# P01: 0||1背包

题目

有N件物品和一个容量为V的背包。第i件物品的重量是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。特点是：每种物品仅有一件，可以选择放或不放。

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]}

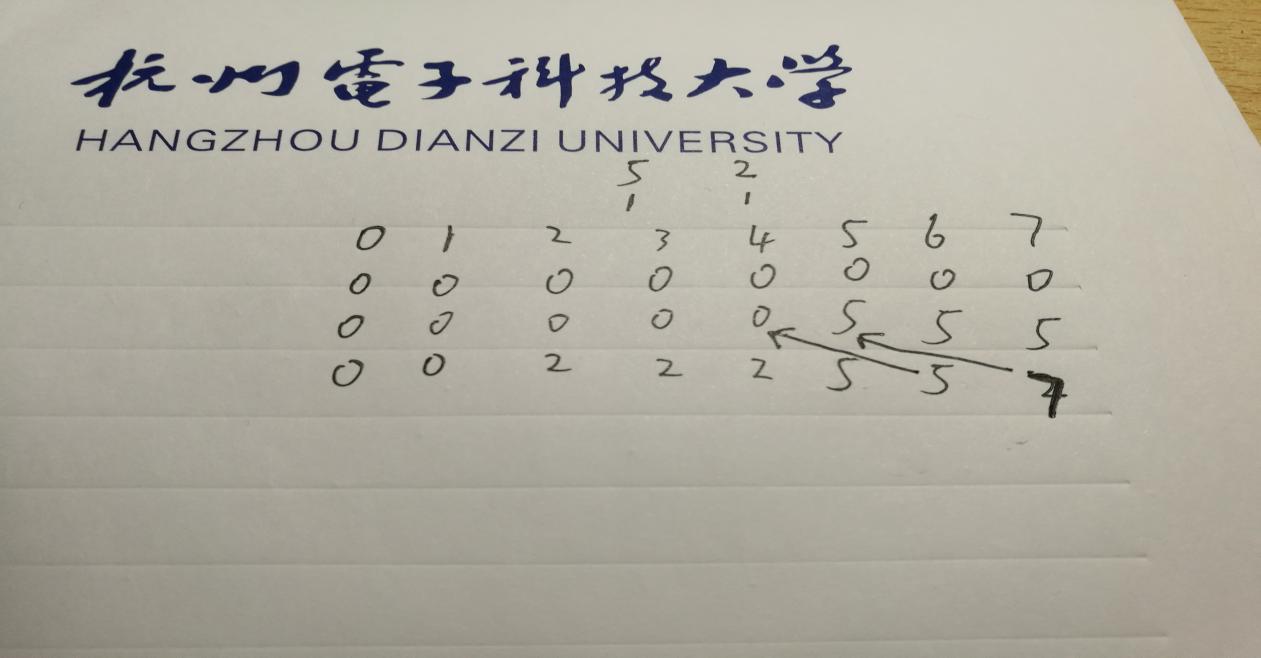
解释：将前i件物品放入容量为v的背包中”这个子问题，若只考虑第i件物品的策略（放或不放），那么就可以转化为一个只牵扯前i-1件物品的问题。如果不放第i件物品，那么问题就转化为“前i-1件物品放入容量为v的背包中”，价值为f[i-1][v]；如果放第i件物品，那么问题就转化为“前i-1件物品放入剩下的容量为v-c[i]的背包中”，此时能获得的最大价值就是f[i-1][v-c[i]]再加上通过放入第i件物品获得的价值w[i]。

伪代码如下：

for i=1..N

    for v=V..0

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



## 初始化的细节问题

要求恰好装满背包（容量必须全部用完），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（每一个子背包的容量都不一定用完），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。

# P02: 完全背包@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。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。

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}

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。

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]}

f[i][v]=max{f[i-1][v](f[i][0]),f[i][1]+w[i],f[i][2]+2\*w[i],f[i][3]+3\*w[i]...f[i][V/c[i]]+(V/c[i])\*w[i]}

# 完全背包@2

procedure CompletePack(cost,weight)

 for v=cost..V

        f[v]=max{f[v],f[v-c[i]]+w[i]}

我们可以理解为 0||1背包是有放与不放两种选择的，而完全背包是有放0件，放一件，放两件。。。放V÷c[i]件这么多选择（可以理解为，一定会放，不放的情况是放0件）。

# 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。

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]}

(如果真不想优化，就在0||1背包的第一个循环下面再加一个循环用来表示每一种物品的加入个数)

## 优化

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。

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<num

        ZeroOnePack(k\*cost,k\*weight)

amount=amount-k

k=k\*2

这是一个多重背包的模板，也是十分好用的一种模板，因为这个比直接拆除01 背包来做要省些时间。这是为啥呢，首先先由我讲一下为什么能换成01 背包吧。

 老杨我就举个栗子吧。 假如给了我们价值为 2，但是数量却是10 的物品，我们应该把10给拆开，要知道二进制可是能够表示任何数的，所以10 就是可以有1，2, 4，8之内的数把它组成，一开始我们选上 1了，然后让10-1=9，再选上2, 9-2=7，在选上4, 7-4=3，而这时的3<8了，所以我们就是可以得出10由 1,2,4,3，来组成，就是这个数量为1,2,3,4的物品了，那么他们的价值是什么呢，是2,4,6,8，也就说给我们的价值为2，数量是10的这批货物，已经转化成了价值分别是2,4,6,8元的货物了，每种只有一件哎！！！！这就是二进制优化的思想。

 那为什么会有完全背包和01 背包的不同使用加判断呢？原因也很简单啊，当数据很大，大于背包的容纳量时，我们就是在这个物品中取上几件就是了，取得量时不知道的，也就理解为无限的啦，这就是完全背包啦，反而小于容纳量的就是转化为01背包来处理就是了，可以大量的省时间。

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#define N 1000 //物品个数

#define M 100000000 //所有物品可能的最大价值

int m[N],c[N],w[N],f[M];

int V;

int max(int a,int b){return a>b?a:b;}

void ZeroOnePack(int cost,int weight)

{

int v;

for(v=V;v>=cost;v--) f[v]=max(f[v],f[v-cost]+weight);

}

void CompletePack(int cost,int weight)

{

int v;

for(v=cost;v<=V;v++)

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

}

void MultiplePack(int cost,int weight,int amount)

{

int k;

if(cost\*amount>=V)

{

CompletePack(cost,weight);

return;

}

k=1;

while(k<amount)

{

ZeroOnePack(k\*cost,k\*weight);

amount=amount-k;

k=k\*2;

}

ZeroOnePack(amount\*cost,amount\*weight);

}

int main()

{

int n,i;

scanf("%d %d",&n,&V);

// 两种不同的初始化方式，根据情况自行选择

//memset(f,0,sizeof(f[0])\*(V+1)); // 只希望价格尽量大

//memset(f,-M,sizeof(f[0])\*(V+1));f[0]=0; // 要求恰好装满背包

for(i=0;i<n;i++) scanf("%d %d %d",m+i,c+i,w+i);

for(i=0;i<n;i++) MultiplePack(c[i],w[i],m[i]);

printf("%d\n",f[V]);

system("PAUSE");

return 0;

}

# 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？

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。

# 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。

//初始化dp

for(i=0;i<=M;i++)

for(j=0;j<=L;j++)

{

if(i==0)

dp[i][j]=0;

else

dp[i][j]=*-*inf;

}

//二维背包dp

for(j=0;j<N;j++) //注意枚举N个物品的变量必须放在最外层

for(k=L;k>=t[j];k--)

for(i=M;i>= 1;i--)

dp[i][k]=max(dp[i][k],dp[i-1][k-t[j]]+v[j]);

//判断

if(dp[M][L]>0)

printf("%d\n",dp[M][L]);

else

printf("0\n");

## 物品总个数的限制（精辟）

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。

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。

# 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。

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 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。

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。

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。

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。

# 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。

# P09: 背包问题问法的变化

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。

## 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]

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][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]])

# 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(NVK)。

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。