

背包问题九讲2.0

崔添翼 (Tianyi Cui, a.k.a. dd_engi)

September 15, 2011

Contents

1	01背包问题	2
1.1	题目	2
1.2	基本思路	2
1.3	优化空间复杂度	2
1.4	初始化的细节问题	3
1.5	一个常数优化	3
1.6	小结	3
2	完全背包问题	4
2.1	题目	4
2.2	基本思路	4
2.3	一个简单有效的优化	4
2.4	转化为01背包问题求解	4
2.5	$O(VN)$ 的算法	5
2.6	总结	5
3	多重背包问题	6
3.1	题目	6
3.2	基本算法	6
3.3	转化为01背包问题	6
3.4	$O(VN)$ 的算法	7
3.5	小结	7
4	混合三种背包问题	7
4.1	问题	7
4.2	01背包与完全背包的混合	7
4.3	再加上多重背包	7
4.4	小结	8
5	二维费用的背包问题	8
5.1	问题	8
5.2	算法	8
5.3	物品总个数的限制	8
5.4	复整数域上的背包问题	9
5.5	小结	9

1 01背包问题

1.1 题目

有 N 件物品和一个容量为 V 的背包。放入第 i 件物品耗费的空间是 C_i ，得到的价值是 W_i 。求解将哪些物品装入背包可使价值总和最大。

1.2 基本思路

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

用子问题定义状态：即 $F[i, v]$ 表示前 i 件物品恰放入一个容量为 v 的背包可以获得的**最大价值**。则其状态转移方程便是：

$$F[i, v] = \max\{F[i-1, v], F[i-1, v-C_i] + W_i\}$$

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下：“将前 i 件物品放入容量为 v 的背包中”这个子问题，若只考虑第 i 件物品的策略（放或不放），那么就可以转化为一个只和前 $i-1$ 件物品相关的问题。如果不放第 i 件物品，那么问题就转化为“前 $i-1$ 件物品放入容量为 v 的背包中”，价值为 $F[i-1, v]$ ；如果放第 i 件物品，那么问题就转化为“前 $i-1$ 件物品放入剩下的容量为 $v-C_i$ 的背包中”，此时能获得的最大价值就是 $F[i-1, v-C_i]$ 再加上通过放入第 i 件物品获得的价值 W_i 。

伪代码如下：

```
F[0..V] = 0
for i = 1 to N
  for v = Ci to V
    F[i, v] = max{F[i-1, v], F[i-1, v-Ci] + Wi}
```

1.3 优化空间复杂度

以上方法的时间和空间复杂度均为 $O(VN)$ ，其中时间复杂度应该已经不能再优化了，但空间复杂度却可以优化到 $O(V)$ 。

先考虑上面讲的基本思路如何实现，肯定是有一个主循环 $i = 1..N$ ，每次算出来二维数组 $F[i, 0..V]$ 的所有值。那么，如果只用一个数组 $F[0..V]$ ，能不能保证第 i 次循环结束后 $F[v]$ 中表示的就是我们定义的状态 $F[i, v]$ 呢？ $F[i, v]$ 是由 $F[i-1, v]$ 和 $F[i-1, v-C_i]$ 两个子问题递推而来，能否保证在推 $F[i, v]$ 时（也即在第 i 次主循环中推 $F[v]$ 时）能够取用 $F[i-1, v]$ 和 $F[i-1, v-C_i]$ 的值呢？事实上，这要求在每次主循环中我们以 $v = V..0$ 的递减顺序计算 $F[v]$ ，这样才能保证推 $F[v]$ 时 $F[v-C_i]$ 保存的是状态 $F[i-1, v-C_i]$ 的值。伪代码如下：

```
F[0..V] = 0
for i = 1 to N
  for v = V to Ci
    F[v] = max{F[v], F[v-Ci] + Wi}
```

其中的 $F[v] = \max\{F[v], F[v-C_i] + W_i\}$ 一句，恰就对应于我们原来的转移方程，因为现在的 $F[v-C_i]$ 就相当于原来的 $F[i-1, v-C_i]$ 。如果将 v 的循环顺序

从上面的逆序改成顺序的话，那么则成了 $F[i, v]$ 由 $F[i, v - C_i]$ 推导得到，与本题意不符。

事实上，使用一维数组解01背包的程序在后面会被多次用到，所以这里抽象出一个处理一件01背包中的物品过程，以后的代码中直接调用不加说明。

```
def ZeroOnePack(F, C, W)
    for v = V to C
        F[v] = max(F[v], f[v - C] + W)
```

有了这个过程以后，01背包问题的伪代码就可以这样写：

```
for i = 1 to N
    ZeroOnePack(F, Ci, Wi)
```

1.4 初始化的细节问题

我们看到的求最优解的背包问题题目中，事实上有两种不太相同的问法。有的题目要求“恰好装满背包”时的最优解，有的题目则并没有要求必须把背包装满。一种区别这两种问法的实现方法是在初始化的时候有所不同。

如果是第一种问法，要求恰好装满背包，那么在初始化时除了 $F[0]$ 为0，其它 $F[1..V]$ 均设为 $-\infty$ ，这样就可以保证最终得到的 $F[V]$ 是一种恰好装满背包的最优解。

如果并没有要求必须把背包装满，而是只希望价格尽量大，初始化时应该将 $F[0..V]$ 全部设为0。

这是为什么呢？可以这样理解：初始化的 F 数组事实上就是在没有任何物品可以放入背包时的合法状态。如果要求背包恰好装满，那么此时只有容量为0的背包可以在什么也不装且价值为0的情况下被“恰好装满”，其它容量的背包均没有合法的解，属于未定义的状态，应该被赋值为 $-\infty$ 了。如果背包并非必须被装满，那么任何容量的背包都有一个合法解“什么都不装”，这个解的价值为0，所以初始时状态的值也就全部为0了。

这个小技巧完全可以推广到其它类型的背包问题，后面也就不再对进行状态转移之前的初始化进行讲解。

1.5 一个常数优化

上面伪代码中的

```
for i = 1 to N
    for v = V to Ci
```

中第二重循环的下限可以改进。它可以被优化为

```
for i = 1 to N
    for v = V to max(V -  $\sum_i^N W_i$ , Ci)
```

这个优化之所以成立的原因请读者自己思考。（提示：使用二维的转移方程思考较易。）

1.6 小结

01背包问题是最基本的背包问题，它包含了背包问题中设计状态、方程的最基本思想。另外，别的类型的背包问题往往也可以转换成01背包问题求解。故一定要仔细体会上面基本思路的得出方法，状态转移方程的意义，以及空间复杂度怎样被优化。

2 完全背包问题

2.1 题目

有 N 种物品和一个容量为 V 的背包，每种物品都有无限件可用。放入第 i 种物品的耗费的空间是 C_i ，得到的价值是 W_i 。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。

2.2 基本思路

这个问题非常类似于01背包问题，所不同的是每种物品有无限件。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是有取0件、取1件、取2件……直至取 $\lfloor V/C_i \rfloor$ 件等很多种。

如果仍然按照解01背包时的思路，令 $F[i, v]$ 表示前 i 种物品恰放入一个容量为 v 的背包的最大权值。仍然可以按照每种物品不同的策略写出状态转移方程，像这样：

$$F[i, v] = \max\{F[i-1, v - kC_i] + kW_i \mid 0 \leq kC_i \leq v\}$$

这跟01背包问题一样有 $O(VN)$ 个状态需要求解，但求解每个状态的时间已经不是常数了，求解状态 $F[i, v]$ 的时间是 $O(\frac{v}{C_i})$ ，总的复杂度可以认为是 $O(NV \sum \frac{V}{C_i})$ ，是比较大的。

将01背包问题的基本思路加以改进，得到了这样一个清晰的方法。这说明01背包问题的方程的确很重要，可以推及其它类型的背包问题。但我们还是要试图改进这个复杂度。

2.3 一个简单有效的优化

完全背包问题有一个很简单有效的优化，是这样的：若两件物品 i 、 j 满足 $C_i \leq C_j$ 且 $W_i \geq W_j$ ，则可以将物品 j 直接去掉，不用考虑。

这个优化的正确性是显然的：任何情况下都可将价值小耗费高的 j 换成物美价廉的 i ，得到的方案至少不会更差。对于随机生成的数据，这个方法往往会大大减少物品的件数，从而加快速度。然而这个并不能改善最坏情况的复杂度，因为有可能特别设计的数据可以一件物品也去不掉。

这个优化可以简单的 $O(N^2)$ 地实现，一般都可以承受。另外，针对背包问题而言，比较不错的一种方法是：首先将费用大于 V 的物品去掉，然后使用类似计数排序的做法，计算出费用相同的物品中价值最高的是哪个，可以 $O(V + N)$ 地完成这个优化。这个不太重要的过程就不给出伪代码了，希望你能独立思考写出伪代码或程序。

2.4 转化为01背包问题求解

01背包问题是最基本的背包问题，我们可以考虑把完全背包问题转化为01背包问题来解。

最简单的想法是，考虑到第 i 种物品最多选 $\lfloor \frac{V}{C_i} \rfloor$ 件，于是可以把第 i 种物品转化为 $\lfloor \frac{V}{C_i} \rfloor$ 件费用及价值均不变的物品，然后求解这个01背包问题。这样的做法完全没有改进时间复杂度，但这种方法也指明了将完全背包问题转化为01背包问题的思路：将一种物品拆成多件只能选0件或1件的01背包中的物品。

更高效的转化方法是：把第 i 种物品拆成费用为 $C_i 2^k$ 、价值为 $W_i 2^k$ 的若干件物品，其中 k 取遍满足 $C_i 2^k \leq V$ 的非负整数。

这是二进制的思想。因为，不管最优策略选几件第 i 种物品，其件数写成二进制后，总可以表示成若干个 2^k 件物品的和。这样一来就把每种物品拆成 $O(\log \frac{V}{C_i})$ 件物品，是一个很大的改进。

2.5 $O(VN)$ 的算法

这个算法使用一维数组，先看伪代码：

```

F[0..V] = 0
for i = 1 to N
    for v = Ci to V
        F[v] = max(F[v], F[v - Ci] + Wi)

```

你会发现，这个伪代码与01背包问题的伪代码只有 v 的循环次序不同而已。

为什么这个算法就可行呢？首先想想为什么01背包中要按照 v 递减的次序来循环。让 v 递减是为了保证第 i 次循环中的状态 $F[i, v]$ 是由状态 $F[i - 1, v - C_i]$ 递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第 i 件物品”这件策略时，依据的是一个绝无已经选入第 i 件物品的子结果 $F[i - 1, v - C_i]$ 。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第 i 种物品”这种策略时，却正需要一个可能已选入第 i 种物品的子结果 $F[i, v - C_i]$ ，所以就可以并且必须采用 v 递增的顺序循环。这就是这个简单的程序为何成立的道理。

值得一提的是，上面的伪代码中两层for循环的次序可以颠倒。这个结论有可能会带来算法时间常数上的优化。

这个算法也可以由另外的思路得出。例如，将基本思路中求解 $F[i, v - C_i]$ 的状态转移方程显式地写出来，代入原方程中，会发现该方程可以等价地变形成这种形式：

$$F[i, v] = \max(F[i - 1, v], F[i, v - C_i] + W_i)$$

将这个方程用一维数组实现，便得到了上面的伪代码。

最后抽象出处理一件完全背包类物品的过程伪代码：

```

def CompletePack(F, C, W)
    for v = C to V
        F[v] = max{F[v], f[v - C] + W}

```

2.6 总结

完全背包问题也是一个相当基础的背包问题，它有两个状态转移方程。希望你能够对这两个状态转移方程都仔细地体会，不仅记住，也要弄明白它们是怎么得出来的，最好能够自己想一种得到这些方程的方法。

事实上，对每一道动态规划题目都思考其方程的意义以及如何得来，是加深对动态规划的理解、提高动态规划功力的好方法。

3 多重背包问题

3.1 题目

有 N 种物品和一个容量为 V 的背包。第 i 种物品最多有 M_i 件可用，每件耗费的空间是 C_i ，价值是 W_i 。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。

3.2 基本算法

这题目和完全背包问题很类似。基本的方程只需将完全背包问题的方程略微一改即可。

因为对于第 i 种物品有 $M_i + 1$ 种策略：取0件，取1件……取 M_i 件。令 $F[i, v]$ 表示前 i 种物品恰放入一个容量为 v 的背包的最大价值，则有状态转移方程：

$$F[i, v] = \max\{F[i-1, v-k \cdot C_i] + k \cdot W_i \mid 0 \leq k \leq M_i\}$$

复杂度是 $O(V \sum M_i)$ 。

3.3 转化为01背包问题

另一种好想好写的基本方法是转化为01背包求解：把第 i 种物品换成 M_i 件01背包中的物品，则得到了物品数为 $\sum M_i$ 的01背包问题。直接求解之，复杂度仍然是 $O(V \sum M_i)$ 。

但是我们期望将它转化为01背包问题之后，能够像完全背包一样降低复杂度。

仍然考虑二进制的思想，我们考虑把第 i 种物品换成若干件物品，使得原问题中第 i 种物品可取的每种策略——取 $0 \dots M_i$ 件——均能等价于取若干件代换以后的物品。另外，取超过 M_i 件的策略必不能出现。

方法是：将第 i 种物品分成若干件01背包中的物品，其中每件物品有一个系数。这件物品的费用和价值均是原来的费用和价值乘以这个系数。令这些系数分别为 $1, 2, 2^2 \dots 2^{k-1}, M_i - 2^k + 1$ ，且 k 是满足 $M_i - 2^k + 1 > 0$ 的最大整数。例如，如果 M_i 为13，则相应的 $k = 3$ ，这种最多取13件的物品应被分成系数分别为1, 2, 4, 6的四件物品。

分成的这几件物品的系数和为 M_i ，表明不可能取多于 M_i 件的第 i 种物品。另外这种方法也能保证对于 $0 \dots M_i$ 间的每一个整数，均可以用若干个系数的和表示。这里算法正确性的证明可以分 $0 \dots 2^{k-1}$ 和 $2^k \dots M_i$ 两段来分别讨论得出，希望读者自己思考尝试一下。

这样就将第 i 种物品分成了 $O(\log M_i)$ 种物品，将原问题转化为了复杂度为 $O(V \sum \log M_i)$ 的01背包问题，是很大的改进。

下面给出 $O(\log M)$ 时间处理一件多重背包中物品的过程：

```
def MultiplePack(F, C, W, M)
    if  $C \cdot M \geq V$ 
        CompletePack(F, C, W)
    return
     $k := 1$ 
    while  $k < M$ 
        ZeroOnePack( $kC, kW$ )
         $M := M - k$ 
         $k := 2k$ 
```

ZeroOnePack($C \cdot M, W \cdot M$)

希望你仔细体会这个伪代码，如果不太理解的话，不妨翻译成程序代码以后，单步执行几次，或者头脑加纸笔模拟一下，以加深理解。

3.4 $O(VN)$ 的算法

多重背包问题同样有 $O(VN)$ 复杂度的算法。这个算法基于基本算法的状态转移方程，但应用单调队列的方法使每个状态的值可以以均摊 $O(1)$ 的时间求解。我最初了解到这个方法是在楼天成的“男人八题”幻灯片上。（TODO：是否在此插入单调队列的讲解呢？）

3.5 小结

在这一讲中，我们看到了将一个算法的复杂度由 $O(V \sum M_i)$ 改进到 $O(V \sum \log M_i)$ 的过程，还知道了存在复杂度为 $O(VN)$ 的算法。

希望你特别注意“拆分物品”的思想和方法，自己证明一下它的正确性，并将完整的程序代码写出来。

4 混合三种背包问题

4.1 问题

如果将前面的三种背包问题混合起来。也就是说，有的物品只可以取一次（01背包），有的物品可以取无限次（完全背包），有的物品可以取的次数有一个上限（多重背包）。应该怎么求解呢？

4.2 01背包与完全背包的混合

考虑到01背包和完全背包中给出的伪代码只有一处不同，故如果只有两类物品：一类物品只能取一次，另一类物品可以取无限次，那么只需在对每个物品应用转移方程时，根据物品的类别选用顺序或逆序的循环即可，复杂度是 $O(VN)$ 。伪代码如下：

```
for  $i = 1$  to  $N$ 
  if 第 $i$ 件物品属于01背包
    for  $v = V$  to  $C_i$ 
       $F[v] = \max(F[v], F[v - C_i] + W_i)$ 
  else if 第 $i$ 件物品属于完全背包
    for  $v = C_i$  to  $V$ 
       $F[v] = \max(F[v], F[v - C_i] + W_i)$ 
```

4.3 再加上多重背包

如果再加上最多可以取有限次的多重背包式的物品，那么利用单调队列，也可以给出均摊 $O(VN)$ 的解法。

但如果不考虑单调队列算法的话，用将每个这类物品分成 $O(\log M_i)$ 个01背包的物品的的方法也已经很优了。

当然，最清晰的写法是调用我们前面给出的三个过程。

```

for  $i = 1$  to  $N$ 
  if 第 $i$ 件物品属于01背包
    ZeroOnePack( $F, C_i, W_i$ )
  else if 第 $i$ 件物品属于完全背包
    CompletePack( $F, C_i, W_i$ )
  else if 第 $i$ 件物品属于多重背包
    MultiplePack( $F, C_i, W_i, N_i$ )

```

在最初写出这三个过程的时候，可能完全没有想到它们会在这里混合应用。我想这体现了编程中抽象的威力。如果你一直就是以这种“抽象出过程”的方式写每一类背包问题的，也非常清楚它们的实现中细微的不同，那么在遇到混合三种背包问题的题目时，一定能很快想到上面简洁的解法，对吗？

4.4 小结

有人说，困难的题目都是由简单的题目叠加而来的。这句话是否公理暂且存之不论，但它在本讲中已经得到了充分的体现。本来01背包、完全背包、多重背包都不是什么难题，但将它们简单地组合起来以后就得到了这样一道一定能吓倒不少人的题目。但只要基础扎实，领会三种基本背包问题的思想，就可以做到把困难的题目拆分成简单的题目来解决。

5 二维费用的背包问题

5.1 问题

二维费用的背包问题是指：对于每件物品，具有两种不同的空间耗费，选择这件物品必须同时付出这两种代价。对于每种代价都有一个可付出的最大值（背包容量）。问怎样选择物品可以得到最大的价值。

设这两种代价分别为代价一和代价二，第 i 件物品所需的两种代价分别为 C_i 和 D_i 。两种代价可付出的最大值（两种背包容量）分别为 V 和 U 。物品的价值为 W_i 。

5.2 算法

费用加了一维，只需状态也加一维即可。设 $F[i, v, u]$ 表示前 i 件物品付出两种代价分别为 v 和 u 时可获得的最大价值。状态转移方程就是：

$$F[i, v, u] = \max\{F[i-1, v, u], F[i-1, v-C_i, u-D_i] + W_i\}$$

如前述优化空间复杂度的方法，可以只使用二维的数组：当每件物品只可以取一次时变量 v 和 u 采用逆序的循环，当物品有如完全背包问题时采用顺序的循环，当物品有如多重背包问题时拆分物品。

这里就不再给出伪代码了，相信有了前面的基础，读者应该能够自己实现出这个问题的程序。

5.3 物品总个数的限制

有时，“二维费用”的条件是以这样一种隐含的方式给出的：最多只能取 U 件物品。这事实上相当于每件物品多了一种“件数”的费用，每个物品的件数费用均为1，可以付出的最大件数费用为 U 。换句话说，设 $F[v, u]$ 表示付

出费用 v 、最多选 u 件时可得到的最大价值，则根据物品的类型（01、完全、多重）用不同的方法循环更新，最后在 $f[0 \dots V, 0 \dots U]$ 范围内寻找答案。

5.4 复整数域上的背包问题

另一种看待二维背包问题的思路是：将它看待成复整数域上的背包问题。也就是说，背包的容量以及每件物品的费用都是一个复整数。而常见的一维背包问题则是自然数域上的背包问题。所以说，一维背包的种种思想方法，往往可以应用于二位背包问题的求解中，因为只是数域扩大了而已。

作为这种思想的练习，你可以尝试将后文中提到的“子集和问题”扩展到二维，并试图用同样的复杂度解决。

5.5 小结

当发现由熟悉的动态规划题目变形得来的题目时，在原来的状态中加一维以满足新的限制是一种比较通用的方法。希望你能从本讲中初步体会到这种方法。