经典动态规划：0-1 背包问题

Stars 108k | B站 @labuladong | 配套PDF和插件 下载 | 打卡挑战 报名 | 精品课程 查看

微信搜一搜 labuladong公众号

通知：数据结构精品课 V1.6 持续更新中，第八期打卡挑战 开始报名，算法私教课 开始预约。

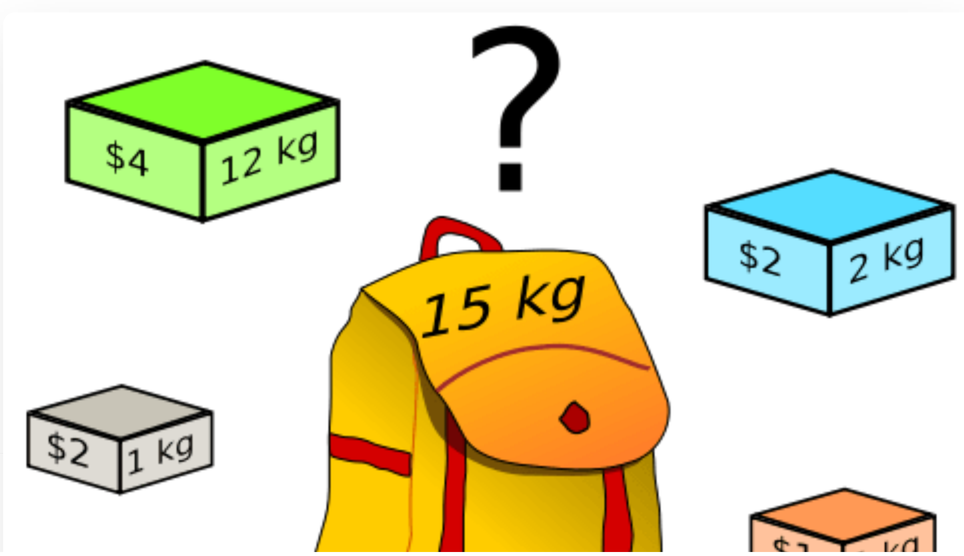
———

本文有视频版：0-1背包问题详解

后台天天有人问背包问题，这个问题其实不难啊，如果我们号动态规划系列的十几篇文章你都看过，借助框架，遇到背包问题可以说是手到擒来好吧。无非就是状态 + 选择，也没啥特别之处嘛。

今天就来说一下背包问题吧，就讨论最常说的 0-1 背包问题。描述：

给你一个可装载重量为 `W` 的背包和 `N` 个物品，每个物品有重量和价值两个属性。其中第 `i` 个物品的重量为 `wt[i]`，价值为 `val[i]`，现在让你用这个背包装物品，最多能装的价值是多少？



举个简单的例子，输入如下：

```
N = 3, W = 4
wt = [2, 1, 3]
val = [4, 2, 3]
```

算法返回 6，选择前两件物品装进背包，总重量 3 小于 `W`，可以获得最大价值 6。

题目就是这么简单，一个典型的动态规划问题。这个题目中的物品不可以分割，要么装进包里，要么不装，不能说切成两块装一半。这就是 0-1 背包这个名词的来历。

解决这个问题没有什么排序之类巧妙的方法，只能穷举所有可能，根据我们 动态规划详解 中的套路，直接走流程就行了。

动规标准套路

看来每篇动态规划文章都得重复一遍套路，历史文章中的动态规划问题都是按照下面的套路来的。

第一步要明确两点，「状态」和「选择」。

先说状态，如何才能描述一个问题局面？只要给几个物品和一个背包的容量限制，就形成了一个背包问题呀。**所以状态有两个，就是「背包的容量」和「可选择的物品」**。

再说选择，也很容易想到啊，对于每件物品，你能选择什么？**选择就是「装进背包」或者「不装进背包」嘛**。

明白了状态和选择，动态规划问题基本上就解决了，只要往这个框架套就完事儿了：

```
for 状态1 in 状态1的所有取值：
```

```
dp[状态1][状态2][...] = 择优(选择1，选择2...)
```

> PS：此框架出自历史文章 团灭 LeetCode 股票问题。

第二步要明确 `dp` 数组的定义。

首先看看刚才找到的「状态」，有两个，也就是说我们需要一个二维 `dp` 数组。

`dp[i][w]` 的定义如下：对于前 `i` 个物品，当前背包的容量为 `w`，这种情况下可以装的最大价值是 `dp[i][w]`。

比如说，如果 `dp[3][5] = 6`，其含义为：对于给定的一系列物品中，若只对前 3 个物品进行选择，当背包容量为 5 时，最多可以装下的价值为 6。

> PS：为什么要这么定义？便于状态转移，或者说这就是套路，记下来就行了。建议看一下我们的动态规划系列文章，几种套路都被扒得清清楚楚了。

根据这个定义，我们想求的最终答案就是 `dp[N][W]`。base case 就是 `dp[0][..] = dp[..][0] = 0`，因为没有物品或者背包没有空间的时候，能装的最大价值就是 0。

细化上面的框架：

```
int[][] dp[N+1][W+1]
dp[0][..] = 0
dp[..][0] = 0

for i in [1..N]:
    for w in [1..W]:
        dp[i][w] = max(
            把物品 i 装进背包,
            不把物品 i 装进背包
        )
return dp[N][W]
```

第三步，根据「选择」，思考状态转移的逻辑。

简单说就是，上面伪码中「把物品 `i` 装进背包」和「不把物品 `i` 装进背包」怎么用代码体现出来

这就要结合对 `dp` 数组的定义，看看这两种选择会对状态产生什么影响：

先重申一下刚才我们的 `dp` 数组的定义：

`dp[i][w]` 表示：对于前 `i` 个物品（从 1 开始计数），当前背包的容量为 `w` 时，这种情况下可以装下的最大价值是 `dp[i][w]`。

如果你没有把这第 `i` 个物品装入背包，那么很显然，最大价值 `dp[i][w]` 应该等于 `dp[i-1][w]`，继承之前的结果。

如果你把这第 `i` 个物品装入了背包，那么 `dp[i][w]` 应该等于 `val[i-1] + dp[i-1][w - wt[i-1]]`。

首先，由于数组索引从 0 开始，而我们定义中的 `i` 是从 1 开始计数的，所以 `val[i-1]` 和 `wt[i-1]` 表示第 `i` 个物品的价值和重量。

你如果选择将第 `i` 个物品装进背包，那么第 `i` 个物品的价值 `val[i-1]` 肯定就到手了，接下来你就要在剩余容量 `w - wt[i-1]` 的限制下，在前 `i - 1` 个物品中挑选，求最大价值，即 `dp[i-1][w - wt[i-1]]`。

综上就是两种选择，我们都已经分析完毕，也就是写出来了状态转移方程，可以进一步细化代码：

```
for i in [1..N]:
    for w in [1..W]:
        dp[i][w] = max(
            dp[i-1][w],
            dp[i-1][w - wt[i-1]] + val[i-1]
        )
return dp[N][W]
```

最后一步，把伪码翻译成代码，处理一些边界情况。

我用 Java 写的代码，把上面的思路完全翻译了一遍，并且处理了 `w - wt[i-1]` 可能小于 0 导致数组索引越界的问题：

```
int knapsack(int W, int N, int[] wt, int[] val) {
    // base case 已初始化
```

```
            for (int w = 1; w <= W; w++) {
                if (w - wt[i - 1] < 0) {
                    // 这种情况下只能选择不装入背包
                    dp[i][w] = dp[i - 1][w];
                } else {
                    // 装入或者不装入背包，择优
                    dp[i][w] = Math.max(
                        dp[i - 1][w - wt[i-1]] + val[i-1],
                        dp[i - 1][w]
                    );
                }
            }
        }

        return dp[N][W];
    }
```

至此，背包问题就解决了，相比而言，我觉得这是比较简单的动态规划问题，因为状态转移的推导比较自然，基本上你明确了 `dp` 数组的定义，就可以理所当然地确定状态转移了。

接下来可阅读：

- 完全背包问题之零钱兑换
- 背包问题变体之子集分割

▸ **引用本文的文章**