第九章 动态规划

第二节 背包问题

第二节 背包问题

一、01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（即体积，下同）是w[i]，价值是c[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(部分或全部)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-w[i]]+c[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-w[i]的背包中”，此时能获得的最大价值就是f [i-1][v-w[i]]再加上通过放入第i件物品获得的价值c[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-1][v]，这样就可以保证f[N][V]就是最后的答案。但是若将所有f[i][j]的初始值都赋为0，你会发现f[n][v]也会是最后的答案。为什么呢？因为这样你默认了最开始f[i][j]是有意义的，只是价值为0，就看作是无物品放的背包价值都为0，所以对最终价值无影响，这样初始化后的状态表示就可以把“恰”字去掉。

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-w[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-w[i]]的值呢？事实上，这要求在每次主循环中我们以v=V..0的逆序推f[v]，这样才能保证推f[v]时f[v-w[i]]保存的是状态f[i-1][v-w[i]]的值。

伪代码如下：

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

其中f[v]=max{f[v],f[v-w[i]]+c[i]}相当于转移方程f[i][v]=max{f[i-1][v],f[i-1][v-w[i]]+c[i]}，因为现在的f[v-w[i]]就相当于原来的f[i-1][v-w[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-w[i]]推知，与本题意不符，但它却是另一个重要的完全背包问题最简捷的解决方案，故学习只用一维数组解01背包问题是十分必要的。

【例1】 0/1背包

【问题描述】

一个旅行者有一个最多能用m公斤的背包，现在有n件物品，它们的重量分别是W1，W2，...,Wn,它们的价值分别为C1,C2,...,Cn.若每种物品只有一件求旅行者能获得最大总价值。

【输入格式】

- 第一行：两个整数，M(背包容量，M<=200)和N(物品数量，N<=30)；
- 第2..N+1行：每行二个整数Wi,Ci，表示每个物品的重量和价值。

【输出格式】

- 仅一行，一个数，表示最大总价值。

【样例输入】package.in

- 10 4
- 2 1
- 3 3
- 4 5
- 7 9

【样例输出】package.out

- 12

【解法一】设f[i][v]表示前i件物品，总重量不超过v的最优价值，则f[i][v]=max(f[i-1][v-w[i]]+c[i]，f[i-1][v])；f[n][m]即为最优解,给出程序：

```
#include<cstdio>
using namespace std;
const int maxm = 201, maxn = 31;
int m, n;
int w[maxn], c[maxn];
int f[maxn][maxm];

int max(int x,int y)  { x>y?x:y;}    //求x和y最大值

int main(){
    scanf("%d%d",&m, &n);          //背包容量m和物品数量n
    for (int i = 1; i <= n; i++)          //在初始化循环变量部分，定义一个变量并初始化
      scanf("%d%d",&w[i],&c[i]);    //每个物品的重量和价值
    for (int i = 1; i <= n; i++)          // f[i][v]表示前i件物品，总重量不超过v的最优价值
        for (int v = m; v > 0; v--)
            if (w[i] <= v)  f[i][v] = max(f[i-1][v],f[i-1][v-w[i]]+c[i]);
                else  f[i][v] = f[i-1][v];
     printf("%d",f[n][m]);                 // f[n][m]为最优解
     return 0;
}
```

使用二维数组存储各子问题时方便，但当maxm较大时，如maxm=2000时不能定义二维数组f，怎么办，其实可以用一维数组。

【解法二】本问题的数学模型如下：设 f[v]表示重量不超过v公斤的最大价值， 则f[v]=max{f[v]，f[v-w[i]]+c[i]}， 当v>=w[i]，1<=i<=n 。程序如下：

```
#include<cstdio>
using namespace std;

const int maxm = 2001, maxn = 31;
int m, n;
int w[maxn], c[maxn];
int f[maxm];
int main(){
    scanf("%d%d",&m, &n);          //背包容量m和物品数量n
    for (int i=1; i <= n; i++)
        scanf("%d%d",&w[i],&c[i]);  //每个物品的重量和价值

    for (int i=1; i <= n; i++)          //设f(v)表示重量不超过v公斤的最大价值
        for (int v = m; v >= w[i]; v--)
            if (f[v-w[i]]+c[i]>f[v])
                f[v] = f[v-w[i]]+c[i];
printf("%d",f[m]);                      // f(m)为最优解
return 0;
}
```

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。

二、完全背包问题

问题:

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w[i]，价值是c[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*w[i]]+k*c[i]|0<=k*w[i]<= v}。

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。

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-w[i]]+c[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。为什么这样一改就可行呢？首先想想为什么01背包问题中要按照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-w[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-w[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-w[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-w[i]]+c[i]}，将这个方程用一维数组实现，便得到了上面的伪代码。

【例9-12】、完全背包问题

【问题描述】

设有n种物品，每种物品有一个重量及一个价值。但每种物品的数量是无限的，同时有一个背包，最大载重量为M，今从n种物品中选取若干件(同一种物品可以多次选取)，使其重量的和小于等于M，而价值的和为最大。

【输入格式】

第一行：两个整数，M(背包容量，M<=200)和N(物品数量，N<=30)；

第2..N+1行：每行二个整数Wi,Ci，表示每个物品的重量和价值。

【输出格式】

仅一行，一个数，表示最大总价值。

【样例输入】knapsack.in

```
10  4
2  1
3  3
4  5
7  9
```

【样例输出】knapsack.out

```
max=12
```

【解法一】

设f[i][v]表示前i件物品，总重量不超过v的最优价值，则f[i][v]=max(f[i][v-w[i]]+c[i]，f[i-1][v])；f[n][m]即为最优解。

【参考程序】

```
#include<cstdio>
using namespace std;

const int maxm = 201, maxn = 31;
int m, n;
int w[maxn], c[maxn];
int f[maxn][maxm];
int main()
{
   scanf("%d%d",&m, &n);            //背包容量m和物品数量n
   for (int i = 1; i <= n; i++)
      scanf(“%d%d”,&w[i],&c[i]);    //每个物品的重量和价值
   for (int i = 1; i <= n; i++)     //f[i][v]表示前i件物品，总重量不超过v的最优价值
      for (int v = 1; v <= m; v++)
         if (v < w[i])  f[i][v] = f[i-1][v];
         else
            if (f[i-1][v] > f[i][v-w[i]]+c[i])  f[i][v] = f[i-1][v];
            else f[i][v] = f[i][v-w[i]]+c[i];
   printf("max=%d",f[n][m]);          // f[n][m]为最优解
   return 0;
}
```

【解法二】

本问题的数学模型如下：

设 f(v)表示重量不超过v公斤的最大价值， 则 f(v)=max{f(v)，f(v-w[i])+c[i]}（v>=w[i]，1<=i<=n）。

【参考程序】

```
#include<cstdio>
using namespace std;
const int maxm=2001,maxn=31;
int n,m,v,i;
int c[maxn],w[maxn];
int f[maxm];

int main()
{
   scanf("%d%d",&m,&n);          //背包容量m和物品数量n
   for(i=1;i<=n;i++)
      scanf("%d%d",&w[i],&c[i]);
   for(i=1;i<=n;i++)
      for(v=w[i];v<=m;v++)          //设 f[v]表示重量不超过v公斤的最大价值
         if(f[v-w[i]]+c[i]>f[v])  f[v]=f[v-w[i]]+c[i];
   printf("max=%d\n",f[m]);        // f[m]为最优解
   return 0;
}
```

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w[i]<=w[j]且c[i]>=c[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的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/w[i]件，于是可以把第i种物品转化为V/w[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w[i]*2^k、价值为c[i]*2^k的若干件物品，其中k满足w[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/w[i])+1)件物品，是一个很大的改进。

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。

三、多重背包问题

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w[i]，价值是c[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*w[i]]+ k*c[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(注意：这些系数已经可以组合出1~n[i]内的所有数字)。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(logn[i])种物品，将原问题转化为了复杂度为O(V*∑logn[i])的01背包问题，是很大的改进。

【例3】庆功会

【问题描述】

- 为了庆贺班级在校运动会上取得全校第一名成绩，班主任决定开一场庆功会，为此拨款购买奖品犒劳运动员。期望拨款金额能购买最大价值的奖品，可以补充他们的精力和体力。

【输入格式】

- 第一行二个数n(n<=500)，m(m<=6000)，其中n代表希望购买的奖品的种数，m表示拨款金额。
- 接下来n行，每行3个数，v、w、s，分别表示第I种奖品的价格、价值（价格与价值是不同的概念）和购买的数量（**买0件到s件均可**），其中v<=100，w<=1000，s<=10。

【输出格式】

- 第一行：一个数，表示此次购买能获得的最大的价值（注意！不是价格）。

【输入样例】

- 5 1000
- 80 20 4
- 40 50 9
- 30 50 7
- 40 30 6
- 20 20 1

【输出样例】

- 1040

【解法一】朴素算法

【参考程序】

```
#include<cstdio>
using namespace std;

int v[6002], w[6002], s[6002];
int f[6002];
int n, m;

int max(int x,int y)  {
    if (x < y) return y;
      else return x;
}

int main(){
    scanf("%d%d",&n,&m);
    for (int i = 1; i <= n; i++)
        scanf("%d%d%d",&v[i],&w[i],&s[i]);
    for (int i = 1; i <= n; i++)
       for (int j = m; j >= 0; j--)
          for (int k = 0; k <= s[i]; k++){
               if (j-k*v[i]<0) break;
               f[j] = max(f[j],f[j-k*v[i]]+k*w[i]);
           }
    printf("%d",f[m]);
    return 0;
}
```

【解法二】进行二进制优化，转换为01背包

【参考程序】

```
#include<cstdio>
int v[10001],w[10001];
int f[6001];
int n,m,n1;
int max(int a,int b){
    return a>b?a:b;                          //这句话等于：if (a>b) return a; else return b;
}
int main(){
    scanf("%d%d",&n,&m);
    for(int i=1;i<=n;i++){
        int x,y,s,t=1;
        scanf("%d%d%d",&x,&y,&s);
        while (s>=t) {
            v[++n1]=x*t;                     //相当于n1++;  v[n1]=x*t;
            w[n1]=y*t;
            s-=t;
            t*=2;
        }
        v[++n1]=x*s;
        w[n1]=y*s;                           //把s以2的指数分堆：1，2，4，…，2^(k-1)，s-2^k+1,
    }
```

```
 for(int i=1;i<=n1;i++)
        for(int j=m;j>=v[i];j--)
            f[j]=max(f[j],f[j-v[i]]+w[i]);
    printf("%d\n",f[m]);
    return 0;
}
```

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*∑logn[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。

四、混合三种背包问题

问题

如果将01背包、完全背包、多重背包混合起来。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01背包和完全背包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-w[i]]+c[i]};
    else if 第i件物品是完全背包
        for v=0..V
            f[v]=max{f[v],f[v-w[i]]+c[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，用多重背包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。

【例4】混合背包

【问题描述】

- 一个旅行者有一个最多能用V公斤的背包，现在有n件物品，它们的重量分别是W1，W2，...,Wn，它们的价值分别为C1,C2,...,Cn。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）。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。

【输入格式】

- 第一行：二个整数，V(背包容量，V<=200)，N(物品数量，N<=30)；
- 第2..N+1行：每行三个整数Wi,Ci,Pi，前两个整数分别表示每个物品的重量，价值，第三个整数若为0，则说明此物品可以购买无数件，若为其他数字，则为此物品可购买的最多件数(Pi)。

【输出格式】

- 仅一行，一个数，表示最大总价值。

【样例输入】mix.in

- 10 3
- 2 1 0
- 3 3 1
- 4 5 4

【样例输出】mix.out

- 11

【样例解释】

- 选第一件物品1件和第三件物品2件。

【参考程序】

```
#include<cstdio>
using namespace std;
int m, n;
int w[31], c[31], p[31];
int f[201];

int max(int x,int y) {  return  x>y?x:y; }
int main(){
    scanf("%d%d",&m,&n);
    for (int i = 1; i <= n; i++)
        scanf("%d%d%d",&w[i],&c[i],&p[i]);
    for (int i = 1; i <= n; i++)
        if (p[i] == 0)  {                              //完全背包
            for (int j = w[i]; j <= m; j++)
                f[j] = max(f[j], f[j-w[i]]+c[i]);
        }
        else {
            for (int j = 1; j <= p[i]; j++)           //01背包和多重背包
                for (int k = m; k >= w[i]; k--)
                    f[k] = max(f[k],f[k-w[i]]+c[i]);
        }
    printf("%d",f[m]);
    return 0;
}
```

- 小结
- 有人说，困难的题目都是由简单的题目叠加而来的。这句话是否是公理暂且存之不论，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。

- 五、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c[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]]+c[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。

- 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。

【例5】潜水员

【问题描述】

- 潜水员为了潜水要使用特殊的装备。他有一个带2种气体的气缸：一个为氧气，一个为氮气。让潜水员下潜的深度需要各种的数量的氧和氮。潜水员有一定数量的气缸。每个气缸都有重量和气体容量。潜水员为了完成他的工作需要特定数量的氧和氮。他完成工作所需气缸的总重的最低限度的是多少？
- 例如：潜水员有5个气缸。每行三个数字为：氧，氮的（升）量和气缸的重量：
- 3 36 120
- 10 25 129
- 5 50 250
- 1 45 130
- 4 20 119
- 如果潜水员需要5升的氧和60升的氮则总重最小为249（1，2或者4，5号气缸）。
- 你的任务就是计算潜水员为了完成他的工作需要的气缸的重量的最低值。

【输入格式】

- 第一行有2整数m,n（1<=m<=21,1<=n<=79）。它们表示氧，氮各自需要的量。
- 第二行为整数k（1<=n<=1000）表示气缸的个数。
- 此后的k行，每行包括ai，bi，ci（1<=ai<=21，1<=bi<=79，1<=ci<=800）3整数。这些各自是：第i个气缸里的氧和氮的容量及汽缸重量。

【输出格式】

- 仅一行包含一个整数，为潜水员完成工作所需的气缸的重量总和的最低值。

【参考程序】

```
#include<cstdio>
#include<cstring>                          //初始化memset要用到
using namespace std;
int v, u, k;
int a[1001], b[1001], c[1001];
int f[101][101];

int main(){
    memset(f,127,sizeof(f));               //初始化为一个很大的正整数
    f[0][0] = 0;
    scanf("%d%d%d",&v,&u,&k);
    for (int i = 1; i <= k; i++)
        scanf("%d%d%d",&a[i],&b[i],&c[i]);
    for (int i = 1; i <= k; i++)
      for (int j = v; j >= 0; j--)
        for (int l = u; l >= 0; l--)
        {
            int t1 = j+ a[i],t2 = l + b[i];
            if (t1 > v)  t1 = v;           //若氮、氧含量超过需求，可直接用需求量代换，
            if (t2> u)  t2 = u;            //不影响最优解
            if (f[t1][t2] > f[j][l] + c[i])  f[t1][t2] = f[j][l] + c[i];
        }
    printf("%d",f[v][u]);
    return 0;
  }
```

- 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。

- 六、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w[i]，价值是c[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-w[i]]+c[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-w[i]]+c[i]}
- 注意这里的三层循环的顺序，“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，显然可以对每组中的物品应用完全背包中“一个简单有效的优化”。

【例6】分组背包

【问题描述】

- 一个旅行者有一个最多能用V公斤的背包，现在有n件物品，它们的重量分别是W1，W2，...,Wn，它们的价值分别为C1,C2,...,Cn。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。

【输入格式】

- 第一行：三个整数，V(背包容量，V<=200)，N(物品数量，N<=30)和T(最大组号，T<=10)；
- 第2..N+1行：每行三个整数Wi,Ci,P，表示每个物品的重量，价值，所属组号。

【输出格式】

- 仅一行，一个数，表示最大总价值。

【样例输入】group.**in**

- 10 6 3
- 2 1 1
- 3 3 1
- 4 8 2
- 6 9 2
- 2 8 3
- 3 9 3

【样例输出】group.**out**

- 20

【参考程序】

```
#include<cstdio>
using namespace std;
int v, n, t;
int w[31], c[31];
int a[11][32], f[201];
int main(){
    scanf("%d%d%d",&v,&n,&t);
    for (int i = 1; i <= n; i++){
        int p;
        scanf("%d%d%d",&w[i],&c[i],&p);
        a[p][++a[p][0]] = i;
    }
    for (int k = 1; k <= t; k++)
        for (int j = v; j >= 0; j--)
            for (int i = 1; i <= a[k][0]; i++)
                if (j >= w[a[k][i]]) {
                    int tmp = a[k][i];
                    if (f[j] < f[j-w[tmp]]+c[tmp])
                        f[j] = f[j-w[tmp]]+c[tmp];
                }
    printf("%d",f[v]);
    return 0;
}
```

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。

七、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**分组的背包**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**分组的背包**中的一句话： 可以对每组中的物品应用**完全背包**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-w[i]所有这些值时相应的最大价值f'[0..V-w[i]]。那么这个主件及它的附件集合相当于V-w[i]+1个物品的物品组，其中费用为w[i]+k的物品的价值为f'[k]+c[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-w[i]+1个物品的物品组，就可以直接应用**分组的背包**的算法解决问题了。

- **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，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。

八、背包问题的方案总数

- 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-w[i]]+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。

【例9-17】、货币系统

【问题描述】

给你一个n种面值的货币系统，求组成面值为m的货币有多少种方案。样例：设n=3，m=10，要求输入和输出的格式如下：

【样例输入】money.in

```
3  10                    //3种面值组成面值为10的方案
1                        //面值1
2                        //面值2
5                        //面值5
```

【样例输出】money.out

```
10                       //有10种方案
```

【算法分析1】

设f[j]表示面值为j的最大方案数， 如果f[j-k*a[i]]!=0则f[j]=f[j]+f[j-k*a[i]]，当1<=i<=n，m>=j>= a[i]，1<=k<=j / a[i]。

【参考程序1】

```
#include<cstdio>
int m, n;
int a[1001];
long long f[10001];                    //注意要用long long

int main()
{
   scanf("%d%d",&n,&m);                //n种面值的货币，组成面值为m
   for (int i = 1; i <= n; i++)
      scanf("%d",&a[i]);               //输入每一种面值
   f[0] = 1;
   for (int i = 1; i <= n; i++)
      for (int j = m; j >= a[i]; j--)  //f[j]表示面值为j的总方案数
         for (int k = 1; k <= j / a[i]; k++)
            f[j] += f[j-k*a[i]];
   printf("%lld",f[m]);                // f[m]为最优解
   return 0;
}
```

【算法分析2】

设f[j]表示面值为j的总方案数，如果f[j-a[i]]!=0则f[j]=f[j]+f[j-a[i]],1<=i<=n,a[i]<=j<=m。

【参考程序2】

```
#include<cstdio>
using namespace std;

int n, m;
int a[101];
long long f[10001];

int main(){
   scanf("%d%d",&n,&m);
   for (int i = 1; i <= n; i++)
      scanf("%d",&a[i]);
   f[0] = 1;
   for (int i = 1; i <= n; i++)
      for (int j = a[i]; j <= m; j++)
         f[j] += f[j-a[i]];
   printf("%lld",f[m]);
   return 0;
}
```

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。