经典动态规划：0-1 背包问题

原创 labuladong labuladong 2020年03月09日 11:59

点击上方蓝字设为星标✨
东哥带你手把手撕力扣~

作者：labuladong

公众号：labuladong

若已授权白名单也必须保留以上来源信息

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，简单描述一下吧：

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。**这也许就是 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 动态规划套路详解 中的套路，直接走流程就行了。

动规标准套路

看来我得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，今天再来手把手演示一下：

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？只要给定几个可选物品和一个背包的容量限制，就形成了一个背包问题，对不对？**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：
    for 状态2 in 状态2的所有取值：
        for ...
            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 股票买卖问题。

第二步要明确 `dp` 数组的定义。

`dp` 数组是什么？其实就是描述问题局面的一个数组。换句话说，我们刚才明确问题有什么「状态」，现在需要用 `dp` 数组把状态表示出来。

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，一维表示可选择的物品，一维表示背包的容量。

`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，几种动规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呢？

这一步要结合对 dp 数组的定义和我们的算法逻辑来分析：

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，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] 。你不装嘛，那就继承之前的结果。

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

首先，由于 i 是从 1 开始的，所以对 val 和 wt 的取值是 i-1 。

而 dp[i-1][w-wt[i-1]] 也很好理解：你如果想装第 i 个物品，你怎么计算这时候的最大价值？**换句话说，在装第 i 个物品的前提下，背包能装的最大价值是多少？**

显然，你应该寻求剩余重量 w-wt[i-1] 限制下能装的最大价值，加上第 i 个物品的价值 val[i-1] ，这就是装第 i 个物品的前提下，背包可以装的最大价值。

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 C++ 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, vector<int>& wt, vector<int>& val) {
    // vector 全填入 0，base case 已初始化
    vector<vector<int>> dp(N + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= N; i++) {
        for (int w = 1; w <= W; w++) {
            if (w - wt[i-1] < 0) {
                // 当前背包容量装不下，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] = max(dp[i - 1][w - wt[i-1]] + val[i-1],
                               dp[i - 1][w]);
```

```
            }
        }
    }

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

现在你看这个解法代码，是不是感觉非常简单，就是把我们刚才分析的思路原封不动翻译了一下而已。

所以说，明确了动态规划的套路，思路就显得行云流水，非常自然就出答案了。

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逻辑比较容易想到，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。

手把手刷动态规划 31 二维动态规划 16 背包问题 3

手把手刷动态规划 · 目录

上一篇
动态规划答疑篇

下一篇
经典动态规划：0-1背包问题的变体

阅读原文