/*
背包问题（空间优化）经典代码
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(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。

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。

*/

```
#include<stdio.h>
#include<string.h>
#define MINUSINF 0x80000000
#define MAXN 100
#define MAXV 1000
int max(int a,int b)
{
	return a>b?a:b;
}
//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]，装满与否要求为 full
//算法 1：经典 DP 二维数组解法，时间复杂度及空间复杂度均为 O(nv)
int ZeroOnePack1(int n,int v,int c[],int w[],int full)
{
	int i,j;
	int f[MAXN][MAXV];
	if(full)
	{
		for(i=0;i<=n;i++)
			for(j=0;j<=v;j++)
				f[i][j]=MINUSINF;
		f[0][0]=0;
	}
	else memset(f,0,sizeof(f));
	for(i=1;i<=n;i++)
	{
		for(j=0;j<=v;j++)
		{
			if(j>=c[i])
				f[i][j]=max(f[i-1][j],f[i-1][j-c[i]]+w[i]);
			else f[i][j]=f[i-1][j];
		}
	}
```

```
    if(f[n][v]<0) return -1;
    else return f[n][v];
}
//算法 2：算法 1 的一维数组解法，时间复杂度为 O(nv)，空间复杂度为 O(v)
int ZeroOnePack2(int n,int v,int c[],int w[],int full)
{
    int i,j;
    int f[MAXV];
    if(full)
    {
        f[0]=0;
        for(i=1;i<=v;i++)
            f[i]=MINUSINF;
    }
    else memset(f,0,sizeof(f));
    for(i=1;i<=n;i++)
    {
        for(j=v;j>=0;j--)
        {
            if(j>=c[i])
                f[j]=max(f[j],f[j-c[i]]+w[i]);
        }
    }
    if(f[v]<0) return -1;
    else return f[v];
}
//算法 3：算法 2 的优化，去掉了无必要的判断，时间复杂度为 O(nv)，空间复杂度为 O(v)
int ZeroOnePack3(int n,int v,int c[],int w[],int full)
{
    int i,j;
    int f[MAXV];
    if(full)
    {
        f[0]=0;
        for(i=1;i<=v;i++)
            f[i]=MINUSINF;
    }
    else memset(f,0,sizeof(f));
    for(i=1;i<=n;i++)
    {
        for(j=v;j>=c[i];j--)
        {
            f[j]=max(f[j],f[j-c[i]]+w[i]);
        }
```

```
    }
    if(f[v]<0) return -1;
    else return f[v];
}
//算法 4：算法 3 的常数优化，在 v 较大时优势明显，时间复杂度为 O(nv)，空间复杂度为 O(v)
int ZeroOnePack4(int n,int v,int c[],int w[],int full)
{
    int i,j,sum=0,bound;
    int f[MAXV];
    if(full)
    {
        f[0]=0;
        for(i=1;i<=v;i++)
            f[i]=MINUSINF;
    }
    else memset(f,0,sizeof(f));
    for(i=1;i<=n;i++) sum+=w[i];
    for(i=1;i<=n;i++)
    {
        if(i>1) sum-=w[i-1];
        bound=max(v-sum,c[i]);
        for(j=v;j>=bound;j--)
        {
            f[j]=max(f[j],f[j-c[i]]+w[i]);
        }
    }
    if(f[v]<0) return -1;
    else return f[v];
}
int main()
{
    int i,j;
    int n,v,c[MAXN],w[MAXN];
    while(scanf("%d %d",&n,&v)!=EOF)
    {
        for(i=1;i<=n;i++) scanf("%d %d",&c[i],&w[i]);
        printf("%d\n",ZeroOnePack1(n,v,c,w,0));
    }
    return 0;
}
```

/*完全背包问题*/
/* 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(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]<=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。

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。

```
*/


#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MINUSINF 0x80000000
#define MAXN 100
#define MAXV 1000
int max(int a,int b)
{
	return a>b?a:b;
}
struct Pack
{
	int c;
	int w;
};
int cmp( const void *a ,const void *b)
{
	struct Pack *d=(Pack *)a;
	struct Pack *e=(Pack *)b;
	if(d->c!=e->c) return d->c-e->c;
	else return e->w-d->w;
}
//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]，装满与否要求为 full
//算法 1：基本思路解法，时间复杂度为 O(v*Σ(v/c[i]))，空间复杂度为 O(nv)
int CompletePack1(int n,int v,int c[],int w[],int full)
{
	int i,j,k,current;
	int f[MAXN][MAXV];
	if(full)
	{
		for(i=0;i<=n;i++)
			for(j=0;j<=v;j++)
				f[i][j]=MINUSINF;
		f[0][0]=0;
	}
	else memset(f,0,sizeof(f));
	for(i=1;i<=n;i++)
	{
		for(j=0;j<=v;j++)
		{
```

```
                current=MINUSINF;
                for(k=0;k<=j/c[i];k++)
                {
                    f[i][j]=max(current,f[i-1][j-k*c[i]]+k*w[i]);
                    current=f[i][j];
                }
            }
        }
        if(f[n][v]<0) return -1;
        else return f[n][v];
}
//算法 2：基本思路解法的简单优化，时间复杂度为 O(v*Σ(v/c[i]))，空间复杂度为 O(nv)
int Optimization(int n,int v,int c[],int w[],int selected[])
{
        Pack pack[MAXN];
        int i;
        memset(selected,0,sizeof(selected));
        for(i=0;i<n;i++)
        {
            pack[i].c=c[i+1];
            pack[i].w=w[i+1];
        }
        qsort(pack,n,sizeof(pack[0]),cmp);
        for(i=0;i<n;i++)
        {
            c[i+1]=pack[i].c;
            w[i+1]=pack[i].w;
        }
        for(i=1;i<n;i++)
        {
            if(c[i]!=c[i+1]) continue;
            else selected[i+1]=-1;
        }
}
int CompletePack2(int n,int v,int c[],int w[],int full)
{
        int i,j,k,current;
        int f[MAXN][MAXV];
        int selected[MAXN];
        if(full)
        {
            for(i=0;i<=n;i++)
                for(j=0;j<=v;j++)
                    f[i][j]=MINUSINF;
```

```
            f[0][0]=0;
        }
        else memset(f,0,sizeof(f));
        Optimization(n,v,c,w,selected);
        for(i=1;i<=n;i++)
        {
            if(selected[i]==-1) continue;
            for(j=0;j<=v;j++)
            {
                current=MINUSINF;
                for(k=0;k<=j/c[i];k++)
                {
                    f[i][j]=max(current,f[i-1][j-k*c[i]]+k*w[i]);
                    current=f[i][j];
                }
            }
        }
        if(f[n][v]<0) return -1;
        else return f[n][v];
}
//算法 3：一维数组解法，时间复杂度为 O(nv)，空间复杂度为 O(v)
int CompletePack3(int n,int v,int c[],int w[],int full)
{
        int i,j;
        int f[MAXV];
        if(full)
        {
            f[0]=0;
            for(i=1;i<=v;i++)
                f[i]=MINUSINF;
        }
        else memset(f,0,sizeof(f));
        for(i=1;i<=n;i++)
        {
            for(j=c[i];j<=v;j++)
            {
                f[j]=max(f[j],f[j-c[i]]+w[i]);
            }
        }
        if(f[v]<0) return -1;
        else return f[v];
}
int main()
{
```

```
    int i,j;
    int n,v,c[MAXN],w[MAXN];
    while(scanf("%d %d",&n,&v)!=EOF)
    {
        for(i=1;i<=n;i++) scanf("%d %d",&c[i],&w[i]);
        printf("%d\n",CompletePack1(n,v,c,w,0));
    }
    return 0;
}
```