

「代码随想录」背包问题专题精讲

作者：[程序员Carl](#)

目前「代码随想录」已经发布了如下手册：

- [二叉树学习手册PDF](#)
- [回溯算法学习手册PDF](#)
- [贪心算法学习手册PDF](#)

每一本手册都广受好评，这也是Carl花费大量时间写题解的动力，感谢大家的支持！

这本pdf是由公众号「代码随想录」背包专题的文章整理而来。

共4万字，18篇精品文章，详细讲解了10道leetcode背包问题经典题目

背包问题是动态规划里的非常重要的一部分，目前「代码随想录」还在更新动态规划专题，但背包已经更新完了，考虑到录友们想尽快看PDF，所以先把背包问题整理出来。

依旧保持「代码随想录」严谨缜密的风格，这是全网最全最详细的背包专题讲解！

如何使用这本PDF？

就是按顺序刷就可以了！

题目顺序都编排好了，按照pdf里排好的题目顺序来刷效果最好，这份刷题顺序已经陪伴上万录友（代码随想录的朋友们）。

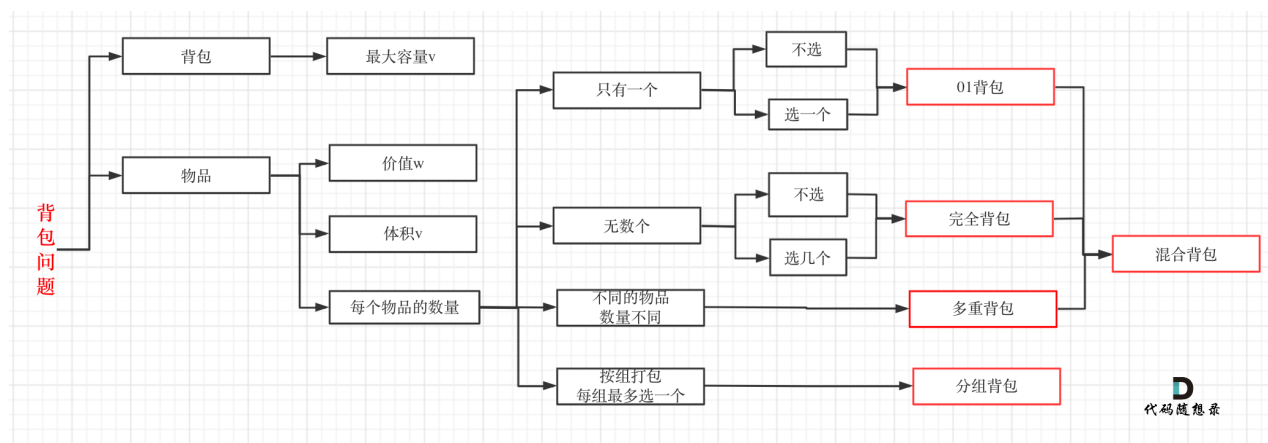
本PDF力扣题目大纲如下：



大家可以发现这里详细介绍了01背包的完全背包，而没有多重背包的题目，其实力扣上并没对应的题目，所以我把多重背包的理论基础讲一讲。

至于背包九讲里面还有混合背包，二维费用背包，分组背包等等这些，大家感兴趣可以自己去学习学习，这里也不做介绍了，面试也不会考。

关于这几种常见的背包，其关系如下：



通过这个图，可以很清晰分清这几种常见背包之间的关系。

在讲解背包问题的时候，都是围绕着动态五部曲进行讲解，把这五部都搞透才算是对动规理解深入了。

动规五部曲如下：

1. 确定dp数组（dp table）以及下标的含义
2. 确定递推公式
3. dp数组如何初始化
4. 确定遍历顺序
5. 举例推导dp数组

这五部里哪一步都很关键！，大家仔细看完PDF就会感受出来了。

跟进最新文章，关注微信公众号「代码随想录」，你会发现相见恨晚！



很多小伙伴刷题的时候面对力扣上近两千道题目，感觉无从下手，我花费半年时间整理了Github项目：「力扣刷题攻略」<https://github.com/youngyangyang04/leetcode-master>。里面有100多道经典算法题目刷题顺序、配有40w字的详细图解，常用算法模板总结，以及难点视频讲解，按照list一道一道刷就可以了！star支持一波吧！

公众号：[代码随想录](#)

B站：[代码随想录](#)

Github：[leetcode-master](#)

知乎：[代码随想录](#)

后续将在公众号陆续发布其他算法专题的pdf版本，敬请期待！

动态规划：关于01背包问题，你该了解这些！

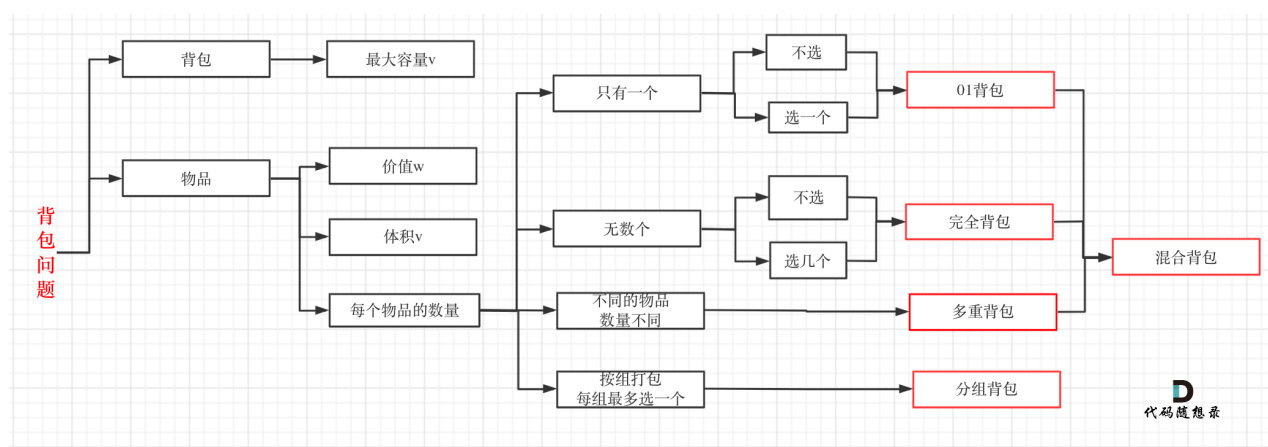
这周我们正式开始讲解背包问题！

背包问题的经典资料当然是：背包九讲。在公众号「代码随想录」后台回复：背包九讲，就可以获得背包九讲的PDF。

但说实话，背包九讲对于小白来说确实不太友好，看起来还是有点费劲的，而且都是伪代码理解起来也吃力。

对于面试的话，其实掌握01背包，和完全背包，就够用了，最多可以再来一个多重背包。

如果这几种背包，分不清，我这里画了一个图，如下：



至于背包九讲其他背包，面试几乎不会问，都是竞赛级别的了，leetcode上连多重背包的题目都没有，所以题库也告诉我们，01背包和完全背包就够用了。

而完全背包又是也是01背包稍作变化而来，即：完全背包的物品数量是无限的。

所以背包问题的理论基础重中之重是01背包，一定要理解透！

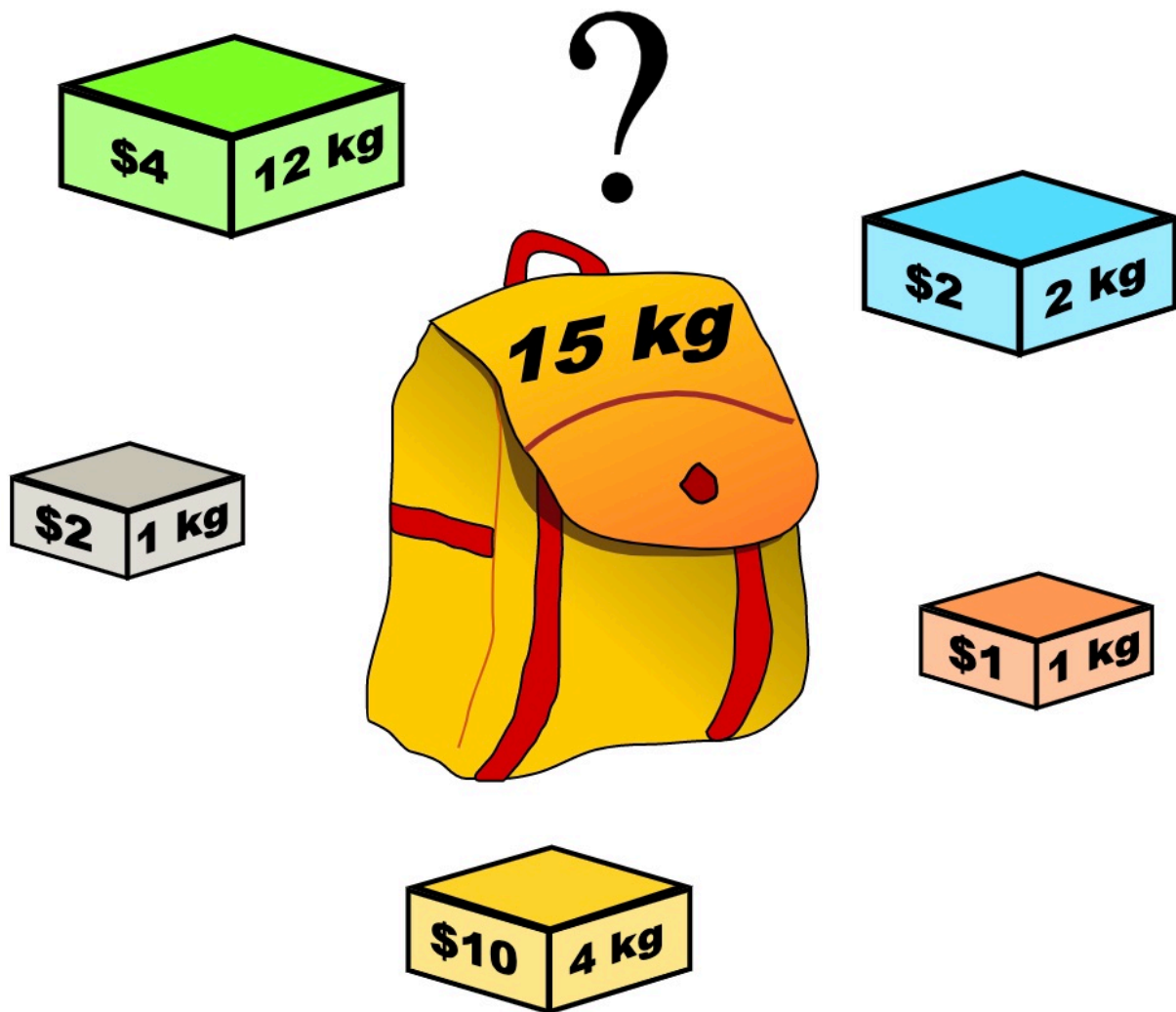
leetcode上没有纯01背包的问题，都是01背包应用方面的题目，也就是需要转化为01背包问题。

所以我先通过纯01背包问题，把01背包原理讲清楚，后续再讲解leetcode题目的时候，重点就是讲解如何转化为01背包问题了。

之前可能有些录友已经可以熟练写出背包了，但只要把这个文章仔细看完，相信你会意外收获！

01 背包

有N件物品和一个最多能被重量为W 的背包。第i件物品的重量是weight[i]，得到的价值是value[i]。每件物品只能用一次，求解将哪些物品装入背包里物品价值总和最大。



这是标准的背包问题，以至于很多同学看了这个自然就会想到背包，甚至都不知道暴力的解法应该怎么解了。

这样其实是没有从底向上去思考，而是习惯性想到了背包，那么暴力的解法应该是怎么样的呢？

每一件物品其实只有两个状态，取或者不取，所以可以使用回溯法搜索出所有的情况，那么时间复杂度就是 $O(2^n)$ ，这里的n表示物品数量。

所以暴力的解法是指数级别的时间复杂度。进而才需要动态规划的解法来进行优化！

在下面的讲解中，我举一个例子：

背包最大重量为4。

物品为：

	重量	价值
物品0	1	15
物品1	3	20
物品2	4	30

问背包能背的物品最大价值是多少？

以下讲解和图示中出现的数字都是以这个例子为例。

二维dp数组01背包

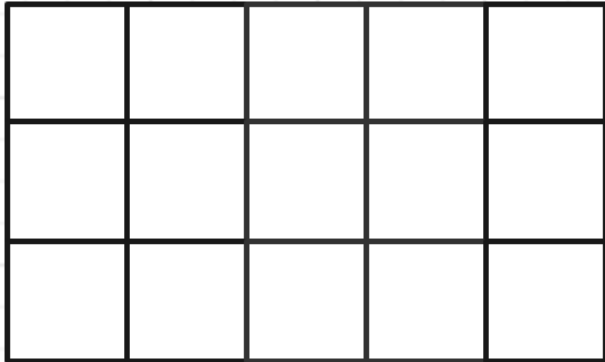
依然动规五部曲分析一波。

1. 确定dp数组以及下标的含义

对于背包问题，有一种写法，是使用二维数组，即 $dp[i][j]$ 表示从下标为 $[0-i]$ 的物品里任意取，放进容量为 j 的背包，价值总和最大是多少。

只看这个二维数组的定义，大家一定会有点懵，看下面这个图：

$dp[i][j]$		背包重量j:				
		0	1	2	3	4
物品0:						
物品1:						
物品2:						



要时刻记着这个dp数组的含义，下面的一些步骤都围绕这dp数组的含义进行的，如果哪里看懵了，就来回顾一下 i 代表什么， j 又代表什么。

2. 确定递推公式

再回顾一下 $dp[i][j]$ 的含义：从下标为 $[0-i]$ 的物品里任意取，放进容量为 j 的背包，价值总和最大是多少。

那么可以有两个方向推出来 $dp[i][j]$,

- 由 $dp[i-1][j]$ 推出，即背包容量为 j ，里面不放物品 i 的最大价值，此时 $dp[i][j]$ 就是 $dp[i-1][j]$
- 由 $dp[i-1][j - \text{weight}[i]]$ 推出， $dp[i-1][j - \text{weight}[i]]$ 为背包容量为 $j - \text{weight}[i]$ 的时候不放物品 i 的最大价值，那么 $dp[i-1][j - \text{weight}[i]] + \text{value}[i]$ （物品 i 的价值），就是背包放物品 i 得到的最大价值

如何确定递推公式？参考01背包问题详尽入门讲解


所以递归公式： $dp[i][j] = \max(dp[i-1][j], dp[i-1][j - \text{weight}[i]] + \text{value}[i])$;

3. dp数组如何初始化

关于初始化，一定要和dp数组的定义吻合，否则到递推公式的时候就会越来越乱。

首先从 $dp[i][j]$ 的定义触发，如果背包容量j为0的话，即 $dp[i][0]$ ，无论是选取哪些物品，背包价值总和一定为0。如图：

$dp[i][j]$		背包重量j:				
		0	1	2	3	4
物品0:	0					
物品1:	0					
物品2:	0					



在看其他情况。

状态转移方程 $dp[i][j] = \max(dp[i-1][j], dp[i-1][j - \text{weight}[i]] + \text{value}[i])$; 可以看出i 是由 i-1 推导出来，那么i为0的时候就一定要初始化。

$dp[0][j]$ ，即：i为0，存放编号0的物品的時候，各个容量的背包所能存放的最大价值。

代码如下：

```
// 倒叙遍历
for (int j = bagweight; j >= weight[0]; j--) {
    dp[0][j] = dp[0][j - weight[0]] + value[0]; // 初始化i为0时候的情况
}
```

大家应该发现，这个初始化为什么是倒叙的遍历的？正序遍历就不行么？

正序遍历还真就不行， $dp[0][j]$ 表示容量为j的背包存放物品0时候的最大价值，物品0的价值就是15，因为题目中说了每个物品只有一个！所以 $dp[0][j]$ 如果不是初始值的话，就应该都是物品0的价值，也就是15。

但如果一旦正序遍历了，那么物品0就会被重复加入多次！例如代码如下：

```
// 正序遍历
for (int j = weight[0]; j <= bagweight; j++) {
    dp[0][j] = dp[0][j - weight[0]] + value[0];
}
```

例如dp[0][1] 是15, 到了dp[0][2] = dp[0][2 - 1] + 15; 也就是dp[0][2] = 30 了, 那么就是物品0被重复放入了。

所以一定要倒叙遍历, 保证物品0只被放入一次! 这一点对01背包很重要, 后面在讲解滚动数组的时候, 还会用到倒叙遍历来保证物品使用一次!

此时dp数组初始化情况如图所示:

dp[i][j]		背包重量j:				
		0	1	2	3	4
物品0:		0	15	15	15	15
物品1:		0				
物品2:		0				



dp[0][j] 和 dp[i][0] 都已经初始化了, 那么其他下标应该初始化多少呢?

dp[i][j]在推导的时候一定是取价值最大的数, 如果题目给的价值都是正整数那么非0下标都初始化为0就可以了, 因为0就是最小的了, 不会影响取最大价值的结果。

如果题目给的价值有负数, 那么非0下标就要初始化为负无穷了。例如: 一个物品的价值是-2, 但对应的位置依然初始化为0, 那么取最大值的时候, 就会取0而不是-2了, 所以要初始化为负无穷。

这样才能让dp数组在递归公式的过程中取最大的价值, 而不是被初始值覆盖了。

最后初始化代码如下:


```
// 初始化 dp
vector<vector<int>> dp(weight.size() + 1, vector<int>(bagweight + 1, 0));
for (int j = bagweight; j >= weight[0]; j--) {
    dp[0][j] = dp[0][j - weight[0]] + value[0];
}
```

费了这么大的功夫, 才把如何初始化讲清楚, 相信不少同学平时初始化dp数组是凭感觉来的, 但有时候感觉是不靠谱的。

4. 确定遍历顺序

在如下图中，可以看出，有两个遍历的维度：物品与背包重量

dp[i][j]		背包重量j:				
		0	1	2	3	4
物品0:		0	15	15	15	15
物品1:		0	0	0	0	0
物品2:		0	0	0	0	0



那么问题来了，先遍历 物品还是先遍历背包重量呢？

其实都可以！！但是先遍历物品更好理解。

那么我先给出先遍历物品，然后遍历背包重量的代码。

```
// weight数组的大小 就是物品个数
for(int i = 1; i < weight.size(); i++) { // 遍历物品
    for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
        if (j < weight[i]) dp[i][j] = dp[i - 1][j]; // 这个是为了展现dp数组里元素的变化
        else dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);
    }
}
```

先遍历背包，再遍历物品，也是可以的！（注意我这里使用的二维dp数组）

例如这样：

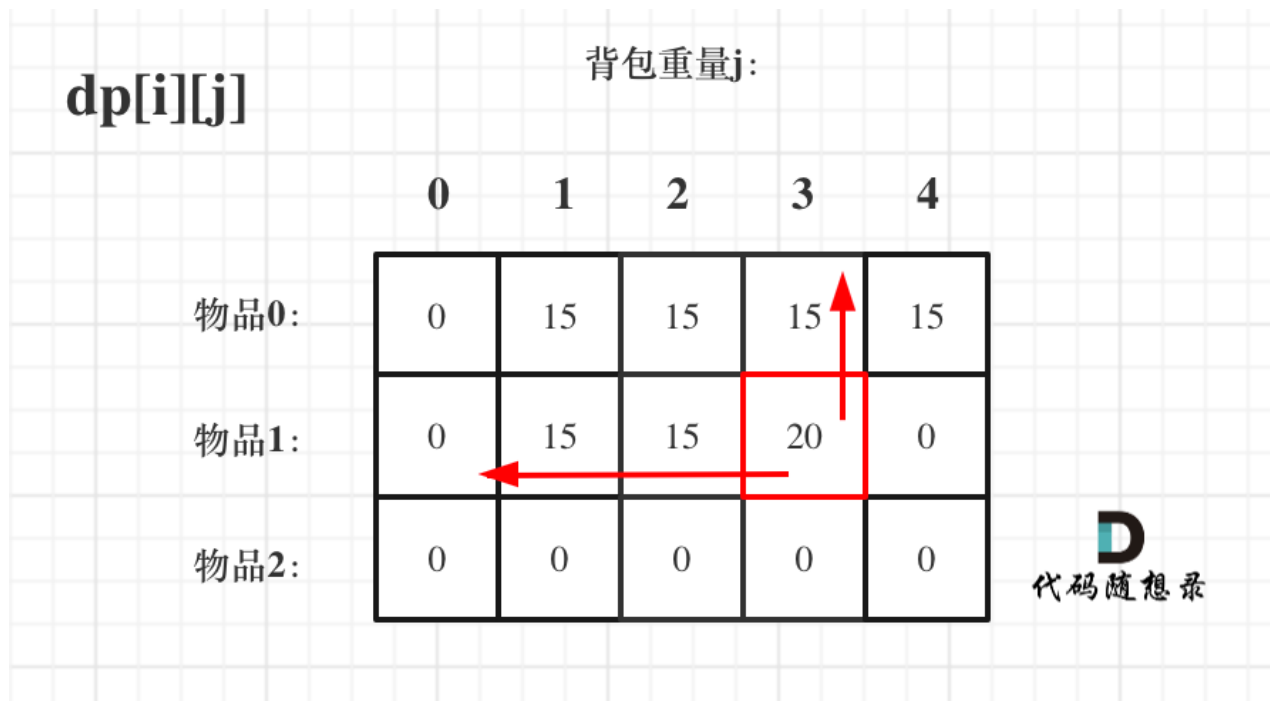
```
// weight数组的大小 就是物品个数
for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
    for(int i = 1; i < weight.size(); i++) { // 遍历物品
        if (j < weight[i]) dp[i][j] = dp[i - 1][j];
        else dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);
    }
}
```

为什么也是可以的呢？

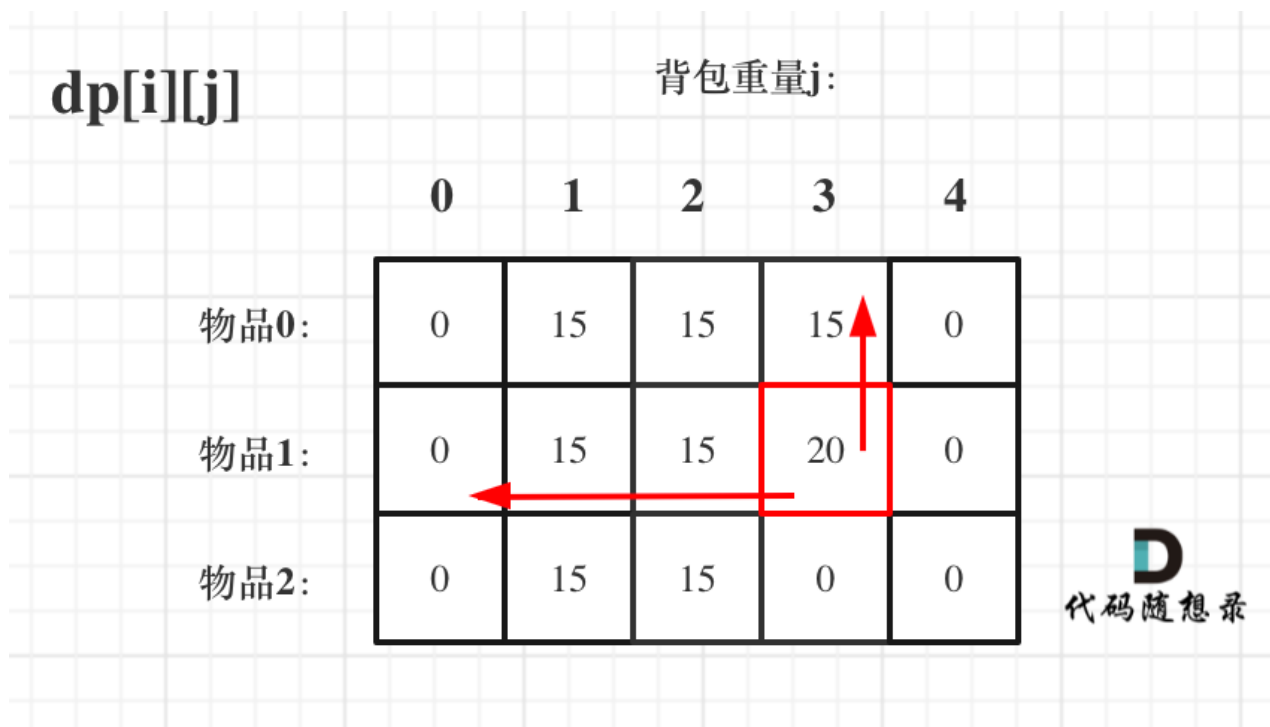
要理解递归的本质和递推的方向。

$dp[i][j] = \max(dp[i-1][j], dp[i-1][j - \text{weight}[i]] + \text{value}[i])$; 递归公式中可以看出 $dp[i][j]$ 是靠 $dp[i-1][j]$ 和 $dp[i-1][j - \text{weight}[i]]$ 推导出来的。

$dp[i-1][j]$ 和 $dp[i-1][j - \text{weight}[i]]$ 都在 $dp[i][j]$ 的左上角方向（包括正左和正上两个方向），那么先遍历物品，再遍历背包的过程如图所示：



再来看看先遍历背包，再遍历物品呢，如图：



大家可以看出，虽然两个for循环遍历的次序不同，但是 $dp[i][j]$ 所需要的数据就是左上角，根本不影响 $dp[i][j]$ 公式的推导！

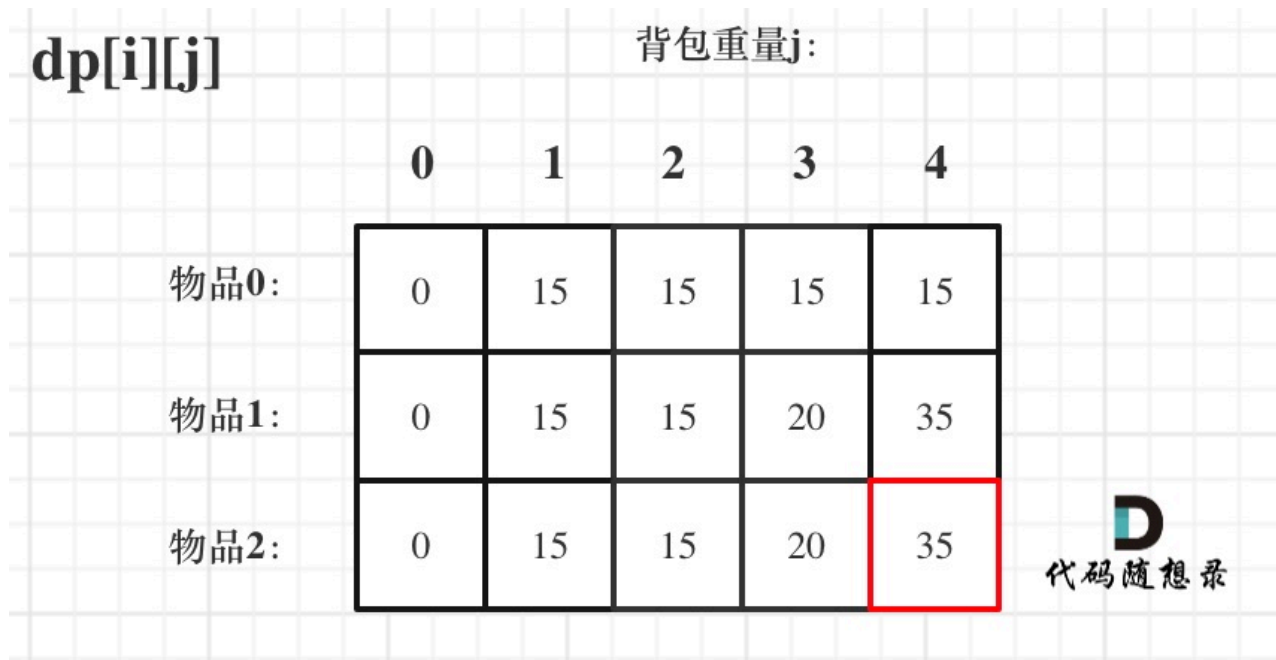
但先遍历物品再遍历背包这个顺序更好理解。

其实背包问题里，两个for循环的先后顺序是非常有讲究的，理解遍历顺序其实比理解推导公式难多了。

5. 举例推导dp数组

来看一下对应的dp数组的数值，如图：

dp[i][j]		背包重量j:				
		0	1	2	3	4
物品0:		0	15	15	15	15
物品1:		0	15	15	20	35
物品2:		0	15	15	20	35



最终结果就是dp[2][4]。

建议大家此时自己在纸上推导一遍，看看dp数组里每一个数值是不是这样的。

做动态规划的题目，最好的过程就是自己在纸上举一个例子把对应的dp数组的数值推导一下，然后在动手写代码！

很多同学做dp题目，遇到各种问题，然后凭感觉东改改西改改，怎么改都不对，或者稀里糊涂就改过了。

主要就是自己没有动手推导一下dp数组的演变过程，如果推导明白了，代码写出来就算有问题，只要把dp数组打印出来，对比一下和自己推导的有什么差异，很快就可以发现问题了。

完整C++测试代码

```
void test_2_we_i_bag_problem1() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    int bagweight = 4;

    // 二维数组
    vector<vector<int>> dp(weight.size() + 1, vector<int>(bagweight + 1, 0));

    // 初始化
    for (int j = bagweight; j >= weight[0]; j--) {
        dp[0][j] = dp[0][j - weight[0]] + value[0];
    }
}
```

```

// weight数组的大小 就是物品个数
for(int i = 1; i < weight.size(); i++) { // 遍历物品
    for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
        if (j < weight[i]) dp[i][j] = dp[i - 1][j];
        else dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] +
value[i]);
    }
}

cout << dp[weight.size() - 1][bagweight] << endl;
}

int main() {
    test_2_weigh_bag_problem1();
}

```

0	15	15	15	15
0	15	15	20	35
0	15	15	20	35

以上遍历的过程也可以这么写：

```

// 遍历过程
for(int i = 1; i < weight.size(); i++) { // 遍历物品
    for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
        if (j - weight[i] >= 0) {
            dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);
        }
    }
}
}

```

这么写打印出来的dp数据这就是这样：

$dp[i][j]$

背包重量j:

	0	1	2	3	4
物品0:	0	15	15	15	15
物品1:	0	0	0	20	35
物品2:	0	0	0	0	35

 代码随想录

空出来的0其实是用不上的，版本一 能把完整的dp数组打印出来，出来我用版本一来讲解。

总结

讲了这么多才刚刚把二维dp的01背包讲完，这里大家其实可以发现最简单的是推导公式了，推导公式估计看一遍就记下来了，但难就难在如何初始化和遍历顺序上。

可能有的同学并没有注意到初始化和遍历顺序的重要性，我们后面做力扣上背包面试题目的时候，大家就会感受出来了。

下一篇 还是理论基础，我们再来讲一维dp数组实现的01背包（滚动数组），分析一下和二维有什么区别，在初始化和遍历顺序上又有什么差异，敬请期待！

动态规划：关于01背包问题，你该了解这些！（滚动数组）

昨天[动态规划：关于01背包问题，你该了解这些！](#)中是用二维dp数组来讲解01背包。

今天我们就来说一说滚动数组，其实在前面的题目中我们已经用到过滚动数组了，就是把二维dp降为一维dp，一些录友当时还表示比较困惑。

那么我们通过01背包，来彻底讲一讲滚动数组！

接下来还是用如下这个例子来进行讲解

背包最大重量为4。

物品为：

	重量	价值
物品0	1	15
物品1	3	20
物品2	4	30

问背包能背的物品最大价值是多少？

一维dp数组（滚动数组）

对于背包问题其实状态都是可以压缩的。

在使用二维数组的时候，递推公式： $dp[i][j] = \max(dp[i-1][j], dp[i-1][j - \text{weight}[i]] + \text{value}[i])$;

其实可以发现如果把 $dp[i-1]$ 那一层拷贝到 $dp[i]$ 上，表达式完全可以是： $dp[i][j] = \max(dp[i][j], dp[i][j - \text{weight}[i]] + \text{value}[i])$;

于其把 $dp[i-1]$ 这一层拷贝到 $dp[i]$ 上，不如只用一个一维数组了，只用 $dp[j]$ （一维数组，也可以理解是一个滚动数组）。

这就是滚动数组的由来，需要满足的条件是上一层可以重复利用，直接拷贝到当前层。

读到这里估计大家都忘了 $dp[i][j]$ 里的 i 和 j 表达的是怎么了， i 是物品， j 是背包容量。

$dp[i][j]$ 表示从下标为 $[0-i]$ 的物品里任意取，放进容量为 j 的背包，价值总和最大是多少。

一定要时刻记住这里 i 和 j 的含义，要不然很容易看懵了。

动规五部曲分析如下：

1. 确定dp数组的定义

在一维dp数组中， $dp[j]$ 表示：容量为 j 的背包，所背的物品价值可以最大为 $dp[j]$ 。

2. 一维dp数组的递推公式

$dp[j]$ 为 容量为 j 的背包所背的最大价值，那么如何推导 $dp[j]$ 呢？

$dp[j]$ 可以通过 $dp[j - \text{weight}[i]]$ 推导出来， $dp[j - \text{weight}[i]]$ 表示容量为 $j - \text{weight}[i]$ 的背包所背的最大价值。

$dp[j - \text{weight}[i]] + \text{value}[i]$ 表示 容量为 $j - \text{物品}i\text{重量}$ 的背包 加上 物品 i 的价值。（也就是容量为 j 的背包，放入物品 i 了之后的价值即： $dp[j]$ ）

此时 $dp[j]$ 有两个选择，一个是取自己 $dp[j]$ ，一个是取 $dp[j - \text{weight}[i]] + \text{value}[i]$ ，指定是取最大的，毕竟是求最大价值，

所以递归公式为：

$$dp[j] = \max(dp[j], dp[j - \text{weight}[i]] + \text{value}[i]);$$

可以看出相对于二维dp数组的写法，就是把 $dp[i][j]$ 中 i 的维度去掉了。

3. 一维dp数组如何初始化

关于初始化，一定要和dp数组的定义吻合，否则到递推公式的时候就会越来越乱。

dp[j]表示：容量为j的背包，所背的物品价值可以最大为dp[j]，那么dp[0]就应该是0，因为背包容量为0所背的物品的最大价值就是0。

那么dp数组除了下标0的位置，初始为0，其他下标应该初始化多少呢？

看一下递归公式：dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);

dp数组在推导的时候一定是取价值最大的数，如果题目给的价值都是正整数那么非0下标都初始化为0就可以了，如果题目给的价值有负数，那么非0下标就要初始化为负无穷。

这样才能让dp数组在递归公式的过程中取的最大的价值，而不是被初始值覆盖了。

那么我假设物品价值都是大于0的，所以dp数组初始化的时候，都初始为0就可以了。

4. 一维dp数组遍历顺序

代码如下：

```
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = bagweight; j >= weight[i]; j--) { // 遍历背包容量
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
}
```

这里大家发现和二维dp的写法中，遍历背包的顺序是不一样的！

二维dp遍历的时候，背包容量是从小到大，而一维dp遍历的时候，背包是从大到小。

为什么呢？

倒叙遍历是为了保证物品i只被放入一次！，在[动态规划：关于01背包问题，你该了解这些！](#)中讲解二维dp数组初始化dp[0][j]时候已经讲解到过一次。

举一个例子：物品0的重量weight[0] = 1，价值value[0] = 15

如果正序遍历

dp[1] = dp[1 - weight[0]] + value[0] = 15

dp[2] = dp[2 - weight[0]] + value[0] = 30

此时dp[2]就已经是30了，意味着物品0，被放入了两次，所以不能正序遍历。

为什么倒叙遍历，就可以保证物品只放入一次呢？

倒叙就是先算dp[2]

dp[2] = dp[2 - weight[0]] + value[0] = 15 （dp数组已经都初始化为0）

dp[1] = dp[1 - weight[0]] + value[0] = 15

所以从后往前循环，每次取得状态不会和之前取得状态重合，这样每种物品就只取一次了。

那么问题又来了，为什么二维dp数组历的时候不用倒叙呢？

因为对于二维dp， $dp[i][j]$ 都是通过上一层即 $dp[i-1][j]$ 计算而来，本层的 $dp[i][j]$ 并不会被覆盖！

（如何这里读不懂，大家就要动手试一试了，空想还是不靠谱的，实践出真知！）

再来看看两个嵌套for循环的顺序，代码中是先遍历物品嵌套遍历背包容量，那可不可以先遍历背包容量嵌套遍历物品呢？

不可以！

因为一维dp的写法，背包容量一定是要倒序遍历（原因上面已经讲了），如果遍历背包容量放在上一层，那么每个 $dp[j]$ 就只会放入一个物品，即：背包里只放入了一个物品。


（这里如果读不懂，就在回想一下 $dp[j]$ 的定义，或者就把两个for循环顺序颠倒一下试试！）

所以一维dp数组的背包在遍历顺序上和二维其实是有很大差异的！，这一点大家一定要注意。

5. 举例推导dp数组

一维dp，分别用物品0，物品1，物品2 来遍历背包，最终得到结果如下：

用物品0，遍历背包：	0	15	15	15	15
用物品1，遍历背包：	0	15	15	20	35
用物品2，遍历背包：	0	15	15	20	35


代码随想录

一维dp01背包完整C++测试代码

```
void test_1_wei_bag_problem() {  
    vector<int> weight = {1, 3, 4};  
    vector<int> value = {15, 20, 30};  
    int bagweight = 4;  
  
    // 初始化  
    vector<int> dp(bagweight + 1, 0);  
    for(int i = 0; i < weight.size(); i++) { // 遍历物品  
        for(int j = bagweight; j >= weight[i]; j--) { // 遍历背包容量
```

```
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
}
cout << dp[bagweight] << endl;
}

int main() {
    test_1_wei_bag_problem();
}
```

可以看出，一维dp的01背包，要比二维简洁的多！初始化 和 遍历顺序相对简单了。

所以我倾向于使用一维dp数组的写法，比较直观简洁，而且空间复杂度还降了一个数量级！

在后面背包问题的讲解中，我都直接使用一维dp数组来进行推导。

总结

以上的讲解可以开发一道面试题目（毕竟力扣上没原题）。

就是本文中的题目，要求先实现一个纯二维的01背包，如果写出来了，然后再问为什么两个for循环的嵌套顺序这么写？反过来写行不行？再讲一讲初始化的逻辑。

然后要求实现一个一维数组的01背包，最后再问，一维数组的01背包，两个for循环的顺序反过来写行不行？为什么？

注意以上问题都是在候选人把代码写出来的情况下才问的。

就是纯01背包的题目，都不用考01背包应用类的题目就可以看出候选人对算法的理解程度了。

相信大家读完这篇文章，应该对以上问题都有了答案！

此时01背包理论基础就讲完了，我用了两篇文章把01背包的dp数组定义、递推公式、初始化、遍历顺序从二维数组到一维数组统统深度剖析了一遍，没有放过任何难点。

大家可以发现其实信息量还是挺大的。

如果把[动态规划：关于01背包问题，你该了解这些！](#)和本篇的内容都理解了，后面我们在做01背包的题目，就会发现非常简单了。

不用再凭感觉或者记忆去写背包，而是有自己的思考，了解其本质，代码的方方面面都在自己的掌控之中。

即使代码没有通过，也会有自己的逻辑去debug，这样就思维清晰了。

接下来就要开始用这两天的理论基础去做力扣上的背包面试题目了，录友们握紧扶手，我们要上高速啦！

相信很多小伙伴刷题的时候面对力扣上近两千道题目，感觉无从下手，我花费半年时间整理了Github项目：「力扣刷题攻略」<https://github.com/youngyangyang04/leetcode-master>。里面有100多道经典算法题目刷题顺序、配有40w字的详细图解，常用算法模板总结，以及难点视频讲解，按照list一道一道刷就可以了！star支持一波吧！

- 公众号: [代码随想录](#)
- B站: [代码随想录](#)
- Github: [leetcode-master](#)
- 知乎: [代码随想录](#)



关注「代码随想录」

学习算法不迷路!

动态规划：分割等和子集可以用01背包！

416. 分割等和子集

题目链接: <https://leetcode-cn.com/problems/partition-equal-subset-sum/>

题目难易: 中等

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

这道题目初步看，是如下两题几乎是一样的，大家可以用回溯法，解决如下两题

- 698.划分为k个相等的子集
- 473.火柴拼正方形

这道题目是要找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。

那么只要找到集合里能够出现 $\text{sum} / 2$ 的子集总和，就算是分割成两个相同元素和子集了。

本题是可以回溯暴力搜索出所有答案的，但最后超时了，也不想再优化了，放弃回溯，直接上01背包吧。

如果对01背包不够了解，建议仔细看完如下两篇：

- [动态规划：关于01背包问题，你该了解这些！](#)
- [动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)

01背包问题

背包问题，大家都知道，有N件物品和一个最多能被重量为W 的背包。第i件物品的重量是 $\text{weight}[i]$ ，得到的价值是 $\text{value}[i]$ 。每件物品只能用一次，求解将哪些物品装入背包里物品价值总和最大。

背包问题有多种背包方式，常见的有：**01背包**、完全背包、多重背包、分组背包和混合背包等等。

要注意题目描述中商品是不是可以重复放入。

即一个商品如果可以重复多次放入是完全背包，而只能放入一次是**01背包**，写法还是不一样的。

要明确本题中我们要使用的是**01背包**，因为元素我们只能用一次。

回归主题：首先，本题要求集合里能否出现总和为 $\text{sum} / 2$ 的子集。

那么来一一对应一下本题，看看背包问题如果来解决。

只有确定了如下四点，才能把**01背包问题**套到本题上来。

- 背包的体积为 $\text{sum} / 2$
- 背包要放入的商品（集合里的元素）重量为 元素的数值，价值也为元素的数值
- 背包如何正好装满，说明找到了总和为 $\text{sum} / 2$ 的子集。
- 背包中每一个元素是不可重复放入。

背包体积 $\text{sum}/2$ ，
价值为 $\text{sum}/2$

以上分析完，我们就可以套用01背包，来解决这个问题了。

动规五部曲分析如下：

1. 确定dp数组以及下标的含义

01背包中， $\text{dp}[j]$ 表示：容量为j的背包，所背的物品价值可以最大为 $\text{dp}[j]$ 。

套到本题， $\text{dp}[i]$ 表示 背包总容量是i，最大可以凑成i的**子集总和为 $\text{dp}[i]$** 。

2. 确定递推公式

01背包的递推公式为： $\text{dp}[j] = \max(\text{dp}[j], \text{dp}[j - \text{weight}[i]] + \text{value}[i])$;

本题，相当于背包里放入数值，那么物品i的重量是nums[i]，其价值也是nums[i]。

所以递推公式： $dp[j] = \max(dp[j], dp[j - \text{nums}[i]] + \text{nums}[i])$;

3. dp数组如何初始化

在01背包，一维dp如何初始化，已经讲过，

从dp[j]的定义来看，首先dp[0]一定是0。

如果如果题目给的价值都是正整数那么非0下标都初始化为0就可以了，如果题目给的价值有负数，那么非0下标就要初始化为负无穷。

这样才能让dp数组在递归公式的过程中取的最大的价值，而不是被初始值覆盖了。

本题题目中 只包含正整数的非空数组，所以非0下标的元素初始化为0就可以了。

代码如下：

```
// 题目中说：每个数组中的元素不会超过 100，数组的大小不会超过 200
// 总和不会大于20000，背包最大只需要其中一半，所以10001大小就可以了
vector<int> dp(10001, 0);
```

4. 确定遍历顺序

在[动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)中就已经说明：如果使用一维dp数组，物品遍历的for循环放在外层，遍历背包的for循环放在内层，且内层for循环倒叙遍历！

代码如下：

```
// 开始 01背包
for(int i = 0; i < nums.size(); i++) {
    for(int j = target; j >= nums[i]; j--) { // 每一个元素一定是不可重复放入，所以从大到小遍历
        dp[j] = max(dp[j], dp[j - nums[i]] + nums[i]);
    }
}
```

5. 举例推导dp数组

dp[i]的数值一定是小于等于i的。

如果dp[i] == i说明，集合中的子集总和正好可以凑成总和i，理解这一点很重要。

用例1，输入[1,5,11,5]为例，如图：

输入: [1,5,11,5], target = (1+5+11+5)/2 = 11

下标i: 0 1 2 3 4 5 6 7 8 9 10 11

0	1	1	1	1	5	6	6	6	6	10	11
---	---	---	---	---	---	---	---	---	---	----	----

dp[11] == 11

D
代码随想录

最后dp[11] == 11, 说明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。

综上分析完毕, C++代码如下:

```
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int sum = 0;

        // dp[i]中的i表示背包内总和
        // 题目中说: 每个数组中的元素不会超过 100, 数组的大小不会超过 200
        // 总和不会大于20000, 背包最大只需要其中一半, 所以10001大小就可以了
        vector<int> dp(10001, 0);
        for (int i = 0; i < nums.size(); i++) {
            sum += nums[i];
        }
        if (sum % 2 == 1) return false;
        int target = sum / 2;

        // 开始 01背包
        for(int i = 0; i < nums.size(); i++) {
            for(int j = target; j >= nums[i]; j--) { // 每一个元素一定是不可重复放入, 所以从大到小遍历
                dp[j] = max(dp[j], dp[j - nums[i]] + nums[i]);
            }
        }
        // 集合中的元素正好可以凑成总和target
        if (dp[target] == target) return true;
        return false;
    }
};
```

- 时间复杂度: $O(n^2)$
- 空间复杂度: $O(n)$, 虽然dp数组大小为一个常数, 但是大常数

总结

这道题目就是一道01背包应用类的题目，需要我们拆解题目，然后套入01背包的场景。

01背包相对于本题，主要要理解，题目中物品是nums[i]，重量是nums[i]，价值也是nums[i]，背包体积是sum/2。

看代码的话，就可以发现，基本就是按照01背包的写法来的。

动态规划：最后一块石头的重量 II

1049. 最后一块石头的重量 II

题目链接：<https://leetcode-cn.com/problems/last-stone-weight-ii/>

题目难度：中等

有一堆石头，每块石头的重量都是正整数。

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 $x \leq y$ 。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 $x \neq y$ ，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：

输入：[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。

提示：

- $1 \leq \text{stones.length} \leq 30$
- $1 \leq \text{stones}[i] \leq 1000$

思路

如果对背包问题不都熟悉先看这两篇：

- [动态规划：关于01背包问题，你该了解这些！](#)
- [动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)

本题其实就**是尽量让石头分成重量相同的两堆，相撞之后剩下的石头最小，这样就化解成01背包问题了。**

是不是感觉和昨天讲解的[416. 分割等和子集](#)非常像了。

本题物品的重量为store[i]，物品的价值也为store[i]。

对应着01背包里的物品重量weight[i]和 物品价值value[i]。

接下来进行动规五步曲：

1. 确定dp数组以及下标的含义

dp[j]表示容量（这里说容量更形象，其实就是重量）为j的背包，最多可以背dp[j]这么重的石头。

2. 确定递推公式

01背包的递推公式为： $dp[j] = \max(dp[j], dp[j - \text{weight}[i]] + \text{value}[i])$;

本题则是： **$dp[j] = \max(dp[j], dp[j - \text{stones}[i]] + \text{stones}[i])$** ;

一些同学可能看到这 $dp[j - \text{stones}[i]] + \text{stones}[i]$ 中又有 $-\text{stones}[i]$ 又有 $+\text{stones}[i]$ ，看着有点晕乎。

还是要牢记dp[j]的含义，要知道 $dp[j - \text{stones}[i]]$ 为容量为 $j - \text{stones}[i]$ 的背包最大所背重量。

3. dp数组如何初始化

既然dp[j]中的j表示容量，那么最大容量（重量）是多少呢，就是所有石头的重量和。

因为提示中给出 $1 \leq \text{stones.length} \leq 30$, $1 \leq \text{stones}[i] \leq 1000$ ，所以最大重量就是 $30 * 1000$ 。

而我们要求的target其实只是最大重量的一半，所以dp数组开到15000大小就可以了。

当然也可以把石头遍历一遍，计算出石头总重量 然后除2，得到dp数组的大小。

我这里就直接用15000了。

接下来就是如何初始化dp[j]呢，因为重量都不会是负数，所以dp[j]都初始化为0就可以了，这样在递归公式 $dp[j] = \max(dp[j], dp[j - \text{stones}[i]] + \text{stones}[i])$ 中dp[j]才不会初始值所覆盖。

代码为：

```
vector<int> dp(15001, 0);
```

4. 确定遍历顺序

在[动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)中就已经说明：如果使用一维dp数组，物品遍历的for循环放在外层，遍历背包的for循环放在内层，且内层for循环倒叙遍历！

代码如下：

```
for (int i = 0; i < stones.size(); i++) { // 遍历物品
    for (int j = target; j >= stones[i]; j--) { // 遍历背包
        dp[j] = max(dp[j], dp[j - stones[i]] + stones[i]);
    }
}
```

5. 举例推导dp数组

举例，输入：[2,4,1,1]，此时 $\text{target} = (2 + 4 + 1 + 1)/2 = 4$ ，dp数组状态图如下：

输入: [2,4,1,1], target = 4

下标i: 0 1 2 3 4

用store[0], 遍历背包:

0	0	2	2	2
---	---	---	---	---

用store[1], 遍历背包:

0	0	2	2	4
---	---	---	---	---

用store[2], 遍历背包:

0	1	2	3	4
---	---	---	---	---

用store[3], 遍历背包:

0	1	2	3	4
---	---	---	---	---



公众号:代码随想录

最后dp[target]里是容量为target的背包所能背的最大重量。

那么分成两堆石头, 一堆石头的总重量是dp[target], 另一堆就是sum - dp[target]。

在计算target的时候, $target = sum / 2$ 因为是向下取整, 所以 $sum - dp[target]$ 一定是大于等于dp[target]的。

那么相撞之后剩下的最小石头重量就是 $(sum - dp[target]) - dp[target]$ 。

以上分析完毕, C++代码如下:

```
class Solution {
public:
    int lastStoneweightII(vector<int>& stones) {
        vector<int> dp(15001, 0);
        int sum = 0;
        for (int i = 0; i < stones.size(); i++) sum += stones[i];
        int target = sum / 2;
        for (int i = 0; i < stones.size(); i++) { // 遍历物品
            for (int j = target; j >= stones[i]; j--) { // 遍历背包
                dp[j] = max(dp[j], dp[j - stones[i]] + stones[i]);
            }
        }
        return sum - dp[target] - dp[target];
    }
}
```

```
};
```

- 时间复杂度： $O(m * n)$ ， m 是石头总重量（准确的说是总重量的一半）， n 为石头块数
- 空间复杂度： $O(m)$

总结

本题其实和[416. 分割等和子集](#)几乎是一样的，只是最后对 $dp[target]$ 的处理方式不同。

[416. 分割等和子集](#)相当于是求背包是否正好装满，而本题是求背包最多能装多少。

相信很多小伙伴刷题的时候面对力扣上近两千道题目，感觉无从下手，我花费半年时间整理了Github项目：「力扣刷题攻略」<https://github.com/youngyangyang04/leetcode-master>。里面有100多道经典算法题目刷题顺序、配有40w字的详细图解，常用算法模板总结，以及难点视频讲解，按照list一道一道刷就可以了！star支持一波吧！

- 公众号：代码随想录
- B站：代码随想录
- Github：leetcode-master
- 知乎：代码随想录



推荐给朋友



关注「代码随想录」

学习算法不迷路！

本周小结！（动态规划系列三）

本周我们正式开始讲解背包问题，也是动规里非常重要的一类问题。

背包问题其实有很多细节，如果了解个大概，然后也能一气呵成把代码写出来，但稍稍变变花样可能会陷入迷茫了。

开始回顾一下本周的内容吧！

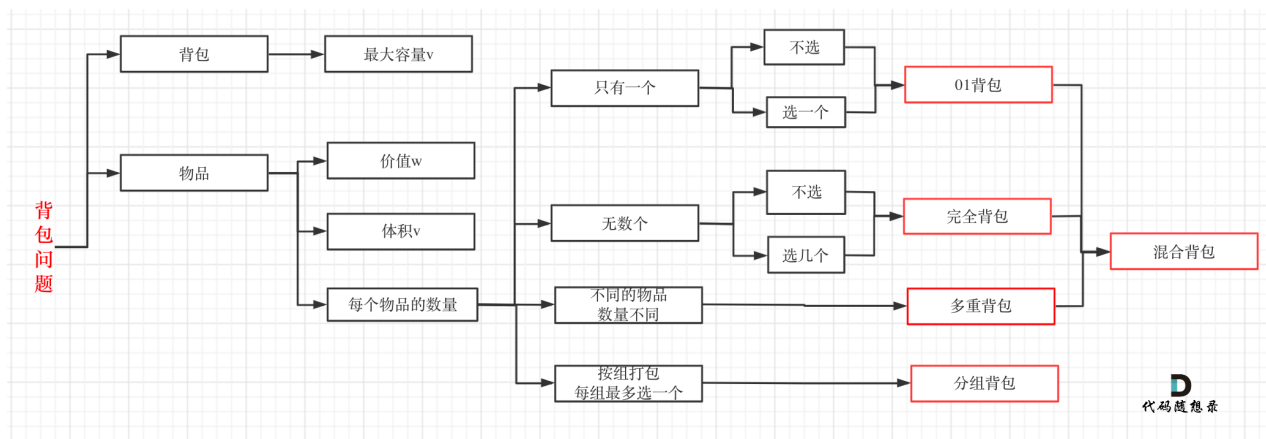
周一

[动态规划：关于01背包问题，你该了解这些！](#)中，我们开始介绍了背包问题。

首先对于背包的所有问题中，01背包是最基础的，其他背包也是在01背包的基础上稍作变化。

所以我才花费这么大精力去讲解01背包。

关于其他几种常用的背包，大家看这张图就了然于胸了：



本文用动规五部曲详细讲解了01背包的二维dp数组的实现方法，大家其实可以发现最简单的是推导公式了，推导公式估计看一遍就记下来了，但难就难在确定初始化和遍历顺序上。

1. 确定dp数组以及下标的含义

$dp[i][j]$ 表示从下标为 $[0-i]$ 的物品里任意取，放进容量为 j 的背包，价值总和最大是多少。

2. 确定递推公式

$dp[i][j] = \max(dp[i-1][j], dp[i-1][j - \text{weight}[i]] + \text{value}[i]);$

3. dp数组如何初始化

```
// 初始化 dp
vector<vector<int>> dp(weight.size() + 1, vector<int>(bagweight + 1, 0));
for (int j = bagweight; j >= weight[0]; j--) {
    dp[0][j] = dp[0][j - weight[0]] + value[0];
}
```

4. 确定遍历顺序

01背包二维dp数组在遍历顺序上，外层遍历物品，内层遍历背包容量 和 外层遍历背包容量，内层遍历物品 都是可以的！

但是先遍历物品更好理解。代码如下：

```
// weight数组的大小 就是物品个数
for(int i = 1; i < weight.size(); i++) { // 遍历物品
    for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
        if (j < weight[i]) dp[i][j] = dp[i - 1][j]; // 这个是为了展现dp数组里元素的变化
        else dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);
    }
}
```

5. 举例推导dp数组

背包最大重量为4。

物品为：

	重量	价值
物品0	1	15
物品1	3	20
物品2	4	30

来看一下对应的dp数组的数值，如图：

dp[i][j]		背包重量j:				
		0	1	2	3	4
物品0:		0	15	15	15	15
物品1:		0	15	15	20	35
物品2:		0	15	15	20	35

D
代码随想录

最终结果就是dp[2][4]。

周二

[动态规划：关于01背包问题，你该了解这些！（滚动数组）](#) 中把01背包的一维dp数组（滚动数组）实现详细讲解了一遍。

分析一下和二维dp数组有什么区别，在初始化和遍历顺序上又有什么差异？

最后总结了一道朴实无华的背包面试题。

要求候选人先实现一个纯二维的01背包，如果写出来了，然后再问为什么两个for循环的嵌套顺序这么写？反过来写行不行？再讲一讲初始化的逻辑。

然后要求实现一个一维数组的01背包，最后再问，一维数组的01背包，两个for循环的顺序反过来写行不行？为什么？

这几个问题就可以考察出候选人的算法功底了。

01背包一维数组分析如下：

1. 确定dp数组的定义

在一维dp数组中， $dp[j]$ 表示：容量为j的背包，所背的物品价值可以最大为 $dp[j]$ 。

2. 一维dp数组的递推公式

```
dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
```

3. 一维dp数组如何初始化

如果物品价值都是大于0的，所以dp数组初始化的时候，都初始为0就可以了。

4. 一维dp数组遍历顺序

代码如下：

```
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = bagweight; j >= weight[i]; j--) { // 遍历背包容量
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
}
```

5. 举例推导dp数组

一维dp，分别用物品0，物品1，物品2 来遍历背包，最终得到结果如下：

用物品0，遍历背包：

0	15	15	15	15
---	----	----	----	----

用物品1，遍历背包：

0	15	15	20	35
---	----	----	----	----

用物品2，遍历背包：

0	15	15	20	35
---	----	----	----	----



周三

[动态规划：416. 分割等和子集](#)中我们开始用01背包来解决问题。

只有确定了如下四点，才能把01背包问题套到本题上来。

- 背包的体积为 $\text{sum} / 2$
- 背包要放入的商品（集合里的元素）重量为 元素的数值，价值也为元素的数值
- 背包如何正好装满，说明找到了总和为 $\text{sum} / 2$ 的子集。
- 背包中每一个元素是不可重复放入。

接下来就是一个完整的01背包问题，大家应该可以轻松做出了。

周四

[动态规划：1049. 最后一块石头的重量 II](#)这道题目其实和[动态规划：416. 分割等和子集](#)是非常像的。

本题其实就是尽量让石头分成重量相同的两堆，相撞之后剩下的石头最小，这样就化解成01背包问题了。

[动态规划：416. 分割等和子集](#)相当于是求背包是否正好装满，而本题是求背包最多能装多少。

这两道题目是对 $\text{dp}[\text{target}]$ 的处理方式不同。这也考验的对 $\text{dp}[i]$ 定义的理解。

总结

总体来说，本周信息量还是比较大的，特别对于对动态规划还不够了解的同学。

但如果坚持下来把，我在文章中列出的每一个问题，都仔细思考，消化为自己的知识，那么进步一定是飞速的。

有的同学可能看了看背包递推公式，上来就能撸它几道题目，然后背包问题就这么过去了，其实这样是很不牢固的。

就像是我们讲解01背包的时候，花了那么大力气才把每一个细节都讲清楚，这里其实是基础，后面的背包问题怎么变，基础比较牢固自然会有自己的一套思考过程。

动态规划：目标和！

494. 目标和

题目链接：<https://leetcode-cn.com/problems/target-sum/>

难度：中等

给定一个非负整数数组， a_1, a_2, \dots, a_n ，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。

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：

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。

提示：

- 数组非空，且长度不会超过 20 。 $2^{20}=1,048,576$
- 初始的数组的和不会超过 1000 。
- 保证返回的最终结果能被 32 位整数存下。 $2^{31}-1=2,147,483,647$

思路

如果跟着「代码随想录」一起学过[回溯算法系列](#)的录友，看到这道题，应该有一种直觉，就是感觉好像回溯法可以爆搜出来。

事实确实如此，下面我也会给出相应的代码，只不过会超时，哈哈。

这道题目咋眼一看和动态规划背包啥的也没啥关系。

本题要如何使表达式结果为target，

既然为target，那么就一定有 left组合 - right组合 = target。

left + right等于sum，而sum是固定的。

公式来了， $\text{left} - (\text{sum} - \text{left}) = \text{target} \rightarrow \text{left} = (\text{target} + \text{sum})/2$ 。

target是固定的，sum是固定的，left就可以求出来。

此时问题就是在集合nums中找出和为left的组合。

回溯算法

在回溯算法系列中，一起学过这道题目[回溯算法：39. 组合总和](#)的录友应该感觉很熟悉，这不就是组合总和问题么？

此时可以套组合总和的回溯法代码，几乎不用改动。

当然，也可以转变成序列区间选+ 或者 -，使用回溯法，那就是另一个解法。

我也把代码给出来吧，大家可以了解一下，回溯的解法，以下是本题转变为组合总和问题的回溯法代码：

```
class Solution {
private:
    vector<vector<int>> result;
    vector<int> path;
    void backtracking(vector<int>& candidates, int target, int sum, int
startIndex) {
        if (sum == target) {
            result.push_back(path);
        }
        // 如果 sum + candidates[i] > target 就终止遍历
        for (int i = startIndex; i < candidates.size() && sum + candidates[i]
<= target; i++) {
            sum += candidates[i];
            path.push_back(candidates[i]);
            backtracking(candidates, target, sum, i + 1);
            sum -= candidates[i];
            path.pop_back();
        }
    }
public:
    int findTargetSumWays(vector<int>& nums, int S) {
        int sum = 0;
        for (int i = 0; i < nums.size(); i++) sum += nums[i];
        if (S > sum) return 0; // 此时没有方案
        if ((S + sum) % 2) return 0; // 此时没有方案，两个int相加的时候要各位小心数值
溢出的问题
        int bagSize = (S + sum) / 2; // 转变为组合总和问题，bagSize就是要求的和

        // 以下为回溯法代码
        result.clear();
        path.clear();
        sort(nums.begin(), nums.end()); // 需要排序
        backtracking(nums, bagSize, 0, 0);
        return result.size();
    }
}
```

```
};
```

当然以上代码超时了。

也可以使用记忆化回溯，但这里我就不再回溯上下功夫了，直接看动规吧

动态规划

如何转化为01背包问题呢。

假设加法的总和为 x ，那么减法对应的总和就是 $\text{sum} - x$ 。

所以我们要求的是 $x - (\text{sum} - x) = S$

$$x = (S + \text{sum}) / 2$$

此时问题就转化为，装满容量为 x 背包，有几种方法。

大家看到 $(S + \text{sum}) / 2$ 应该担心计算的过程中向下取整有没有影响。

这么担心就对了，例如 sum 是5， S 是2的话其实就是无解的，所以：

```
if ((S + sum) % 2 == 1) return 0; // 此时没有方案
```

看到这种表达式，应该本能的反应，两个`int`相加数值可能溢出的问题，当然本题并没有溢出。

再回归到01背包问题，为什么是01背包呢？

因为每个物品（题目中的1）只用一次！

这次和之前遇到的背包问题不一样了，之前都是求容量为 j 的背包，最多能装多少。

本题则是装满有几种方法。其实这就是一个组合问题了。

1. 确定dp数组以及下标的含义

$\text{dp}[j]$ 表示：填满 j （包括 j ）这么大容积的包，有 $\text{dp}[j]$ 种方法

其实也可以使用二维dp数组来求解本题， $\text{dp}[i][j]$ ：使用 下标为 $[0, i]$ 的 $\text{nums}[i]$ 能够凑满 j （包括 j ）这么大容量的包，有 $\text{dp}[i][j]$ 种方法。

下面我都是统一使用一维数组进行讲解，二维降为一维（滚动数组），其实就是上一层拷贝下来，这个我在[动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)也有介绍。

2. 确定递推公式

有哪些来源可以推出 $\text{dp}[j]$ 呢？

不考虑 $\text{nums}[i]$ 的情况下，填满容量为 $j - \text{nums}[i]$ 的背包，有 $\text{dp}[j - \text{nums}[i]]$ 中方法。

那么只要搞到 $\text{nums}[i]$ 的话，凑成 $\text{dp}[j]$ 就有 $\text{dp}[j - \text{nums}[i]]$ 种方法。

举一个例子, $\text{nums}[i] = 2$ ： $\text{dp}[3]$ ，填满背包容量为3的话，有 $\text{dp}[3]$ 种方法。

那么只需要搞到一个2（ $\text{nums}[i]$ ），有 $\text{dp}[3]$ 方法可以凑齐容量为3的背包，相应的就有多少种方法可以凑齐容量为5的背包。

$\text{dp}[j - \text{nums}[i]]$

还是不太明白为什么累加和？

那么需要把 这些方法累加起来就可以了， $dp[j] += dp[j - nums[i]]$

所以求组合类问题的公式，都是类似这种：

```
dp[j] += dp[j - nums[i]]
```

这个公式在后面在讲解背包解决排列组合问题的时候还会用到！

3. dp数组如何初始化

从递归公式可以看出，在初始化的时候 $dp[0]$ 一定要初始化为1，因为 $dp[0]$ 是在公式中一切递推结果的起源，如果 $dp[0]$ 是0的话，递归结果将都是0。

$dp[0] = 1$ ，理论上也很好解释，装满容量为0的背包，有1种方法，就是装0件物品。

$dp[j]$ 其他下标对应的数值应该初始化为0，从递归公式也可以看出， $dp[j]$ 要保证是0的初始值，才能正确的由 $dp[j - nums[i]]$ 推导出来。

4. 确定遍历顺序

在[动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)中，我们讲过对于01背包问题一维dp的遍历， $nums$ 放在外循环， $target$ 在内循环，且内循环倒序。

5. 举例推导dp数组

输入：nums: [1, 1, 1, 1, 1], S: 3

bagSize = $(S + sum) / 2 = (3 + 5) / 2 = 4$

dp数组状态变化如下：

dp[k]

下标k: 0 1 2 3 4

nums[0]遍历背包

1	1	0	0	0
---	---	---	---	---

nums[1]遍历背包

1	2	1	0	0
---	---	---	---	---

nums[2]遍历背包

1	3	3	1	0
---	---	---	---	---

nums[3]遍历背包

1	4	6	4	1
---	---	---	---	---

nums[4]遍历背包

1	5	10	10	5
---	---	----	----	---



公众号:代码随想录

C++代码如下:

```
class solution {
public:
    int findTargetSumWays(vector<int>& nums, int S) {
        int sum = 0;
        for (int i = 0; i < nums.size(); i++) sum += nums[i];
        if (S > sum) return 0; // 此时没有方案
        if ((S + sum) % 2 == 1) return 0; // 此时没有方案
        int bagSize = (S + sum) / 2;
        vector<int> dp(bagSize + 1, 0);
        dp[0] = 1;
        for (int i = 0; i < nums.size(); i++) {
            for (int j = bagSize; j >= nums[i]; j--) {
                dp[j] += dp[j - nums[i]];
            }
        }
        return dp[bagSize];
    }
};
```

- 时间复杂度 $O(n * m)$, n 为正数个数, m 为背包容量
- 空间复杂度: $O(m)$ m 为背包容量

总结

此时大家应该不仅想起, 我们之前讲过的[回溯算法: 39. 组合总和](#)是不是应该也可以用dp来做啊?

是的, 如果仅仅是求个数的话, 就可以用dp, 但[回溯算法: 39. 组合总和](#)要求的是把所有组合列出来, 还是要使用回溯法爆搜的。

本地还是有点难度, 大家也可以记住, 在求装满背包有几种方法的情况下, 递推公式一般为:

```
dp[j] += dp[j - nums[i]];
```

后面我们在讲解完全背包的时候, 还会用到这个递推公式!

相信很多小伙伴刷题的时候面对力扣上近两千道题目, 感觉无从下手, 我花费半年时间整理了Github项目:「力扣刷题攻略」<https://github.com/youngyangyang04/leetcode-master>。里面有100多道经典算法题目刷题顺序、配有40w字的详细图解, 常用算法模板总结, 以及难点视频讲解, 按照list一道一道刷就可以了! star支持一波吧!

- 公众号: [代码随想录](#)
- B站: [代码随想录](#)
- Github: [leetcode-master](#)
- 知乎: [代码随想录](#)



关注「代码随想录」

学习算法不迷路!

动态规划：一和零！

474.一和零

题目链接：<https://leetcode-cn.com/problems/ones-and-zeroes/>

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：

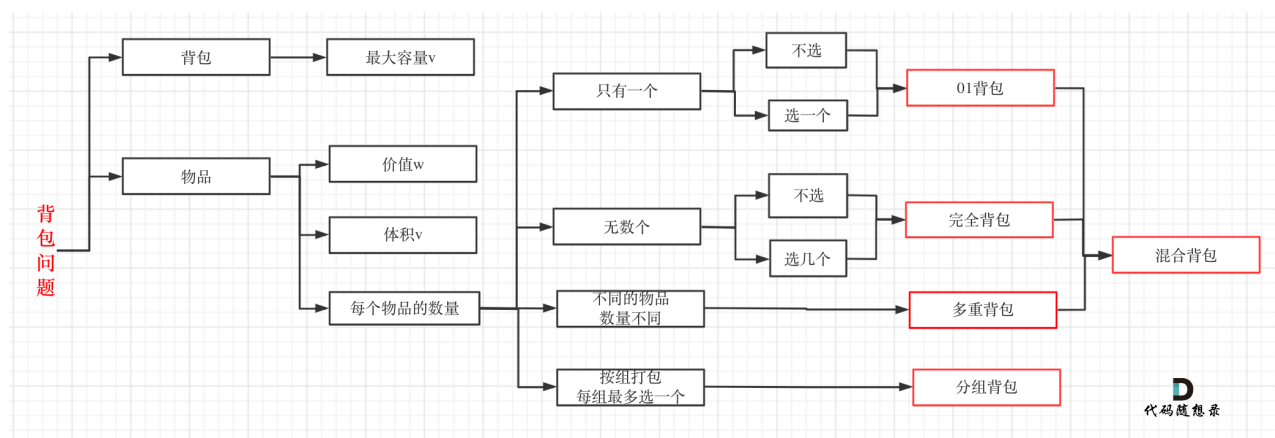
- $1 \leq \text{strs.length} \leq 600$
- $1 \leq \text{strs}[i].\text{length} \leq 100$
- strs[i] 仅由 '0' 和 '1' 组成
- $1 \leq m, n \leq 100$

思路

这道题目，还是比较难的，也有点像程序员自己给自己出个脑筋急转弯，程序员何苦为难程序员呢哈哈。

来说题，本题不少同学会认为是多重背包，一些题解也是这么写的。

其实本题并不是多重背包，再来看一下这个图，捋清几种背包的关系



多重背包是每个物品，数量不同的情况。

本题中strs数组里的元素就是物品，每个物品都是一个！

而m和n相当于是一个背包，两个维度的背包。

理解成多重背包的同学主要是把m和n混淆为物品了，感觉这是不同数量的物品，所以以为是多重背包。

但本题其实是01背包问题！

这不过这个背包有两个维度，一个是m 一个是n，而不同长度的字符串就是不同大小的待装物品。

开始动规五部曲：

1. 确定dp数组（dp table）以及下标的含义

dp[i][j]：最多有i个0和j个1的strs的最大子集的大小为dp[i][j]。

2. 确定递推公式

dp[i][j] 可以由前一个strs里的字符串推导出来，strs里的字符串有zeroNum个0，oneNum个1。

dp[i][j] 就可以是 $dp[i - \text{zeroNum}][j - \text{oneNum}] + 1$ 。

然后我们在遍历的过程中，取dp[i][j]的最大值。

一维01背包，解决遍历物品的循环和下标问题

所以递推公式： $dp[i][j] = \max(dp[i][j], dp[i - \text{zeroNum}][j - \text{oneNum}] + 1)$;

此时大家可以回想一下01背包的递推公式： $dp[j] = \max(dp[j], dp[j - \text{weight}[i]] + \text{value}[i])$;

对比一下就会发现，字符串的zeroNum和oneNum相当于物品的重量（weight[i]），字符串本身的个数相当于物品的价值（value[i]）。

这就是一个典型的01背包！只不过物品的重量有了两个维度而已。

3. dp数组如何初始化

在[动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)中已经讲解了，01背包的dp数组初始化为0就可以。

因为物品价值不会是负数，初始为0，保证递推的时候dp[i][j]不会被初始值覆盖。

4. 确定遍历顺序

在[动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)中，我们讲到了01背包为什么一定是外层for循环遍历物品，内层for循环遍历背包容量且从后向前遍历！

那么本题也是，物品就是strs里的字符串，背包容量就是题目描述中的m和n。

代码如下：

```

for (string str : strs) { // 遍历物品
    int oneNum = 0, zeroNum = 0;
    for (char c : str) {
        if (c == '0') zeroNum++;
        else oneNum++;
    }
    for (int i = m; i >= zeroNum; i--) { // 遍历背包容量且从后向前遍历!
        for (int j = n; j >= oneNum; j--) {
            dp[i][j] = max(dp[i][j], dp[i - zeroNum][j - oneNum] + 1);
        }
    }
}
}

```

有同学可能想，那个遍历背包容量的两层for循环先后循序有没有什么讲究？
没讲究，都是物品重量的一个维度，先遍历那个都行！

5. 举例推导dp数组

以输入：["10","0001","111001","1","0"], m = 3, n = 3为例

最后dp数组的状态如下所示：

输入：["10","0001","111001","1","0"]
m = 3, n = 3

dp[i][j]		0	1	2	3
0		0	1	1	1
1		1	2	2	2
2		1	2	3	3
3		1	2	3	3



公众号：代码随想录

以上动规五部曲分析完毕，C++代码如下：

```

class Solution {
public:
    int findMaxForm(vector<string>& strs, int m, int n) {
        vector<vector<int>>> dp(m + 1, vector<int> (n + 1, 0)); // 默认初始化0
        for (string str : strs) { // 遍历物品
            int oneNum = 0, zeroNum = 0;
            for (char c : str) {
                if (c == '0') zeroNum++;
                else oneNum++;
            }
            for (int i = m; i >= zeroNum; i--) { // 遍历背包容量且从后向前遍历!
                for (int j = n; j >= oneNum; j--) {
                    dp[i][j] = max(dp[i][j], dp[i - zeroNum][j - oneNum] + 1);
                }
            }
        }
        return dp[m][n];
    }
};

```

总结

不少同学刷过这道题，可能没有总结这究竟是什么背包。

这道题的本质是有两个维度的01背包，如果大家认识到这一点，对这道题的理解就比较深入了。

动态规划：关于完全背包，你该了解这些！

完全背包

有N件物品和一个最多能背重量为W的背包。第i件物品的重量是weight[i]，得到的价值是value[i]。每件物品都有无限个（也就是可以放入背包多次），求解将哪些物品装入背包里物品价值总和最大。

完全背包和01背包问题唯一不同的地方就是，每种物品有无限件。

同样leetcode上没有纯完全背包问题，都是需要完全背包的各种应用，需要转化成完全背包问题，所以我这里还是以纯完全背包问题进行讲解理论和原理。

在下面的讲解中，我依然举这个例子：

背包最大重量为4。

物品为：

	重量	价值
物品0	1	15
物品1	3	20
物品2	4	30

每件商品都有无限个！

问背包能背的物品最大价值是多少？

01背包和完全背包唯一不同就是体现在遍历顺序上，所以本文就不去做动规五部曲了，我们直接针对遍历顺序进行分析！

关于01背包我如下两篇已经进行深入分析了：

- [动态规划：关于01背包问题，你该了解这些！](#)
- [动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)

首先在回顾一下01背包的核心代码

```
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = bagweight; j >= weight[i]; j--) { // 遍历背包容量
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
}
```

我们知道01背包内嵌的循环是从大到小遍历，为了保证每个物品仅被添加一次。

而完全背包的物品是可以添加多次的，所以要从小到大去遍历，即：

```
// 先遍历物品，再遍历背包
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = weight[i]; j <= bagweight ; j++) { // 遍历背包容量
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
}
```

至于为什么，我在[动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)中也做了讲解。

dp状态图如下：

$dp[j]$

背包容量j:

用物品0，遍历背包:

0	15	30	45	60
---	----	----	----	----

用物品1，遍历背包:

0	15	30	45	60
---	----	----	----	----

用物品2，遍历背包:

0	15	30	45	60
---	----	----	----	----



公众号:代码随想录

相信很多同学看网上的文章，关于完全背包介绍基本就到为止了。

其实还有一个很重要的问题，为什么遍历物品在外层循环，遍历背包容量在内层循环？

这个问题很多题解关于这里都是轻描淡写就略过了，大家都默认 遍历物品在外层，遍历背包容量在内层，好像本应该如此一样，那么为什么呢？

难道就不能遍历背包容量在外层，遍历物品在内层？

看过这两篇的话：

- [动态规划：关于01背包问题，你该了解这些！](#)
- [动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)

就知道了，01背包中二维dp数组的两个for遍历的先后循序是可以颠倒了，一位dp数组的两个for循环先后循序一定是先遍历物品，再遍历背包容量。

在完全背包中，对于一维dp数组来说，其实两个for循环嵌套顺序同样无所谓！

因为 $dp[j]$ 是根据 下标j之前所对应的 $dp[j]$ 计算出来的。 只要保证下标j之前的 $dp[j]$ 都是经过计算的就可以了。

遍历物品在外层循环，遍历背包容量在内层循环，状态如图：

dp[j]

背包容量j:

用物品0，遍历背包:

0	15	30	45	60
---	----	----	----	----

用物品1，遍历背包:

0	15	30	45	
---	----	----	----	--

用物品2，遍历背包:

--	--	--	--	--

D
公众号:代码随想录

遍历背包容量在外层循环，遍历物品在内层循环，状态如图：

dp[j]

背包容量j:

用物品0，遍历背包:

0	15	30	45	
---	----	----	----	--

用物品1，遍历背包:

0	15	30	45	
---	----	----	----	--

用物品2，遍历背包:

0	15	15		
---	----	----	--	--

D
公众号:代码随想录

看了这两个图，大家就会理解，完全背包中，两个for循环的先后循序，都不影响计算dp[j]所需要的值（这个值就是下标j之前所对应的dp[j]）。

先遍历被背包在遍历物品，代码如下：

```
// 先遍历背包，再遍历物品
for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
    for(int i = 0; i < weight.size(); i++) { // 遍历物品
        if (j - weight[i] >= 0) dp[j] = max(dp[j], dp[j - weight[i]] +
value[i]);
    }
    cout << endl;
}
}
```

C++测试代码

完整的C++测试代码如下：

```
// 先遍历物品，在遍历背包
void test_CompletePack() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    int bagweight = 4;
    vector<int> dp(bagweight + 1, 0);
    for(int i = 0; i < weight.size(); i++) { // 遍历物品
        for(int j = weight[i]; j <= bagweight; j++) { // 遍历背包容量
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
        }
    }
    cout << dp[bagweight] << endl;
}

int main() {
    test_CompletePack();
}
```

```
// 先遍历背包，再遍历物品
void test_CompletePack() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    int bagweight = 4;

    vector<int> dp(bagweight + 1, 0);

    for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
        for(int i = 0; i < weight.size(); i++) { // 遍历物品
            if (j - weight[i] >= 0) dp[j] = max(dp[j], dp[j - weight[i]] +
value[i]);
        }
    }
    cout << dp[bagweight] << endl;
}
```

```
}  
int main() {  
    test_CompletePack();  
}
```

总结

细心的同学可能发现，全文我说的都是对于纯完全背包问题，其for循环的先后循环是可以颠倒的！

但如果题目稍稍有点变化，就会体现在遍历顺序上。

如果问装满背包有几种方式的话？那么两个for循环的先后顺序就有很大的区别了，而leetcode上的题目都是这种稍有变化的类型。

这个区别，我将在后面讲解具体leetcode题目中给大家介绍，因为这块如果不结合具体题目，单纯的介绍原理估计很多同学会越看越懵！

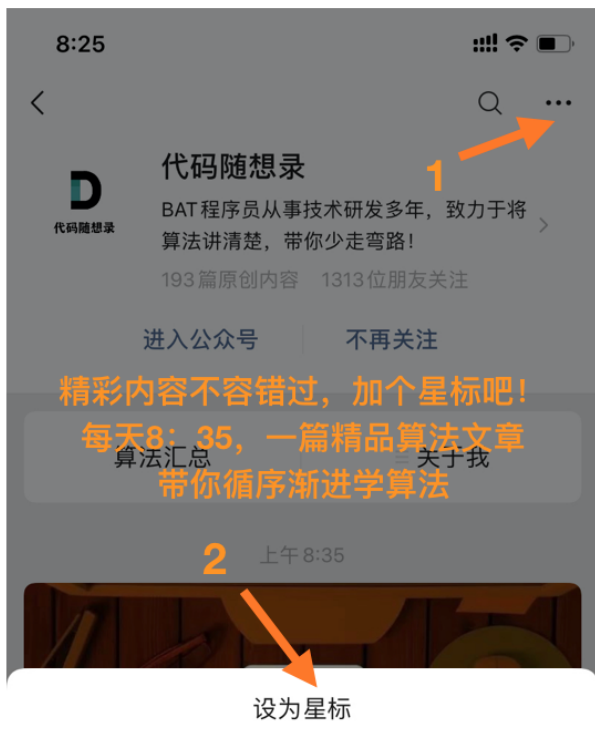
别急，下一篇就是了！哈哈

最后，又可以出一道面试题了，就是纯完全背包，要求先用二维dp数组实现，然后再用一维dp数组实现，最后在问，两个for循环的先后是否可以颠倒？为什么？

这个简单的完全背包问题，估计就可以难住不少候选人了。

相信很多小伙伴刷题的时候面对力扣上近两千道题目，感觉无从下手，我花费半年时间整理了Github项目：「力扣刷题攻略」<https://github.com/youngyangyang04/leetcode-master>。里面有100多道经典算法题目刷题顺序、配有40w字的详细图解，常用算法模板总结，以及难点视频讲解，按照list一道一道刷就可以了！star支持一波吧！

- 公众号：[代码随想录](#)
- B站：[代码随想录](#)
- Github：[leetcode-master](#)
- 知乎：[代码随想录](#)



关注「代码随想录」

学习算法不迷路！

动态规划：给你一些零钱，你要怎么凑？

518. 零钱兑换 II

链接：<https://leetcode-cn.com/problems/coin-change-2/>

难度：中等

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：

- $0 \leq \text{amount}$ (总金额) ≤ 5000
- $1 \leq \text{coin}$ (硬币面额) ≤ 5000
- 硬币种类不超过 500 种
- 结果符合 32 位符号整数

思路

这是一道典型的背包问题，一看到钱币数量不限，就知道这是一个完全背包。

对完全背包还不了解的同学，可以看这篇：[动态规划：关于完全背包，你该了解这些！](#)

但本题和纯完全背包不一样，**纯完全背包是能否凑成总金额，而本题是要求凑成总金额的个数！**

注意题目描述中是凑成总金额的硬币组合数，为什么强调是组合数呢？

例如示例一：

$$5 = 2 + 2 + 1$$

$$5 = 2 + 1 + 2$$

这是一种组合，都是 2 2 1。

如果问的是排列数，那么上面就是两种排列了。

组合不强调元素之间的顺序，排列强调元素之间的顺序。 其实这一点我们在讲解回溯算法专题的时候就讲过了哈。

那我为什么要介绍这些呢，因为这和下文讲解遍历顺序息息相关！

回归本题，动规五步曲来分析如下：

1. 确定dp数组以及下标的含义

$\text{dp}[j]$ ：凑成总金额j的货币组合数为 $\text{dp}[j]$

2. 确定递推公式

$\text{dp}[j]$ （考虑 $\text{coins}[i]$ 的组合总和）就是所有的 $\text{dp}[j - \text{coins}[i]]$ （不考虑 $\text{coins}[i]$ ）相加。

所以递推公式： $\text{dp}[j] += \text{dp}[j - \text{coins}[i]]$;

这个递推公式大家应该不陌生了，我在讲解01背包题目的时候在这篇[动态规划：目标和！](#)中就讲解了，求装满背包有几种方法，一般公式都是： $\text{dp}[j] += \text{dp}[j - \text{nums}[i]]$;

3. dp数组如何初始化

首先 $\text{dp}[0]$ 一定要为1， $\text{dp}[0] = 1$ 是 递归公式的基础。

从 $\text{dp}[i]$ 的含义上来讲就是，凑成总金额0的货币组合数为1。

下标非0的 $\text{dp}[j]$ 初始化为0，这样累计加 $\text{dp}[j - \text{coins}[i]]$ 的时候才不会影响真正的 $\text{dp}[j]$

4. 确定遍历顺序

本题中我们是外层for循环遍历物品（钱币），内层for遍历背包（金钱总额），还是外层for遍历背包（金钱总额），内层for循环遍历物品（钱币）呢？

我在[动态规划：关于完全背包，你该了解这些！](#)中讲解了完全背包的两个for循环的先后顺序都是可以的。

但本题就不行了！

因为纯完全背包求得是能否凑成总和，和凑成总和的元素有没有顺序没关系，即：有顺序也行，没有顺序也行！

而本题要求凑成总和的组合数，元素之间要求没有顺序。

所以纯完全背包是能凑成总和就行，不用管怎么凑的。

本题是求凑出来的方案个数，且每个方案个数是为组合数。

那么本题，两个for循环的先后顺序可就有说法了。

我们先来看 外层for循环遍历物品（钱币），内层for遍历背包（金钱总额）的情况。

代码如下：

```
for (int i = 0; i < coins.size(); i++) { // 遍历物品
    for (int j = coins[i]; j <= amount; j++) { // 遍历背包容量
        dp[j] += dp[j - coins[i]];
    }
}
```

假设：coins[0] = 1, coins[1] = 5。

那么就是先把1加入计算，然后再把5加入计算，得到的方法数量只有{1, 5}这种情况。而不会出现{5, 1}的情况。

所以这种遍历顺序中dp[j]里计算的是组合数！

如果把两个for交换顺序，代码如下：

```
for (int j = 0; j <= amount; j++) { // 遍历背包容量
    for (int i = 0; i < coins.size(); i++) { // 遍历物品
        if (j - coins[i] >= 0) dp[j] += dp[j - coins[i]];
    }
}
```

背包容量的每一个值，都是经过 1 和 5 的计算，包含了{1, 5} 和 {5, 1}两种情况。

此时dp[j]里算出来的就是排列数！

可能这里很多同学还不是很理解，**建议动手把这两种方案的dp数组数值变化打印出来，对比看一看！（实践出真知）**

5. 举例推导dp数组

输入: amount = 5, coins = [1, 2, 5]，dp状态图如下：

输入: amount = 5, coins = [1, 2, 5]

dp[i]

下标: 0 1 2 3 4 5

coins[0]加入遍历

1	1	1	1	1	1
---	---	---	---	---	---

coins[1]加入遍历

1	1	2	2	3	3
---	---	---	---	---	---

coins[2]加入遍历

1	1	2	2	3	4
---	---	---	---	---	---



公众号: 代码随想录

最后红色框dp[amount]为最终结果。

以上分析完毕, C++代码如下:

```
class Solution {
public:
    int change(int amount, vector<int>& coins) {
        vector<int> dp(amount + 1, 0);
        dp[0] = 1;
        for (int i = 0; i < coins.size(); i++) { // 遍历物品
            for (int j = coins[i]; j <= amount; j++) { // 遍历背包
                dp[j] += dp[j - coins[i]];
            }
        }
        return dp[amount];
    }
};
```

是不是发现代码如此精简, 哈哈

总结

本题的递推公式, 其实我们在[动态规划: 目标和!](#)中就已经讲过了, 而难点在于遍历顺序!

在求装满背包有几种方案的时候, 认清遍历顺序是非常关键的。

如果求组合数就是外层for循环遍历物品, 内层for遍历背包。

如果求排列数就是外层for遍历背包, 内层for循环遍历物品。

可能说到排列数录友们已经有点懵了，后面Carl还会安排求排列数的题目，到时候在对比一下，大家就会发现神奇所在！

本周小结！（动态规划系列四）

周一

动态规划：目标和！ 要求在数列之间加入+ 或者 -，使其和为S。

所有数的总和为sum，假设加法的总和为x，那么可以推出 $x = (S + \text{sum}) / 2$ 。

S 和 sum都是固定的，那此时问题就转化为01背包问题（数列中的数只能使用一次）：给你一些物品（数字），装满背包（就是x）有几种方法。

1. 确定dp数组以及下标的含义

dp[j] 表示：填满j（包括j）这么大容积的包，有dp[j]种方法

2. 确定递推公式

$\text{dp}[i] += \text{dp}[j - \text{nums}[i]]$

注意：求装满背包有几种方法类似的题目，递推公式基本都是这样的。

3. dp数组如何初始化

dp[0] 初始化为1，dp[j]其他下标对应的数值应该初始化为0。

4. 确定遍历顺序

01背包问题一维dp的遍历，nums放在外循环，target在内循环，且内循环倒序。

5. 举例推导dp数组

输入：nums: [1, 1, 1, 1, 1], S: 3

$\text{bagSize} = (S + \text{sum}) / 2 = (3 + 5) / 2 = 4$

dp数组状态变化如下：

dp[k]

下标k: 0 1 2 3 4

nums[0]遍历背包

1	1	0	0	0
---	---	---	---	---

nums[1]遍历背包

1	2	1	0	0
---	---	---	---	---

nums[2]遍历背包

1	3	3	1	0
---	---	---	---	---

nums[3]遍历背包

1	4	6	4	1
---	---	---	---	---

nums[4]遍历背包

1	5	10	10	5
---	---	----	----	---



公众号:代码随想录

周二

这道题目[动态规划：一和零！](#)算有点难度。

不少同学都以为是多重背包，其实这是一道标准的01背包。

这不过这个背包有两个维度，一个是m 一个是n，而不同长度的字符串就是不同大小的待装物品。

所以这是一个二维01背包！

1. 确定dp数组（dp table）以及下标的含义

dp[i][j]：最多有i个0和j个1的strs的最大子集的大小为dp[i][j]。

2. 确定递推公式

$dp[i][j] = \max(dp[i][j], dp[i - \text{zeroNum}][j - \text{oneNum}] + 1);$

字符串集合中的一个字符串0的数量为zeroNum，1的数量为oneNum。

3. dp数组如何初始化

因为物品价值不会是负数，初始为0，保证递推的时候dp[i][j]不会被初始值覆盖。

4. 确定遍历顺序

01背包一定是外层for循环遍历物品，内层for循环遍历背包容量且从后向前遍历！


5. 举例推导dp数组

以输入：["10","0001","111001","1","0"], m = 3, n = 3为例

最后dp数组的状态如下所示：

输入：["10","0001","111001","1","0"]
m = 3, n = 3

dp[i][j]	0	1	2	3
0	0	1	1	1
1	1	2	2	2
2	1	2	3	3
3	1	2	3	3


公众号: 代码随想录

周三

此时01背包我们就讲完了，正式开始完全背包。

在[动态规划：关于完全背包，你该了解这些！](#)中我们讲解了完全背包的理论基础。

其实完全背包和01背包区别就是完全背包的物品是无限数量。

递推公式也是一样的，但难点在于遍历顺序上！

完全背包的物品是可以添加多次的，所以遍历背包容量要从小到大去遍历，即：

```
// 先遍历物品，再遍历背包
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = weight[i]; j < bagweight ; j++) { // 遍历背包容量
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
}
```

基本网上题的题解介绍到这里就到此为止了。

那么为什么要先遍历物品，在遍历背包呢？（灵魂拷问）

其实对于纯完全背包，先遍历物品，再遍历背包 与 先遍历背包，再遍历物品都是可以的。我在文中[动态规划：关于完全背包，你该了解这些！](#)也给出了详细的解释。

这个细节是很多同学忽略掉的点，其实也不算细节了，相信不少同学在写背包的时候，两层for循环的先后顺序搞不清楚，靠感觉来的。

所以理解究竟是先遍历啥，后遍历啥非常重要，这也体现出遍历顺序的重要性！

在文中，我也强调了是对纯完全背包，两个for循环先后顺序无所谓，那么题目稍有变化，可就有所谓了。

周四

在[动态规划：给你一些零钱，你要怎么凑？](#)中就是给你一堆零钱（零钱个数无限），为凑成amount的组合数有几种。

注意这里组合数和排列数的区别！

看到无限零钱个数就知道是完全背包，

但本题不是纯完全背包了（求是否能装满背包），而是求装满背包有几种方法。

这里在遍历顺序上可就有说法了。

- 如果求组合数就是外层for循环遍历物品，内层for遍历背包。
- 如果求排列数就是外层for遍历背包，内层for循环遍历物品。

这里同学们需要理解一波，我在文中也给出了详细的解释，下周我们将介绍求排列数的完全背包题目来加深对这个遍历顺序的理解。

总结

相信通过本周的学习，大家已经初步感受到遍历顺序的重要性！

很多对动规理解不深入的同学都会感觉：动规嘛，就是把递推公式推出来其他都easy了。

其实这是一种错觉，或者说对动规理解的不够深入！

我在动规专题开篇介绍[关于动态规划，你该了解这些！](#)中就强调了 递推公式仅仅是 动规五部曲里的一小部分，dp数组的定义、初始化、遍历顺序，哪一点没有搞透的话，即使知道递推公式，遇到稍稍难一点的动规题目立刻会感觉写不出来了。

此时相信大家对于动规五部曲也有更深的理解了，同样也验证了Carl之前讲过的：简单题是用来学习方法论的，而遇到难题才体现出方法论的重要性！

相信很多小伙伴刷题的时候面对力扣上近两千道题目，感觉无从下手，我花费半年时间整理了Github项目：「力扣刷题攻略」<https://github.com/youngyangyang04/leetcode-master>。里面有100多道经典算法题目刷题顺序、配有40w字的详细图解，常用算法模板总结，以及难点视频讲解，按照list一道一道刷就可以了！star支持一波吧！

- 公众号：代码随想录
- B站：代码随想录
- Github：leetcode-master
- 知乎：代码随想录



关注「代码随想录」

学习算法不迷路！

动态规划：Carl称它为排列总和！

377. 组合总和 IV

题目链接：<https://leetcode-cn.com/problems/combination-sum-iv/>

难度：中等

给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。

示例：

nums = [1, 2, 3]

target = 4

所有可能的组合为：

(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)

请注意，顺序不同的序列被视作不同的组合。

因此输出为 7。

思路

本题题目描述说是求组合，但又说是可以元素相同顺序不同的组合算两个组合，**其实就是求排列！**

弄清什么是组合，什么是排列很重要。

组合不强调顺序，(1,5)和(5,1)是同一个组合。

排列强调顺序，(1,5)和(5,1)是两个不同的排列。

大家在公众号里学习回溯算法专题的时候，一定做过这两道题目[回溯算法：39.组合总和](#)和[回溯算法：40.组合总和II](#)会感觉这两题和本题很像！

但其本质是本题求的是排列总和，而且仅仅是求排列总和的个数，并不是把所有的排列都列出来。

如果本题要把排列都列出来的话，只能使用回溯算法爆搜。

动规五部曲分析如下：

1. 确定dp数组以及下标的含义

dp[i]: 凑成目标正整数为i的排列个数为dp[i]

2. 确定递推公式

dp[i]（考虑nums[j]）可以由 dp[i - nums[j]]（不考虑nums[j]）推导出来。

因为只要得到nums[j]，排列个数dp[i - nums[j]]，就是dp[i]的一部分。

在[动态规划：494.目标和](#)和 [动态规划：518.零钱兑换II](#)中我们已经讲过了，求装满背包有几种方法，递推公式一般都是dp[i] += dp[i - nums[j]];

本题也一样。

3. dp数组如何初始化

因为递推公式dp[i] += dp[i - nums[j]]的缘故，dp[0]要初始化为1，这样递归其他dp[i]的时候才会有数值基础。

至于dp[0] = 1 有没有意义呢？

其实没有意义，所以我也不去强行解释它的意义了，因为题目中也说了：给定目标值是正整数！所以dp[0] = 1是没有意义的，仅仅是为了推导递推公式。

至于非0下标的dp[i]应该初始为多少呢？

初始化为0，这样才不会影响dp[i]累加所有的dp[i - nums[j]]。

4. 确定遍历顺序

个数可以不限使用，说明这是一个完全背包。

得到的集合是排列，说明需要考虑元素之间的顺序。

本题要求的是排列，那么这个for循环嵌套的顺序可以有说法了。

在[动态规划：518.零钱兑换II](#)中就已经讲过了。

如果求组合数就是外层for循环遍历物品，内层for遍历背包。

如果求排列数就是外层for遍历背包，内层for循环遍历物品。

如果把遍历nums（物品）放在外循环，遍历target的作为内循环的话，举一个例子：计算dp[4]的时候，结果集只有{1,3}这样的集合，不会有{3,1}这样的集合，因为nums遍历放在外层，3只能出现在1后面！

所以本题遍历顺序最终遍历顺序：**target（背包）**放在外循环，将**nums（物品）**放在内循环，内循环从前到后遍历。

5. 举例来推导dp数组

我们再来用示例中的例子推导一下：

输入：nums = [1, 2, 3], target = 4

下标：	0	1	2	3	4
dp[i]:	1	1	2	4	7

dp[0] = 1
dp[1] = dp[0] = 1
dp[2] = dp[1] + dp[0] = 2
dp[3] = dp[2] + dp[1] + dp[0] = 4
dp[4] = dp[3] + dp[2] + dp[1] = 7

公众号：代码随想录

如果代码运行处的结果不是想要的结果，就把dp[i]都打出来，看看和我们推导的一不一样。

以上分析完毕，C++代码如下：

```
class Solution {
public:
    int combinationSum4(vector<int>& nums, int target) {
```

```

vector<int> dp(target + 1, 0);
dp[0] = 1;
for (int i = 0; i <= target; i++) { // 遍历背包
    for (int j = 0; j < nums.size(); j++) { // 遍历物品
        if (i - nums[j] >= 0 && dp[i] < INT_MAX - dp[i - nums[j]]) {
            dp[i] += dp[i - nums[j]];
        }
    }
}
return dp[target];
}
};

```

C++测试用例有超过两个数相加超过int的数据，所以需要在if里加上`dp[i] < INT_MAX - dp[i - num]`。

但java就不用考虑这个限制，java里的int也是四个字节吧，也有可能leetcode后台对不同语言的测试数据不一样。

总结

求装满背包有几种方法，递归公式都是一样的，没有什么差别，但关键在于遍历顺序！

本题与[动态规划：518.零钱兑换II](#)就是一个鲜明的对比，一个是求排列，一个是求组合，遍历顺序完全不同。

如果对遍历顺序没有深度理解的话，做这种完全背包的题目会很懵逼，即使题目刷过了可能也不太清楚具体是怎么过的。

此时大家应该对动态规划中的遍历顺序又有更深的理解了。

动态规划：以前我没得选，现在我选择再爬一次！

之前讲这道题目的时候，因为还没有讲背包问题，所以就只是讲了一下爬楼梯最直接的动规方法（斐波那契）。

这次终于讲到了背包问题，我选择带录友们再爬一次楼梯！

70. 爬楼梯

链接：<https://leetcode-cn.com/problems/climbing-stairs/>

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

示例 1:

输入: 2

输出: 2

解释: 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶
2. 2 阶

示例 2:

输入: 3

输出: 3

解释: 有三种方法可以爬到楼顶。

1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶

思路

这道题目 我们在[动态规划: 爬楼梯](#) 中已经讲过一次了, 原题其实是一道简单动规的题目。

既然这么简单为什么还要讲呢, 其实本题稍加改动就是一道面试好题。

改为: 一步一个台阶, 两个台阶, 三个台阶,, 直到 m 个台阶。问有多少种不同的方法可以爬到楼顶呢?

1阶, 2阶, m 阶就是物品, 楼顶就是背包。

每一阶可以重复使用, 例如跳了1阶, 还可以继续跳1阶。

问跳到楼顶有几种方法其实就是问装满背包有几种方法。

此时大家应该发现这就是一个完全背包问题了!

和昨天的题目[动态规划: 377. 组合总和 IV](#)基本就是一道题了。

动规五部曲分析如下:

1. 确定dp数组以及下标的含义

dp[i]: 爬到有i个台阶的楼顶, 有dp[i]种方法。

2. 确定递推公式

在[动态规划: 494.目标和](#)、[动态规划: 518.零钱兑换II](#)、[动态规划: 377. 组合总和 IV](#)中我们都讲过了, 求装满背包有几种方法, 递推公式一般都是 $dp[i] += dp[i - nums[j]]$;

本题呢, dp[i]有几种来源, dp[i - 1], dp[i - 2], dp[i - 3] 等等, 即: dp[i - j]

那么递推公式为: $dp[i] += dp[i - j]$

3. dp数组如何初始化

既然递推公式是 $dp[i] += dp[i - j]$, 那么dp[0] 一定为1, dp[0]是递归中一切数值的基础所在, 如果dp[0]是0的话, 其他数值都是0了。

下标非0的dp[i]初始化为0，因为dp[i]是靠dp[i-j]累计上来的，dp[i]本身为0这样才不会影响结果

4. 确定遍历顺序

这是背包里求排列问题，即：**1、2步** 和 **2、1步**都是上三个台阶，但是这两种方法不一样！

所以需将target放在外循环，将nums放在内循环。

每一步可以走多次，这是完全背包，内循环需要从前向后遍历。

5. 举例来推导dp数组

介于本题和[动态规划：377. 组合总和 IV](#)几乎是一样的，这里我就不再重复举例了。

以上分析完毕，C++代码如下：

```
class Solution {
public:
    int climbStairs(int n) {
        vector<int> dp(n + 1, 0);
        dp[0] = 1;
        for (int i = 1; i <= n; i++) { // 遍历背包
            for (int j = 1; j <= m; j++) { // 遍历物品
                if (i - j >= 0) dp[i] += dp[i - j];
            }
        }
        return dp[n];
    }
};
```

代码中m表示最多可以爬m个台阶，代码中把m改成2就是本题70.爬楼梯可以AC的代码了。

总结

本题看起来是一道简单题目，稍稍进阶一下其实就是一个完全背包！

如果我来面试的话，我就会先给候选人出一个 本题原题，看其表现，如果顺利写出来，进而在要求每次可以爬[1 - m]个台阶应该怎么写。

顺便再考察一下两个for循环的嵌套顺序，为什么target放外面，nums放里面。

这就能考察对背包问题本质的掌握程度，候选人是不是刷题背公式，一眼就看出来了。

这么一连套下来，如果候选人都能答出来，相信任何一位面试官都是非常满意的。

本题代码不长，题目也很普通，但稍稍一进阶就可以考察完全背包，而且题目进阶的内容在leetcode上并没有原题，一定程度上就可以排除掉刷题党了，简直是面试题目的绝佳选择！

相信很多小伙伴刷题的时候面对力扣上近两千道题目，感觉无从下手，我花费半年时间整理了Github项目：「力扣刷题攻略」<https://github.com/youngyangyang04/leetcode-master>。里面有100多道经典算法题目刷题顺序、配有40w字的详细图解，常用算法模板总结，以及难点视频讲解，按照list一道一道刷就可以了！star支持一波吧！

- 公众号：[代码随想录](#)

- B站: [代码随想录](#)
- Github: [leetcode-master](#)
- 知乎: [代码随想录](#)



关注「代码随想录」

学习算法不迷路！

动态规划： 给我个机会，我再兑换一次零钱

322. 零钱兑换

题目链接: <https://leetcode-cn.com/problems/coin-change/>

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:

- $1 \leq \text{coins.length} \leq 12$
- $1 \leq \text{coins}[i] \leq 2^{31} - 1$
- $0 \leq \text{amount} \leq 10^4$

思路

在[动态规划: 518.零钱兑换II](#)中我们已经兑换一次零钱了, 这次又要兑换, 套路不一样!

题目中说每种硬币的数量是无限的, 可以看出是典型的完全背包问题。

动规五部曲分析如下:

1. 确定dp数组以及下标的含义

dp[j]: 凑足总额为j所需钱币的最少个数为dp[j]

2. 确定递推公式

得到dp[j] (考虑coins[i]), 只有一个来源, dp[j - coins[i]] (没有考虑coins[i])。

凑足总额为j - coins[i]的最少个数为dp[j - coins[i]], 那么只需要加上一个钱币coins[i]即dp[j - coins[i]] + 1就是dp[j] (考虑coins[i])

所以dp[j] 要取所有 dp[j - coins[i]] + 1 中最小的。

递推公式: $dp[j] = \min(dp[j - \text{coins}[i]] + 1, dp[j]);$

3. dp数组如何初始化

首先凑足总金额为0所需钱币的个数一定是0, 那么dp[0] = 0;

其他下标对应的数值呢?

考虑到递推公式的特性, dp[j]必须初始化为一个最大的数, 否则就会在min(dp[j - coins[i]] + 1, dp[j])比较的过程中被初始值覆盖。

所以下标非0的元素都是应该是最大值。

代码如下:

```
vector<int> dp(amount + 1, INT_MAX);  
dp[0] = 0;
```

4. 确定遍历顺序

本题求钱币最小个数, 那么钱币有顺序和没有顺序都可以, 都不影响钱币的最小个数。。

所以本题并不强调集合是组合还是排列。

如果求组合数就是外层for循环遍历物品，内层for遍历背包。

如果求排列数就是外层for遍历背包，内层for循环遍历物品。

在动态规划专题我们讲过了求组合数是[动态规划：518.零钱兑换II](#)，求排列数是[动态规划：377. 组合总和 IV](#)。

所以本题的两个for循环的关系是：外层for循环遍历物品，内层for遍历背包或者外层for遍历背包，内层for循环遍历物品都是可以的！

那么我采用coins放在外循环，target在内循环的方式。

本题钱币数量可以无限使用，那么是完全背包。所以遍历的内循环是正序

综上所述，遍历顺序为：coins（物品）放在外循环，target（背包）在内循环。且内循环正序。

5. 举例推导dp数组

以输入：coins = [1, 2, 5], amount = 5为例

输入：coins = [1, 2, 5], amount = 5

	下标：0	1	2	3	4	5
dp[i]:	0	1	1	2	2	1

 公众号：代码随想录

dp[amount]为最终结果。

C++代码

以上分析完毕，C++ 代码如下：

```
// 版本一
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
```

```

vector<int> dp(amount + 1, INT_MAX);
dp[0] = 0;
for (int i = 0; i < coins.size(); i++) { // 遍历物品
    for (int j = coins[i]; j <= amount; j++) { // 遍历背包
        if (dp[j - coins[i]] != INT_MAX) { // 如果dp[j - coins[i]]是初始
值则跳过
            dp[j] = min(dp[j - coins[i]] + 1, dp[j]);
        }
    }
}
if (dp[amount] == INT_MAX) return -1;
return dp[amount];
}
};

```

对于遍历方式遍历背包放在外循环，遍历物品放在内循环也是可以的，我就直接给出代码了

```

// 版本二
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> dp(amount + 1, INT_MAX);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) { // 遍历背包
            for (int j = 0; j < coins.size(); j++) { // 遍历物品
                if (i - coins[j] >= 0 && dp[i - coins[j]] != INT_MAX ) {
                    dp[i] = min(dp[i - coins[j]] + 1, dp[i]);
                }
            }
        }
        if (dp[amount] == INT_MAX) return -1;
        return dp[amount];
    }
};

```

总结

细心的同学看网上的题解，可能看一篇是遍历背包的for循环放外面，看一篇又是遍历背包的for循环放里面，看多了都看晕了，到底两个for循环应该是什么先后关系。

能把遍历顺序讲明白的文章几乎找不到！

这也是大多数同学学习动态规划的苦恼所在，有的时候递推公式很简单，难在遍历顺序上！

但最终又可以稀里糊涂的把题目过了，也不知道为什么这样可以过，反正就是过了，哈哈

那么这篇文章就把遍历顺序分析的清清楚楚。

[动态规划：518.零钱兑换II](#)中求的是组合数，[动态规划：377.组合总和 IV](#)中求的是排列数。

而本题是要求最少硬币数量，硬币是组合数还是排列数都无所谓！所以两个for循环先后顺序怎样都可以！

这也是我为什么要先讲518.零钱兑换II 然后再讲本题即：322.零钱兑换，这是Carl的良苦用心那。

相信大家看完之后，对背包问题中的遍历顺序又了更深的理解了。

动态规划：一样的套路，再求一次完全平方数

279.完全平方数

题目地址：<https://leetcode-cn.com/problems/perfect-squares/>

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：

- $1 \leq n \leq 10^4$

思路

可能刚看这种题感觉没啥思路，又平方和的，又最小数的。

我来把题目翻译一下：完全平方数就是物品（可以无限件使用），凑个正整数 n 就是背包，问凑满这个背包最少有多少物品？

感受出来了没，这么浓厚的完全背包氛围，而且和昨天的题目[动态规划：322. 零钱兑换](#)就是一样一样的！

动规五部曲分析如下：

1. 确定dp数组（dp table）以及下标的含义

dp[i]：和为i的完全平方数的最少数量为dp[i]

2. 确定递推公式

dp[j] 可以由dp[j - i * i]推出， dp[j - i * i] + 1 便可以凑成dp[j]。

此时我们要选择最小的dp[j]，所以递推公式： $dp[j] = \min(dp[j - i * i] + 1, dp[j])$;

3. dp数组如何初始化

dp[0]表示 和为0的完全平方数的最小数量，那么dp[0]一定是0。

有同学问题，那 $0 * 0$ 也算是一种啊，为啥dp[0] 就是 0呢？

看题目描述，找到若干个完全平方数（比如 1, 4, 9, 16, ...），题目描述中可没说要从0开始，dp[0]=0完全是为了递推公式。

非0下标的dp[j]应该是多少呢？

从递归公式 $dp[j] = \min(dp[j - i * i] + 1, dp[j])$;中可以看出每次dp[j]都要选最小的，所以非0下标的dp[i]一定要初始为最大值，这样dp[j]在递推的时候才不会被初始值覆盖。

4. 确定遍历顺序

我们知道这是完全背包，

如果求组合数就是外层for循环遍历物品，内层for遍历背包。

如果求排列数就是外层for遍历背包，内层for循环遍历物品。

在[动态规划：322. 零钱兑换](#)中我们就深入探讨了这个问题，本题也是一样的，是求最小数！

所以本题外层for遍历背包，里层for遍历物品，还是外层for遍历物品，内层for遍历背包，都是可以的！

我这里先给出外层遍历背包，里层遍历物品的代码：

```
vector<int> dp(n + 1, INT_MAX);
dp[0] = 0;
for (int i = 0; i <= n; i++) { // 遍历背包
    for (int j = 1; j * j <= i; j++) { // 遍历物品
        dp[i] = min(dp[i - j * j] + 1, dp[i]);
    }
}
```

5. 举例推导dp数组

已输入n为5例，dp状态图如下：

输入 $n = 5$

下标:	0	1	2	3	4	5
dp[i]:	0	1	2	3	1	2

$dp[0] = 0$

$dp[1] = \min(dp[0] + 1) = 1$

$dp[2] = \min(dp[1] + 1) = 2$

$dp[3] = \min(dp[2] + 1) = 3$

$dp[4] = \min(dp[3] + 1, dp[0] + 1) = 1$

$dp[5] = \min(dp[4] + 1, dp[1] + 1) = 2$



公众号：代码随想录

$dp[0] = 0$

$dp[1] = \min(dp[0] + 1) = 1$

$dp[2] = \min(dp[1] + 1) = 2$

$dp[3] = \min(dp[2] + 1) = 3$

$dp[4] = \min(dp[3] + 1, dp[0] + 1) = 1$

$dp[5] = \min(dp[4] + 1, dp[1] + 1) = 2$

最后的 $dp[n]$ 为最终结果。

C++代码

以上动规五部曲分析完毕C++代码如下：

```
// 版本一
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n + 1, INT_MAX);
        dp[0] = 0;
        for (int i = 0; i <= n; i++) { // 遍历背包
            for (int j = 1; j * j <= i; j++) { // 遍历物品
                dp[i] = min(dp[i - j * j] + 1, dp[i]);
            }
        }
        return dp[n];
    }
};
```


同样我在给出先遍历物品，在遍历背包的代码，一样的可以AC的。

```
// 版本二
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n + 1, INT_MAX);
        dp[0] = 0;
        for (int i = 1; i * i <= n; i++) { // 遍历物品
            for (int j = 1; j <= n; j++) { // 遍历背包
                if (j - i * i >= 0) {
                    dp[j] = min(dp[j - i * i] + 1, dp[j]);
                }
            }
        }
        return dp[n];
    }
};
```

总结

如果大家认真做了昨天的题目[动态规划：322. 零钱兑换](#)，今天这道就非常简单了，一样的套路一样的味道。

但如果没有按照「代码随想录」的题目顺序来做的话，做动态规划或者做背包问题，上来就做这道题，那还是挺难的！

经过前面的训练这道题已经是简单题了，哈哈哈

相信很多小伙伴刷题的时候面对力扣上近两千道题目，感觉无从下手，我花费半年时间整理了Github项目：「力扣刷题攻略」<https://github.com/youngyangyang04/leetcode-master>。里面有100多道经典算法题目刷题顺序、配有40w字的详细图解，常用算法模板总结，以及难点视频讲解，按照list一道一道刷就可以了！star支持一波吧！

- 公众号：[代码随想录](#)
- B站：[代码随想录](#)
- Github：[leetcode-master](#)
- 知乎：[代码随想录](#)



关注「代码随想录」

学习算法不迷路！

本周小结！（动态规划系列五）

周一

[动态规划：377. 组合总和 IV](#)中给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数（顺序不同的序列被视作不同的组合）。

题目面试虽然是组合，但又强调顺序不同的序列被视作不同的组合，其实这道题目求的是排列数！

递归公式： $dp[i] += dp[i - nums[j]]$;

这个和前上周讲的组合问题又不一样，关键就体现在遍历顺序上！

在[动态规划：518. 零钱兑换 II](#)中就已经讲过了。

如果求组合数就是外层for循环遍历物品，内层for遍历背包。

如果求排列数就是外层for遍历背包，内层for循环遍历物品。

如果把遍历nums（物品）放在外循环，遍历target的作为内循环的话，举一个例子：计算dp[4]的时候，结果集只有{1,3}这样的集合，不会有{3,1}这样的集合，因为nums遍历放在外层，3只能出现在1后面！

所以本题遍历顺序最终遍历顺序：**target**（背包）放在外循环，将**nums**（物品）放在内循环，内循环从前到后遍历。

```
class Solution {
public:
    int combinationSum4(vector<int>& nums, int target) {
        vector<int> dp(target + 1, 0);
```

```

        dp[0] = 1;
        for (int i = 0; i <= target; i++) { // 遍历背包
            for (int j = 0; j < nums.size(); j++) { // 遍历物品
                if (i - nums[j] >= 0 && dp[i] < INT_MAX - dp[i - nums[j]]) {
                    dp[i] += dp[i - nums[j]];
                }
            }
        }
        return dp[target];
    }
};

```

周二

爬楼梯之前我们已经做过了，就是斐波那契数列，很好解，但[动态规划：70. 爬楼梯进阶版（完全背包）](#)中我们进阶了一下。

改为：每次可以爬 1、2、.....、m 个台阶。问有多少种不同的方法可以爬到楼顶呢？

1阶，2阶，.... m阶就是物品，楼顶就是背包。

每一阶可以重复使用，例如跳了1阶，还可以继续跳1阶。

问跳到楼顶有几种方法其实就是问装满背包有几种方法。

此时大家应该发现这就是一个完全背包问题了！

和昨天的题目[动态规划：377. 组合总和 IV](#)基本就是一道题了，遍历顺序也是一样一样的！

代码如下：

```

class Solution {
public:
    int climbStairs(int n) {
        vector<int> dp(n + 1, 0);
        dp[0] = 1;
        for (int i = 1; i <= n; i++) { // 遍历背包
            for (int j = 1; j <= m; j++) { // 遍历物品
                if (i - j >= 0) dp[i] += dp[i - j];
            }
        }
        return dp[n];
    }
};

```

代码中m表示最多可以爬m个台阶，代码中把m改成2就是本题70.爬楼梯可以AC的代码了。

周三

[动态规划：322.零钱兑换](#) 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（每种硬币的数量是无限的）。

这里我们都知道这是完全背包。

递归公式： $dp[j] = \min(dp[j - \text{coins}[i]] + 1, dp[j])$;

关键看遍历顺序。

本题求钱币最小个数，那么钱币有顺序和没有顺序都可以，都不影响钱币的最小个数。。

所以本题并不强调集合是组合还是排列。

那么本题的两个for循环的关系是：外层for循环遍历物品，内层for遍历背包或者外层for遍历背包，内层for循环遍历物品都是可以的！

外层for循环遍历物品，内层for遍历背包：

```
// 版本一
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> dp(amount + 1, INT_MAX);
        dp[0] = 0;
        for (int i = 0; i < coins.size(); i++) { // 遍历物品
            for (int j = coins[i]; j <= amount; j++) { // 遍历背包
                if (dp[j - coins[i]] != INT_MAX) { // 如果dp[j - coins[i]]是初始
值则跳过
                    dp[j] = min(dp[j - coins[i]] + 1, dp[j]);
                }
            }
        }
        if (dp[amount] == INT_MAX) return -1;
        return dp[amount];
    }
};
```

外层for遍历背包，内层for循环遍历物品：

```
// 版本二
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> dp(amount + 1, INT_MAX);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) { // 遍历背包
            for (int j = 0; j < coins.size(); j++) { // 遍历物品
                if (i - coins[j] >= 0 && dp[i - coins[j]] != INT_MAX ) {
                    dp[i] = min(dp[i - coins[j]] + 1, dp[i]);
                }
            }
        }
    }
};
```

```

    }
    if (dp[amount] == INT_MAX) return -1;
    return dp[amount];
}
};

```

周四

[动态规划：279.完全平方数](#) 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（平方数可以重复使用）。

如果按顺序把前面的文章都看了，这道题目就是简单题了。dp[i]的定义，递推公式，初始化，遍历顺序，都是和[动态规划：322. 零钱兑换](#) 一样一样的。

要是没有前面的基础上来做这道题，那这道题目就有点难度了。

这也体现了刷题顺序的重要性。

先遍历背包，在遍历物品：

```

// 版本一
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n + 1, INT_MAX);
        dp[0] = 0;
        for (int i = 0; i <= n; i++) { // 遍历背包
            for (int j = 1; j * j <= i; j++) { // 遍历物品
                dp[i] = min(dp[i - j * j] + 1, dp[i]);
            }
        }
        return dp[n];
    }
};

```

先遍历物品，在遍历背包：

```

// 版本二
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n + 1, INT_MAX);
        dp[0] = 0;
        for (int i = 1; i * i <= n; i++) { // 遍历物品
            for (int j = 1; j <= n; j++) { // 遍历背包
                if (j - i * i >= 0) {
                    dp[j] = min(dp[j - i * i] + 1, dp[j]);
                }
            }
        }
    }
};

```

```
    }  
    return dp[n];  
}  
};
```

总结

本周的主题其实就是背包问题中的遍历顺序！

我这里做一下总结：

求组合数：[动态规划：518.零钱兑换II](#)

求排列数：[动态规划：377. 组合总和 IV](#)、[动态规划：70. 爬楼梯进阶版（完全背包）](#)

求最小数：[动态规划：322. 零钱兑换](#)、[动态规划：279.完全平方数](#)

此时我们就已经把完全背包的遍历顺序研究的透透的了！

动态规划：单词拆分

139.单词拆分

题目链接：<https://leetcode-cn.com/problems/word-break/>

给定一个非空字符串 s 和一个包含非空单词的列表 $wordDict$ ，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

拆分时可以重复使用字典中的单词。

你可以假设字典中没有重复的单词。

示例 1：

输入： $s = \text{"leetcode"}$, $wordDict = [\text{"leet"}, \text{"code"}]$

输出: true

解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

示例 2：

输入： $s = \text{"applepenapple"}$, $wordDict = [\text{"apple"}, \text{"pen"}]$

输出: true

解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。

注意你可以重复使用字典中的单词。

示例 3：

输入： $s = \text{"catsanddog"}$, $wordDict = [\text{"cats"}, \text{"dog"}, \text{"sand"}, \text{"and"}, \text{"cat"}]$

输出: false

思路

看到这道题目的时候，大家应该回想起我们之前讲解回溯法专题的时候，讲过的一道题目[回溯算法：分割回文串](#)，就是枚举字符串的所有分割情况。

[回溯算法：分割回文串](#)：是枚举分割后的所有子串，判断是否回文。

本道是枚举分割所有字符串，判断是否在字典里出现过。

那么这里我也给出回溯法C++代码：

```
class Solution {
private:
    bool backtracking (const string& s, const unordered_set<string>& wordSet,
int startIndex) {
        if (startIndex >= s.size()) {
            return true;
        }
        for (int i = startIndex; i < s.size(); i++) {
            string word = s.substr(startIndex, i - startIndex + 1);
            if (wordSet.find(word) != wordSet.end() && backtracking(s, wordSet,
i + 1)) {
                return true;
            }
        }
        return false;
    }
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
        return backtracking(s, wordSet, 0);
    }
};
```

- 时间复杂度： $O(2^n)$ ，因为每一个单词都有两个状态，切割和不切割
- 空间复杂度： $O(n)$ ，算法递归系统调用栈的空间

那么以上代码很明显要超时了，超时的数据如下：

```
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab"
["a", "aa", "aaa", "aaaa", "aaaaa", "aaaaaa", "aaaaaaa", "aaaaaaa", "aaaaaaa", "aaaaa
aaaaa"]
```

递归的过程中有很多重复计算，可以使用数组保存一下递归过程中计算的结果。

这个叫做记忆化递归，这种方法我们之前已经提过很多次了。

使用memory数组保存每次计算的以startIndex起始的计算结果，如果memory[startIndex]里已经被赋值了，直接用memory[startIndex]的结果。

C++代码如下：

```

class Solution {
private:
    bool backtracking (const string& s,
                      const unordered_set<string>& wordSet,
                      vector<int>& memory,
                      int startIndex) {
        if (startIndex >= s.size()) {
            return true;
        }
        // 如果memory[startIndex]不是初始值了, 直接使用memory[startIndex]的结果
        if (memory[startIndex] != -1) return memory[startIndex];
        for (int i = startIndex; i < s.size(); i++) {
            string word = s.substr(startIndex, i - startIndex + 1);
            if (wordSet.find(word) != wordSet.end() && backtracking(s, wordSet,
memory, i + 1)) {
                memory[startIndex] = 1; // 记录以startIndex开始的子串是可以被拆分的
                return true;
            }
        }
        memory[startIndex] = 0; // 记录以startIndex开始的子串是不可以被拆分的
        return false;
    }
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
        vector<int> memory(s.size(), -1); // -1 表示初始化状态
        return backtracking(s, wordSet, memory, 0);
    }
};

```

这个时间复杂度其实也是： $O(2^n)$ 。只不过对于上面那个超时测试用例优化效果特别明显。

这个代码就可以AC了，当然回溯算法不是本题的主菜，背包才是！

背包问题

单词就是物品，字符串s就是背包，单词能否组成字符串s，就是问物品能不能把背包装满。

拆分时可以重复使用字典中的单词，说明就是一个完全背包！

动规五部曲分析如下：

1. 确定dp数组以及下标的含义

dp[i]：字符串长度为i的话，**dp[i]**为true，表示可以拆分为一个或多个在字典中出现的单词。

2. 确定递推公式

如果确定dp[j] 是true，且 [j, i] 这个区间的子串出现在字典里，那么dp[i]一定是true。（ $j < i$ ）。

所以递推公式是 if([j, i] 这个区间的子串出现在字典里 && dp[j]是true) 那么 dp[i] = true。

3. dp数组如何初始化

从递归公式中可以看出， $dp[i]$ 的状态依靠 $dp[j]$ 是否为true，那么 $dp[0]$ 就是递归的根基， $dp[0]$ 一定要为true，否则递归下去后面都是false了。

那么 $dp[0]$ 有没有意义呢？

$dp[0]$ 表示如果字符串为空的话，说明出现在字典里。

但题目中说了“给定一个非空字符串 s ” 所以测试数据中不会出现 i 为0的情况，那么 $dp[0]$ 初始为true完全就是为了推导公式。

下标非0的 $dp[i]$ 初始化为false，只要没有被覆盖说明都是不可拆分为一个或多个在字典中出现的单词。

4. 确定遍历顺序

题目中说是拆分为一个或多个在字典中出现的单词，所以这是完全背包。

还要讨论两层for循环的前后循序。

如果求组合数就是外层for循环遍历物品，内层for遍历背包。

如果求排列数就是外层for遍历背包，内层for循环遍历物品。

对这个结论还有疑问的同学可以看这篇[本周小结！（动态规划系列五）](#)，这篇本周小节中，我做了如下总结：

求组合数：[动态规划：518.零钱兑换II](#)

求排列数：[动态规划：377. 组合总和 IV](#)、[动态规划：70. 爬楼梯进阶版（完全背包）](#)

求最小数：[动态规划：322. 零钱兑换](#)、[动态规划：279.完全平方数](#)

本题最终要求的是是否都出现过，所以对出现单词集合里的元素是组合还是排列，并不在意！

那么本题使用求排列的方式，还是求组合的方式都可以。

即：外层for循环遍历物品，内层for遍历背包 或者 外层for遍历背包，内层for循环遍历物品 都是可以的。

但本题还有特殊性，因为是要求子串，最好是遍历背包放在外循环，将遍历物品放在内循环。

如果要是外层for循环遍历物品，内层for遍历背包，就需要把所有的子串都预先放在一个容器里。（如果不理解的话，可以自己尝试这么写一写就理解了）

所以最终我选择的遍历顺序为：遍历背包放在外循环，将遍历物品放在内循环。内循环从前到后。

5. 举例推导 $dp[i]$

以输入： $s = \text{"leetcode"}$, $\text{wordDict} = [\text{"leet"}, \text{"code"}]$ 为例， dp 状态如图：

输入 "leetcode"
["leet", "code"]

下标:	0	1	2	3	4	5	6	7	8
dp[i]:	1	0	0	0	1	0	0	0	1

 公众号: 代码随想录

dp[s.size()]就是最终结果。

动规五部曲分析完毕, C++代码如下:

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
        vector<bool> dp(s.size() + 1, false);
        dp[0] = true;
        for (int i = 1; i <= s.size(); i++) {    // 遍历背包
            for (int j = 0; j < i; j++) {        // 遍历物品
                string word = s.substr(j, i - j); // substr(起始位置, 截取的个数)
                if (wordSet.find(word) != wordSet.end() && dp[j]) {
                    dp[i] = true;
                }
            }
        }
        return dp[s.size()];
    }
};
```

- 时间复杂度: $O(n^3)$, 因为substr返回子串的副本是 $O(n)$ 的复杂度 (这里的n是substring的长度)
- 空间复杂度: $O(n)$

总结

本题和我们之前讲解回溯专题的[回溯算法: 分割回文串](#)非常像, 所以我也给出了对应的回溯解法。

稍加分析, 便可知道本题是完全背包, 而且是求能否组成背包, 所以遍历顺序理论上讲 两层for循环谁先谁后都可以!

但因为分割子串的特殊性, 遍历背包放在外循环, 将遍历物品放在内循环更方便一些。

本题其实递推公式都不是重点，遍历顺序才是重点，如果我直接把代码贴出来，估计同学们也会想两个for循环的顺序理所当然就是这样，甚至都不会想为什么遍历背包的for循环为什么在外层。

不分析透彻不是Carl的风格啊，哈哈

相信很多小伙伴刷题的时候面对力扣上近两千道题目，感觉无从下手，我花费半年时间整理了Github项目：「力扣刷题攻略」<https://github.com/youngyangyang04/leetcode-master>。里面有100多道经典算法题目刷题顺序、配有40w字的详细图解，常用算法模板总结，以及难点视频讲解，按照list一道一道刷就可以了！star支持一波吧！

- 公众号：代码随想录
- B站：代码随想录
- Github：leetcode-master
- 知乎：代码随想录



关注「代码随想录」

学习算法不迷路！

动态规划：关于多重背包，你该了解这些！

之前我们已经系统的讲解了01背包和完全背包，如果没有看过的录友，建议先把如下三篇文章仔细阅读一波。

- [动态规划：关于01背包问题，你该了解这些！](#)
- [动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)
- [动态规划：关于完全背包，你该了解这些！](#)

这次我们再来说一说多重背包

多重背包

对于多重背包，我在力扣上还没发现对应的题目，所以这里就做一下简单介绍，大家大概了解一下。

有N种物品和一个容量为V的背包。第i种物品最多有Mi件可用，每件耗费的空间是Ci，价值是Wi。求解将哪些物品装入背包可使这些物品的耗费的空间总和不超过背包容量，且价值总和最大。

多重背包和01背包是非常像的，为什么和01背包像呢？

每件物品最多有Mi件可用，把Mi件摊开，其实就是一个01背包问题了。

例如：

背包最大重量为10。

物品为：

	重量	价值	数量
物品0	1	15	2
物品1	3	20	3
物品2	4	30	2

问背包能背的物品最大价值是多少？

和如下情况有区别么？

	重量	价值	数量
物品0	1	15	1
物品0	1	15	1
物品1	3	20	1
物品1	3	20	1
物品1	3	20	1
物品2	4	30	1
物品2	4	30	1

毫无区别，这就转成了一个01背包问题了，且每个物品只用一次。

这种方式来实现多重背包的代码如下：

```
void test_multi_pack() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    vector<int> nums = {2, 3, 2};
    int bagweight = 10;
    for (int i = 0; i < nums.size(); i++) {
        while (nums[i] > 1) { // nums[i]保留到1，把其他物品都展开
            weight.push_back(weight[i]);
        }
    }
}
```

```

        value.push_back(value[i]);
        nums[i]--;
    }
}

vector<int> dp(bagweight + 1, 0);
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = bagweight; j >= weight[i]; j--) { // 遍历背包容量
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
    for (int j = 0; j <= bagweight; j++) {
        cout << dp[j] << " ";
    }
    cout << endl;
}
cout << dp[bagweight] << endl;

}
int main() {
    test_multi_pack();
}

```

- 时间复杂度： $O(m * n * k)$ m ：物品种类个数， n 背包容量， k 单类物品数量

也有另一种实现方式，就是把每种商品遍历的个数放在01背包里面在遍历一遍。

代码如下：（详看注释）

```

void test_multi_pack() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    vector<int> nums = {2, 3, 2};
    int bagweight = 10;
    vector<int> dp(bagweight + 1, 0);

    for(int i = 0; i < weight.size(); i++) { // 遍历物品
        for(int j = bagweight; j >= weight[i]; j--) { // 遍历背包容量
            // 以上为01背包，然后加一个遍历个数
            for (int k = 1; k <= nums[i] && (j - k * weight[i]) >= 0; k++) { //
遍历个数
                dp[j] = max(dp[j], dp[j - k * weight[i]] + k * value[i]);
            }
        }
        // 打印一下dp数组
        for (int j = 0; j <= bagweight; j++) {
            cout << dp[j] << " ";
        }
    }
}

```

```

        cout << endl;
    }
    cout << dp[bagweight] << endl;
}
int main() {
    test_multi_pack();
}

```

- 时间复杂度： $O(m * n * k)$ m ：物品种类个数， n 背包容量， k 单类物品数量

从代码里可以看出是01背包里面在加一个for循环遍历一个每种商品的数量。和01背包还是如出一辙的。

当然还有那种二进制优化的方法，其实就是把每种物品的数量，打包成一个个独立的包。

和以上在循环遍历上有所不同，因为是分拆为各个包最后可以组成一个完整背包，具体原理我就不做过多解释了，大家了解一下就行，面试的话基本不会考完这个深度了，感兴趣可以自己深入研究一波。

总结

多重背包在面试中基本不会出现，力扣上也没有对应的题目，大家对多重背包的掌握程度知道它是一种01背包，并能在01背包的基础上写出对应代码就可以了。

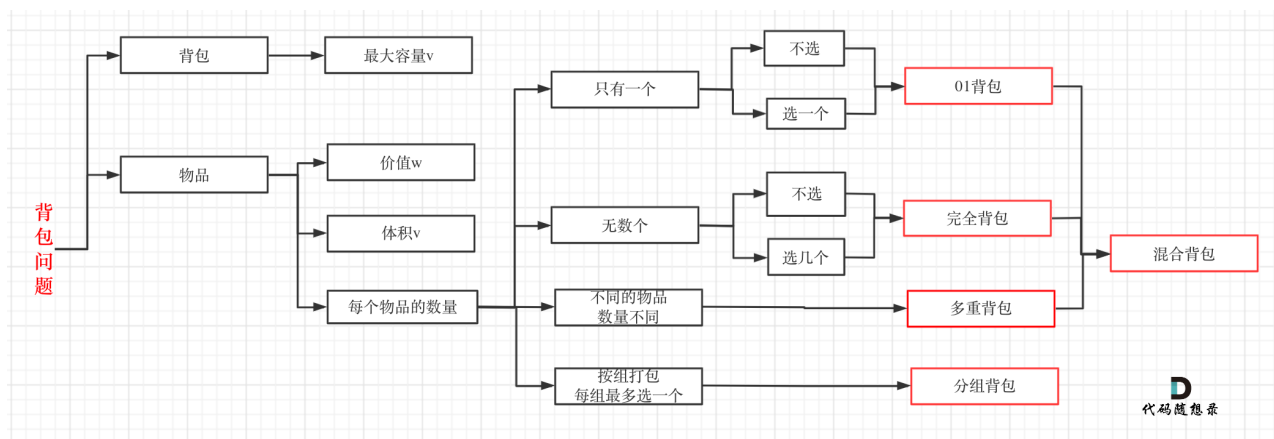
至于背包九讲里面还有混合背包，二维费用背包，分组背包等等这些，大家感兴趣可以自己去学习学习，这里也不做介绍了，面试也不会考。

听说背包问题很难？这篇总结篇来拯救你了

年前我们已经把背包问题都讲完了，那么现在我们要对背包问题进行总结一番。

背包问题是动态规划里的非常重要的一部分，所以我把背包问题单独总结一下，等动态规划专题更新完之后，我们还会在整体总结一波动态规划。

关于这几种常见的背包，其关系如下：



通过这个图，可以很清晰分清这几种常见背包之间的关系。

在讲解背包问题的时候，我们都是按照如下五部来逐步分析，相信大家也体会到，把这五部都搞透了，算是对动规来理解深入了。

1. 确定dp数组（dp table）以及下标的含义
2. 确定递推公式
3. dp数组如何初始化
4. 确定遍历顺序
5. 举例推导dp数组

其实这五部里哪一步都很关键，但确定递推公式和确定遍历顺序都具有规律性和代表性，所以下面我从这两点来对背包问题做一做总结。

背包递推公式

问能否能装满背包（或者最多装多少）： $dp[j] = \max(dp[j], dp[j - \text{nums}[i]] + \text{nums}[i])$ ；，对应题目如下：

- [动态规划：416.分割等和子集](#)
- [动态规划：1049.最后一块石头的重量 II](#)

问装满背包有几种方法： $dp[j] += dp[j - \text{nums}[i]]$ ，对应题目如下：

- [动态规划：494.目标和](#)
- [动态规划：518.零钱兑换 II](#)
- [动态规划：377.组合总和IV](#)
- [动态规划：70.爬楼梯进阶版（完全背包）](#)

问背包装满最大价值： $dp[j] = \max(dp[j], dp[j - \text{weight}[i]] + \text{value}[i])$ ；，对应题目如下：

- [动态规划：474.一和零](#)

问装满背包所有物品的最小个数： $dp[j] = \min(dp[j - \text{coins}[i]] + 1, dp[j])$ ；，对应题目如下：

- [动态规划：322.零钱兑换](#)
- [动态规划：279.完全平方数](#)

遍历顺序

01背包

在[动态规划：关于01背包问题，你该了解这些！](#)中我们讲解二维dp数组01背包先遍历物品还是先遍历背包都是可以的，且第二层for循环是从小到大遍历。

和[动态规划：关于01背包问题，你该了解这些！（滚动数组）](#)中，我们讲解一维dp数组01背包只能先遍历物品再遍历背包容量，且第二层for循环是从大到小遍历。

一维dp数组的背包在遍历顺序上和二维dp数组实现的01背包其实是有很大差异的，大家需要注意！

完全背包

说完01背包，再看看完全背包。

在[动态规划：关于完全背包，你该了解这些！](#)中，讲解了纯完全背包的一维dp数组实现，先遍历物品还是先遍历背包都是可以的，且第二层for循环是从小到大遍历。

但是仅仅是纯完全背包的遍历顺序是这样的，题目稍有变化，两个for循环的先后顺序就不一样了。

如果求组合数就是外层for循环遍历物品，内层for遍历背包。

如果求排列数就是外层for遍历背包，内层for循环遍历物品。

相关题目如下：

- 求组合数： [动态规划：518.零钱兑换II](#)
- 求排列数： [动态规划：377. 组合总和 IV](#)、[动态规划：70. 爬楼梯进阶版（完全背包）](#)

如果求最小数，那么两层for循环的先后顺序就无所谓了，相关题目如下：

- 求最小数： [动态规划：322. 零钱兑换](#)、[动态规划：279.完全平方数](#)

对于背包问题，其实递推公式算是容易的，难是难在遍历顺序上，如果把遍历顺序搞透，才算是真正理解了。

总结

这篇背包问题总结篇是对背包问题的高度概括，讲最关键的两部：递推公式和遍历顺序，结合力扣上的题目全都抽象出来了。

而且每一个点，我都给出了对应的力扣题目。

最后如果你想了解多重背包，可以看这篇[动态规划：关于多重背包，你该了解这些！](#)，力扣上还没有多重背包的题目，也不是面试考察的重点。

如果把我本篇总结出来的内容都掌握的话，可以说对背包问题理解的就很深刻了，用来对付面试中的背包问题绰绰有余！

相信很多小伙伴刷题的时候面对力扣上近两千道题目，感觉无从下手，我花费半年时间整理了Github项目：「力扣刷题攻略」<https://github.com/youngyangyang04/leetcode-master>。里面有100多道经典算法题目刷题顺序、配有40w字的详细图解，常用算法模板总结，以及难点视频讲解，按照list一道一道刷就可以了！star支持一波吧！

- 公众号： [代码随想录](#)
- B站： [代码随想录](#)
- Github： [leetcode-master](#)
- 知乎： [代码随想录](#)



关注「代码随想录」

学习算法不迷路！