



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

算法导论期末作业

生活中背包问题及优化分析

蒋薇

年级：2021 级

专业：计算机科学与技术

指导教师：苏明

2023 年 6 月 3 日

摘要

背包问题是一类经典的动态规划问题，其基本思想是在限定总体积或总重量的前提下，选择一些物品放入背包中，使得背包内物品的总价值最大化。

关键字：背包问题，动态规划，优化

目录

一、 问题描述	1
二、 设计思想	1
(一) 暴力解法	1
(二) 二维算法	2
1. 确定递推公式	2
2. dp 数组初始化	2
3. 确定遍历顺序	2
(三) 优化一维数组	3
1. 确定 dp 数组定义	3
2. 一维 dp 数组递推公式	3
3. 数组初始化	4
4. 一维数组遍历顺序	4
(四) 实验结果	4
(五) 实验分析	6
(六) 引申	7

一、 问题描述

假设你要去徒步旅行，你需要带上一些必要的物品，包括帐篷、睡袋、衣服、食品等。你的背包容量有限，不能超过一定重量。你需要在这些物品中选择一些，使得它们的总重量不超过背包容量，同时满足你的旅行需求，例如保暖、饱腹等。同时，你也希望这些物品的总价值尽可能高。

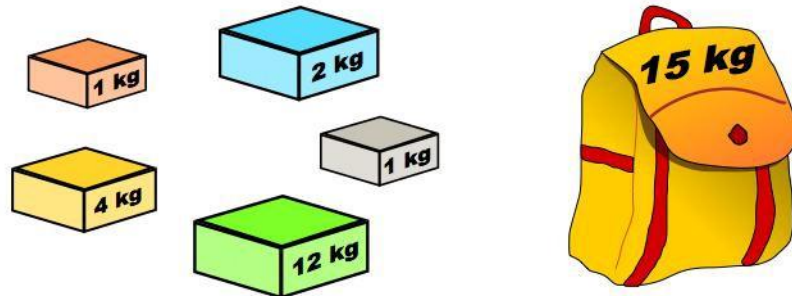


图 1: Question

数学描述：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。

name	weight(kg)	value(yuan)
thing1	1	15
thing2	3	20
thing3	4	30

表 1: 物品概况表 1

name	weight	value
camp	3	200
sleeping bag	2	150
clothes	1	80
food	5	160

表 2: 物品概况表 2

二、 设计思想

(一) 暴力解法

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。

(二) 二维算法

1. 确定递推公式

$dp[i][j]$ 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 $dp[i][j]$ ，当 $j < weight[i]$ 时， $f(i, j) = f(i-1, j)$ 。即当前的背包容量无法容纳第 i 件物品，只能选择不装入。

当 $j \geq weight[i]$ 时， $f(i, j) = \max f(i-1, j), f(i-1, j-weight[i]) + value[i]$ 。即当前的背包容量可以容纳第 i 件物品，我们就需要判断装入还是不装入背包，取两者之间的最大值。

最终的结果为 $f(N, W)$ ，即将所有物品装入容量为 W 的背包中所能获得的最大价值。所以递归公式：

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

2. dp 数组初始化

$dp[i][j]$ 的定义出发，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。

状态转移方程 $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-weight[i]] + value[i])$ ；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。

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

图 2: 初始化

注意： $dp[i][j]$ 在推导的时候一定是取价值最大的数，如果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。

而背包问题的物品价值都是正整数，所以初始化为 0，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。

3. 确定遍历顺序

主要代码

先遍历物品，然后遍历背包重量

```

1 // weight数组的大小 就是物品个数
2 for(int i = 1; i < weight.size(); i++) { // 遍历物品
3     for(int j = 0; j <= bagWeight; j++) { // 遍历背包容量
4         if (j < weight[i])
5             dp[i][j] = dp[i - 1][j]; // 这个是为了展现dp数组里元素的变化
6         else
7             dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i])
8         ;
9     }
10 }

```

主要代码

先遍历背包，再遍历物品

```

1 // weight数组的大小 就是物品个数
2 for(int j = 0; j <= bagWeight; j++) { // 遍历背包容量
3     for(int i = 1; i < weight.size(); i++) { // 遍历物品
4         if (j < weight[i])
5             dp[i][j] = dp[i - 1][j];
6         else
7             dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i])
8         ;
9     }
10 }

```

本文采用先遍历物品，然后遍历背包重量。

(三) 优化一维数组

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

1. 确定 dp 数组定义

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 数组递推公式

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

$dp[j - weight[i]] + value[i]$ 表示容量为 j - 物品 i 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 $dp[j - weight[i]] + value[i]$ ，指定是取最大的，毕竟是求最大价值，所以递归公式为：

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

3. 数组初始化

$dp[0]$ 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。递归公式: $dp[j] = \max(dp[j], dp[j - \text{weight}[i]] + \text{value}[i])$; dp 数组在推导的时候一定是取价值最大的数, 如果给的价值都是正整数那么非 0 下标都初始化为 0 就可以, 如果给的价值有负数, 那么非 0 下标要初始化为负无穷。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。

4. 一维数组遍历顺序

主要代码

先遍历物品, 然后遍历背包重量

```
1  for(int i = 0; i < weight.size(); i++) { // 遍历物品
2      for(int j = bagWeight; j >= weight[i]; j--) { // 遍历背包容量
3          dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
4      }
5  }
```

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。**倒叙遍历是为了保证物品 i 只被放入一次!。但如果一旦正序遍历了, 那么物品 0 就会被重复加入多次!** 举例: 物品 0 的重量 $\text{weight}[0] = 1$, 价值 $\text{value}[0] = 15$ 如果正序遍历 $dp[1] = dp[1 - \text{weight}[0]] + \text{value}[0] = 15$ $dp[2] = dp[2 - \text{weight}[0]] + \text{value}[0] = 30$ 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。

二维 dp 数组历的时候不用倒叙: 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。

不可以先遍历背包容量嵌套遍历物品: 因为一维 dp 的写法, 背包容量一定是要倒序遍历, 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, 背包里只放入了一个物品。

(四) 实验结果

功能实现代码及对比代码 [请点击这里](#)

```
请输入物品数目: 4
您输入的物品数目为 4
请输入背包最大重量: 10
您输入的背包最大重量为 10
请依次输入第 1 件物品的名称、重量、价值:
camp 3 200
请依次输入第 2 件物品的名称、重量、价值:
bag 2 150
请依次输入第 3 件物品的名称、重量、价值:
clothes 1 80
请依次输入第 4 件物品的名称、重量、价值:
food 5 160
您输入的信息如下
名称      重量      价值
camp      3         200
bag       2         150
clothes   1         80
food      5         160
最大价值为 510
您的最优挑选方案如下:
共挑选 3 件物品
它们分别是 camp bag food
二维背包算法时间为 0.0177ms
可优化为一维背包算法
最大价值为 510
您的最优挑选方案如下:
共挑选 3 件物品
它们分别是 camp bag food
一维优化算法时间为 0.0127ms
打印二维数组
0 0 0 0 0 0 0 0 0 0
0 0 0 200 200 200 200 200 200 200
0 0 150 200 200 350 350 350 350 350
0 80 150 230 280 350 430 430 430 430
0 80 150 230 280 350 430 430 430 510
打印一维数组
0 80 150 230 280 350 430 430 430 440 510
C:\Users\HONOR\source\repos\bag\Debug\bag.exe (进程 21020)已退出, 代码为 0。
请在调试停止时自动关闭控制台。请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

图 3: 实验结果 1

```
Microsoft Visual Studio 调试控制台
请输入物品数目: 3
您输入的物品数目为 3
请输入背包最大重量: 4
您输入的背包最大重量为 4
请依次输入第 1 件物品的名称、重量、价值:
t1 1 15
请依次输入第 2 件物品的名称、重量、价值:
t2 3 20
请依次输入第 3 件物品的名称、重量、价值:
t3 4 30
您输入的信息如下
名称      重量      价值
t1         1         15
t2         3         20
t3         4         30
最大价值为 35
您的最优挑选方案如下:
共挑选 2 件物品
它们分别是 t1 t2
二维背包算法时间为 0.0014ms
可优化为一维背包算法
```

图 4: 实验结果 2.1

```
可优化为一维背包算法
最大价值为35
您的最优挑选方案如下：
共挑选 2 件物品
它们分别是 t1 t2
一维优化算法时间为 0.0006ms

打印二维数组
0 0 0 0
0 15 15 15 15
0 15 15 20 35
0 15 15 20 35

打印一维数组
0 15 15 20 35
C:\Users\HONOR\source\repos\bag\Debug\bag.exe (进程 30864)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

图 5: 实验结果 2.2

(五) 实验分析

```
Microsoft Visual Studio 调试控制台

10 10
n = 10 m = 10
二维背包算法时间为 0.0369ms
一维背包算法时间为 0.0083ms

20 20
n = 20 m = 20
二维背包算法时间为 0.0372ms
一维背包算法时间为 0.0015ms

50 50
n = 50 m = 50
二维背包算法时间为 0.0872ms
一维背包算法时间为 0.0063ms

100 100
n = 100 m = 100
二维背包算法时间为 0.2146ms
一维背包算法时间为 0.029ms

500 500
n = 500 m = 500
二维背包算法时间为 2.8462ms
一维背包算法时间为 1.0849ms

1000 1000
n = 1000 m = 1000
二维背包算法时间为 8.4444ms
一维背包算法时间为 4.479ms
```

图 6: 实验对比

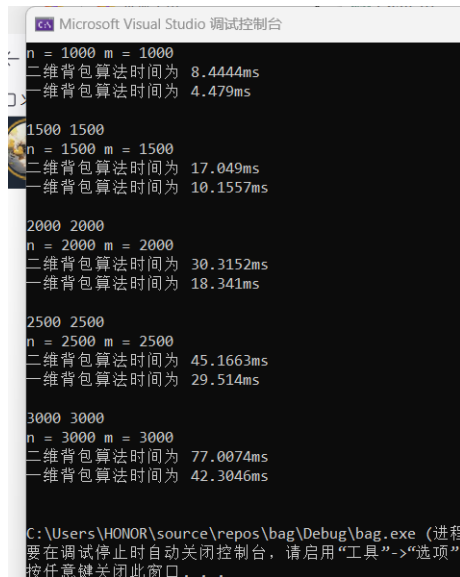


图 7: 实验对比

时间复杂度为 $O(nW)$, 其中 n 是物品数量, W 是背包的最大容量。二维算法空间复杂度 $o(nW)$, 一维算法空间复杂度 $o(W)$ 。一维数组可以节省空间, 但需要注意转移方程中数组的更新顺序, 以免出现重复计算的情况。二维数组则比较直观, 易于理解和实现, 但空间复杂度较高。

动态规划算法是解决生活中的背包问题的有效方法。通过将问题划分为多个子问题, 在每个子问题中选择最优解, 最终得到全局最优解。该算法能够有效地处理大规模的数据集, 同时具有较高的时间和空间效率。

(六) 引申

给定一个价值 $amount$ 和一些面值, 假设每个面值的硬币数都是无限的, 问我们最少能用几个硬币组成给定的价值。如果我们将面值看作是物品, 面值金额看成是物品的重量, 每件物品的价值均为 1, 这样此题就是一个恰好装满的完全背包问题了。不过这里不是求最多装入多少物品而是求最少, 将状态转移方程中的 \max 改成 \min 即可, 又由于是恰好装满, 所以除了 $dp[0]$, 其他都应初始化为 INT_MAX 。

主要代码

解决代码如下:

```

1  int coinChange(vector<int>& coins, int amount) {
2      vector<int> dp(amount + 1, INT_MAX);
3      dp[0] = 0;
4
5      for(int i = 1; i <= coins.size(); i++)
6          for(int j = coins[i-1]; j <= amount; j++){
7              // 下行代码会在 1+INT_MAX 时溢出
8              // dp[j] = min(dp[j], 1 + dp[j - coins[i-1]]);
9              if(dp[j] - 1 > dp[j - coins[i-1]])
10                 dp[j] = 1 + dp[j - coins[i-1]];
11          }
12      return dp[amount] == INT_MAX ? -1 : dp[amount];
  
```