背包专题

简介

背包问题是一类非常经典的动态规划问题，日常使用场景非常灵活。背包问题在动态规划中 的比例非常大，以至于我们单独将其从动态规划中抽取出来进行讲解。

百度百科定义：背包问题(Knapsack problem)是一种组合优化的 NP 完全问题。问题的名称 来源于如何选择最合适的物品放置于给定背包中。相似问题经常出现在商业、组合数学，计 算复杂性理论、密码学和应用数学等领域中。也可以将背包问题描述为决定性问题，即在总 重量不超过 W 的前提下，总价值是否能达到 V？它是在 1978 年由 Merkle 和 Hellman 提 出的。

常见题型及对应模版

背包问题可以描述为：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内， 我们如何选择，才能使得物品的总价格最高。

下面给大家归纳几种常用的背包问题（01 背包，完全背包和多重背包三种类型）及其对应 模版。说实话，如果实在理解不了，直接背住模版，把题目对应数据处理一下直接套题目， 练习多了再回过头来复习可能就会**恍然大悟**。

有一点需要大家注意：几乎没有一道题是直接告诉你是背包的，这需要你自己的抽象能力。 将问题抽象为背包，然后使用背包的套路去解决。我们也会在接下来的几天出几个题目，大 家可以尝试将其抽象为背包问题。

01 背包问题

01 背包是最简单的类型，并且完全背包和多重背包都可以转化为 01 背包问题，因此搞清 楚 01 背包是非常重要的。

问题描述

有 n 个物品，每个物品对应的重量为 w，价值为 v，问在不超过背包重量 M 的情况下，能 够装入物品的最大价值，每个物品只能使用一次。

w[i]是第 i 个物品的重量，v[i]是第 i 个物品的价值。

分析

简单的思路是找到**所有物品的组合**（复杂度为指数），然后判断组合的体积和是否大于 M，如果不大于 M，则选择性更新最大价值 max_v（是否更新取决于当前的组合总价值是否 大于 max_v），最后返回 max_v 即可。

n 个物品的组合的数量的“数量级”是2^n，n 稍微大点就很恐怖了，我们不得不考虑进 行优化。

套用 01 背包模型算法可以将复杂度降到多项式，具体来说是 $O(n * W)$，具体怎么做呢？

定义状态 dp[i][j]表示仅考虑前 i 个物品将其装入承重为 j 的背包可以获得的最大价值， 显然最终返回 dp[n][m] 即可。

接下来考虑状态转移，具体会有如下两种情况：

- 当前第 i 件物品我要了（前提背包要装得下）， dp[i][j] 就是当前物品的价值 v[i] + 仅考虑前 i - 1 件剩余容量为 j - w[i](已经装了第 i 件物品)的最大价值（就是 dp[i-1]j-w[i]]）。 `dp[i][j] = dp[i - 1][j - w[i]] + v[i]`
- 当前第 i 件物品我不要就更简单了，dp[i][j] = dp[i - 1][j]

由于我们的目标是价值最大， 那么我们当然要选以上两种情况的最大值。

因此可以得到状态转移方程如下：

$$\mathrm{dp[i][j]} = \max(\mathrm{dp[i-1][j]}, \mathrm{dp[i-1][j-w[i]]} + \mathrm{v[i]}), \mathrm{j} >= \mathrm{[i]}$$

通过上述分析可以很容易写出如下代码：

```
N, M, W, V
dp[0..N][0..M] = 0

for i in 1...N:
    for j in W[i]...M:
        dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - W[i]] + V[i]

return dp[N][M]
```

不难发现当前 i 对应的状态计算只和 i - 1 有关，我们可以用动态规划专题讲到的**滚动 数组**进行优化空间使其降至 M。

模板如下：

```
N, M, W, V
dp[0..M] = 0

for i in 1...N:
    for j in M...W[i]: # 这里必须逆向枚举，如果正向的话i状态会覆盖掉i-1的状态
        dp[j] = max(dp[j], dp[j - W[i]] + V[i]

return dp[M]
```

关于为何此处必须逆向枚举这个问题。简单来说，**如果你不使用滚动数组，那么怎么枚举 都无所谓，但是如果使用了在这里就必须逆序枚举**。原因的话，我在文章末尾给大家解释 。

这就是 01 背包问题。

完全背包问题

问题描述

有 n 个物品，每个物品对应的重量为 w，价值为 v，问在不超过背包重量 M 的情况下，能 够装入物品的最大价值，与 01 背包的区别是每个物品可以使用**无限次**。

分析

完全背包问题状态转移方程和 01 背包问题很类似：

dp[i][j]表示将前 i 个物品装入承重为 j 的背包可以获得的最大价值。

那么 dp[i][j]求解时对应以下两种情况：

- 当前第 i 件物品我要了（前提背包要装得下）： `dp[i][j - w[i]] + v[i]` ， w[i]是第 i 个物品的重量，v[i]是第 i 个物品的价值。（这里注意，这里是和 01 背包的区别所 在，因为当前物品可以无限次被选，因此不应该用 i-1 的状态计算而是继续在 i 状态）

- 当前第 i 件物品我不要：dp[i - 1][j]

因此可以得到状态转移方程如下： $$dp[i][j] = max(dp[i - 1][j], dp[i][j - w[i]] + v[i]), j>=w[i]$$

一样，还是可以用滚动数组进行空间上的优化，大致模版如下：

```
N, M, W, V
dp[0..M] = 0

for i in 1...N:
    for j in W[i]...M: # 这里必须正向枚举，因为当前计算需要dp[i]对应的其他状态来计算
        dp[j] = max(dp[j], dp[j - W[i]] + V[i]

return dp[M]
```

多重背包问题

问题描述

该问题的描述和上面的区别仅仅在于，每个物品的个数有限制。

分析

分析过程和上面也很类似，直接写出状态转移方程：

$$\mathrm{dp[i][j] = max((dp[i-1][j-h*w[i]] + h*v[i])foreveryh)}$$

其中 h 为装入第 i 件物品的个数，h≤min(H[i], j / W[i]), H 为物品及其个数的对应关 系。

因为装入第 i 物品是从 0-h 计算的，因此 dp[i]需要 dp[i - 1]的状态辅助完成，因此可 以采用 01 背包优化的方式来优化空间使用，下面是模板代码：

```
N, M, W, V, H
dp[0..M] = 0

for i in 1...N:
    for j in M...W[i]: # 这里必须逆向枚举，因为当前计算需要dp[i - 1]对应的其他状态来计算
        for h in 0...min(H[i], j / W[i]):
            dp[j] = max(dp[j], dp[j - h * W[i]] + h * V[i])

return dp[M]
```

背包问题几个关键点

1. 为什么 01 背包需要倒序，而完全背包则不可以

实际上，这是一个骚操作，我来详细给你讲一下。

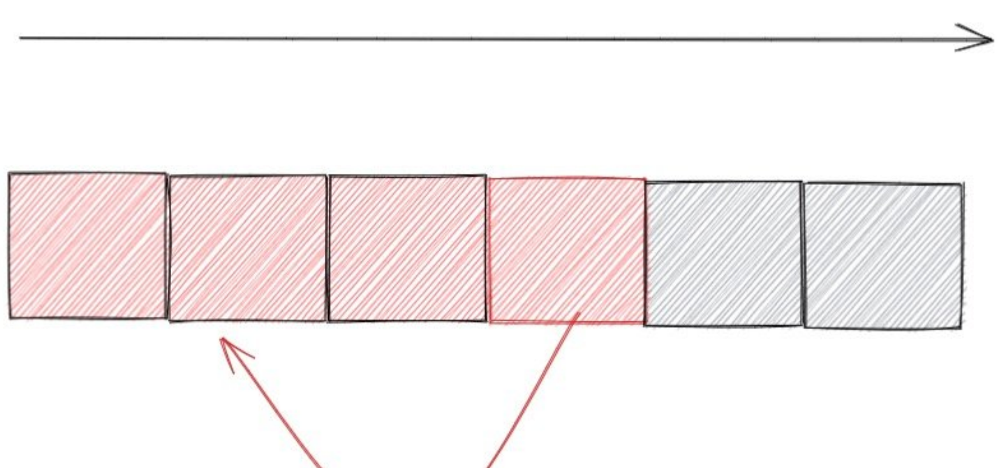
其实要回答这个问题，我要先将 01 背包和完全背包退化二维的情况。

对于 01 背包：

```
for i in 1 to N + 1:
    for j in V to 0:
        dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - cost[i - 1])
```

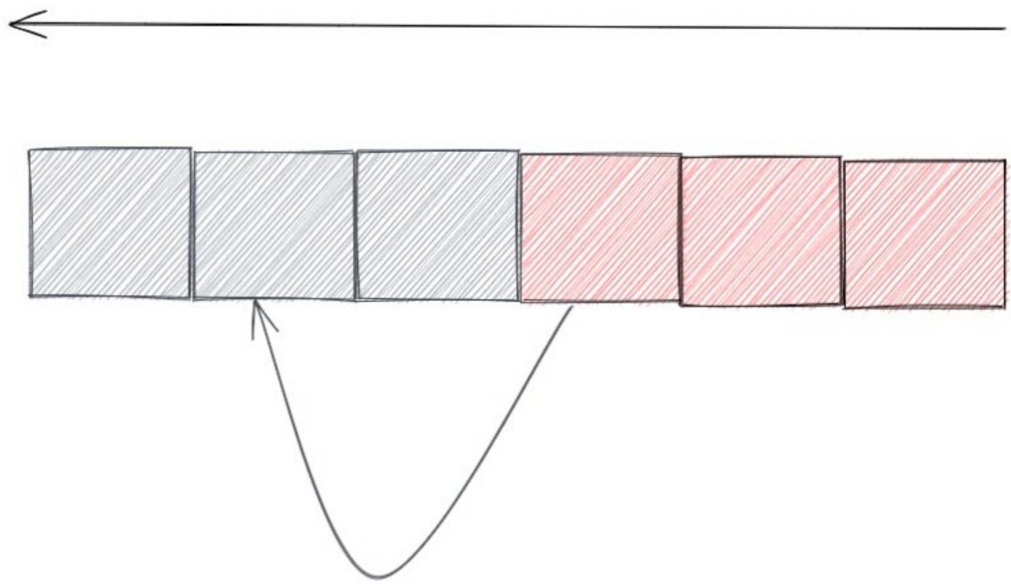
注意等号左边是 i，右边是 i - 1，这很好理解，因为 i 只能取一次嘛。

那么如果我们不倒序遍历会怎么样呢？



如图橙色部分表示已经遍历的部分，而让我们去用[j - cost[i - 1]] 往前面回溯的时候， 实际上回溯的是 dp[i]j - cost[i - 1]]，而不是 dp[i - 1]j - cost[i - 1]]。

如果是倒序就可以了，如图：



这个明白的话，我们继续思考为什么完全背包就要不降序了呢？

我们还是像上面一样写出二维的代码：

```
for i in 1 to N + 1:
   for j in 1 to V + 1:
       dp[i][j] = max(dp[i - 1][j], dp[i][j - cost[i - 1]])
```

由于 i 可以取无数次，那么正序遍历正好可以满足，如上图。

2. 恰好装满 VS 可以不装满

题目有两种可能，一种是要求背包恰好装满，一种是可以不装满（只要不超过容量就行）。 而本题是要求 恰好装满 的。而这两种情况仅仅影响我们 dp数组初始化 。

- 恰好装满。只需要初始化 dp[0] 为 0， 其他初始化为负数即可。

- 可以不装满。 只需要全部初始化为 0，即可，

原因很简单，我多次强调过 dp 数组本质上是记录了一个个自问题。 dp[0]是一个子问题， dp[1]是一个子问题。。。

有了上面的知识就不难理解了。 初始化的时候，我们还没有进行任何选择，那么也就是说 dp[0] = 0，因为我们可以通过什么都不选达到最大值 0。而 dp[1],dp[2]...则在当前什么 都不选的情况下无法达成，也就是无解，因为为了区分，我们可以用负数来表示，当然你可 以用任何可以区分的东西表示，比如 None。

两层循环的位置可以换么?

所以这两层循环的位置起的实际作用是什么？ 代表的含义有什么不同?

本质上:

```
for i in 1 to N + 1:
    for j in V to 0:
        ...
```

这种情况选择物品 1 和物品 3（随便举的例子），是一种方式。选择物品 3 个物品 1（注 意是有顺序的）是同一种方式。 **原因在于你是固定物品，去扫描容量。**

而：

```
for j in V to 0:
    for i in 1 to N + 1:
        ...
```

这种情况选择物品 1 和物品 3（随便举的例子），是一种方式。选择物品 3 个物品 1（注 意是有顺序的）也是一种方式。**原因在于你是固定容量，去扫描物品。**

因此总的来说，如果你认为[1,3]和[3,1]是一种，那么就用方法 1 的遍历，否则用方法 2。

扩展

最后贴几个我写过的背包问题，让大家看看历史是多么的相似。

除了dp[0]全部初始化一个不可能的解

Python3 Code:

```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0

        for i in range(1, amount + 1):
            for j in range(len(coins)):
                if i >= coins[j]:
                    dp[i] = min(dp[i], dp[i - coins[j]] + 1)

        return -1 if dp[-1] == amount + 1 else dp[-1]
```

复杂度分析

- 时间复杂度：$O(amonut * len(coins))$
- 空间复杂度：$O(amount)$

(322. ***找零(完全背包问题))

> 这里内外循环和本题正好是反的，我只是为了"秀技"(好玩)，实际上在这里对答案并不影 响。

Python Code:

```python
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [0] * (amount + 1)
        dp[0] = 1

        for j in range(len(coins)):
            for i in range(1, amount + 1):
                if i >= coins[j]:
                    dp[i] += dp[i - coins[j]]

        return dp[-1]
```

(518. 零钱兑换 II)

> 这里内外循环和本题正好是反的，但是这里必须这么做，否则结果是不对的，具体可以点 进去链接看我那个题解

- **5269. 从栈中取出 K 个硬币的最大面值和**

5269. 从栈中取出 K 个硬币的最大面值和

难度 困难 4 收藏 分享 切换为英文 接收动态 反馈

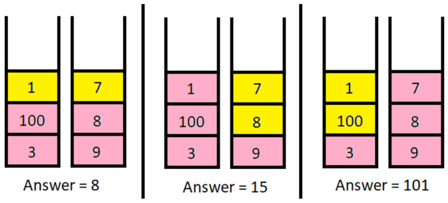
一张桌子上总共有 `n` 个硬币 **栈** 。每个栈有 **正整数** 个带面值的硬币。

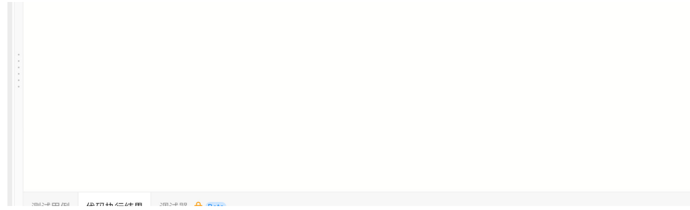
每一次操作中，你可以从任意一个栈的 **顶部** 取出 1 个硬币，从栈中移除它，并放入你的钱包里。

给你一个列表 `piles` ，其中 `piles[i]` 是一个整数数组，分别表示第 `i` 个栈里 **从顶到底** 的硬币面值。同时给你一个正整数 `k` ，请你返回在 **恰好** 进行 `k` 次操作的前提下，你钱包里硬币面值之和 **最大为多少** 。

示例 1：

```python
class Solution:
    def maxValueOfCoins(self, piles: List[List[int]], k: int) -> int:
        dp = [0] * (k + 1)
        pres = []
        for pile in piles:
            pres.append(list(accumulate(pile)))
        for i in range(len(piles)):
            for j in range(k,-1,-1):
                for w in range(min(j, len(piles[i]))):
                    dp[j] = max(dp[j], dp[j - w - 1] + pres[i][w])
        return dp[-1]
```





将 piles 看成是 n 个背包，其中第 i 个背包 piles[i] 有 m 个物品， piles[i]的前缀 和为 pres[i]，那么第 i 个物品的价值可以看成是 pres[i]，体积看为 i + 1。这样问题 转化为背包问题。只不过这道题是多重背包，因此多了一层循环。

总结

万变不离其宗，还有背包的很多变形版本，以及不一定求最大价值，dp 的定义以及初始化 是很灵活的，后面题目会涉及部分知识。

建议大家把模版翻译成自己擅长的语言，关于背包问题的详细介绍还请查阅背包问题经典的 参考资料： **背包九讲第二版**。

上一页 下一页