背包问题九讲

看云文档小组

目　录

前言

> 原文出处：http://love-oriented.com/pack

本篇文章是我(dd_engi)正在进行中的一个雄心勃勃的写作计划的一部分，这个计划的内容是写作一份较为完善的NOIP难度的动态规划总结，名为《解动态规划题的基本思考方式》。现在你看到的是这个写作计划最先发布的一部分。

背包问题是一个经典的动态规划模型。它既简单形象容易理解，又在某种程度上能够揭示动态规划的本质，故不少教材都把它作为动态规划部分的第一道例题，我也将它放在我的写作计划的第一部分。

读本文最重要的是思考。因为我的语言和写作方式向来不以易于理解为长，思路也偶有跳跃的地方，后面更有需要大量思考才能理解的比较抽象的内容。更重要的是：不大量思考，绝对不可能学好动态规划这一信息学奥赛中最精致的部分。

你现在看到的是本文的v1.1版，发布于2007年11月15日。我会长期维护这份文本，把大家的意见和建议融入其中，也会不断加入我在OI学习以及将来可能的ACM-ICPC的征程中得到的新的心得。但目前本文还没有一个固定的发布页面，想了解本文是否有更新版本发布，可以在OIBH论坛中以“背包问题九讲”为关键字搜索贴子，每次比较重大的版本更新都会在这个论坛里发贴公布。也可以用“背包问题九讲”为关键字在搜索引擎中搜索以得到最新版本。

第一讲 01背包问题

题目

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]。求解将哪些物品装入背包可使价值总和最大。

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牵扯前i-1件物品的问题。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]。

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。

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）能够得到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的顺序推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：

```
for i=1..N
    for v=V..0
        f[v]=max{f[v],f[v-c[i]]+w[i]};
```

其中的f[v]=max{f[v],f[v-c[i]]}一句恰就相当于我们的转移方程`f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]}`，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]]推知，与本题意不符，但它却是另一个重要的背包问题P02最简捷的解决方案，故学习只用一维数组解01背包问题是十分必要的。

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。

过程ZeroOnePack，表示处理一件01背包中的物品，两个参数cost、weight分别表明这件物品的费用和价值。

```
procedure ZeroOnePack(cost,weight)
    for v=V..cost
        f[v]=max{f[v],f[v-cost]+weight}
```

注意这个过程里的处理与前面给出的伪代码有所不同。前面的示例程序写成v=V..0是为了在程序中体现每个状态都按照方程求解了，避免不必要的思维复杂度。而这里既然已经抽象成看作黑箱的过程了，就可以加入优化。费用为cost的物品不会影响状态f[0..cost-1]，这是显然的。

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..N
    ZeroOnePack(c[i],w[i]);
```

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-∞，这样就可以保证最终得到的f[N]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。

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的背包可能被价值为0的nothing“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，它们的值就都应该是-∞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。

一个常数优化

前面的伪代码中有 for v=V..1，可以将这个循环的下限进行改进。

由于只需要最后f[v]的值，倒推前一个物品，其实只要知道f[v-w[n]]即可。以此类推，对以第j个背包，其实只需要知道到f[v-sum{w[j..n]}]即可，即代码中的

```
for i=1..N
```

```
    for v=V..0
```

可以改成

```
for i=1..n
    bound=max{V-sum{w[i..n]},c[i]}
    for v=V..bound
```

这对于V比较大时是有用的。

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，以及最后怎样优化的空间复杂度。

首页

第二讲 完全背包问题

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。第i种物品的费用是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费用总和不超过背包容量，且价值总和最大。

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……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-k*c[i]]+k*w[i]|0<=k*c[i]<=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(v/c[i])，总的复杂度可以认为是O(V*Σ(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。但我们还是试图改进这个复杂度。

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]=w[j]，则将物品j去掉，不用考虑。这个优化的正确性显然：任何情况下都可将价值小费用高得j换成物美价廉的i，得到至少不会更差的方案。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。

转化为01背包问题求解

既然01背包问题是最基本的背包问题，那么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V/c[i]件，于是可以把第i种物品转化为V/c[i]件费用及价值均不变的物品，然后求解这个01背包问题。这样完全没有改进基本思路的时间复杂度，但这毕竟给了我们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满足`c[i]*2^k<=V`。这是二进制的思想，因为不管最优策略选几件第i种物品，总可以表示成若干个2^k件

物品的和。这样把每种物品拆成O(log V/c[i])件物品，是一个很大的改进。

但我们有更优的O(VN)的算法。

O(VN)的算法

这个算法使用一维数组，先看伪代码：

```
for i=1..N
    for v=0..V
        f[v]=max{f[v],f[v-cost]+weight}
```

你会发现，这个伪代码与P01的伪代码只有v的循环次序不同而已。为什么这样一改就可行呢？首先想想为什么P01中要按照v=V..0的逆序来循环。这是因为要保证第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=0..V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。

这个算法也可以以另外的思路得出。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：

```
procedure CompletePack(cost,weight)
    for v=cost..V
        f[v]=max{f[v],f[v-c[i]]+w[i]}
```

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，分别在“基本思路”以及“O(VN)的算法“的小节中给出。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。

首页

第三讲 多重背包问题

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n[i]件可用，每件费用是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费用总和不超过背包容量，且价值总和最大。

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n[i]+1种策略：取0件，取1件……取n[i]件。令f[i][v]表示前i种物品恰放入一个容量为v的背包的最大权值，则有状态转移方程：

> `f[i][v]=max{f[i-1][v-k*c[i]]+k*w[i]|0<=k<=n[i]}`

复杂度是O(V*Σn[i])。

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n[i]件01背包中的物品，则得到了物品数为Σn[i]的01背包问题，直接求解，复杂度仍然是O(V*Σn[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..n[i]件——均能等价于取若干件代换以后的物品。另外，取超过n[i]件的策略必不能出现。

方法是：将第i种物品分成若干件物品，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。使这些系数分别为1,2,4,...,2^(k-1),n[i]-2^k+1，且k是满足n[i]-2^k+1>0的最大整数。例如，如果n[i]为13，就将这种物品分成系数分别为1,2,4,6的四件物品。

分成的这几件物品的系数和为n[i]，表明不可能取多于n[i]件的第i种物品。另外这种方法也能保证对于0..n[i]间的每一个整数，均可以用若干个系数的和表示，这个证明可以分0..2^k-1和2^k..n[i]两段来分别讨论得出，并不难，希望你自己思考尝试一下。

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

下面给出O(log amount)时间处理一件多重背包中物品的过程，其中amount表示物品的数量：

```
procedure MultiplePack(cost,weight,amount)
    if cost*amount>=V
        CompletePack(cost,weight)
        return
    integer k=1
    while k<amount
```

```
        ZeroOnePack(k*cost,k*weight)
        amount=amount-k
        k=k*2
    ZeroOnePack(amount*cost,amount*weight)
```

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，也许就会慢慢理解了。

O(VN)的算法

多重背包问题同样有O(VN)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。由于用单调队列优化的DP已超出了NOIP的范围，故本文不再展开讲解。我最初了解到这个方法是在楼天成的“男人八题”幻灯片上。

小结

这里我们看到了将一个算法的复杂度由O(VΣn[i])改进到O(VΣlog n[i])的过程，还知道了存在应用超出NOIP范围的知识的O(VN)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。

首页

第四讲 混合三种背包问题

问题

如果将P01、P02、P03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？

01背包与完全背包的混合

考虑到在P01和P02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..N
    if 第i件物品属于01背包
        for v=V..0
            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=0..V
            f[v]=max{f[v],f[v-c[i]]+w[i]};
```

再加上多重背包

如果再加上有的物品最多可以取有限次，那么原则上也可以给出O(VN)的解法：遇到多重背包类型的物品用单调队列解即可。但如果不考虑超过NOIP范围的算法的话，用P03中将每个这类物品分成O(log n[i])个01背包的物品的方法也已经很优了。

当然，更清晰的写法是调用我们前面给出的三个相关过程。

```
for i=1..N
    if 第i件物品属于01背包
        ZeroOnePack(c[i],w[i])
    else if 第i件物品属于完全背包
        CompletePack(c[i],w[i])
    else if 第i件物品属于多重背包
        MultiplePack(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？

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。

首页

第五讲 二维费用的背包问题

问题

二维费用的背包问题是指：对于每件物品，具有两种不同的费用；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1和代价2，第i件物品所需的两种代价分别为a[i]和b[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]。

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-a[i]][u-b[i]]+w[i]}
```

如前述方法，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，你能够自己实现出这个问题的程序。

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M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M。换句话说，设f[v][m]表示付出费用v、最多选m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..V][0..M]范围内寻找答案。

复数域上的背包问题

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数。而常见的一维背包问题则是实数域上的背包问题。（注意：上面的话其实不严谨，因为事实上我们处理的都只是整数而已。）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P11中提到的“子集和问题”扩展到复数域（即二维），并试图用同样的复杂度解决。

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纬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。

首页

第六讲 分组的背包问题

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]。这些物品被划分为若干组，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。

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]|物品i属于组k}`

使用一维数组的伪代码如下：

```
for 所有的组k
    for v=V..0
        for 所有的i属于组k
            f[v]=max{f[v],f[v-c[i]]+w[i]}
```

注意这里的三层循环的顺序，甚至在本文的第一个beta版中我自己都写错了。“for v=V..0”这一层循环必须在“for 所有的i属于组k”之外。这样才能保证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P02中“一个简单有效的优化”。

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P07），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。

首页

第七讲 有依赖的背包问题

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，i依赖于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。

算法

这个问题由NOIP2006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P06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P06中的一句话： 可以对每组中的物品应用P02中“一个简单有效的优化”。 这提示我们，对于一个物品组中的物品，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，我们可以对主件i的“附件集合”先进行一次01背包，得到费用依次为0..V-c[i]所有这些值时相应的最大价值f'[0..V-c[i]]。那么这个主件及它的附件集合相当于V-c[i]+1个物品的物品组，其中费用为c[i]+k的物品的价值为f'[k]+w[i]。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i转化为V-c[i]+1个物品的物品组，就可以直接应用P06的算法解决问题了。

较一般的问题

更一般的问题是：依赖关系以图论中“森林”的形式给出（森林即多叉树的集合），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DP，其特点是每个父节点都需要对它的各个儿子的属性进行一次DP以求得自己的相关属性。这已经触及到了“泛化物品”的思想。看完P08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。

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。

我想说：失败不是什么丢人的事情，从失败中全无收获才是。

首页

第八讲 泛化物品

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。

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..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..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)=v/cw，其它函数值均为0。如果它是多重背包中重复次数最多为n的物品，那么它对应的泛化物品的函数有h(v)=v/cw仅当v被c整除且v/c<=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..V中的v，若物品组中不存在费用为v的的物品，则h(v)=0，否则h(v)为所有费用为v的物品的最大价值。P07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。

泛化物品的和

如果面对两个泛化物品h和l，要用给定的费用从这两个泛化物品中得到最大的价值，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..V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)|0<=k<=v}`

可以看到，f也是一个由泛化物品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。

由此可以定义泛化物品的和：h、l都是泛化物品，若泛化物品f满足以上关系式，则称f是h与l的和。这个运算的时间复杂度取决于背包的容量，是O(V^2)。

泛化物品的定义表明：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。设此和为s，则答案就是s[0..V]中的最大值。

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域（例如0..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方法就是将它表示为若干泛化物品的和然后求之。

小结

本讲可以说都是我自己的原创思想。具体来说，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。这一讲真的很抽象，也许在“模型的抽象程度”这一方面已经超出了NOIP的要求，所以暂且看不懂也没关系。相信随着你的OI之路逐渐延伸，有一天你会理解的。

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

首页

第九讲 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。

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]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]
```

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。

输出字典序最小的最优方案

这里“字典序最小”的意思是1..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..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..N的子问题。不管答案怎样，子问题的物品都是以i..N而非前所述的1..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逆序排列一下，以下按物品已被逆序排列来叙述。

在这种情况下，可以按照前面经典的状态转移方程来求值，只是输出方案的时候要注意：从N到1输入时，如果f[i][v]==f[i-1][i-v]及f[i][v]==f[i-1][f-c[i]]+w[i]同时成立，应该按照后者（即选择了物品i）来输出方案。

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]=sum{f[i-1][v],f[i][v-c[i]]}`

初始条件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。

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]意义同前述，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：

```
for i=1..N
   for v=0..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])
            inc(g[i][v],g[i-1][v])
        if(f[i][v]==f[i-1][v-c[i]]+w[i])
            inc(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。

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的数组f[i][v][1..K]。其中f[i][v][k]表示前i个物品、背包大小为v时，第k优解的值。“f[i][v]是一个大小为K的数组”这一句，熟悉C语言的同学可能比较好理解，或者也可以简单地理解为在原来的方程中加了一维。显然f[i][v][1..K]这K个数是由大到小排列的，所以我们把它认为是一个有序队列。

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。有序队列f[i-1][v]即f[i-1][v][1..K]，f[i-1][v-c[i]]+w[i]则理解为在f[i-1][v-c[i]][1..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..K]中的复杂度是O(K)。最后的答案是f[N][V][K]。总的复杂度是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的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，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。

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，遇到其它的变形问法，只要题目难度还属于NOIP，应该也不难想出算法。

触类旁通、举一反三，应该也是一个OIer应有的品质吧。

首页

附录一：USACO中的背包问题

USACO是USA Computing Olympiad的简称，它组织了很多面向全球的计算机竞赛活动。

USACO Trainng是一个很适合初学者的题库，我认为它的特色是题目质量高，循序渐进，还配有不错的课文和题目分析。其中关于背包问题的那篇课文 (TEXT Knapsack Problems) 也值得一看。

另外，USACO Contest是USACO常年组织的面向全球的竞赛系列，在此也推荐NOIP选手参加。

我整理了USACO Training中涉及背包问题的题目，应该可以作为不错的习题。其中标加号的是我比较推荐的，标叹号的是我认为对NOIP选手比较有挑战性的。

题目列表

- Inflate (+)（基本01背包）
- Stamps (+)(!)（对初学者有一定挑战性）
- Money
- Nuggets
- Subsets
- Rockers (+)（另一类有趣的“二维”背包问题）
- Milk4 (!)（很怪的背包问题问法，较难用纯DP求解）

题目简解

以下文字来自我所撰的《USACO心得》一文，该文的完整版本，包括我的程序，可在DD的USACO征程中找到。

> Inflate 是加权01 背包问题，也就是说：每种物品只有一件，只可以选择放或者 不放；而且每种物品有对应的权值，目标是使总权值最大或最小。它最朴素的状态 转移方程是：f[k][i] = max{f[k-1][i]，f[k-1][i-v[k]]+w[k]}。f[k][i]表示前k 件物品花费代 价i 可以得到的最大权值。v[k]和w[k]分别是第k 件物品的花费和权值。可以看到， f[k]的求解过程就是使用第k 件物品对f[k-1]进行更新的过程。那么事实上就不用使用 二维数组，只需要定义f[i]，然后对于每件物品k，顺序地检查f[i]与f[i-v[k]]+w[k]的大小，如果后者更大，就对前者进行更新。这是背包问题中典型的优化方法。
>
> 题目stamps 中，每种物品的使用量没有直接限制，但使用物品的总量有限制。 求第一个不能用这有限个物品组成的背包的大小。（可以这样等价地认为）设f[k][i] 表示前k 件物品组成大小为i 的背包，最少需要物品的数量。则f[k][i]= min{f[k-1][i],f[k-1][i-j*s[k]]+j}，其中j 是选择使用第k 件物品的数目，这个方程运用时 可以用和上面一样的方法处理成一维的。求解时先设置一个粗糙的循环上限，即最 大的物品乘最多物品数。
>
> Money 是多重背包问题。也就是每个物品可以使用无限多次。要求解的是构成 一种背包的不同方案总数。基本上就是把一般的多重背包的方程中的min 改成sum 就行了。

> Nuggets 的模型也是多重背包。要求求解所给的物品不能恰好放入的背包大小 的最大值（可能不存在）。只需要根据“若i、j 互质，则关于x、y 的不定方程ix+yj=n 必有正整数解，其中n>ij”这一定理得出一个循环的上限。 Subsets 子集和问题相当于物品大小是前N 个自然数时求大小为N(N+1)/4 的 01 背包的方案数。
>
> Rockers 可以利用求解背包问题的思想设计解法。我的状态转移方程如下： f[i][j][t]=max{f[i][j][t-1]，f[i-1][j][t]，f[i-1][j][t-time[i]]+1，f[i-1][j-1][T]+(t>=time[i])}。其中 f[i][j][t]表示前i 首歌用j 张完整的盘和一张录了t 分钟的盘可以放入的最多歌数，T 是 一张光盘的最大容量，t>=time[i]是一个bool 值转换成int 取值为0 或1。但我后来发 现我当时设计的状态和方程效率有点低，如果换成这样：f[i][j]=(a,b)表示前i 首歌中 选了j 首需要用到a 张完整的光盘以及一张录了b 分钟的光盘，会将时空复杂度都大 大降低。这种将状态的值设为二维的方法值得注意。
>
> Milk4 是这些类背包问题中难度最大的一道了。很多人无法做到将它用纯DP 方 法求解，而是用迭代加深搜索枚举使用的桶，将其转换成多重背包问题再DP。由于 USACO 的数据弱，迭代加深的深度很小，这样也可以AC，但我们还是可以用纯DP 方法将它完美解决的。设f[k]为称量出k 单位牛奶需要的最少的桶数。那么可以用类 似多重背包的方法对f 数组反复更新以求得最小值。然而困难在于如何输出字典序最 小的方案。我们可以对每个i 记录pre_f[i]和pre_v[i]。表示得到i 单位牛奶的过程是 用pre_f[i]单位牛奶加上若干个编号为pre_v[i]的桶的牛奶。这样就可以一步步求得得 到i 单位牛奶的完整方案。为了使方案的字典序最小，我们在每次找到一个耗费桶数 相同的方案时对已储存的方案和新方案进行比较再决定是否更新方案。为了使这种 比较快捷，在使用各种大小的桶对f 数组进行更新时先大后小地进行。USACO 的官 方题解正是这一思路。如果认为以上文字比较难理解可以阅读官方程序或我的程序。

首页

附录二：背包问题的搜索解法

《背包问题九讲》的本意是将背包问题作为动态规划问题中的一类进行讲解。但鉴于的确有一些背包问题只能用搜索来解，所以这里也对用搜索解背包问题做简单介绍。大部分以01背包为例，其它的应该可以触类旁通。

简单的深搜

对于01背包问题，简单的深搜的复杂度是O(2^N)。就是枚举出所有2^N种将物品放入背包的方案，然后找最优解。基本框架如下：

```
procedure SearchPack(i,cur_v,cur_w)
    if(i>N)
        if(cur_w>best)
            best=cur_w
        return
    if(cur_v+v[i]<=V)
        SearchPack(i+1,cur_v+v[i],cur_w+w[i])
    SearchPack(i+1,cur_v,cur_w)
```

其中cur_v和cur_w表示当前解的费用和权值。主程序中调用SearchPack(1,0,0)即可。

搜索的剪枝

基本的剪枝方法不外乎可行性剪枝或最优性剪枝。

可行性剪枝即判断按照当前的搜索路径搜下去能否找到一个可行解，例如：若将剩下所有物品都放入背包仍然无法将背包充满（设题目要求必须将背包充满），则剪枝。

最优性剪枝即判断按照当前的搜索路径搜下去能否找到一个最优解，例如：若加上剩下所有物品的权值也无法得到比当前得到的最优解更优的解，则剪枝。

搜索的顺序

在搜索中，可以认为顺序靠前的物品会被优先考虑。所以利用贪心的思想，将更有可能出现在结果中的物品的顺序提前，可以较快地得出贪心地较优解，更有利于最优性剪枝。所以，可以考虑将按照“性价比”（权值/费用）来排列搜索顺序。

另一方面，若将费用较大的物品排列在前面，可以较快地填满背包，有利于可行性剪枝。

最后一种可以考虑的方案是：在开始搜索前将输入文件中给定的物品的顺序随机打乱。这样可以避免命题人故意设置的陷阱。

以上三种决定搜索顺序的方法很难说哪种更好，事实上每种方法都有适用的题目和数据，也有可能将它们在某种程度上混合使用。

子集和问题

子集和问题是一个NP-Complete问题，与前述的（加权的）01背包问题并不相同。给定一个整数的集合S和一个整数X，问是否存在S的一个子集满足其中所有元素的和为X。

这个问题有一个时间复杂度为O(2^(N/2))的较高效的搜索算法，其中N是集合S的大小。

第一步思想是二分。将集合S划分成两个子集S1和S2，它们的大小都是N/2。对于S1和S2，分别枚举出它们所有的2^(N/2)个子集和，保存到某种支持查找的数据结构中，例如hash set。

然后就要将两部分结果合并，寻找是否有和为X的S的子集。事实上，对于S1的某个和为X1的子集，只需寻找S2是否有和为X-X1的子集。

假设采用的hash set是理想的，每次查找和插入都仅花费O(1)的时间。两步的时间复杂度显然都是O(2^(N/2))。

实践中，往往可以先将第一步得到的两组子集和分别排序，然后再用两个指针扫描的方法查找是否有满足要求的子集和。这样的实现，在可接受的时间内可以解决的最大规模约为N=42。

搜索还是DP?

在看到一道背包问题时，应该用搜索还是动态规划呢？

首先，可以从数据范围中得到命题人意图的线索。如果一个背包问题可以用DP解，V一定不能很大，否则O(VN)的算法无法承受，而一般的搜索解法都是仅与N有关，与V无关的。所以，V很大时（例如上百万），命题人的意图就应该是考察搜索。另一方面，N较大时（例如上百），命题人的意图就很有可能是考察动态规划了。

另外，当想不出合适的动态规划算法时，就只能用搜索了。例如看到一个从未见过的背包中物品的限制条件，无法想出DP的方程，只好写搜索以谋求一定的分数了。

首页

联系方式

如果有任何意见和建议，特别是文章的错误和不足，或者希望为文章添加新的材料，可以通过http://kontactr.com/user/tianyi/这个网页联系我。

值得说明的是，如果有OI方面的问题，例如不明白自己的程序为什么错了或者索要某种算法的源代码，使用这个联系方式可能得不到及时解答。请在OIBH论坛发问。

致谢

感谢以下名单：

- 阿坦
- jason911
- donglixp
- LeafDuo

他们每人都最先指出了本文曾经存在的某个并非无关紧要的错误。谢谢你们如此仔细地阅读拙作并弥补我的疏漏。

感谢 XiaQ，它针对本文的第一个beta版发表了用词严厉的六条建议，虽然我只认同并采纳了其中的两条。在所有读者几乎一边倒的赞扬将我包围的当时，你的贴子是我的一剂清醒剂，让我能清醒起来并用更严厉的眼光审视自己的作品。

sfita 提供了P01中的“一个常数优化”。

当然，还有用各种方式对我表示鼓励和支持的几乎无法计数的同学。不管是当面赞扬，或是在论坛上回复我的贴子，不管是发来热情洋溢的邮件，或是在即时聊天的窗口里竖起大拇指，你们的鼓励和支持是支撑我的写作计划的强大动力，也鞭策着我不断提高自身水平，谢谢你们！

最后，感谢 Emacs 这一世界最强大的编辑器的所有贡献者，感谢它的插件 EmacsMuse 的开发者们，本文的所有编辑工作都借助这两个卓越的自由软件完成。谢谢你们——自由软件社群——为社会提供了如此有生产力的工具。我深深钦佩你们身上体现出的自由软件的精神，没有你们的感召，我不能完成本文。在你们的影响下，采用自由文档的方式发布本文档，也是我对自由社会事业的微薄努力。

首页