f[0][i] = INF;

        // 有硬币的情况
        for (int i = 1; i <= n; i++) {
            int val = cs[i - 1];
            for (int j = 0; j <= cnt; j++) {
                // 不考虑当前硬币的情况
                f[i][j] = f[i - 1][j];

                // 考虑当前硬币的情况（可选当前硬币个数基于当前容量大小）
                for (int k = 1; k * val <= j; k++) {
                    if (f[i - 1][j - k * val] != INF) {
                        f[i][j] = Math.min(f[i][j], f[i-1][j-k*val] + k);
                    }
                }
            }
        }
        return f[n][cnt] == INF ? -1 : f[n][cnt];
    }
}
```

- 时间复杂度：共有 $n * cnt$ 个状态需要转移，每个状态转移最多遍历 cnt 次。整体复杂度为 $O(n * cnt^2)$。
- 空间复杂度：$O(n * cnt)$。

无效状态的定义问题

借这个问题，刚好说一下，我们初始化时，对于无效状态应该如何定义。

可以看到上述解法，将 `INF` 定义为 `INT_MAX` 。

这是因为我们转移时取的是较小值，我们希望无效值不要被转移，所以将 `INF` 定义为较大的数，以代表数学上的 $+\infty$ （正无穷）。

这很合理，但是我们需要注意，如果我们在 `INF` 的基础上进行累加的话，常规的语言会将其变成负数最小值。

也就是在正无穷基础上进行累加，会丢失其正无穷的含义，这与数学上的正无穷概念冲突。

因此，我们才有先判断再使用的习惯：

```
if (f[i-1][j] != INF) {
    f[i][j] = Math.min(f[i][j], f[i-1][j]);
}
```

但事实上，如果每次使用都需要有前置检查的话，是很麻烦的。

于是我们有另外一个技巧，不直接使用 `INT_MAX` 作为 `INF`，而是使用一个比 `INT_MAX` 小的较大数来代表 `INF`。

相当于预留了一些「累加空间」给 `INF`。

比如使用 `0x3f3f3f3f` 作为最大值，这样我们使用 `INF` 做状态转移的时候，就不需要先判断再使用了。

代码：

```
class Solution {
    int INF = 0x3f3f3f3f;
    public int coinChange(int[] cs, int cnt) {
        int n = cs.length;
        int[][] f = new int[n + 1][cnt + 1];
        for (int i = 1; i <= cnt; i++) f[0][i] = INF;
        for (int i = 1; i <= n; i++) {
            int val = cs[i - 1];
            for (int j = 0; j <= cnt; j++) {
                f[i][j] = f[i-1][j];
                for (int k = 0; k * val <= j; k++) {
                        f[i][j] = Math.min(f[i][j], f[i-1][j-k*val] + k);
                }
            }
        }
        return f[n][cnt] == INF ? -1 : f[n][cnt];
    }
}
```

完全背包（一维优化）

显然朴素版的完全背包进行求解复杂度有点高。

在「学习完全背包」和「上一讲练习」中，我们从最朴素背包转移方程出发，从数学的角度去推导一维优化是如何来的。

这十分科学，而绝对严谨。

但每次都这样推导是十分耗时的。

因此，我们这次站在一个「更高」的角度去看「完全背包」问题。

我们知道传统的「完全背包」二维状态转移方程是：

$$f[i][j] = max(f[i-1][j], f[i-1][j-k*w[i]] + k*v[i])$$

经过一维空间优化后的状态转移方程是（同时容量维度遍历顺序为「从小到大」）：

$$f[j] = max(f[j], f[j-w[i]] + v[i])$$

这是我们在 学习完全背包 时推导的，是经过严格证明的，具有一般性的。

然后我们只需要对「成本」&「价值」进行抽象，并结合「换元法」即可得到任意背包问题的一维优化状态转移方程。

拿我们本题的状态转移方程来分析，本题的朴素状态转移方程为：

$$f[i][j] = min(f[i-1][j], f[i-1][j-k*coin] + k)$$

我们将硬币的面值抽象为「成本」，硬币的数量抽象「价值」，再对物品维度进行消除，即可得：

$$f[j] = min(f[j], f[j-coin] + 1)$$

如果还不理解，可以将上述四个状态转移方程「两两成对」结合来看。

代码：

```java
class Solution {
    int INF = 0x3f3f3f3f;
    public int coinChange(int[] cs, int cnt) {
        int n = cs.length;
        int[] f = new int[cnt + 1];
        for (int i = 1; i <= cnt; i++) f[i] = INF;
        for (int i = 1; i <= n; i++) {
            int val = cs[i - 1];
            for (int j = val; j <= cnt; j++) {
                f[j] = Math.min(f[j], f[j - val] + 1);
            }
        }
        return f[cnt] == INF ? -1 : f[cnt];
    }
}
```

- 时间复杂度：共有 $n * cnt$ 个状态需要转移，整体复杂度为 $O(n * cnt)$。
- 空间复杂度：$O(cnt)$。

总结

本节，我们先是从朴素「完全背包」的角度分析并解决了问题。

而在考虑「一维优化」的时候，由于已经有前两节「数学推导优化思路」的基础，我们这次站在了「更高」的角度去看待一维优化。

从抽象「成本」&「价值」，结合「换元法」的角度去理解一维优化过程。

这可以大大节省我们分析推导的时间。

建议大家加强理解 ~

下一节练习篇，我们会继续强化这个过程

题目描述

这是 LeetCode 上的 **416. 分割等和子集(上)** ，难度为 **中等**。

Tag : 「背包 DP」

给你一个 只包含正整数 的 非空 数组 nums 。

请你判断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。

示例 1：

```
输入：nums = [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：nums = [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。
```

提示：

- 1 <= nums.length <= 200
- 1 <= nums[i] <= 100

前言

今天是我们讲解**动态规划专题**中的 *「背包问题」的第二天*。

在众多背包问题中「01 背包问题」是最为核心的，因此我建议你先精读过 背包问题 第一讲 之后再阅读本文。

另外，我在文章结尾处列举了我所整理的关于背包问题的相关题目。

背包问题我会按照编排好的顺序进行讲解（*每 2~3 天更新一篇，确保大家消化*）。

你也先可以尝试做做，也欢迎你向我留言补充，你觉得与背包相关的 DP 类型题目 ~

基本分析

通常「背包问题」相关的题，都是在考察我们的「建模」能力，也就是将问题转换为「背包问题」的能力。

由于本题是问我们能否将一个数组分成两个「等和」子集。

问题等效于**能否从数组中挑选若干个元素，使得元素总和等于所有元素总和的一半**。

这道题如果抽象成「背包问题」的话，应该是：

我们背包容量为 $target = sum/2$，每个数组元素的「价值」与「成本」都是其数值大小，求我们能否装满背包。

转换为 01 背包

由于每个数字（数组元素）只能被选一次，而且每个数字选择与否对应了「价值」和「成本」，求解的问题也与「最大价值」相关。

可以使用「01 背包」的模型来做。

当我们确定一个问题可以转化为「01 背包」之后，就可以直接套用「01 背包」的状态定义进行求解了。

注意，我们积累 DP 模型的意义，就是在于我们可以快速得到可靠的「状态定义」。

在 路径问题 中我教过你通用的 DP 技巧解法，但那是基于我们完全没见过那样的题型才去用的，而对于一些我们见过题型的 DP 题目，我们应该直接套用（或微调）该模型「状态定义」来做。

我们直接套用「01 背包」的状态定义：

$f[i][j]$ 代表考虑前 i 个数值，其选择数字总和不超过 j 的最大价值。

当有了「状态定义」之后，结合我们的「最后一步分析法」，每个数字都有「选」和「不选」两种选择。

因此不难得出状态转移方程：

$$f[i][j] = max(f[i-1][j], f[i-1][j-nums[i]] + nums[i])$$

代码：

```java
class Solution {
    public boolean canPartition(int[] nums) {
        int n = nums.length;

        //「等和子集」的和必然是总和的一半
        int sum = 0;
        for (int i : nums) sum += i;
        int target = sum / 2;

        // 对应了总和为奇数的情况，注定不能被分为两个「等和子集」
        if (target * 2 != sum) return false;

        int[][] f = new int[n][target + 1];
        // 先处理考虑第 1 件物品的情况
        for (int j = 0; j <= target; j++) {
            f[0][j] = j >= nums[0] ? nums[0] : 0;
        }

        // 再处理考虑其余物品的情况
        for (int i = 1; i < n; i++) {
            int t = nums[i];
            for (int j = 0; j <= target; j++) {
                // 不选第 i 件物品
                int no = f[i-1][j];
                // 选第 i 件物品
                int yes = j >= t ? f[i-1][j-t] + t : 0;
                f[i][j] = Math.max(no, yes);
            }
        }
        // 如果最大价值等于 target，说明可以拆分成两个「等和子集」
        return f[n-1][target] == target;
    }
}
```

- 时间复杂度：$target$ 为数组总和的一半，n 数组元素个数。为共有 $n * target$ 个状态需要被转移，复杂度为 $O(n * target)$
- 空间复杂度：$O(n * target)$

「滚动数组」解法

在上一讲我们讲到过「01 背包」具有两种空间优化方式。

其中一种优化方式的编码实现十分固定，只需要固定的修改「物品维度」即可。

代码：

```java
class Solution {
    public boolean canPartition(int[] nums) {
        int n = nums.length;

        //「等和子集」的和必然是总和的一半
        int sum = 0;
        for (int i : nums) sum += i;
        int target = sum / 2;
        if (target * 2 != sum) return false;

        // 将「物品维度」修改为 2
        int[][] f = new int[2][target + 1];
        // 先处理考虑第 1 件物品的情况
        for (int j = 0; j <= target; j++) {
            f[0][j] = j >= nums[0] ? nums[0] : 0;
        }

        // 再处理考虑其余物品的情况
        for (int i = 1; i < n; i++) {
            int t = nums[i];
            for (int j = 0; j <= target; j++) {
                // 不选第 i 件物品，将物品维度的使用加上「&1」
                int no = f[(i-1)&1][j];
                // 选第 i 件物品，将物品维度的使用加上「&1」
                int yes = j >= t ? f[(i-1)&1][j-t] + t : 0;
                f[i&1][j] = Math.max(no, yes);
            }
        }
        // 如果最大价值等于 target，说明可以拆分成两个「等和子集」
        // 将物品维度的使用加上「&1」
        return f[(n-1)&1][target] == target;
    }
}
```

- 时间复杂度：$target$ 为数组总和的一半，n 数组元素个数。为共有 $n * target$ 个状态需要被转移，复杂度为 $O(n * target)$
- 空间复杂度：$O(target)$

「一维空间优化」解法

事实上，我们还能继续进行空间优化：只保留代表「剩余容量」的维度，同时将容量遍历方向修改为「从大到小」。

代码：

```
class Solution {
    public boolean canPartition(int[] nums) {
        int n = nums.length;

        //「等和子集」的和必然是总和的一半
        int sum = 0;
        for (int i : nums) sum += i;
        int target = sum / 2;

        // 对应了总和为奇数的情况，注定不能被分为两个「等和子集」
        if (target * 2 != sum) return false;

        // 将「物品维度」取消
        int[] f = new int[target + 1];
        for (int i = 0; i < n; i++) {
            int t = nums[i];
            // 将「容量维度」改成从大到小遍历
            for (int j = target; j >= 0; j--) {
                // 不选第 i 件物品
                int no = f[j];
                // 选第 i 件物品
                int yes = j >= t ? f[j-t] + t : 0;
                f[j] = Math.max(no, yes);
            }
        }
        // 如果最大价值等于 target，说明可以拆分成两个「等和子集」
        return f[target] == target;
    }
}
```

- 时间复杂度：$target$ 为数组总和的一半，n 数组元素个数。为共有 $n * target$ 个状态需要被转移，复杂度为 $O(n * target)$
- 空间复杂度：$O(target)$

总结

今天我们对昨天学的「01 背包」进行了应用。

可以发现，本题的难点在于**对问题的抽象**，主要考察的是如何将原问题转换为一个「01 背包」

问题。

事实上，无论是 DP 还是图论，对于特定问题，大多都有相应的模型或算法。

难是难在如何将问题转化为我们的模型。

至于如何培养自己的「问题抽象能力」？

首先通常需要我们积累一定的刷题量，并对「转换问题的关键点」做总结。

例如本题，一个转换「01 背包问题」的关键点是我们需要将「划分等和子集」的问题等效于「在某个数组中选若干个数，使得其总和为某个特定值」的问题。

拓展

但这道题到这里还有一个"小问题"。

就是我们最后是通过「判断」来取得答案的。

通过判断取得的最大价值是否等于 $target$ 来决定是否能划分出「等和子集」。

虽然说逻辑上完全成立，但总给我们一种「间接求解」的感觉。

造成这种「间接求解」的感觉，主要是因为我们没有对「01 背包」的「状态定义」和「初始化」做任何改动。

但事实上，我们是可以利用「01 背包」的思想进行「直接求解」的。

因此在下一讲，我们还会再做一遍这道题。

不过却是以「另外一个角度」的「01 背包」思维来解决。

敬请期待 ~

> **🍭 更多精彩内容，欢迎关注：**公众号** / Github / LeetCode / **知乎** **

题目描述

这是 LeetCode 上的 **474. 一和零** ，难度为 **中等**。

Tag : 「01 背包」、「背包问题」、「多维背包」、「动态规划」

给你一个二进制字符串数组 strs 和两个整数 m 和 n 。

请你找出并返回 strs 的最大子集的大小，该子集中 最多 有 m 个 0 和 n 个 1 。

如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。

示例 1：

```
输入：strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3
输出：4
解释：最多有 5 个 0 和 3 个 1 的最大子集是 {"10","0001","1","0"} ，因此答案是 4 。
其他满足题意但较小的子集包括 {"0001","1"} 和 {"10","1","0"} 。{"111001"} 不满足题意，因为它含 4 个 1 ，大于 n 的值 3
```

示例 2：

```
输入：strs = ["10", "0", "1"], m = 1, n = 1
输出：2
解释：最大的子集是 {"0", "1"} ，所以答案是 2 。
```

提示：

- 1 <= strs.length <= 600
- 1 <= strs[i].length <= 100
- strs[i] 仅由 ‘0’ 和 ‘1’ 组成
- 1 <= m, n <= 100

（多维）01 背包

通常与「背包问题」相关的题考察的是 **将原问题转换为「背包问题」的能力**。

要将原问题转换为「背包问题」，往往需要从题目中抽象出「价值」与「成本」的概念。

这道题如果抽象成「背包问题」的话，应该是：

每个字符串的价值都是 1（对答案的贡献都是 1），选择的成本是该字符串中 1 的数量和 0 的数量。

问我们在 1 的数量不超过 m，0 的数量不超过 n 的条件下，最大价值是多少。

由于每个字符串只能被选一次，且每个字符串的选与否对应了「价值」和「成本」，求解的问题也是「最大价值」是多少。

因此可以直接套用 01 背包的「状态定义」来做：

$f[k][i][j]$ 代表考虑前 k 件物品，在数字 1 容量不超过 i，数字 0 容量不超过 j 的条件下的「最大价值」（每个字符串的价值均为 1）。

有了「状态定义」之后，「转移方程」也很好推导：

$$f[k][i][j] = \max(f[k-1][i][j], f[k-1][i-cnt[k][0]][j-cnt[k][1]]+1)$$

其中 cnt 数组记录的是字符串中出现的 01 数量。

代码（为了方便理解，$P1$ 将第一件物品的处理单独抽了出来，也可以不抽出来，只需要将让物品下标从 1 开始即可，见 $P2$）：

```java
class Solution {
    public int findMaxForm(String[] strs, int m, int n) {
        int len = strs.length;
        // 预处理每一个字符包含 0 和 1 的数量
        int[][] cnt = new int[len][2];
        for (int i = 0; i < len; i++) {
            String str = strs[i];
            int zero = 0, one = 0;
            for (char c : str.toCharArray()) {
                if (c == '0') {
                    zero++;
                } else {
                    one++;
                }
            }
            cnt[i] = new int[]{zero, one};
        }

        // 处理只考虑第一件物品的情况
        int[][][] f = new int[len][m + 1][n + 1];
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                f[0][i][j] = (i >= cnt[0][0] && j >= cnt[0][1]) ? 1 : 0;
            }
        }

        // 处理考虑其余物品的情况
        for (int k = 1; k < len; k++) {
            int zero = cnt[k][0], one = cnt[k][1];
            for (int i = 0; i <= m; i++) {
                for (int j = 0; j <= n; j++) {
                    // 不选择第 k 件物品
                    int a = f[k-1][i][j];
                    // 选择第 k 件物品（前提是有足够的 m 和 n 额度可使用）
                    int b = (i >= zero && j >= one) ? f[k-1][i-zero][j-one] + 1 : 0;
                    f[k][i][j] = Math.max(a, b);
                }
            }
        }
        return f[len-1][m][n];
    }
}
```

```java
class Solution {
    public int findMaxForm(String[] strs, int m, int n) {
        int len = strs.length;
        int[][] cnt = new int[len][2];
        for (int i = 0; i < len; i++) {
            String str = strs[i];
            int zero = 0, one = 0;
            for (char c : str.toCharArray()) {
                if (c == '0') zero++;
                else one++;

            }
            cnt[i] = new int[]{zero, one};
        }
        int[][][] f = new int[len + 1][m + 1][n + 1];
        for (int k = 1; k <= len; k++) {
            int zero = cnt[k - 1][0], one = cnt[k - 1][1];
            for (int i = 0; i <= m; i++) {
                for (int j = 0; j <= n; j++) {
                    int a = f[k - 1][i][j];
                    int b = (i >= zero && j >= one) ? f[k - 1][i - zero][j - one] + 1 : 0;
                    f[k][i][j] = Math.max(a, b);
                }
            }
        }
        return f[len][m][n];
    }
}
```

- 时间复杂度：预处理字符串的复杂度为 $O(\sum_{i=0}^{k-1} len(strs[i]))$，处理状态转移的 $O(k * m * n)$。整体复杂度为：$O(k * m * n + \sum_{i=0}^{k-1} len(strs[i]))$
- 空间复杂度：$O(k * m * n)$

滚动数组

根据「状态转移」可知，更新某个物品的状态时，只依赖于上一个物品的状态。

因此，可以使用「滚动数组」的方式进行空间优化。

代码（为了方便理解，$P1$ 将第一件物品的处理单独抽了出来，也可以不抽出来，只需要将让物品下标从 1 开始即可，见 $P2$）：

```java
class Solution {
    public int findMaxForm(String[] strs, int m, int n) {
        int len = strs.length;
        // 预处理每一个字符包含 0 和 1 的数量
        int[][] cnt = new int[len][2];
        for (int i = 0; i < len; i++) {
            String str = strs[i];
            int zero = 0, one = 0;
            for (char c : str.toCharArray()) {
                if (c == '0') {
                    zero++;
                } else {
                    one++;
                }
            }
            cnt[i] = new int[]{zero, one};
        }

        // 处理只考虑第一件物品的情况
        // 「物品维度」修改为 2
        int[][][] f = new int[2][m + 1][n + 1];
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                f[0][i][j] = (i >= cnt[0][0] && j >= cnt[0][1]) ? 1 : 0;
            }
        }

        // 处理考虑其余物品的情况
        for (int k = 1; k < len; k++) {
            int zero = cnt[k][0], one = cnt[k][1];
            for (int i = 0; i <= m; i++) {
                for (int j = 0; j <= n; j++) {
                    // 不选择第 k 件物品
                    // 将 k-1 修改为 (k-1)&1
                    int a = f[(k-1)&1][i][j];
                    // 选择第 k 件物品（前提是有足够的 m 和 n 额度可使用）
                    // 将 k-1 修改为 (k-1)&1
                    int b = (i >= zero && j >= one) ? f[(k-1)&1][i-zero][j-one] + 1 : 0;
                    f[k&1][i][j] = Math.max(a, b);
                }
            }
        }
        // 将 len-1 修改为 (len-1)&1
        return f[(len-1)&1][m][n];
    }
}
```

```java
class Solution {
    public int findMaxForm(String[] strs, int m, int n) {
        int len = strs.length;
        int[][] cnt = new int[len][2];
        for (int i = 0; i < len; i++) {
            String str = strs[i];
            int zero = 0, one = 0;
            for (char c : str.toCharArray()) {
                if (c == '0') zero++;
                else one++;
            }
            cnt[i] = new int[]{zero, one};
        }
        int[][][] f = new int[2][m + 1][n + 1];
        for (int k = 1; k <= len; k++) {
            int zero = cnt[k - 1][0], one = cnt[k - 1][1];
            for (int i = 0; i <= m; i++) {
                for (int j = 0; j <= n; j++) {
                    int a = f[(k-1) & 1][i][j];
                    int b = (i >= zero && j >= one) ? f[(k-1) & 1][i - zero][j - one] + 1
                    f[k&1][i][j] = Math.max(a, b);
                }
            }
        }
        return f[len&1][m][n];
    }
}
```

- 时间复杂度：预处理字符串的复杂度为 $O(\sum_{i=0}^{k-1} len(strs[i]))$，处理状态转移的 $O(k * m * n)$。整体复杂度为：$O(k * m * n + \sum_{i=0}^{k-1} len(strs[i]))$
- 空间复杂度：$O(m * n)$

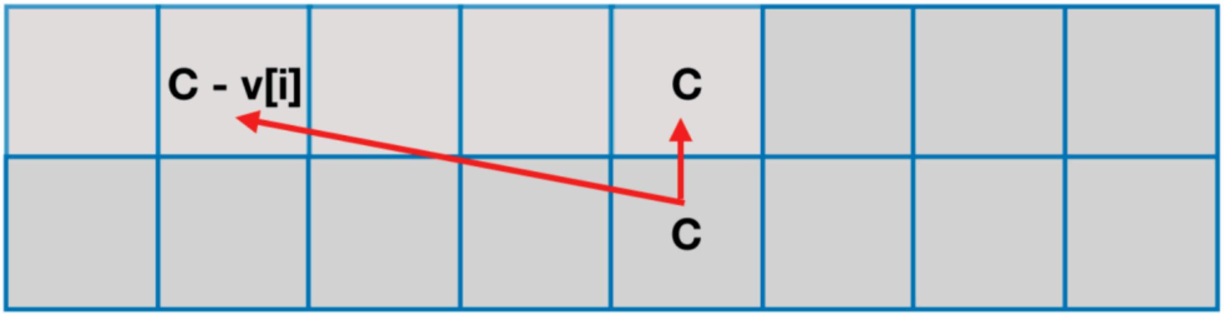
一维空间优化

事实上，我们还能继续进行空间优化。

再次观察我们的「状态转移方程」发现：$f[k][i][j]$ **不仅仅依赖于上一行，还明确依赖于比 i 小和比 j 小的状态。**

即可只依赖于「上一行」中「正上方」的格子，和「正上方左边」的格子。

对应到「朴素的 01 背包问题」依赖关系如图：



因此可直接参考「01 背包的空间优化」方式：取消掉「物品维度」，然后调整容量的遍历顺序。

代码：

```java
class Solution {
    public int findMaxForm(String[] strs, int m, int n) {
        int len = strs.length;
        int[][] cnt = new int[len][2];
        for (int i = 0; i < len; i++) {
            int zero = 0, one = 0;
            for (char c : strs[i].toCharArray()) {
                if (c == '0') zero++;
                else one++;
            }
            cnt[i] = new int[]{zero, one};
        }
        int[][] f = new int[m + 1][n + 1];
        for (int k = 0; k < len; k++) {
            int zero = cnt[k][0], one = cnt[k][1];
            for (int i = m; i >= zero; i--) {
                for (int j = n; j >= one; j--) {
                    f[i][j] = Math.max(f[i][j], f[i - zero][j - one] + 1);
                }
            }
        }
        return f[m][n];
    }
}
```

- 时间复杂度：预处理字符串的复杂度为 $O(\sum_{i=0}^{k-1} len(strs[i]))$，处理状态转移的 $O(k * m * n)$。整体复杂度为：$O(k * m * n + \sum_{i=0}^{k-1} len(strs[i]))$
- 空间复杂度：$O(m * n)$

> **🍭 更多精彩内容，欢迎关注：**公众号** / Github / LeetCode / **知乎** **

题目描述

这是 LeetCode 上的 **494. 目标和** ，难度为 **中等**。

Tag : 「DFS」、「记忆化搜索」、「背包 DP」、「01 背包」

给你一个整数数组 nums 和一个整数 target 。

向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，可以构造一个 表达式 ：

- 例如，nums = [2, 1] ，可以在 2 之前添加 '+' ，在 1 之前添加 '-' ，然后串联起来得到表达式 "+2-1" 。
  返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。

示例 1：

```
输入：nums = [1,1,1,1,1], target = 3
输出：5
解释：一共有 5 种方法让最终目标和为 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
```

示例 2：

```
输入：nums = [1], target = 1
输出：1
```

提示：

- 1 <= nums.length <= 20
- 0 <= nums[i] <= 1000
- 0 <= sum(nums[i]) <= 100
- -1000 <= target <= 100

DFS

数据范围只有 20，而且每个数据只有 $+/-$ 两种选择，因此可以直接使用 DFS 进行「爆搜」。

而 DFS 有「使用全局变量维护」和「接收返回值处理」两种形式。

代码：

```
class Solution {
    public int findTargetSumWays(int[] nums, int t) {
        return dfs(nums, t, 0, 0);
    }
    int dfs(int[] nums, int t, int u, int cur) {
        if (u == nums.length) {
            return cur == t ? 1 : 0;
        }
        int left = dfs(nums, t, u + 1, cur + nums[u]);
        int right = dfs(nums, t, u + 1, cur - nums[u]);
        return left + right;
    }
}
```

```java
class Solution {
    int ans = 0;
    public int findTargetSumWays(int[] nums, int t) {
        dfs(nums, t, 0, 0);
        return ans;
    }
    void dfs(int[] nums, int t, int u, int cur) {
        if (u == nums.length) {
            ans += cur == t ? 1 : 0;
            return;
        }
        dfs(nums, t, u + 1, cur + nums[u]);
        dfs(nums, t, u + 1, cur - nums[u]);
    }
}
```

- 时间复杂度：$O(2^n)$
- 空间复杂度：忽略递归带来的额外空间消耗。复杂度为 $O(1)$

记忆化搜索

不难发现，在 DFS 的函数签名中只有「数值下标 `u`」和「当前结算结果 `cur`」为可变参数，考虑将其作为记忆化容器的两个维度，返回值作为记忆化容器的记录值。

由于 `cur` 存在负权值，为了方便，我们这里不设计成静态数组，而是使用「哈希表」进行记录。

以上分析都在 （题解）403. 青蛙过河 完整讲过。

代码：

```java
class Solution {
    public int findTargetSumWays(int[] nums, int t) {
        return dfs(nums, t, 0, 0);
    }
    Map<String, Integer> cache = new HashMap<>();
    int dfs(int[] nums, int t, int u, int cur) {
        String key = u + "_" + cur;
        if (cache.containsKey(key)) return cache.get(key);
        if (u == nums.length) {
            cache.put(key, cur == t ? 1 : 0);
            return cache.get(key);
        }
        int left = dfs(nums, t, u + 1, cur + nums[u]);
        int right = dfs(nums, t, u + 1, cur - nums[u]);
        cache.put(key, left + right);
        return cache.get(key);
    }
}
```

- 时间复杂度：$O(n * \sum_{i=0}^{n-1} abs(nums[i]))$
- 空间复杂度：忽略递归带来的额外空间消耗。复杂度为 $O(n * \sum_{i=0}^{n-1} abs(nums[i]))$

动态规划

能够以「递归」的形式实现动态规划（记忆化搜索），自然也能使用「递推」的方式进行实现。

根据记忆化搜索的分析，我们可以定义：

$f[i][j]$ 代表考虑前 i 个数，当前计算结果为 j 的方案数，令 `nums` 下标从 1 开始。

那么 $f[n][target]$ 为最终答案，$f[0][0] = 1$ 为初始条件：代表不考虑任何数，凑出计算结果为 0 的方案数为 1 种。

根据每个数值只能搭配 $+/-$ 使用，可得状态转移方程：

$$f[i][j] = f[i-1][j-nums[i-1]] + f[i-1][j+nums[i-1]]$$

到这里，既有了「状态定义」和「转移方程」，又有了可以滚动下去的「有效值」（起始条件）。

距离我们完成所有分析还差最后一步。

当使用递推形式时，我们通常会使用「静态数组」来存储动规值，因此还需要考虑维度范围的：

- 第一维为物品数量：范围为 `nums` 数组长度
- 第二维为中间结果：令 `s` 为所有 `nums` 元素的总和（题目给定了 `nums[i]` 为非负数的条件，否则需要对 `nums[i]` 取绝对值再累加），那么中间结果的范围为 $[-s, s]$

因此，我们可以确定动规数组的大小。**同时在转移时，对第二维度的使用做一个 `s` 的右偏移，以确保「负权值」也能够被合理计算/存储。**

代码：

```
class Solution {
    public int findTargetSumWays(int[] nums, int t) {
        int n = nums.length;
        int s = 0;
        for (int i : nums) s += Math.abs(i);
        if (t > s) return 0;
        int[][] f = new int[n + 1][2 * s + 1];
        f[0][0 + s] = 1;
        for (int i = 1; i <= n; i++) {
            int x = nums[i - 1];
            for (int j = -s; j <= s; j++) {
                if ((j - x) + s >= 0) f[i][j + s] += f[i - 1][(j - x) + s];
                if ((j + x) + s <= 2 * s) f[i][j + s] += f[i - 1][(j + x) + s];
            }
        }
        return f[n][t + s];
    }
}
```

- 时间复杂度：$O(n * \sum_{i=0}^{n-1} abs(nums[i]))$
- 空间复杂度：$O(n * \sum_{i=0}^{n-1} abs(nums[i]))$

动态规划（优化）

在上述「动态规划」分析中，我们总是尝试将所有的状态值都计算出来，当中包含很多对「目标状态」不可达的“额外”状态值。

即达成某些状态后，不可能再回到我们的「目标状态」。

例如当我们的 $target$ 不为 $-s$ 和 s 时，$-s$ 和 s 就是两个对「目标状态」不可达的“额外”状态值，到达 $-s$ 或 s 已经使用所有数值，对 $target$ 不可达。

那么我们如何规避掉这些“额外”状态值呢？

我们可以从哪些数值使用哪种符号来分析，即划分为「负值部分」&「非负值部分」，令「负值部分」的绝对值总和为 m，即可得：

$$(s-m)-m=s-2*m=target$$

变形得：

$$m=\frac{s-target}{2}$$

问题转换为：**只使用 $+$ 运算符，从 `nums` 凑出 m 的方案数。**

这样「原问题的具体方案」和「转换问题的具体方案」具有一一对应关系：「转换问题」中凑出来的数值部分在实际计算中应用 $-$，剩余部分应用 $+$，从而实现凑出来原问题的 $target$ 值。

另外，由于 `nums` 均为非负整数，因此我们需要确保 $s-target$ 能够被 2 整除。

同时，由于问题转换为 **从 `nums` 中凑出 m 的方案数，因此「状态定义」和「状态转移」都需要进行调整（01 背包求方案数）：**

定义 $f[i][j]$ 为从 `nums` 凑出总和「恰好」为 j 的方案数。

最终答案为 $f[n][m]$，$f[0][0]=1$ 为起始条件：代表不考虑任何数，凑出计算结果为 0 的方案数为 1 种。

每个数值有「选」和「不选」两种决策，转移方程为：

$$f[i][j]=f[i-1][j]+f[i-1][j-nums[i-1]]$$

代码：

```java
class Solution {
    public int findTargetSumWays(int[] nums, int t) {
        int n = nums.length;
        int s = 0;
        for (int i : nums) s += Math.abs(i);
        if (t > s || (s - t) % 2 != 0) return 0;
        int m = (s - t) / 2;
        int[][] f = new int[n + 1][m + 1];
        f[0][0] = 1;
        for (int i = 1; i <= n; i++) {
            int x = nums[i - 1];
            for (int j = 0; j <= m; j++) {
                f[i][j] += f[i - 1][j];
                if (j >= x) f[i][j] += f[i - 1][j - x];
            }
        }
        return f[n][m];
    }
}
```

- 时间复杂度：$O(n * (\sum_{i=0}^{n-1} abs(nums[i]) - target))$
- 空间复杂度：$O(n * (\sum_{i=0}^{n-1} abs(nums[i]) - target))$

> **🍭 更多精彩内容，欢迎关注：**公众号** / Github / LeetCode / **知乎** **

题目描述

这是 LeetCode 上的 **518. 零钱兑换 II** ，难度为 **中等**。

Tag : 「背包问题」、「完全背包」、「动态规划」

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。

示例 1:

```
输入: amount = 5, coins = [1, 2, 5]
输出: 4
解释: 有四种方式可以凑成总金额:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

示例 2:

```
输入: amount = 3, coins = [2]
输出: 0
解释: 只用面额2的硬币不能凑成总金额3。
```

示例 3:

```
输入: amount = 10, coins = [10]
输出: 1
```

注意:

你可以假设：

- 0 <= amount (总金额) <= 5000
- 1 <= coin (硬币面额) <= 5000
- 硬币种类不超过 500 种
- 结果符合 32 位符号整数

完全背包（朴素解法）

在 322. 零钱兑换 中，我们求的是「取得特定价值所需要的最小物品个数」。

对于本题，我们求的是「取得特定价值的方案数量」。

求的东西不一样，但问题的本质没有发生改变，同样属于「组合优化」问题。

你可以这样来理解什么是组合优化问题：

被选物品之间不需要满足特定关系，只需要选择物品，以达到「全局最优」或者「特定状态」即可。

同时硬币相当于我们的物品，每种硬币可以选择「无限次」，很自然的想到「完全背包」。

这时候可以将「完全背包」的状态定义搬过来进行“微调”：

定义 $f[i][j]$ 为考虑前 i 件物品，凑成总和为 j 的方案数量。

为了方便初始化，我们一般让 $f[0][x]$ 代表不考虑任何物品的情况。

因此我们有显而易见的初始化条件：$f[0][0] = 1$，其余 $f[0][x] = 0$。

代表当没有任何硬币的时候，存在凑成总和为 0 的方案数量为 1；凑成其他总和的方案不存在。

当「状态定义」与「基本初始化」有了之后，我们不失一般性的考虑 $f[i][j]$ 该如何转移。

对于第 i 个硬币我们有两种决策方案：

- 不使用该硬币：

$$f[i-1][j]$$

- 使用该硬币：由于每个硬币可以被选择多次（容量允许的情况下），因此方案数量应当是选择「任意个」该硬币的方案总和：

$$\sum_{k=1}^{\lfloor j/val \rfloor} f[i-1][j-k*val], val = nums[i-1]$$

代码：

```java
class Solution {
    public int change(int cnt, int[] cs) {
        int n = cs.length;
        int[][] f = new int[n + 1][cnt + 1];
        f[0][0] = 1;
        for (int i = 1; i <= n; i++) {
            int val = cs[i - 1];
            for (int j = 0; j <= cnt; j++) {
                f[i][j] = f[i - 1][j];
                for (int k = 1; k * val <= j; k++) {
                    f[i][j] += f[i - 1][j - k * val];
                }
            }
        }
        return f[n][cnt];
    }
}
```

- 时间复杂度：共有 $n * cnt$ 个状态需要转移，每个状态转移最多遍历 cnt 次。整体复杂度为 $O(n * cnt^2)$。
- 空间复杂度：$O(n * cnt)$。

完全背包（一维优化）

显然二维完全背包求解方案复杂度有点高。

n 的数据范围为 10^2，cnt 的数据范围为 10^3，总的计算量为 10^8 以上，处于超时边缘（实际测试可通过）。

我们需要对其进行「降维优化」，可以使用最开始讲的 数学分析方式，或者上一讲讲的 换元优化方式 进行降维优化。

由于 数学分析方式 十分耗时，我们用得更多的 换元优化方式。两者同样具有「可推广」特性。

因为后者更为常用，所以我们再来回顾一下如何进行「直接上手写一维空间优化的版本」：

1. 在二维解法的基础上，直接取消「物品维度」
2. 确保「容量维度」的遍历顺序为「从小到大」（适用于「完全背包」）
3. 将形如 $f[i-1][j-k*val]$ 的式子更替为 $f[j-val]$，同时解决「数组越界」

问题（将物品维度的遍历修改为从 val 开始）

代码：

```
class Solution {
    public int change(int cnt, int[] cs) {
        int n = cs.length;
        int[] f = new int[cnt + 1];
        f[0] = 1;
        for (int i = 1; i <= n; i++) {
            int val = cs[i - 1];
            for (int j = val; j <= cnt; j++) {
                f[j] += f[j - val];
            }
        }
        return f[cnt];
    }
}
```

- 时间复杂度：共有 $n * cnt$ 个状态需要转移，整体复杂度为 $O(n * cnt)$。
- 空间复杂度：$O(cnt)$。

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **879. 盈利计划** ，难度为 **困难**。

Tag : 「动态规划」、「容斥原理」、「数学」、「背包问题」、「多维背包」

集团里有 `n` 名员工，他们可以完成各种各样的工作创造利润。

第 `i` 种工作会产生 `profit[i]` 的利润，它要求 `group[i]` 名成员共同参与。如果成员参与了其中一项工作，就不能参与另一项工作。

工作的任何至少产生 `minProfit` 利润的子集称为 盈利计划 。并且工作的成员总数最多为 `n` 。

有多少种计划可以选择？因为答案很大，所以 返回结果模 $10^9 + 7$ 的值。

示例 1：

```
输入：n = 5, minProfit = 3, group = [2,2], profit = [2,3]
输出：2
解释：至少产生 3 的利润，该集团可以完成工作 0 和工作 1 ，或仅完成工作 1 。
总的来说，有两种计划。
```

示例 2：

```
输入：n = 10, minProfit = 5, group = [2,3,5], profit = [6,7,8]
输出：7
解释：至少产生 5 的利润，只要完成其中一种工作就行，所以该集团可以完成任何工作。
有 7 种可能的计划：(0)，(1)，(2)，(0,1)，(0,2)，(1,2)，以及 (0,1,2) 。
```

提示：

- 1 <= n <= 100
- 0 <= minProfit <= 100
- 1 <= group.length <= 100
- 1 <= group[i] <= 100
- profit.length == group.length
- 0 <= profit[i] <= 100

动态规划

这是一类特殊的多维费用背包问题。

将每个任务看作一个「物品」，完成任务所需要的人数看作「成本」，完成任务得到的利润看作「价值」。

其特殊在于存在一维容量维度需要满足「不低于」，而不是常规的「不超过」。这需要我们对于某些状态作等价变换。

定义 $f[i][j][k]$ 为考虑前 i 件物品，使用人数不超过 j，所得利润至少为 k 的方案数。

对于每件物品（令下标从 1 开始），我们有「选」和「不选」两种决策：

- 不选：显然有：

$$f[i-1][j][k]$$

- 选：首先需要满足人数达到要求（ $j >= group[i-1]$ ），还需要考虑「至少利润」负值问题：
  如果直接令「利润维度」为 $k - profit[i-1]$ 可能会出现负值，那么负值是否为合法状态呢？这需要结合「状态定义」来看，由于是「利润至少为 k」，因此属于「合法状态」，需要参与转移。
  由于我们没有设计动规数组存储「利润至少为负权」状态，我们需要根据「状态定义」做一个等价替换，将这个「状态」映射到 $f[i][j][0]$。这主要是利用所有的任务利润都为“非负数”，所以不可能出现利润为负的情况，这时候「利润至少为某个负数 k」的方案数其实是完全等价于「利润至少为 0」的方案数。

$$f[i-1][j-group[i-1]][\max(k-profit[i-1], 0)]$$

最终 $f[i][j][k]$ 为上述两种情况之和.

然后考虑「如何构造有效起始值」问题，还是结合我们的「状态定义」来考虑：

当不存在任何物品（任务）时，所得利用利润必然为 0（满足至少为 0），同时对人数限制没有要求。

因此可以让所有 $f[0][x][0] = 1$。

代码（一维空间优化代码见 P2）：

```java
class Solution {
    int mod = (int)1e9+7;
    public int profitableSchemes(int n, int min, int[] gs, int[] ps) {
        int m = gs.length;
        long[][][] f = new long[m + 1][n + 1][min + 1];
        for (int i = 0; i <= n; i++) f[0][i][0] = 1;
        for (int i = 1; i <= m; i++) {
            int a = gs[i - 1], b = ps[i - 1];
            for (int j = 0; j <= n; j++) {
                for (int k = 0; k <= min; k++) {
                    f[i][j][k] = f[i - 1][j][k];
                    if (j >= a) {
                        int u = Math.max(k - b, 0);
                        f[i][j][k] += f[i - 1][j - a][u];
                        f[i][j][k] %= mod;
                    }
                }
            }
        }
        return (int)f[m][n][min];
    }
}
```

```java
class Solution {
    int mod = (int)1e9+7;
    public int profitableSchemes(int n, int min, int[] gs, int[] ps) {
        int m = gs.length;
        int[][] f = new int[n + 1][min + 1];
        for (int i = 0; i <= n; i++) f[i][0] = 1;
        for (int i = 1; i <= m; i++) {
            int a = gs[i - 1], b = ps[i - 1];
            for (int j = n; j >= a; j--) {
                for (int k = min; k >= 0; k--) {
                    int u = Math.max(k - b, 0);
                    f[j][k] += f[j - a][u];
                    if (f[j][k] >= mod) f[j][k] -= mod;
                }
            }
        }
        return f[n][min];
    }
}
```

- 时间复杂度：$O(m * n * min)$
- 空间复杂度：$O(m * n * min)$

动态规划（作差法）

这个方案足足调了快一个小时 🤣

先是爆 `long`，然后转用高精度后被卡内存，最终改为滚动数组后~~勉强过了~~（不是，稳稳的过了，之前调得久是我把 `N` 多打了一位，写成 1005 了，`N` 不打错的话，不滚动也是能过的 😭😭😭 ）

基本思路是先不考虑最小利润 `minProfit`，求得所有只受「人数限制」的方案数 `a`，然后求得考虑「人数限制」同时，利润低于 `minProfit`（不超过 `minProfit - 1`）的所有方案数 `b`。

由 `a` - `b` 即是答案。

代码：

```java
import java.math.BigInteger;
class Solution {
    static int N = 105;
    static BigInteger[][] f = new BigInteger[2][N];
    static BigInteger[][][] g = new BigInteger[2][N][N];
    static BigInteger mod = new BigInteger("1000000007");

    public int profitableSchemes(int n, int min, int[] gs, int[] ps) {
        int m = gs.length;

        for (int j = 0; j <= n; j++) {
            f[0][j] = new BigInteger("1");
            f[1][j] = new BigInteger("0");
        }
        for (int j = 0; j <= n; j++) {
            for (int k = 0; k <= min; k++) {
                g[0][j][k] = new BigInteger("1");
                g[1][j][k] = new BigInteger("0");
            }
        }

        for (int i = 1; i <= m; i++) {
            int a = gs[i - 1], b = ps[i - 1];
            int x = i & 1, y = (i - 1) & 1;
            for (int j = 0; j <= n; j++) {
                f[x][j] = f[y][j];
                if (j >= a) {
                    f[x][j] = f[x][j].add(f[y][j - a]);
                }
            }
        }
        if (min == 0) return (f[m&1][n]).mod(mod).intValue();

        for (int i = 1; i <= m; i++) {
            int a = gs[i - 1], b = ps[i - 1];
            int x = i & 1, y = (i - 1) & 1;
            for (int j = 0; j <= n; j++) {
                for (int k = 0; k < min; k++) {
                    g[x][j][k] = g[y][j][k];
                    if (j - a >= 0 && k - b >= 0) {
                        g[x][j][k] = g[x][j][k].add(g[y][j - a][k - b]);
                    }
                }
            }
        }
```

```
        return f[m&1][n].subtract(g[m&1][n][min - 1]).mod(mod).intValue();
    }
}
```

- 时间复杂度：第一遍 `DP` 复杂度为 $O(m * n)$；第二遍 `DP` 复杂度为 $O(m * n * min)$。整体复杂度为 $O(m * n * min)$
- 空间复杂度：$O(m * n * min)$

> **🍭 更多精彩内容，欢迎关注：**公众号** / Github / LeetCode / **知乎** **

题目描述

这是 LeetCode 上的 **1049. 最后一块石头的重量 II** ，难度为 **中等**。

Tag : 「动态规划」、「背包问题」、「01 背包」、「数学」

有一堆石头，用整数数组 stones 表示。其中 stones[i] 表示第 i 块石头的重量。

每一回合，从中选出任意两块石头，然后将它们一起粉碎。假设石头的重量分别为 x 和 y，且 x <= y。那么粉碎的可能结果如下：

- 如果 x == y，那么两块石头都会被完全粉碎；
- 如果 x != y，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 y-x。

最后，最多只会剩下一块 石头。返回此石头 最小的可能重量 。如果没有石头剩下，就返回 0。

示例 1：

```
输入：stones = [2,7,4,1,8,1]
输出：1
解释：
组合 2 和 4，得到 2，所以数组转化为 [2,7,1,8,1]，
组合 7 和 8，得到 1，所以数组转化为 [2,1,1,1]，
组合 2 和 1，得到 1，所以数组转化为 [1,1,1]，
组合 1 和 1，得到 0，所以数组转化为 [1]，这就是最优值。
```

示例 2：

```
输入：stones = [31,26,33,21,40]
输出：5
```

示例 3：

```
输入：stones = [1,2]
输出：1
```

提示：

- 1 <= stones.length <= 30
- 1 <= stones[i] <= 100

基本分析

看到标题，心里咯噔了一下 🤣

一般性的石子合并问题通常是只能操作相邻的两个石子，要么是「区间 DP」要么是「四边形不等式」，怎么到 LeetCode 就成了中等难度的题目（也太卷了 🤣

仔细看了一下题目，可对任意石子进行操作，重放回的重量也不是操作石子的总和，而是操作石子的差值。

哦，那没事了 ~ 🤣

也是基于此启发，我们可以这样进行分析。

假设想要得到最优解，我们需要按照如下顺序操作石子：$[(sa, sb), (sc, sd), ..., (si, sj), (sp, sq)]$。

其中 $abcdijpq$ 代表了石子编号，**字母顺序不代表编号的大小关系**。

如果不考虑「有放回」的操作的话，我们可以划分为两个石子堆（正号堆/负号堆）：

- **将每次操作中「重量较大」的石子放到「正号堆」，代表在这次操作中该石子重量在「最终运算结果」中应用 $+$ 运算符**
- **将每次操作中「重量较少/相等」的石子放到「负号堆」，代表在这次操作中该石子重量在「最终运算结果」中应用 $-$ 运算符**

这意味我们最终得到的结果，可以为原来 $stones$ 数组中的数字添加 $+/-$ 符号，所形成的「计算表达式」所表示。

那有放回的石子重量如何考虑？

其实所谓的「有放回」操作，只是触发调整「某个原有石子」所在「哪个堆」中，并不会真正意义上的产生「新的石子重量」。

什么意思呢？

假设有起始石子 a 和 b，且两者重量关系为 $a \geq b$，那么首先会将 a 放入「正号堆」，将 b 放入「负号堆」。重放回操作可以看作产生一个新的重量为 $a - b$ 的“虚拟石子”，将来这个“虚拟石子”也会参与某次合并操作，也会被添加 $+/-$ 符号：

- 当对“虚拟石子”添加 $+$ 符号，即可 $+(a-b)$，展开后为 $a-b$，即起始石子 a 和 b 所在「石子堆」不变
- 当对“虚拟石子”添加 $-$ 符号，即可 $-(a-b)$，展开后为 $b-a$，即起始石子 a 和 b 所在「石子堆」交换

因此所谓不断「合并」&「重放」，本质只是在构造一个折叠的计算表达式，最终都能展开扁平化为非折叠的计算表达式。

综上，即使是包含「有放回」操作，最终的结果仍然可以使用「为原来 $stones$ 数组中的数字添加 $+/-$ 符号，形成的“计算表达式”」所表示。

动态规划

有了上述分析后，问题转换为：**为 $stones$ 中的每个数字添加 $+/-$，使得形成的「计算表达式」结果绝对值最小。**

与（题解）494. 目标和 类似，需要考虑正负号两边时，其实只需要考虑一边就可以了，使用总和 sum 减去决策出来的结果，就能得到另外一边的结果。

同时，由于想要「计算表达式」结果绝对值，因此我们需要将石子划分为差值最小的两个堆。

其实就是对「计算表达式」中带 $-$ 的数值提取公因数 -1，进一步转换为两堆石子相减总和，绝对值最小。

这就将问题彻底切换为 01 背包问题：**从 $stones$ 数组中选择，凑成总和不超过 $\frac{sum}{2}$ 的最大价值。**

其中「成本」&「价值」均为数值本身。

整理一下：

定义 $f[i][j]$ 代表考虑前 i 个物品（数值），凑成总和不超过 j 的最大价值。

每个物品都有「选」和「不选」两种决策，转移方程为：

$$f[i][j] = \max(f[i-1][j], f[i-1][j - stones[i-1]] + stones[i-1])$$

与完全背包不同，01 背包的几种空间优化是不存在时间复杂度上的优化，因此写成 朴素二维、滚动数组、一维优化 都可以。

建议直接上手写「一维空间优化」版本，是其他背包问题的基础。

代码：

```
class Solution {
    public int lastStoneWeightII(int[] ss) {
        int n = ss.length;
        int sum = 0;
        for (int i : ss) sum += i;
        int t = sum / 2;
        int[][] f = new int[n + 1][t + 1];
        for (int i = 1; i <= n; i++) {
            int x = ss[i - 1];
            for (int j = 0; j <= t; j++) {
                f[i][j] = f[i - 1][j];
                if (j >= x) f[i][j] = Math.max(f[i][j], f[i - 1][j - x] + x);
            }
        }
        return Math.abs(sum - f[n][t] - f[n][t]);
    }
}
```

```java
class Solution {
    public int lastStoneWeightII(int[] ss) {
        int n = ss.length;
        int sum = 0;
        for (int i : ss) sum += i;
        int t = sum / 2;
        int[][] f = new int[2][t + 1];
        for (int i = 1; i <= n; i++) {
            int x = ss[i - 1];
            int a = i & 1, b = (i - 1) & 1;
            for (int j = 0; j <= t; j++) {
                f[a][j] = f[b][j];
                if (j >= x) f[a][j] = Math.max(f[a][j], f[b][j - x] + x);
            }
        }
        return Math.abs(sum - f[n&1][t] - f[n&1][t]);
    }
}
```

```java
class Solution {
    public int lastStoneWeightII(int[] ss) {
        int n = ss.length;
        int sum = 0;
        for (int i : ss) sum += i;
        int t = sum / 2;
        int[] f = new int[t + 1];
        for (int i = 1; i <= n; i++) {
            int x = ss[i - 1];
            for (int j = t; j >= x; j--) {
                f[j] = Math.max(f[j], f[j - x] + x);
            }
        }
        return Math.abs(sum - f[t] - f[t]);
    }
}
```

- 时间复杂度：$O(n * \sum_{i=0}^{n-1} stones[i])$
- 空间复杂度：$O(n * \sum_{i=0}^{n-1} stones[i])$

题目描述

这是 LeetCode 上的 **1155. 掷骰子的N种方法** ，难度为 **中等**。

Tag : 「背包问题」、「动态规划」、「分组背包」

这里有 d 个一样的骰子，每个骰子上都有 f 个面，分别标号为 `1,2,...,f` 。

我们约定：掷骰子的得到总点数为各骰子面朝上的数字的总和。

如果需要掷出的总点数为 target，请你计算出有多少种不同的组合情况（所有的组合情况总共有 f^d 种），模 $10^9 + 7$ 后返回。

示例 1：

```
输入：d = 1, f = 6, target = 3

输出：1
```

示例 2：

```
输入：d = 2, f = 6, target = 7

输出：6
```

示例 3：

```
输入：d = 2, f = 5, target = 10

输出：1
```

示例 4：

```
输入：d = 1, f = 2, target = 3

输出：0
```

示例 5：

```
输入：d = 30, f = 30, target = 500

输出：222616187
```

提示：

- 1 <= d, f <= 30
- 1 <= target <= 1000

分组背包

在 分组背包问题 中我们提到，分组背包不仅仅有「组内物品最多选择一个」的情况，还存在「组内物品必须选择一个」的情况。

对于本题，可以将每个骰子看作一个物品组，且每次 **必须** 从物品组中选择一个物品（所掷得的数值大小视作具体物品）。

这样就把问题转换为：**用 d 个骰子（物品组）进行掷，掷出总和（取得的总价值）为 t 的方案数。**

虽然，我们还没专门讲过「背包问题求方案数」，但基本分析与「背包问题求最大价值」并无本质区别。

我们可以套用「分组背包求最大价值」的状态定义来微调：**$f[i][j]$ 表示考虑前 i 个物品组，凑成价值为 j 的方案数。**

为了方便，我们令物品组的编号从 1 开始，因此有显而易见的初始化条件 $f[0][0] = 1$。

代表在不考虑任何物品组的情况下，只有凑成总价值为 0 的方案数为 1，凑成其他总价值的方案不存在。

不失一般性考虑 $f[i][j]$ 该如何转移，也就是考虑第 i 个物品组有哪些决策。

根据题意，对于第 i 个物品组而言，可能决策的方案有：

- 第 i 个骰子的结果为 1，有 $f[i][j] = f[i-1][j-1]$
- 第 i 个骰子的结果为 2，有 $f[i][j] = f[i-1][j-2]$

  ...

- 第 i 个骰子的结果为 m，有 $f[i][j] = f[i-1][j-m]$

$f[i][j]$ 则是上述所有可能方案的方案数总和，即有：

$$f[i][j] = \sum_{k=1}^{m} f[i-1][j-k], j >= k$$

朴素二维

代码：

```
class Solution {
    int mod = (int)1e9+7;
    public int numRollsToTarget(int n, int m, int t) {
        int[][] f = new int[n + 1][t + 1];
        f[0][0] = 1;
        // 枚举物品组（每个骰子）
        for (int i = 1; i <= n; i++) {
            // 枚举背包容量（所掷得的总点数）
            for (int j = 0; j <= t; j++) {
                // 枚举决策（当前骰子所掷得的点数）
                for (int k = 1; k <= m; k++) {
                    if (j >= k) {
                        f[i][j] = (f[i][j] + f[i-1][j-k]) % mod;
                    }
                }
            }
        }
        return f[n][t];
    }
}
```

- 时间复杂度：$O(n * m * t)$
- 空间复杂度：$O(n * t)$

滚动数组

根据状态转移方程，我们发现 $f[i][j]$ 明确只依赖于 $f[i-1][x]$，且 $x < j$。

因此我们可以使用之前学过的「滚动数组」，用很机械的方式将空间从 $O(n * t)$ 优化至 $O(t)$。

需要注意的是，由于我们直接是在 $f[i][j]$ 格子的基础上进行方案数累加，因此在计算 $f[i][j]$

记得手动置零。

代码：

```
class Solution {
    int mod = (int)1e9+7;
    public int numRollsToTarget(int n, int m, int t) {
        int[][] f = new int[2][t + 1];
        f[0][0] = 1;
        for (int i = 1; i <= n; i++) {
            int a = i & 1, b = (i - 1) & 1;
            for (int j = 0; j <= t; j++) {
                f[a][j] = 0; // 先手动置零
                for (int k = 1; k <= m; k++) {
                    if (j >= k) {
                        f[a][j] = (f[a][j] + f[b][j-k]) % mod;
                    }
                }
            }
        }
        return f[n&1][t];
    }
}
```

- 时间复杂度：$O(n * m * t)$
- 空间复杂度：$O(t)$

一维空间优化

更进一步，利用「$f[i][j]$ 明确只依赖于 $f[i-1][x]$，且 $x < j$ 」，我们能通过「01 背包」一维空间优化方式：**将物品维度取消，调整容量维度遍历顺序为「从大到小」。**

代码：

```java
class Solution {
    int mod = (int)1e9+7;
    public int numRollsToTarget(int n, int m, int t) {
        int[] f = new int[t + 1];
        f[0] = 1;
        for (int i = 1; i <= n; i++) {
            for (int j = t; j >= 0; j--) {
                f[j] = 0;
                for (int k = 1; k <= m; k++) {
                    if (j >= k) {
                        f[j] = (f[j] + f[j-k]) % mod;
                    }
                }
            }
        }
        return f[t];
    }
}
```

- 时间复杂度：$O(n*m*t)$
- 空间复杂度：$O(t)$

总结

不难发现，不管是「组内物品最多选一件」还是「组内物品必须选一件」。

我们都是直接套用分组背包基本思路 *「枚举物品组-枚举容量-枚举决策」* 进行求解。

分组背包的空间优化并不会降低时间复杂度，所以对于分组背包问题，我们可以直接写方便调试的朴素多维版本（在空间可接受的情况下），如果遇到卡空间，再通过机械的方式改为「滚动数组」形式。

另外今天我们使用「分组背包问题求方案数」来作为「分组背包问题求最大价值」的练习题。

可以发现，两者其实并无本质区别，都是套用「背包问题求最大价值」的状态定义来微调。

更多的关于「背包问题求方案数」相关内容，在后面也会继续细讲。

> **🍭 更多精彩内容，欢迎关注：**公众号** / **Github** / **LeetCode** / **知乎** **

题目描述

这是 LeetCode 上的 **1449. 数位成本和为目标值的最大数字** ，难度为 **困难**。

Tag : 「完全背包」、「背包问题」、「动态规划」

给你一个整数数组 cost 和一个整数 target 。请你返回满足如下规则可以得到的 最大 整数：

给当前结果添加一个数位（i + 1）的成本为 cost[i] （cost 数组下标从 0 开始）。
总成本必须恰好等于 target 。
添加的数位中没有数字 0 。
由于答案可能会很大，请你以字符串形式返回。

如果按照上述要求无法得到任何整数，请你返回 "0" 。

示例 1：

```
输入：cost = [4,3,2,5,6,7,2,5,5], target = 9
输出："7772"
解释：添加数位 '7' 的成本为 2 ，添加数位 '2' 的成本为 3 。所以 "7772" 的代价为 2*3+ 3*1 = 9 。 "977" 也是满足要求的数字
 数字     成本
  1  ->   4
  2  ->   3
  3  ->   2
  4  ->   5
  5  ->   6
  6  ->   7
  7  ->   2
  8  ->   5
  9  ->   5
```

示例 2：

```
输入：cost = [7,6,5,5,5,6,8,7,8], target = 12
输出："85"
解释：添加数位 '8' 的成本是 7 ，添加数位 '5' 的成本是 5 。"85" 的成本为 7 + 5 = 12 。
```

示例 3：

```
输入：cost = [2,4,6,2,4,6,4,4,4], target = 5
输出："0"
解释：总成本是 target 的条件下，无法生成任何整数。
```

示例 4：

```
输入：cost = [6,10,15,40,40,40,40,40,40], target = 47
输出："32211"
```

提示：

- cost.length == 9
- 1 <= cost[i] <= 5000
- 1 <= target <= 5000

基本分析

根据题意：给定 $1 \sim 9$ 几个数字，每个数字都有选择成本，求给定费用情况下，凑成的最大数字是多少。

通常我们会如何比较两数大小关系？

首先我们 **根据长度进行比较，长度较长数字较大**；再者，对于长度相等的数值，**从高度往低位进行比较，找到第一位不同，不同位值大的数值较大。**

其中规则一的比较优先级要高于规则二。

基于此，我们可以将构造分两步进行。

动态规划 + 贪心

具体的，先考虑「数值长度」问题，每个数字有相应选择成本，所能提供的长度均为 1。

问题转换为：**有若干物品，求给定费用的前提下，花光所有费用所能选择的最大价值（物品个数）为多少。**

每个数字可以被选择多次，属于完全背包模型。

当求得最大「数值长度」后，考虑如何构造答案。

根据规则二，**应该尽可能让高位的数值越大越好**，因此我们可以从数值 9 开始往数值 1 遍历，如果状态能够由该数值转移而来，则选择该数值。

PS. 写了几天两维版本了，大家应该都掌握了叭，今天赶着出门，直接写一维。

代码：

```
class Solution {
    public String largestNumber(int[] cost, int t) {
        int[] f = new int[t + 1];
        Arrays.fill(f, Integer.MIN_VALUE);
        f[0] = 0;
        for (int i = 1; i <= 9; i++) {
            int u = cost[i - 1];
            for (int j = u; j <= t; j++) {
                f[j] = Math.max(f[j], f[j - u] + 1);
            }
        }
        if (f[t] < 0) return "0";
        String ans = "";
        for (int i = 9, j = t; i >= 1; i--) {
            int u = cost[i - 1];
            while (j >= u && f[j] == f[j - u] + 1) {
                ans += String.valueOf(i);
                j -= u;
            }
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n * t)$
- 空间复杂度：$O(t)$

思考 & 进阶

懂得分两步考虑的话，这道题还是挺简单。虽然是「DP」+「贪心」，但两部分都不难。

其实这道题改改条件/思路，也能衍生出几个版本：

0. **【思考】如何彻底转化为「01 背包」或者「多重背包」来处理？**
   完全背包经过一维优化后时间复杂度为 $O(N * C)$。是否可以在不超过此复杂度的前提下，通过预处理物品将问题转换为另外两种传统背包？
   - 对于「多重背包」答案是可以的。由于给定的最终费用 t，我们可以明确算出每个物品最多被选择的次数，可以在 $O(N)$ 的复杂度内预处理额外的 $s[]$ 数组。然后配合「单调队列优化」，做到 $O(N * C)$ 复杂度，整体复杂度不会因此变得更差。
     但转换增加了「预处理」的计算量。为了让转换变成“更有意义”，我们可以在「预处理」时顺便做一个小优化：**对于相同成本的数字，只保留数值大的数字**。不难证明，当成本相同时，选择更大的数字不会让结果变差。
   - 对于「01 背包」答案是不可以。原因与「多重背包」单纯转换为「01 背包」不会降低复杂度一致。因此本题转换成「01 背包」会使得 N 发生非常数级别的增大。
1. **【进阶】不再是给定数值 1~9（取消 $cost$ 数组），转为给定 $nums$ 数组（代表所能选择的数字，不包含 0），和相应 $price$ 数组（长度与 $nums$ 一致，代表选择 $nums[i]$ 所消耗的成本为 $price[i]$）。现有做法是否会失效？**
   此时 $nums$ 中不再是只有长度为 1 的数值了。但我们「判断数值大小」的两条规则不变。因此「第一步」不需要做出调整，但在进行「第二步」开始前，我们要先对物品进行「自定义规则」的排序，确保「贪心」构造答案过程是正确的。规则与证明都不难请自行思考。
2. **【进阶】在进阶 1 的前提下，允许 $nums$ 出现 0，且确保答案有解（不会返回答案 0）,该如何求解？**
   增加数值 0 其实只会对最高位数字的决策产生影响。
   我们可以**通过预处理转换为「分组 & 树形」背包问题**：将 $nums$ 中的非 0 作为一组「主件」（分组背包部分：必须选择一个主件），所有数值作为「附属件」（树形背包部分：能选择若干个，选择附属件必须同时选择主件）。

载链接。

觉得专题不错，可以请作者吃糖🍭🍭🍭：

版权声明：任何形式的转载请保留出处 Wiki。