背包问题

背包问题:有多个物品,重量不同、价值不同,以及一个容量有限的背包,选择一些物品装到背包中,问怎样装才能使装进背包的物品总价值最大.根据不同的限定条件,可以把背包问题分为很多种,常见的有下面两种:

(1). 如果每个物体可以切分,称为一般背包问题,用**贪心法**求最优解.比如吃自助餐,在饭量一定的情况下,怎样吃才能使吃到肚子里的最值钱?显然应该从最贵的食物吃起,吃完最贵的再吃第二贵的,这是贪心法.

(2). 如果每个物体不可分割,称为 **0/1 背包问题**.仍以吃自助餐为例,这次的食物都是一份份的,每一份必须吃完.如果最贵的食物一份就超过了你的饭量,那就只好放弃.这种问题无法用贪心法求最优解.

0/1 背包问题

问题导入:有 N 件物品和一个容量是 V 的背包. 每件物品只能使用一次. 第 i 件物品的体积是 v_i, 价值是 w_i. 求解将哪些物品装入背包, 可使这些物品的总体积不超过背包容量, 且总价值最大. 输出最大价值.

分析:设 x_i 表示物品 i 装入背包的情况, 当 $x_i = 0$ 时不装入背包, 当 $x_i = 1$ 时装入背包, 有以下约束条件和目标函数.

约束条件: $\sum_{i=1}^{n} v_i x_i \le V, x_i \in \{0,1\}, 1 \le i \le n$　　　　目标函数: $\max \sum_{i=1}^{n} w_i x_i$

举例: 有 4 个物品, 其体积分别为 2、3、6、5, 其价值分别为 6、3、5、4, 背包的容量为 9.

引进一个 $(N+1)\times(V+1)$ 的二维表 $dp[][]$, 可以把每个 $dp[i][j]$ 都看成一个背包, $dp[i][j]$ 表示把前 i 个物品装入容量为 j 的背包中获得的最大价值, i 为纵坐标, j 为横坐标.

填表按照只装第一个物品、只装前两个物品、只装前三个物品和装前四个物品的顺序. 这是从小问题扩展到大问题的过程.

	背包容量	0	1	2	3	4	5	6	7	8	9
不装	0										
装第1个	1										
装第2个	2										
装第3个	3										
装第4个	4										

步骤 1:只装第一个物品.

由于物品 1 的体积为 2, 所以背包容量小于 2 的都放不进去, 得 $dp[1][0] = dp[1][1] = 0$; 物品 1 的体积等于背包容量, 装进去, 背包的价值等于物品 1 的价值, 得 $dp[1][2] = 6$, 容量大于 2 的背包, 多余的容量用不到, 所以价值和容量 2 的背包一样, 得 $dp[1][j] = 6, 3 \le j \le 9$.

	背包容量	0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
$v_1 = 2, w_1 = 6$	1	0	0	6	6	6	6	6	6	6	6

步骤 2: 只装前两个物品.

	背包容量	0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
$v_1 = 2, w_1 = 6$	1	0	0	6	6	6	6	6	6	6	6
$v_2 = 3, w_2 = 3$	2	0	0	0	0+3						

装物品2

	背包容量	0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
$v_1 = 2, w_1 = 6$	1	0	0	6	6	6	6	6	6	6	6
$v_2 = 3, w_2 = 3$	2	0	0	6	6						

不装物品2

后续步骤: 继续以上过程, 最后得到如下图（图中箭头代表例子）.

	背包容量	0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
$v_1 = 2, w_1 = 6$	1	0	0	6	6	6	6	6	6	6	6
$v_2 = 3, w_2 = 3$	2	0	0	6	6	6	9	9	9	9	9
$v_3 = 6, w_3 = 5$	3	0	0	6	6	6	9	9	9	11	11
$v_4 = 5, w_4 = 4$	4	0	0	6	6	6	9	9	10	11	11

最后答案是 $dp[4][9]$, 即把 4 个物品装到容量为 9 的背包, 最大价值是 11. 其算法复杂度是 $O(NV)$.

输出 0-1 背包方案（扩展） 回头看具体装了哪些物品, 需要倒过来观察:

➢ $dp[4][9] = \max\{dp[3][4] + 4, dp[3][9]\} = dp[3][9]$, 说明没有装物品 4, 用 $x_4 = 0$ 表

示；

- $dp[3][9]=\max\{dp[2][3]+5,dp[2][9]\}=dp[2][3]+5=11$，说明装了物品 3，用 $x_3=1$ 表示；
- $dp[2][3]=\max\{dp[1][0]+3,dp[1][3]\}=dp[1][3]$，说明没有装物品 2，用 $x_2=0$ 表示；
- $dp[1][3]=\max(dp[0][1]+6,dp[0][3])=dp[0][1]+6=6$，说明装了物品 1，用 $x_1=1$ 表示.

	背包容量	0	1	2	3	4	5	6	7	8	9	
	0	0	0	0	0	0	0	0	0	0	0	
$v_1=2,w_1=6$	1	0	0	6	6	6	6	6	6	6	6	$x_1=1$
$v_2=3,w_2=3$	2	0	0	6	6	6	9	9	9	9	9	$x_2=0$
$v_3=6,w_3=5$	3	0	0	6	6	6	9	9	9	11	11	$x_3=1$
$v_4=5,w_4=4$	4	0	0	6	6	6	9	9	10	11	11	$x_4=0$

①二维

(1) 状态 $dp[i][j]$ 表示：前 i 个物品，背包容量 j 下的最优解（最大价值）

当前的状态依赖于之前的状态，可以理解为初始状态 $dp[0][0]=0$ 开始决策，有 N 件物品，则需要 N 次决策，每一次对第 i 件物品的决策，状态 $dp[i][j]$ 不断由之前的状态更新而来.

(2) 当前背包**容量不够（$j<v[i]$）**，没得选，因此前 i 个物品最优解即为前 $i-1$ 个物品最优解，

即状态转移方程为：$dp[i][j]=dp[i-1][j]$

(3) 当前背包容量够，可以选，因此需要决策选与不选第 i 个物品.

选：$f[i][j]=f[i-1][j-v[i]]+w[i]$　　不选：$dp[i][j]=dp[i-1][j]$

决策是如何取得最大价值，因此以上两种情况取 $\max()$.　　代码链接

```
    for(int i=1;i<=n;i++)
        for(int j=0;j<=V;j++){
            if(j<v[i])
                dp[i][j]=dp[i-1][j];
            else dp[i][j]=max(dp[i-1][j],dp[i-1][j-v[i]]+w[i]);
        }
```

②一维

将状态 $dp[i][j]$ 优化到一维 $dp[j]$，实际只需要做一个**等价变形**.

为什么可以这样变形呢？我们定义的状态 $dp[i][j]$ 可以求得任意合法的 i 与 j 最优解，但只**需要求得最终状态 $dp[N][V]$，因此只需要一维的空间来更新状态**

(1) 状态 $dp[j]$ 表示：N 件物品，背包容量 j 下的最优解(最大价值).

(2) 注意**枚举背包容量 j 必须从 V 开始**.

(3) 为什么一维情况下枚举背包容量需要**逆序**？在二维情况下，状态 $dp[i][j]$ 是由上一轮 $i-1$ 的状态得来的，$dp[i][j]$ 与 $dp[i-1][j]$ 是独立的；而优化到一维后，如果仍按正序计算，则有 $dp[\text{较小体积}]$ 更新到 $dp[\text{较大体积}]$，则有可能本应该用第 $i-1$ 轮的状态却用的是第 i 轮的状态.

(4) 例如：一维状态第 i 轮对体积为 3 的物品进行决策，则 $dp[7]$ 由 $dp[4]$ 更新而来，这里的 $dp[4]$ 正确应为 $dp[i-1][4]$，但从小到大枚举 j 这里的 $dp[4]$ 在第 i 轮计算却变成了 $dp[i][4]$. 当逆序枚举背包容量 j 时，求 $dp[7]$ 同样由 $dp[4]$ 更新，但由于逆序，这里的 $dp[4]$ 还没有在第 i 轮计算，所以此时实际计算的 $dp[4]$ 仍然是 $dp[i-1][4]$.

(5) 一维情况**正序更新**状态 $dp[j]$ 需要用到前面计算的状态已经**被污染**，逆序则不会造成问题. 状态转移方程为：$dp[j]=\max(dp[j],dp[j-v[i]]+w[i])$.

```
    for(int i=1;i<=n;i++)
        for(int j=V;j>=0;j--){
            if(j<v[i])
                dp[j]=dp[j];
            else dp[j]=max(dp[j],dp[j-v[i]]+w[i]);
        }
```

实际上，只有当枚举的背包容量 $\geq v[i]$ 时才会更新状态，可修改终止条件进一步优化.

```
    for(int i=1;i<=n;i++)
        for(int j=V;j>=v[i];j--)
            dp[j]=max(dp[j],dp[j-v[i]]+w[i]);
```

③优化输入

当处理数据时，是一个物品一个物品，一个一个体积和价值的枚举，因此可以不必开两个

数组记录体积和价值, 可**边输入边处理**.

```
    for(int i=1;i<=n;i++){
        int v,w;
        scanf("%d %d",&v,&w);
        for(int j=V;j>=v;j--)
            dp[j]=max(dp[j],dp[j-v]+w);
    }
```

完全背包问题

问题导入:有 N 种物品和一个容量是 V 的背包,每种物品都**有无限件可用**.第 i 个物品的体积是 v_i,价值是 w_i.求解将哪些物品装入背包,可使这些物品的总体积不超过背包容量,且总价值最大.输出最大价值.

分析:思路同 0/1 背包问题,区别在于 0/1 背包对于每种物品只有选或不选,而完全背包则对于每种物品可以多次选择.

代码链接

①暴力做法——时间复杂度为 $O(n^3)$

状态 $dp[i][j]$ 表示(同 0/1 问题):前 i 个物品,背包容量 j 下的最优解(最大价值)

每一轮循环 i 都可以看作是对第 i 件物品的决策——选择多少个(范围 $0 \sim \left\lfloor \frac{j}{v_i} \right\rfloor$)第 i 个物品.

稍微不同的是完全背包允许多次选择一次物品,计算状态方程需要枚举选择第 i 个物品.

```
    for(int i=1;i<=n;i++)
        for(int j=0;j<=V;j++){
            int amount=j/v[i];
            for(int k=0;k<=amount;k++)
                dp[i][j]=max(dp[i][j],dp[i-1][j-k*v[i]]+k*w[i]);
        }
```

②优化时间——时间复杂度为 $O(n^2)$.

实际上,在计算状态方程时不必多一个循环去单独枚举选择第 i 个物品的个数.
状态转移方程的推导过程:

$$dp[i][j] = \max\{dp[i-1][j], dp[i-1][j-v_i]+w_i, dp[i-1][j-2\times v_i]+2\times w_i, \cdots, dp[i-1][j-\left\lfloor \frac{j}{v_i} \right\rfloor \times v_i]+\left\lfloor \frac{j}{v_i} \right\rfloor \times w_i\}$$

上述状态转移方程可以理解为:前 i 个物品背包容量 j 的最优解 $dp[i][j]$,而前 $i-1$ 个物品的最优解 $dp[i-1][j]$ 在上一轮循环中都已计算完毕,现在只需判断选择几个第 i 种物品得到的价值最大.

将 j 变为 $j-v_i$,则有:

$$dp[i][j-v_i] = \max\{dp[i-1][j-v_i], dp[i-1][j-2\times v_i]+w_i, dp[i-1][j-3\times v_i]+2\times w_i, \cdots, dp[i-1][j-\left\lfloor \frac{j}{v_i} \right\rfloor \times v_i]+(\left\lfloor \frac{j}{v_i} \right\rfloor -1)\times w_i\}$$

由以上两个式子可以得到状态转移方程：

$$dp[i][j]=\max\{dp[i-1][j],dp[i][j-v_i]+w_i\}$$

此处**枚举体积 j 应从小到大，在计算 $dp[i][j]$ 时，$dp[i][\text{较大体积}]$ 总是由 $dp[i][\text{较小体积}]$ 更新而来**.

```
    for(int i=1;i<=n;i++)
        for(int j=0;j<=V;j++){
            dp[i][j]=dp[i-1][j];
            if(j>=v[i])
                dp[i][j]=max(dp[i][j],dp[i][j-v[i]]+w[i]);
                //dp[i][j]=max(dp[i-1][j],dp[i][j-v[i]]+w[i]);
        }
```

③一维

状态转移方程为 $dp[j]=\max(dp[j],dp[j-v_i]+w_i)$（状态方程同 0/1 背包，但枚举的方向是**从小到大**，其次 j 可以从 v_i 开始枚举）.

```
    for(int i=1;i<=n;i++)
        for(int j=v[i];j<=V;j++)
            dp[j]=max(dp[j],dp[j-v[i]]+w[i]);
```

④优化输入

同 0/1 背包，也可边输入边处理

```
    for(int i=1;i<=n;i++){
        int v,w;
        scanf("%d %d",&v,&w);
        for(int j=v;j<=V;j++)
            dp[j]=max(dp[j],dp[j-v]+w);
    }
```

多重背包问题

问题导入：有 N 种物品和一个容量是 V 的背包. 第 i 种物品**最多有 s_i 件**, 每件体积是 v_i, 价值是 w_i. 求解将哪些物品装入背包, 可使物品体积总和不超过背包容量, 且价值总和最大. 输出最大价值.

代码链接

分析：

①参照完全背包, 无非就**物品个数受到限制**——时间复杂度为 $O(n\times V\times s)$

```
for(int i=1;i<=n;i++)
    for(int j=0;j<=V;j++)
        for(int k=0;k<=s[i]&&k<=j/v[i];k++)
            dp[i][j]=max(dp[i][j],dp[i-1][j-v[i]*k]+w[i]*k);
```

注：**不能用完全背包的优化方式来优化多重背包**

完全背包(个数无限, 只受背包容积 j 限制, 而每次都是同步的 k 个 v_i)：

$$dp[i][j]=\max\{dp[i-1][j],dp[i-1][j-v_i]+w_i,dp[i-1][j-2\times v_i]+2\times w_i,\cdots,dp[i-1][j-\left\lfloor\frac{j}{v_i}\right\rfloor\times v_i]+\left\lfloor\frac{j}{v_i}\right\rfloor\times w_i\}$$

$$dp[i][j-v_i]=\max\{dp[i-1][j-v_i],dp[i-1][j-2\times v_i]+w_i,dp[i-1][j-3\times v_i]+2\times w_i,\cdots,dp[i-1][j-\left\lfloor\frac{j}{v_i}\right\rfloor\times v_i]+(\left\lfloor\frac{j}{v_i}\right\rfloor-1)\times w_i\}$$

多重背包(受个数限制, 永远最后多一个)：

$$dp[i][j]=\max\{dp[i-1][j],dp[i-1][j-v_i]+w_i,dp[i-1][j-2\times v_i]+2\times w_i,\cdots,dp[i-1][j-s_i\times v_i]+s_i\times w_i\}$$

$$dp[i][j-v_i]=\max\{dp[i-1][j-v_i],dp[i-1][j-2\times v_i]+w_i,dp[i-1][j-3\times v_i]+2\times w_i,$$
$$\cdots,dp[i-1][j-s_i\times v_i]+(s_i-1)\times w_i,dp[i-1][j-v_i-s_i\times v_i]+s_i\times w_i\}$$

完全背包由于对每种物品没有选择个数的限制, 所以只要体积够用就可以一直选, 没有最后一项, 相当于 $dp[i][j]$ 和 $dp[i][j-v_i]$ 中对于第 i 个物品每次多拿的个数 k 是同步的, 容积都是 $j-k\times v_i$.

多重背包也不受容积的限制, 但多重背包每种物品的数量是有范围的, 当容积足够时, $dp[i][j-v_i]$ 一定会多出个 $dp[i-1][j-(s_i+1)\times v_i]+s_i\times w_i$.

知道 $dp[i][j-v_i]$ 的最大值，也知道了最后一项 $dp[i-1][j-(s_i+1)\times v_i]+s_i\times w_i$ 的最大值，但是无法得出前面若干项的最大值，因为是 max 操作：求所有项的最大一项.

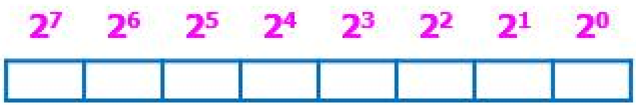
故**不能用完全背包的优化方式来优化多重背包.**

② **二进制优化**——时间复杂度为 $O(nV\log s)$

原理：

将 s 个物品打包成 $\log s$ 个新的物品组，用它们可以凑出从 $0\sim s$ 的任何一个数，不用一个一个凑 $0\sim s$ 中的数. 时间复杂度由 $O(s)$ 优化为 $O(\log s)$.

打包方法： $1+2+4+8+16+\cdots+2^{k-1}+2^k+c=s, c<2^{k+1}$，可凑出 $0\sim s$ 的任何一个数.



假设有一组商品，一共共有 11 个. 十进制数字 11 可以这样表示

$11=1011(B)=0111(B)+(11-0111(B))=0111(B)+0100(B)$

正常背包的思路下，需要枚举 12 次（枚举 $0,1,2,3,\cdots,11$ 个），如果将 11 个商品分别打包成含商品分别打包成含商品个数为 1 个，2 个，4 个，4 个（分别对应 $0001,0010,0100,0100$）的四个"**新的商品**"，将问题转化为 0/1 背包，对于每个"**新的商品**"，都只需要枚举一次，故只需要枚举四次，就可以找出这组商品的最优解，大大降低枚举次数.

这种优化对于大数尤其明显，例如有 1024 个商品，在正常情况下要枚举 1025 次，二进制思想下转化成 0/1 背包只需要枚举 10 次.

优化的合理性的证明：

上面的 $1,2,4,4$ 是可以通过组合来表示出 $0\sim11$ 中任何一个数的，拿 11 证明一下：

首先，11 可以这样分成两个二进制数的组合：

$11=1011(B)=0111(B)+(11-0111(B))=0111(B)+0100(B)$，其中 0111 通过枚举这三个 1 的取或不取（即对 $0001(B),0010(B),0100(B)$ 的组合），可以表示十进制数 $0\sim7$，$0\sim7$ 的枚举再组合上 $0100(B)$（即十进制的 4），可以表示十进制数 $0\sim11$，其他情况也可以证明.

```
#include<iostream>
#include<cstring>
#include<algorithm>
```

```
#define MAXN 2010

using namespace std;

int n,V,cnt;
int v[10*MAXN],w[10*MAXN];
int dp[MAXN];

int main(){
    scanf("%d %d",&n,&V);
    for(int i=1;i<=n;i++){
        int temp_v,temp_w,s;
        scanf("%d %d %d",&temp_v,&temp_w,&s);
        for(int k=1;k<=s;k*=2){
            s-=k;
            v[++cnt]=k*temp_v;
            w[cnt]=k*temp_w;
        }
        if(s>0)
            v[++cnt]=s*temp_v,w[cnt]=s*temp_w;
    }

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

③**单调队列**(不讲)——时间复杂度为 $O(nV)$ 代码链接

回顾之前的状态转移方程:

$dp[i][j]$ 表示将前 i 种物品放入容量为 j 的背包中所得到的最大价值

$$dp[i]][j] = \max\{\text{不放物品}i,\text{放入1个物品}i,\text{放入2个物品}i,\cdots,\text{放入}k\text{个物品}i\}$$

其中 k 满足: $k \le s_i \text{且} j \ge k \times v_i$

$$dp[i][j] = \max\{dp[i-1][j], dp[i-1][j-v_i]+w_i, dp[i-1][j-2\times v_i]+2\times w_i, \cdots, dp[i-1][j-s_i\times v_i]+s_i\times w_i\}$$

实际上并不需要二维的 dp 的数组，适当调整循环条件，可重复利用 dp 数组来保存上一轮的信息.

令 $dp[j]$ 表示容量为 j 的获得的最大价值

针对每一类物品 i，更新 $dp[V] \to dp[0]$ 的值，最后 $dp[V]$ 是一个全局最优解

$$dp[V] = \max\{dp[V], dp[V-v]+w, dp[V-2v]+2w, dp[V-3v]+3w, \cdots\}$$

将 $dp[0] \to dp[V]$ 写成下面形式：

```
dp[0], dp[v],   dp[2*v],   dp[3*v],   ... , dp[k*v]
dp[1], dp[v+1], dp[2*v+1], dp[3*v+1], ... , dp[k*v+1]
dp[2], dp[v+2], dp[2*v+2], dp[3*v+2], ... , dp[k*v+2]
...
dp[j], dp[v+j], dp[2*v+j], dp[3*v+j], ... , dp[k*v+j]
```

显然 V 一定等于 $k \times v + j$，其中 $0 \le j < v$.

将 dp 数组划分为 j 类，每一类的值，都是在同类之间转换得到的.

即 $dp[k \times v + j]$ 只依赖于 $\{dp[j], d[j+v], dp[j+2\times v], dp[j+3\times v], \cdots, dp[j+k\times v]\}$.

故需要的是 $\{dp[j], d[j+v], dp[j+2\times v], dp[j+3\times v], \cdots, dp[j+k\times v]\}$ 中的最大值，可通过维护一个单调队列来得到结果. 问题转化为 j 个单调队列的问题.

故可以得到：

```
dp[j]    =     dp[j]
dp[j+v]  = max(dp[j] +  w,  dp[j+v])
dp[j+2v] = max(dp[j] + 2w,  dp[j+v] +  w, dp[j+2v])
dp[j+3v] = max(dp[j] + 3w,  dp[j+v] + 2w, dp[j+2v] + w, dp[j+3v])
...
```

这个队列中前面的数，每次都会增加一个 w，故进行一些转换

```
dp[j]    =     dp[j]
dp[j+v]  = max(dp[j], dp[j+v] - w) + w
dp[j+2v] = max(dp[j], dp[j+v] - w, dp[j+2v] - 2w) + 2w
dp[j+3v] = max(dp[j], dp[j+v] - w, dp[j+2v] - 2w, dp[j+3v] - 3w) + 3w
...
```

这样每次入队的是 $dp[j+kv]-kw$.（注意是将 kw 提出最大值操作）

单调队列问题，最重要的两点：

1）维护队列元素的个数，如果不能继续入队，弹出队头元素；

2）维护队列的单调性，即：$尾值 \ge dp[j+k\times v]-k\times w$

本题中，队列中元素的个数应该为 $s+1$ 个，即 $0 \sim s$ 个物品 i

分组背包问题

问题导入:有 N **组物品**和一个容量是 V 的背包. 每组物品有 s_i 个, 同一组内的物品最多只能选一个. 每件物品的体积是 v_{ij}, 价值是 w_{ij}, 其中 i 是组号, j 是组内编号. 求解将哪些物品装入背包, 可使物品总体积不超过背包容量, 且总价值最大. 输出最大价值.

代码链接

分析:

①**二维**:

状态 $dp[i][j]$ 表示从前 i 组中选出容量为 j 的物品放入背包物品的最大价值和.

状态转移方程: $dp[i][j]=\max\begin{cases} dp[i-1][j], \text{不选第}i\text{组的物品} \\ \max\limits_{1\le k\le s_i} dp[i-1,j-v_{ik}]+w_{ik}, \text{选第}i\text{组的某个物品}k \end{cases}$.

```
for(int i=1;i<=n;i++)//枚举第i组
    for(int j=0;j<=V;j++){//枚举体积
        dp[i][j]=dp[i-1][j];//不选
        for(int k=1;k<=s[i];k++)//枚举第i组的第k个物品
            if(j>=v[i][k])
                dp[i][j]=max(dp[i][j],dp[i-1][j-v[i][k]]+w[i][k]);
    }
```

②**一维优化**:

思想同 0/1 背包优化, 注意是逆序枚举体积

```
for(int i=1;i<=n;i++)//枚举第i组
    for(int j=V;j>=0;j--){//枚举体积(思想同01背包)
        for(int k=1;k<=s[i];k++)//枚举第i组的第k个物品
            if(j>=v[i][k])//选第i组的第k个物品
                dp[j]=max(dp[j],dp[j-v[i][k]]+w[i][k]);
    }
```

③ **优化输入**

```
for(int i=1;i<=n;i++){
    scanf("%d",&s[i]);
    for(int j=1;j<=s[i];j++)
        scanf("%d %d",&v[j],&w[j]);
    for(int j=V;j>=0;j--)//枚举体积(思想同01背包)
        for(int k=1;k<=s[i];k++)//枚举第i组的第k个物品
            if(j>=v[k])
                dp[j]=max(dp[j],dp[j-v[k]]+w[k]);
}
```