背包问题九讲 2.0 alpha1

崔添翼 (Tianyi Cui, a.k.a. dd_engi)

September 15, 2011

本文题为《背包问题九讲》，从属于《动态规划的思考艺术》系列。

这系列文章的第一版于2007年下半年使用EmacsMuse制作，以HTML格式发布到网上，转载众多，有一定影响力。

2011年9月，本系列文章由原作者用LaTeX重新制作并全面修订，您现在看到的是2.0 alpha1版本，修订历史及最新版本请访问 https://github.com/tianyicui/pack 查阅。

Contents

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：

$F[0,0..V]=0$
for i = 1 to N
  for v = C_i to V
    $F[i,v]$ = max$\{F[i-1,v],F[i-1,v-C_i]+W_i\}$

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

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,C_i,W_i)

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-∞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 C_i
```

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 - Σ_i^N W_i, C_i)
```

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：

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

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$O(NV\Sigma\frac{V}{C_i})$，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是很重要，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$C_i\le C_j$且$W_i\ge W_j$，则将可以将物品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$\left\lfloor\frac{V}{C_i}\right\rfloor$件，于是可以把第i种物品转化为$\left\lfloor\frac{V}{C_i}\right\rfloor$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_i2^k、价值为W_i2^k的若干件物品，其中k取遍满足$C_i2^k\le V$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 = C_i to V
    F[v] = max(F[v], F[v − C_i] + W_i)
```

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 小结

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：

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

复杂度是$O(V\Sigma M_i)$。

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ΣM_i的01背包问题。直接求解之，复杂度仍然是$O(V\Sigma M_i)$。

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$0\ldots M_i$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$1, 2, 2^2 \ldots 2^{k-1}, M_i - 2^k + 1$，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$0\ldots M_i$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$0\ldots 2^{k-1}$和$2^k \ldots M_i$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$O(V\Sigma\log \mathrm{M_i})$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 · M ≥ V
    CompletePack(F,C,W)
    return
  k := 1
  while k < M
    ZeroOnePack(kC,kW)
    M := M − k
    k := 2k
  ZeroOnePack(C · M,W · 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$O(V\Sigma M_i)$改进到$O(V\Sigma\log \mathrm{M_i})$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1、2、3中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(logM_i)个01背包的物品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：

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

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$f[0\ldots V,0\ldots U]$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。

6 分组的背包问题

6.1 问题

有N件物品和一个容量为V的背包。第i件物品的费用是C_i，价值是W_i。这些物品被划分为K组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

6.2 算法

这个问题变成了每组物品有若干种策略：是选择本组的某一件，还是一件都不选。也就是说设$F[k,v]$表示前k组物品花费费用v能取得的最大权值，则有：

$$F[k,v] = \max\{F[k-1,v], F[k-1,v-C_i] + W_i \mid \text{item } i \in \text{group } k\}$$

使用一维数组的伪代码如下：

```
for k = 1 to K
  for v = V to 0
    for item i in group k
      F[v] = max{F[v], F[v - C_i] + W_i}
```

这里三层循环的顺序保证了每一组内的物品最多只有一个会被添加到背包中。

另外，显然可以对每组内的物品应用2.3中的优化。

6.3 小结

分组的背包问题将彼此互斥的若干物品称为一个组，这建立了一个很好的模型。不少背包问题的变形都可以转化为分组的背包问题（例如7），由分组的背包问题进一步可定义“泛化物品”的概念，十分有利于解题。

7 有依赖的背包问题

7.1 简化的问题

这种背包问题的物品间存在某种“依赖”的关系。也就是说，物品i依赖于物品j，表示若选物品i，则必须选物品j。为了简化起见，我们先设没有某个物品既依赖于别的物品，又被别的物品所依赖；另外，没有某件物品同时依赖多件物品。

7.2 算法

这个问题由NOIP2006题目中金明的预算方案一题扩展而来。遵从该题的提法，将不依赖于别的物品的物品称为“主件”，依赖于某主件的物品称为“附件”。由这个问题的简化条件可知所有的物品由若干主件和依赖于每个主件的一个附件集合组成。

按照背包问题的一般思路，仅考虑一个主件和它的附件集合。可是，可用的策略非常多，包括：一个也不选，仅选择主件，选择主件后再选择一个附件，选择主件后再选择两个附件……无法用状态转移方程来表示如此多的策略。事实上，设有n个附件，则策略有2^n+1个，为指数级。

考虑到所有这些策略都是互斥的（也就是说，你只能选择一种策略），所以一个主件和它的附件集合实际上对应于6中的一个物品组，每个选择了主件又选择了若干个附件的策略对应于这个物品组中的一个物品，其费用和价值都是这个策略中的物品的值的和。但仅仅是这一步转化并不能给出一个好的算法，因为物品组中的物品还是像原问题的策略一样多。

再考虑对每组内的物品应用2.3中的优化。我们可以想到，对于第k个物品组中的物品，所有费用相同的物品只留一个价值最大的，不影响结果。所以，可以对主件k的“附件集合”先进行一次01背包，得到费用依次为$0 \ldots V - C_k$所有这些值时相应的最大价值$F_k[0 \ldots V - C_k]$。那么，这个主件及它的附件集合相当于$V - C_k + 1$个物品的物品组，其中费用为v的物品的价值为$F_k[v - C_k] + W_k$，v的取值范围是$C_k \leq v \leq V$。

也就是说，原来指数级的策略中，有很多策略都是冗余的，通过一次01背包后，将主件k及其附件转化为$V - C_k + 1$个物品的物品组，就可以直接应用6的算法解决问题了。

7.3 较一般的问题

更一般的问题是：依赖关系以图论中“森林”[1]的形式给出。也就是说，主件的附件仍然可以具有自己的附件集合。限制只是每个物品最多只依赖于一个物品（只有一个主件）且不出现循环依赖。

解决这个问题仍然可以用将每个主件及其附件集合转化为物品组的方式。唯一不同的是，由于附件可能还有附件，就不能将每个附件都看作一个一般的01背包中的物品了。若这个附件也有附件集合，则它必定要被先转化为物品组，然后用分组的背包问题解出主件及其附件集合所对应的附件组中各个费用的附件所对应的价值。

事实上，这是一种树形动态规划，其特点是，在用动态规划求每个父节点的属性之前，需要对它的各个儿子的属性进行一次动态规划式的求值。这已经触及到了“泛化物品”的思想。看完8后，你会发现这个“依赖关系树”每一个子树都等价于一件泛化物品，求某节点为根的子树对应的泛化物品相当于求其所有儿子的对应的泛化物品之和。

7.4 小结

NOIP2006的那道背包问题我做得很失败，写了上百行的代码，却一分未得。后来我通过思考发现通过引入“物品组”和“依赖”的概念可以加深对这题的理解，还可以解决它的推广问题。用物品组的思想考虑那题中极其特殊的依赖关系：物品不能既作主件又作附件，每个主件最多有两个附件，可以发现一个主件和它的两个附件等价于一个由四个物品组成的物品组，这便揭示了问题的某种本质。

后来，我在《背包问题九讲》第一版中总结此事时说：“失败不是什么丢人的事情，从失败中全无收获才是。”NOIP2007我得了满分。

8 泛化物品

8.1 定义

考虑这样一种物品，它并没有固定的费用和价值，而是它的价值随着你分配给它的费用而变化。这就是泛化物品的概念。

[1] 即多叉树的集合

更严格的定义之。在背包容量为V的背包问题中，泛化物品是一个定义域为$0\dots V$中的整数的函数h，当分配给它的费用为v时，能得到的价值就是$h(v)$。

这个定义有一点点抽象，另一种理解是一个泛化物品就是一个数组$h[0\dots V]$，给它费用v，可得到价值$h[v]$。

一个费用为c价值为w的物品，如果它是01背包中的物品，那么把它看成泛化物品，它就是除了$h(c)=w$外，其它函数值都为0的一个函数。如果它是完全背包中的物品，那么它可以看成这样一个函数，仅当v被c整除时有$h(v)=w\cdot\frac{v}{c}$，其它函数值均为0。如果它是多重背包中重复次数最多为m的物品，那么它对应的泛化物品的函数有$h(v)=w\cdot\frac{v}{c}$仅当v被c整除且$\frac{v}{c}\le n$，其它情况函数值均为0。

一个物品组可以看作一个泛化物品h。对于一个$0\dots V$中的v，若物品组中不存在费用为v的物品，则$h(v)=0$，否则$h(v)$取值为所有费用为v的物品的最大价值。6中每个主件及其附件集合等价于一个物品组，自然也可看作一个泛化物品。

8.2 泛化物品的和

如果给定了两个泛化物品h和l，要用一定的费用从这两个泛化物品中得到最大的价值，这个问题怎么求呢？事实上，对于一个给定的费用v，只需枚举将这个费用如何分配给两个泛化物品就可以了。同样的，对于$0\dots V$中的每一个整数v，可以求得费用v分配到h和l中的最大价值$f(v)$。也即

$$f(v)=\max\{h(k)+l(v-k)\mid 0\le k\le v\}$$

可以看到，这里的f是一个由泛化物品h和l决定的定义域为0...V的函数，也就是说，f是一个由泛化物品h和l决定的泛化物品。

我们将f定义为泛化物品h和l的和：h、l都是泛化物品，若函数f满足以上关系式，则称f是h与l的和。泛化物品和运算的时间复杂度取决于背包的容量，是$O(V^2)$。

由泛化物品的定义可知：在一个背包问题中，若将两个泛化物品代以它们的和，不影响问题的答案。事实上，对于其中的物品都是泛化物品的背包问题，求它的答案的过程也就是求所有这些泛化物品之和的过程。若问题的和为s，则答案就是$s(0\dots V)$中的最大值。

8.3 背包问题的泛化物品

一个背包问题中，可能会给出很多条件，包括每种物品的费用、价值等属性，物品之间的分组、依赖等关系等。但肯定能将问题对应于某个泛化物品。也就是说，给定了所有条件以后，就可以对每个非负整数v求得：若背包容量为v，将物品装入背包可得到的最大价值是多少，这可以认为是定义在非负整数集上的一件泛化物品。这个泛化物品——或者说问题所对应的一个定义域为非负整数的函数——包含了关于问题本身的高度浓缩的信息。一般而言，求得这个泛化物品的一个子定义域（例如$0\dots V$）的值之后，就可以根据这个函数的取值得到背包问题的最终答案。

综上所述，一般而言，求解背包问题，即求解这个问题所对应的一个函数，即该问题的泛化物品。而求解某个泛化物品的一种常用方法就是将它表示为若干泛化物品的和然后求之。

8.4 小结

本讲是我在学习函数式编程的Scheme语言时，用函数编程的眼光审视各类背包问题得出的理论。

我想说：“思考”是一个程序员最重要的品质。简单的问题，深入思考以后，也能发现更多。

9 背包问题问法的变化

以上涉及的各种背包问题都是要求在背包容量（费用）的限制下求可以取到的最大价值，但背包问题还有很多种灵活的问法，在这里值得提一下。但是我认为，只要深入理解了求背包问题最大价值的方法，即使问法变化了，也是不难想出算法的。

例如，求解最多可以放多少件物品或者最多可以装满多少背包的空间。这都可以根据具体问题利用前面的方程求出所有状态的值（F数组）之后得到。

还有，如果要求的是“总价值最小”“总件数最小”，只需简单的将上面的状态转移方程中的max改成min即可。

下面说一些变化更大的问法。

9.1 输出方案

一般而言，背包问题是要求一个最优值，如果要求输出这个最优值的方案，可以参照一般动态规划问题输出方案的方法：记录下每个状态的最优值是由状态转移方程的哪一项推出来的，换句话说，记录下它是由哪一个策略推出来的。便可根据这条策略找到上一个状态，从上一个状态接着向前推即可。

还是以01背包为例，方程为$F[i,v]=\max\{F[i-1,v],F[i-1,v-C_i]+W_i\}$。再用一个数组$G[i,v]$，设$G[i,v]=0$表示推出$F[i,v]$的值时是采用了方程的前一项（也即$F[i,v]=F[i-1,v]$），$G[i,v]=1$表示采用了方程的后一项。注意这两项分别表示了两种策略：未选第i个物品及选了第i个物品。那么输出方案的伪代码可以这样写（设最终状态为$F[N,V]$）：

```
i := N
v := V
while i > 0
  if G[i,v] = 0
    print 未选第i项物品
  else if G[i,v] = 1
    print 选了第i项物品
    v := v - C_i
  i := i - 1
```

另外，采用方程的前一项或后一项也可以在输出方案的过程中根据$F[i,v]$的值实时地求出来。也即，不须纪录G数组，将上述代码中的$G[i,v]=0$改成$F[i,v]=F[i-1,v]$，$G[i,v]=1$改成$F[i,v]=F[i-1][v-C_i]+W_i$也可。

9.2 输出字典序最小的最优方案

这里“字典序最小”的意思是$1\ldots N$号物品的选择方案排列出来以后字典序最小。以输出01背包最小字典序的方案为例。

一般而言，求一个字典序最小的最优方案，只需要在转移时注意策略。

首先，子问题的定义要略改一些。我们注意到，如果存在一个选了物品1的最优方案，那么答案一定包含物品1，原问题转化为一个背包容量为$V - C_1$，物品为$2 \ldots N$的子问题。反之，如果答案不包含物品1，则转化成背包容量仍为V，物品为$2 \ldots N$的子问题。

不管答案怎样，子问题的物品都是以$i \ldots N$而非前所述的$1 \ldots i$的形式来定义的，所以状态的定义和转移方程都需要改一下。

但也许更简易的方法是，先把物品编号做$x := N + 1 - x$的变换，在输出方案时再变换回来。在做完物品编号的变换后，可以按照前面经典的转移方程来求值。只是在输出方案时要注意，如果$F[i, v] = F[i - 1, v]$和$F[i, v] = F[i - 1][v - C_i] + W_i$都成立，应该按照后者来输出方案，即选择了物品$i$，输出其原来的编号$N - 1 - i$。

9.3 求方案总数

对于一个给定了背包容量、物品费用、物品间相互关系（分组、依赖等）的背包问题，除了再给定每个物品的价值后求可得到的最大价值外，还可以得到装满背包或将背包装至某一指定容量的方案总数。

对于这类改变问法的问题，一般只需将状态转移方程中的max改成sum即可。例如若每件物品均是完全背包中的物品，转移方程即为

$$F[i, v] = \text{sum}F[i - 1, v], F[i, v - C_i]$$

初始条件是$F[0, 0] = 1$。

事实上，这样做可行的原因在于状态转移方程已经考察了所有可能的背包组成方案。

9.4 最优方案的总数

这里的最优方案是指物品总价值最大的方案。以01背包为例。

结合求最大总价值和方案总数两个问题的思路，最优方案的总数可以这样求：$F[i, v]$代表该状态的最大价值，$G[i, v]$表示这个子问题的最优方案的总数，则在求$F[i, v]$的同时求$G[i, v]$的伪代码如下：

```
G[0, 0] = 1
for i = 1 to N
  for v = 0 to V
    F[i, v] := max{F[i - 1, v], F[i - 1, v - C_i] + W_i}
    G[i, v] := 0
    if F[i, v] = F[i - 1, v]
      G[i, v]+ = G[i - 1][v]
    if F[i, v] = F[i - 1, v - C_i] + W_i
      G[i, v]+ = G[i - 1][v - C_i]
```

如果你是第一次看到这样的问题，请仔细体会上面的伪代码。

9.5 求次优解、第K优解

对于求次优解、第K优解类的问题，如果相应的最优解问题能写出状态转移方程、用动态规划解决，那么求次优解往往可以相同的复杂度解决，第K优解则比求最优解的复杂度上多一个系数K。

其基本思想是，将每个状态都表示成有序队列，将状态转移方程中的max/min转化成有序队列的合并。

这里仍然以01背包为例讲解一下。

首先看01背包求最优解的状态转移方程：$F[i,v] = \max\{F[i-1,v], F[i-1,v-C_i]+W_i\}$。如果要求第$K$优解，那么状态$F[i,v]$就应该是一个大小为$K$的队列$F[i,v,1\ldots K]$。其中$F[i,v,k]$表示前$i$个物品中，背包大小为$v$时，第$k$优解的值。这里也可以简单地理解为在原来的方程中加了一维来表示结果的优先次序。显然$f[i,v,1\ldots K]$这K个数是由大到小排列的，所以它可看作是一个有序队列。

然后原方程就可以解释为：$F[i,v]$这个有序队列是由$F[i-1,v]$和$F[i-1,v-C_i]+W_i$这两个有序队列合并得到的。前者$F[i-1][V]$即$F[i-1,v,1\ldots K]$，后者$F[i-1,v-C_i]+W_i$则理解为在$F[i-1,v-C_i,1\ldots K]$的每个数上加上W_i后得到的有序队列。合并这两个有序队列并将结果的前K项储存到$f[i,v,1\ldots K]$中的复杂度是$O(K)$。最后的第K优解的答案是$F[N,V,K]$。总的时间复杂度是$O(VNK)$。

为什么这个方法正确呢？实际上，一个正确的状态转移方程的求解过程遍历了所有可用的策略，也就覆盖了问题的所有方案。只不过由于是求最优解，所以其它在任何一个策略上达不到最优的方案都被忽略了。如果把每个状态表示成一个大小为K的数组，并在这个数组中有序地保存该状态可取到的前K个最优值。那么，对于任两个状态的max运算等价于两个由大到小的有序队列的合并。

另外还要注意题目对于“第K优解”的定义，是要求将策略不同但权值相同的两个方案是看作同一个解还是不同的解。如果是前者，则维护有序队列时要保证队列里的数没有重复的。

9.6 小结

显然，这里不可能穷尽背包类动态规划问题所有的问法。甚至还存在一类将背包类动态规划问题与其它领域（例如数论、图论）结合起来的问题，在这篇论背包问题的专文中也不会论及。但只要深刻领会前述所有类别的背包问题的思路和状态转移方程，遇到其它的变形问题，应该也不难想出算法。

触类旁通、举一反三，应该也是一个程序员应有的品质吧。

你好，在这强烈推荐一个良心微信公众号：

如果你想学习 算法，数据结构，计算机基础（计算机网络+操作系统+Linux+MySQL）、编程学习方法、校招准备、面试等，
获取你找的各类编程电子书，那么可以关注公众号『**编程指北**』，微信搜索公众号名称，就可以找到了。

在这个公众号里，专注于分享算法，计算机基础等优质文章，而这些知识，是每个程序员的必修内功。

同时我也整理基本覆盖所有常用学习编程的电子书，在我公众号回复「pdf」即可获取大部分，但是书籍太多，几百本，所以我也会不断更新在Github仓库：

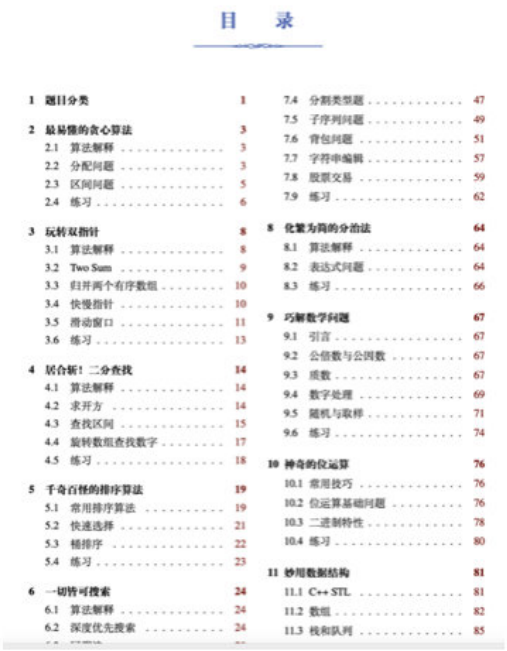
欢迎star： https://github.com/imarvinle/awesome-cs-books

感谢小伙伴们！

你好呀，恭喜你读完了这本书。

在这里也送大家一份我整理的电子书库，绝不是在网上那种打包下载的，而是自己需要学到某个方向知识的时候，去网上挨个找的，最后汇总而成。这部分我是会不断把它完善的，当成自己的小电子书库，不多，但贵在精：

还有两份谷歌大佬的算法笔记，助你拿下算法：



目　录

电子书库和刷题笔记获取的方式，很简单

微信搜索关注「**编程指北**」公众号，后台回复「**end**」，即可获取上面所有资料，或者扫描下方二维码：

👆微信扫描上方二维码 2 秒， 回复「end 」即可获取上面提到的所有资料