P01: 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”；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]。

注意f[i][v]有意义当且仅当存在一个前i件物品的子集，其费用总和为v。所以按照这个方程递推完毕后，最终的答案并不一定是f[N] [V]，而是f[N][0..V]的最大值。如果将状态的定义中的“恰”字去掉，在转移方程中就要再加入一项f[i][v-1]，这样就可以保证f[N] [V]就是最后的答案。至于为什么这样就可以，由你自己来体会了。

优化空间复杂度

以上方法的时间和空间复杂度均为O(N\*V)，其中时间复杂度基本已经不能再优化了，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。

总结

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。

摘自Tianyi Cui童鞋的《背包问题九讲》，稍作修改，方便理解。

**01背包问题描述**

已知:有一个容量为V的背包和N件物品，第i件物品的重量是weight[i]，收益是cost[i]。

限制:每种物品只有一件，可以选择放或者不放

问题:在不超过背包容量的情况下，最多能获得多少价值或收益

相似问题:在恰好装满背包的情况下，最多能获得多少价值或收益

这里，我们先讨论在不超过背包容量的情况下，最多能获得多少价值或收益。

**基本思路**

01背包的特点:每种物品只有一件，可以选择放或者不放

子问题定义状态

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. f[i][v] : 前i件物品放到一个容量为v的背包中可以获得最大价值

状态转移方程

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. f[i][v] = max(f[i - 1][v],f[i - 1][v - weight[i]] + cost[i])

分析

考虑我们的子问题，将前i件物品放到容量为v的背包中，若我们只考虑第i件物品时，它有两种选择，放或者不放。

1) 如果第i件物品不放入背包中，那么问题就转换为：将前i - 1件物品放到容量为v的背包中，带来的收益f[i - 1][v]

2) 如果第i件物品能放入背包中，那么问题就转换为：将前i - 1件物品放到容量为v - weight[i]的背包中，带来的收益f[i - 1][v - weight[i]] + cost[i]

代码

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. #include <iostream>
2. **using** **namespace** std;
4. **const** **int** N = 3;//物品个数
5. **const** **int** V = 5;//背包最大容量
6. **int** weight[N + 1] = {0,3,2,2};//物品重量
7. **int** value[N + 1] = {0,5,10,20};//物品价值
9. **int** f[N + 1][V + 1] = {{0}};
11. **int** Max(**int** x,**int** y)
12. {
13. **return** x > y ? x : y;
14. }
16. /\*
17. 目标：在不超过背包容量的情况下，最多能获得多少价值
19. 子问题状态:f[i][j]:表示前i件物品放入容量为j的背包得到的最大价值
21. 状态转移方程:f[i][j] = max{f[i - 1][j],f[i - 1][j - weight[i]] + value[i]}
23. 初始化:f数组全设置为0
24. \*/
25. **int** Knapsack()
26. {
27. //初始化
28. memset(f,0,**sizeof**(f));
29. //递推
30. **for** (**int** i = 1;i <= N;i++) //枚举物品
31. {
32. **for** (**int** j = 0;j <= V;j++) //枚举背包容量
33. {
34. f[i][j] = f[i - 1][j];
35. **if** (j >= weight[i])
36. {
37. f[i][j] = Max(f[i - 1][j],f[i - 1][j - weight[i]] + value[i]);
38. }
39. }
40. }
41. **return** f[N][V];
42. }
44. **int** main()
45. {
46. cout<<Knapsack()<<endl;
47. system("pause");
48. **return** 1;
49. }

效率分析：

此算法的时间复杂度为O(N\*V)，空间复杂度也为O(N\*V)。其中，N 表示物品个数，V 表示背包容量这里，时间复杂度不可以在优化了，但是空间复杂度可以继续优化到O(V)

**优化空间复杂度**

上述的方法，我们使用二维数组 f[i][v] 保存中间状态，这里我们可以使用一维数组f[v]保存中间状态就能得到结果

分析

我们现在使用f[v]保存中间状态，我们想要达到的效果是，第i次循环后，f[v]中存储的是前i个物体放到容量v时的最大价值

在回顾下之前讲过的状态转移方程：

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. f[i][v] = max(f[i - 1][v],f[i - 1][v - weight[i]] + cost[i])

我们可以看到，要想得到 f[i][v]，我们需要知道 f[i - 1][v] 和 f[i - 1][v - weight[i]]，由于我们使用二维数组保存中间状态，所以可以直接取出这两个状态。

当我们使用一维数组存储状态时，f[v]表示，在执行i次循环后(此时已经处理i个物品)，前i个物体放到容量v时的最大价值，即之前的f[i][v]。与二维相比较，它把第一维隐去了，但是二者表达的含义还是相同的，只不过针对不同的i，f[v]一直在重复使用，所以，也会出现第i次循环可能会覆盖第i - 1次循环的结果。

为了求f[v],我们需要知道，前i - 1个物品放到容量v的背包中带来的收益，即之前的f[i - 1][v]  和 前i - 1件物品放到容量为v - weight[i]的背包中带来的收益，即之前的f[i - 1][v - weight[i]] + cost[i]。

难点：由于我们只使用一维数组存储，则在求这两个子问题时就没有直接取出那么方便了，因为，第i次循环可能会覆盖第i - 1次循环的结果。

现在我们来求这两个值

1）前i - 1个物品放到容量v的背包中带来的收益，即之前的f[i - 1][v] ：

由于，在执行在i次循环时，f[v]存储的是前i个物体放到容量v时的最大价值，在求前i个物体放到容量v时的最大价值(即之前的f[i][v])时，我们是正在执行第 i 次循环，f[ v ]的值还是在第 i - 1  次循环时存下的值，在此时取出的 f[ v ]就是前i - 1个物体放到容量v时的最大价值，即f[i - 1][v]。

2）前i - 1件物品放到容量为v - weight[i]的背包中带来的收益，即之前的f[i - 1][v - weight[i]] + cost[i]

由于，在执行第i次循环前，f[0 ~ V]中保存的是第i - 1次循环的结果，即是前i - 1个物体分别放到容量0 ~ V时的最大价值，即f[i - 1][0 ~ V]。

则，在执行第i次循环前，f 数组中v - weight[i]的位置存储就是我们要找的 前i - 1件物品放到容量为v - weight[i]的背包中带来的收益 (即之前的f[i - 1][v - weight[i]])，这里假设物品是从数组下标1开始存储的。

伪代码

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. **for** i=1..N //枚举物品
2. **for** v=V..0 //枚举容量，从大到小
3. f[v]=max{f[v],f[v-weight[i]] + cost[i]};

由上面伪代码可知，在执行第 i 次循环时，需要把背包容量由V..0都要遍历一遍，检测第 i 件物品是否能放。

逆序枚举容量的原因：

注意一点，我们是由第 i - 1 次循环的两个状态推出 第 i 个状态的，而且 v  > v - weight[i]，则对于第i次循环，背包容量只有当V..0循环时，才会先处理背包容量为v的状况，后处理背包容量为 v-weight[i] 的情况。

具体来说，由于，在执行v时，还没执行到v - weight[i]的，因此，f[v - weight[i]]保存的还是第i - 1次循环的结果。即在执行第i次循环 且 背包容量为v时，此时的f[v]存储的是 f[i - 1][v] ，此时f[v-weight[i]]存储的是f[i - 1][v-weight[i]]。

相反，如果在执行第 i 次循环时，背包容量按照0..V的顺序遍历一遍，来检测第 i 件物品是否能放。此时在执行第i次循环 且 背包容量为v时，此时的f[v]存储的是 f[i - 1][v] ，但是，此时f[v-weight[i]]存储的是f[i][v-weight[i]]。

因为，v  > v - weight[i]，第i次循环中，执行背包容量为v时，容量为v - weight[i]的背包已经计算过，即f[v - weight[i]]中存储的是f[i][v - weight[i]]。即，对于01背包，按照增序枚举背包容量是不对的。

代码

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. #include <iostream>
2. **using** **namespace** std;
4. **const** **int** N = 3;//物品个数
5. **const** **int** V = 5;//背包最大容量
6. **int** weight[N + 1] = {0,3,2,2};//物品重量
7. **int** value[N + 1] = {0,5,10,20};//物品价值
9. **int** f[V + 1] = {0};
11. **int** Max(**int** x,**int** y)
12. {
13. **return** x > y ? x : y;
14. }
16. /\*
17. 目标：在不超过背包容量的情况下，最多能获得多少价值
19. 子问题状态:f[j]:表示前i件物品放入容量为j的背包得到的最大价值
21. 状态转移方程:f[j] = max{f[j],f[j - weight[i]] + value[i]}
23. 初始化:f数组全设置为0
24. \*/
25. **int** Knapsack()
26. {
27. //初始化
28. memset(f,0,**sizeof**(f));
29. //递推
30. **for** (**int** i = 1;i <= N;i++) //枚举物品
31. {
32. **for** (**int** j = V;j >= weight[i];j--) //枚举背包容量,防越界，j下限为 weight[i]
33. {
34. f[j] = Max(f[j],f[j - weight[i]] + value[i]);
35. }
36. }
37. **return** f[V];
38. }
40. **int** main()
41. {
42. cout<<Knapsack()<<endl;
43. system("pause");
44. **return** 1;
45. }

但是，增序枚举背包容量会达到什么效果：它会重复的装入某个物品，而且尽可能多的，使价值最大，当然不会不超过背包容量

而逆序枚举背包容量：背包中的物品至多装一次，使价值最大，当然不会不超过背包容量

我们首先举例说明：

逆序枚举物品



当i = 2，我们要求 f [5]：表示检测物品2放入容量为5的背包的最大收益

上图表示，当i = 2，求f[5]时f数组的状况，

橙色为数组现在存储的值，这些值是i = 1时(上一次循环)存入数组 f 的。相当于f[i - 1][v]

而黄色使我们要求的值，在求f[5]之前，f[5]= 5，即f[i - 1][5] = 5

现在要求 i = 2 时的f[5] = f[5 - 2] + 10 = 5 + 10 = 15  >  f[i - 1][5] = 5

故，f[5] = 15；

注意一点，在求f[v]时，它引用的 f[v - weight[i]] 和 f[v]都是上一次循环的结果

顺序枚举物品



当i = 2，我们要求 f [5]：表示检测物品2放入容量为5的背包的最大收益

上图表示，当i = 2，求f[5]时f数组的状况，

橙色为数组现在存储的值，这些值是i = 2时(本次循环)存入数组 f 的。相当于f[i][v]

这是由于，我们是增序遍历数组f的，在求f[v]时，v之前的值(0 ~ v - 1)都已经在第i次循环中求出。

而黄色使我们要求的值，在求f[5]之前，f[5]= 5，即f[i - 1][5] = 5

现在要求 i = 2 时的f[5] = f[5 - 2] + 10 =10+ 10 = 20 >  f[i - 1][5] = 5

故，f[5] = 20；

其中引用的f[3]是相当于f[i][3] 而不是正常的f[i - 1][3]

注意一点，在求f[v]时，它引用的 f[v - weight[i]]是本次循环的结果 而f[v]是上一次循环的结果

换个角度说，

在检测 背包容量为5时，看物品2是否加入

由状态转移方程可知，我们f[5]需要引用自己本身和f[3]

由于背包容量为3时，可以装入物品2，且收益比之前的大，所以放入背包了。

在检测f[5]时，肯定要加上物品2的收益，而f[5]在引用f[3]时，f[3]时已经加过一次物品2，

因此，在枚举背包容量时，物品2会加入多次。

进一步说：

我们观察一维状态转移方程：

f[i][v] = max(f[i - 1][v],f[i - 1][v - weight[i]] + cost[i])

首先我们明确三个问题

1) v - weight[i] < v

2) 状态f[i][v] 是由 f[i - 1][v] 和 f[i - 1][v - weight[i]] 两个状态决定

3) 对于物品i，我们在枚举背包容量时，只要背包容量能装下物品i 且 收益比原来的大，就会成功放入物品i。

具体来说，枚举背包容量时，是以递增的顺序的话，由于 v - weight[i] < v,则会先计算 v - weight[i]。在背包容量为v - weight[i]时，一旦装入了物品i，由于求f[v]需要使用f[i - 1][v - weight[i]]，而若求f[v]时也可以装入物品i的话，那么在背包容量为v时，容量为v的背包就装入可两次物品。又若v - weight[i]是由之前的状态推出，它们也成功装入物品i的话，那么容量为v的背包就装入了多次物品i了。

注意，此时，在计算f[v]时，已经把物品i能装入的全装入容量为v的背包了，此时装入物品i的次数为最大啊  
  
其实，顺序枚举容量是完全背包问题最简捷的解决方案。

**初始化的细节问题**

求最优解的背包问题时，有两种问法：

1)在不超过背包容量的情况下，最多能获得多少价值

2)在恰好装满背包的情况下，最多能获得多少价值

主要的区别为是否要求恰好装满背包。但这两种问法的实现方法是在初始化的时候有所不同。

1)恰好装满背包的情况：使用二维数组f[i][v]存储中间状态，其中第一维表示物品，第二维表示背包容量

初始化时，除了f[i][0] = 0（第一列）外，其他全为负无穷。

原因：初始化 f 数组就是表示：在没有任何物品可以放入背包时的合法状态。对于恰好装满背包，只有背包容量为 0（第一列），可以什么物品都不装就能装满，这种情况是合法情况，此时价值为0。其他f[0][v]（第一列）是都不能装满的，此时有容量没物品。而其他位置（除去第一行和第一列的位置），我们为了在计算中比较最大值，也要初始化为负无穷。我们从程序的角度上看，我们只允许装入背包物品的序列的起始位置是从第一列开始，这些起始位置都是合法位置，且能恰好装满的情况收益均为正值，到f[N][V]终止。

注意，我们虽然是求恰好装满，还是需要枚举所有可以装入背包的物品，只要能装入，还需装入，收益有增加。只不过，由于恰好装满的物品的序列肯定是从第一列某行开始的，且之后的收益肯定是正值。对于非恰好装满的物品序列，其实位置肯定是从第一行某位置开始的，由于此时被初始化为负无穷，在和那些恰好装满物品序列带来的价值时，肯定是小的。所以，我们最后能获得最大值。

代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. #include <iostream>
2. **using** **namespace** std;
4. **const** **int** MinNum = 0x80000000;
6. **const** **int** N = 3;//物品个数
7. **const** **int** V = 5;//背包最大容量
8. **int** weight[N + 1] = {0,3,2,2};//物品重量
9. **int** value[N + 1] = {0,5,10,20};//物品价值
11. **int** f[N + 1][V + 1] = {{0}};
13. **int** Max(**int** x,**int** y)
14. {
15. **return** x > y ? x : y;
16. }
18. /\*
19. 目标：在恰好装满背包的情况下，最多能获得多少价值
21. 子问题状态:f[i][j]:表示前i件物品放入容量为j的背包得到的最大价值
23. 状态转移方程:f[i][j] = max{f[i - 1][j],f[i - 1][j - weight[i]] + value[i]}
25. 初始化:f数组全设置为0
26. \*/
27. **int** Knapsack()
28. {
29. //初始化
30. **for** (**int** i = 0;i <= N;i++) //枚举物品
31. {
32. **for** (**int** j = 0;j <= V;j++) //枚举背包容量
33. {
34. f[i][j] = MinNum;
35. }
36. }
37. **for** (**int** i = 0;i <= N;i++)
38. {
39. f[i][0] = 0;//背包容量为0时为合法状态
40. }
41. //递推
42. **for** (**int** i = 1;i <= N;i++) //枚举物品
43. {
44. **for** (**int** j = 1;j <= V;j++) //枚举背包容量
45. {
46. f[i][j] = f[i - 1][j];
47. **if** (j >= weight[i])
48. {
49. f[i][j] = Max(f[i - 1][j],f[i - 1][j - weight[i]] + value[i]);
50. }
51. }
52. }
53. **return** f[N][V];
54. }
56. **int** main()
57. {
58. cout<<Knapsack()<<endl;//输出25
59. system("pause");
60. **return** 1;
61. }

使用一维数组f[v]存储中间状态，维表示背包容量

初始化时，除了f[0] = 0，其他全为负无穷。

原因：只有容量为0 的背包可以什么物品都不装就能装满，此时价值为0。其它容量的背包均没有合法的解，属于未定义的状态，应该被赋值为负无穷了

代码

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. #include <iostream>
2. **using** **namespace** std;
4. **const** **int** MinNum = 0x80000000;//int最小的数
6. **const** **int** N = 3;//物品个数
7. **const** **int** V = 5;//背包最大容量
8. **int** weight[N + 1] = {0,3,2,2};//物品重量
9. **int** value[N + 1] = {0,5,10,20};//物品价值
11. **int** f[V + 1] = {0};
13. **int** Max(**int** x,**int** y)
14. {
15. **return** x > y ? x : y;
16. }
18. /\*
19. 目标：在恰好装满背包容量的情况下，最多能获得多少价值
21. 子问题状态:f[j]:表示前i件物品放入容量为j的背包得到的最大价值
23. 状态转移方程:f[j] = max{f[j],f[j - weight[i]] + value[i]}
25. 初始化:f数组全设置为0
26. \*/
27. **int** Knapsack()
28. {
29. //初始化
30. **for** (**int** i = 0;i <= V;i++)
31. {
32. f[i] = MinNum;
33. }
34. f[0] = 0;//只有背包容量为0时才是合法状态，由合法状态组成的结果才是合法的
36. //递推
37. **for** (**int** i = 1;i <= N;i++) //枚举物品
38. {
39. **for** (**int** j = V;j >= weight[i];j--) //枚举背包容量,防越界，j下限为 weight[i]
40. {
41. f[j] = Max(f[j],f[j - weight[i]] + value[i]);
42. }
43. }
44. **return** f[V];
45. }
47. **int** main()
48. {
49. cout<<Knapsack()<<endl;//输出25
50. system("pause");
51. **return** 1;
52. }

2)不需要把背包装满，只需要收益最大

使用二维数组f[i][v]存储中间状态，其中第一维表示物品，第二维表示背包容量

初始化时，除了f[i][0] = 0（第一列）外，其他全为负无穷。

使用一维数组f[v]存储中间状态，维表示背包容量

原因：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。

代码在最前面已贴，不在此上传。

**一个常数优化**

一维数组描述状态时的伪代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. **for** i=1..N //枚举物品
2. **for** v=V..0 //枚举容量，从大到小
3. f[v]=max{f[v],f[v-weight[i]] + cost[i]};

观察可知，对于第i个物品，枚举背包容量下限时，可以到weight[i]为止。

原因：

1）对于第i物品，在求f[v]时，需要使用的状态是 v ~ v -  weight[i] 这么多，这是由于v取最大容量V时，使用的状态才是v - weight[i]，当v不取最大状态时，使用的状态肯定是在v ~ v -  weight[i]之间的。可以到weight[i]为止。

2）在到weight[i]为止时，还可以不进行if判断，担心v -  weight[i]是否越界

此时，伪代码为

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. **for** i=1..N //枚举物品
2. **for** v=V..weight[i] //枚举容量，从大到小
3. f[v]=max{f[v],f[v-weight[i]] + cost[i]};

注意，对 f 数组，如果是检测第i个物品是否能放入，0 ~ weight[i] - 1的这些位置是不会遍历到的，则此时他们仍表示第i - 1次的状态，即二维的f[i - 1][v]。

还可以继续优化下界为

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. **for** i=1..N //枚举物品
2. bound=max{V-sum{weight[i..n]},weight[i]}//确定需要枚举容量的下界
3. **for** v=V..bound
4. f[v]=max{f[v],f[v-weight[i]] + cost[i]};

原因：

1)网上的说法,不太懂,各位大牛可以指导下下。

对于第i次循环（指外循环），对于背包容量v = V(最大)时，对于f[v]的值，其实只要知道f[v-weight[i]]即可。以此类推，对于背包容量为 j 时，我们只需要知道到f[v-sum{weight[j..n]}]即可

2)还有人说

如果比v-sum{weight[j..n]}这个小，那么即使后面物品的全要也装不满背包。

所以对于物品i，小于v-sum{weight[j..n]}的v值，无意义。

总之是不懂。智商啊

作者说，当V很大是，效果好。

代码

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. #include <iostream>
2. **using** **namespace** std;
4. **const** **int** N = 3;//物品个数
5. **const** **int** V = 5;//背包最大容量
6. **int** weight[N + 1] = {0,3,2,2};//物品重量
7. **int** value[N + 1] = {0,5,10,20};//物品价值
9. **int** f[V + 1] = {0};
11. **int** Max(**int** x,**int** y)
12. {
13. **return** x > y ? x : y;
14. }
16. /\*
17. 目标：在不超过背包容量的情况下，最多能获得多少价值
19. 子问题状态:f[j]:表示前i件物品放入容量为j的背包得到的最大价值
21. 状态转移方程:f[j] = max{f[j],f[j - weight[i]] + value[i]}
23. 初始化:f数组全设置为0
24. \*/
25. **int** Knapsack()
26. {
27. **int** sum = 0;//存储还未处理物品的总容量
28. **int** bound = 0;
29. //初始化
30. memset(f,0,**sizeof**(f));
31. **for** (**int** i = 1;i <= N;i++)
32. {
33. sum += weight[i];
34. }
36. //递推
37. **for** (**int** i = 1;i <= N;i++) //枚举物品
38. {
39. //设置下界
40. **if** (i != 1)
41. {
42. sum -= weight[i - 1];
43. }
44. bound = Max(V - sum,weight[i]);
46. **for** (**int** j = V;j >= bound;j--) //枚举背包容量
47. {
48. **if** (f[j] < f[j - weight[i]] + value[i])
49. {
50. f[j] = f[j - weight[i]] + value[i];
51. }
52. }
53. }
54. **return** f[V];
55. }
57. **int** main()
58. {
59. cout<<Knapsack()<<endl;
60. system("pause");
61. **return** 1;
62. }

**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。

这里我们首先给出01背包的二维状态转移方程

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. f[i][v] = max(f[i - 1][v],f[i - 1][v - weight[i]] + cost[i])

对于状态f[i][v]，它来自两种策略，可以是f[i - 1][v],也可以是f[i - 1][v - weight[i]] + cost[i]

其中，对于第二种情况，就是把物品i放入背包了，这里也是我们要找的情况

根据状态转移方程，我们可以给出两种实现方法

1) 借助存储状态的数组，直接根据状态转移方程倒着推，检测是否满足

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. f[i][v] == f[i - 1][v - weight[i]] + value[i]

如果满足，则把第i件物品放入了，此时我们要检测第i - 1件物品，背包容量为v - weight[i]

不满足则表示没有把第i件物品放入，直接检测第i - 1件物品，此时背包容量还是v

注意，这种方法只适用于存储状态数组不压缩的情况。压缩数组由于数据有覆盖，不能使用

代码

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. #include <iostream>
2. **using** **namespace** std;
4. **const** **int** N = 3;//物品个数
5. **const** **int** V = 5;//背包最大容量
6. **int** weight[N + 1] = {0,3,2,2};//物品重量
7. **int** value[N + 1] = {0,5,10,20};//物品价值
9. **int** f[N + 1][V + 1] = {{0}};
11. **int** Max(**int** x,**int** y)
12. {
13. **return** x > y ? x : y;
14. }
16. /\*
17. 目标：在不超过背包容量的情况下，最多能获得多少价值
19. 子问题状态:f[i][j]:表示前i件物品放入容量为j的背包得到的最大价值
21. 状态转移方程:f[i][j] = max{f[i - 1][j],f[i - 1][j - weight[i]] + value[i]}
23. 初始化:f数组全设置为0
24. \*/
25. **int** Knapsack()
26. {
27. //初始化
28. memset(f,0,**sizeof**(f));
29. //递推
30. **for** (**int** i = 1;i <= N;i++) //枚举物品
31. {
32. **for** (**int** j = 1;j <= V;j++) //枚举背包容量
33. {
34. f[i][j] = f[i - 1][j];
35. **if** (j >= weight[i])
36. {
37. f[i][j] = Max(f[i - 1][j],f[i - 1][j - weight[i]] + value[i]);
38. }
39. }
40. }
41. **return** f[N][V];
42. }
43. /\*
44. 输出顺序:逆序输出物品编号
45. 注意：这里借助状态数组f[i][v]
46. 使用状态转移方程:f[i][j] = max{f[i - 1][j],f[i - 1][j - weight[i]] + value[i]}
47. \*/
48. **void** PrintKnapsack()
49. {
50. **int** i = N;//枚举物品
51. **int** j = V;//枚举空间
53. cout<<"加入背包的物品编号:"<<endl;
54. **while**(i)
55. {
56. **if** (f[i][j] == f[i - 1][j - weight[i]] + value[i])
57. {
58. /\*if不满足，表示第i件物品没装入背包,
59. if条件满足，表示放入背包了\*/
60. cout<<i<<" ";
61. j -= weight[i];//此时容量减少
62. }
63. i--;
64. }
65. cout<<endl;
66. }
68. /\*
69. 输出顺序:顺序输出物品编号
70. 注意：这里借助状态数组f[i][v]
71. 使用状态转移方程:f[i][j] = max{f[i - 1][j],f[i - 1][j - weight[i]] + value[i]}
72. \*/
73. **void** PrintKnapsack\_recursion(**int** i,**int** j)
74. {
75. **if** (i == 0 || j == 0)
76. {
77. **return**;
78. }
79. **if** (f[i][j] == f[i - 1][j - weight[i]] + value[i])
80. {
81. PrintKnapsack\_recursion(i - 1,j - weight[i]);
82. cout<<i<<" ";
83. }
84. }
86. **int** main()
87. {
88. cout<<Knapsack()<<endl;
89. PrintKnapsack();
90. PrintKnapsack\_recursion(N,V);
91. system("pause");
92. **return** 1;
93. }

2) 另外开辟数组，在求解最大收益时做标记位。求解完最大收益后，根据这个新数组倒着推结果

思想：对于现在这个状态的位置，它存储的是该状态上一位置

注意：这种方法均适用存储状态数组不压缩 和 压缩两种情况

代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/insistgogo/article/details/8579597)

1. #include <iostream>
2. **using** **namespace** std;
4. **const** **int** N = 3;//物品个数
5. **const** **int** V = 5;//背包最大容量
6. **int** weight[N + 1] = {0,3,2,2};//物品重量
7. **int** value[N + 1] = {0,5,10,20};//物品价值
9. **int** f[V + 1] = {0};
11. **int** G[N + 1][V + 1] = {{0}};//求背包序列
13. **int** Max(**int** x,**int** y)
14. {
15. **return** x > y ? x : y;
16. }
18. /\*
19. 目标：在不超过背包容量的情况下，最多能获得多少价值
21. 子问题状态:f[j]:表示前i件物品放入容量为j的背包得到的最大价值
23. 状态转移方程:f[j] = max{f[j],f[j - weight[i]] + value[i]}
25. 初始化:f数组全设置为0
26. \*/
27. **int** Knapsack()
28. {
29. //初始化
30. memset(f,0,**sizeof**(f));
31. memset(G,0,**sizeof**(G));
32. //递推
33. **for** (**int** i = 1;i <= N;i++) //枚举物品
34. {
35. **for** (**int** j = V;j >= weight[i];j--) //枚举背包容量
36. {
37. **if** (f[j] < f[j - weight[i]] + value[i])
38. {
39. f[j] = f[j - weight[i]] + value[i];
40. G[i][j] = 1;
41. }
42. }
43. }
44. **return** f[V];
45. }
46. /\*
47. 输出顺序:逆序输出物品编号
48. 注意：这里另外开辟数组G[i][v],标记上一个状态的位置
49. G[i][v] = 1:表示物品i放入背包了，上一状态为G[i - 1][v - weight[i]]
50. G[i][v] = 0:表示物品i没有放入背包，上一状态为G[i - 1][v]
51. \*/
52. **void** PrintKnapsack()
53. {
54. **int** i = N;//枚举物品
55. **int** j = V;//枚举空间
57. cout<<"加入背包的物品编号:"<<endl;
58. **while**(i)
59. {
60. **if** (G[i][j] == 1)
61. {
62. /\*if不满足，表示第i件物品没装入背包,
63. if条件满足，表示放入背包了\*/
64. cout<<i<<" ";
65. j -= weight[i];//此时容量减少
66. }
67. i--;
68. }
69. cout<<endl;
70. }
72. /\*
73. 输出顺序:顺序输出物品编号
74. 注意：这里另外开辟数组G[i][v],标记上一个状态的位置
75. G[i][v] = 1:表示物品i放入背包了，上一状态为G[i - 1][v - weight[i]]
76. G[i][v] = 0:表示物品i没有放入背包，上一状态为G[i - 1][v]
77. \*/
78. **void** PrintKnapsack\_recursion(**int** i,**int** j)
79. {
80. **if** (i == 0 || j == 0)
81. {
82. **return**;
83. }
84. **if** (G[i][j] == 1)
85. {
86. PrintKnapsack\_recursion(i - 1,j - weight[i]);
87. cout<<i<<" ";
88. }
89. }
91. **int** main()
92. {
93. cout<<Knapsack()<<endl;
94. PrintKnapsack();
95. PrintKnapsack\_recursion(N,V);
96. system("pause");
97. **return** 1;
98. }

**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。

P02: 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(N\*V)个状态需要求解，但求解每个状态的时间则不是常数了，求解状态f[i][v]的时间是O(v/c[i])，总的复杂度是超过O(VN)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]<=c[j]且w[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。

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： <pre class"example"> for i=1..N for v=0..V f[v]=max{f[v],f[v-c[i]]+w[i]};

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。

这个算法也可以以另外的思路得出。例如，基本思路中的状态转移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。

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。

P03: 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：f[i][v]=max{f[i-1][v-k\*c[i]]+ k\*w[i]|0<=k<=n[i]}。复杂度是O(V\*∑n[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∑n[i]的01背包问题，直接求解，复杂度仍然是O(V\*∑n[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\*∑log n[i])的01背包问题，是很大的改进。

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(V\*∑n[i])改进到O(V\*∑log 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，并用尽量简洁的程序来实现。

P04: 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中最后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件物品是01背包

for v=V..0

f[v]=max{f[v],f[v-c[i]]+w[i]};

else if 第i件物品是完全背包

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。

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。

P05: 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采用顺序的循环，当物品有如完全背包问题时采用逆序的循环。当物品有如多重背包问题时拆分物品。

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。

另外，如果要求“恰取M件物品”，则在f[0..V][M]范围内寻找答案。

小结

事实上，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。

P06: 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属于第k组}。

使用一维数组的伪代码如下：

for 所有的组k

for 所有的i属于组k

for v=V..0

f[v]=max{f[v],f[v-c[i]]+w[i]}

另外，显然可以对每组中的物品应用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。

P07: 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。

更一般的问题

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。

P08: 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/c\*w，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/c\*w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f(v)=max{h(k)+l(v-k)|0<=k<=v}，则称f是h与l的和，即f=h+l。这个运算的时间复杂度是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。

P09: 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-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01背包中的物品，转移方程即为f[i][v]=sum{f[i-1][v],f[i-1][v-c[i]]+w[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。还是以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。

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。