



浙江财经大学

Zhejiang University Of Finance & Economics



# 背包

信智学院 陈琰宏



# 教学内容



01

动态规划原理

02

01背包问题

03

案例讲解



## 1.3 动态规划问题

---

故事讲到这里先暂停一下，我们重新分析这个故事属于什么问题？

### 典型的动态规划问题、著名的“背包问题”

所谓动态规划，就是把复杂的问题简化成规模较小的子问题，再从简单的子问题自底向上一步一步递推，最终得到复杂问题的最优解。




## 1.3.1 动态规划的基本思想

---

动态规划程序设计是对**解最优化问题**的一种途径、一种方法，而不是一种特殊算法，**没有一个标准的数学表达式和明确清晰的解题方法**。

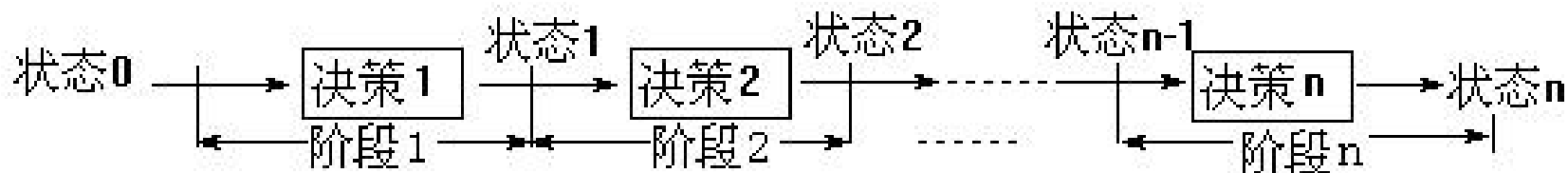
动态规划程序设计往往是针对一种最优化问题，由于各种问题的性质不同，确定最优解的条件也互不相同，因而动态规划的设计方法对不同的问题，有各具特色的解题方法，而不存在一种万能的动态规划算法，可以解决各类最优化问题。

---



## 1.3.1 动态规划的基本思想

和分治法一样，动态规划也是把一个复杂问题分解成相对简单的“子问题”的方式求解，再组合出答案的方法。不过，分治法是将问题划分成一些“独立”的子问题，递归地求解各子问题，然后再合并子问题的解而得到原问题的解，如二分查找、快速排序等问题。与此不同，**动态规划适用于子问题不是独立的情况，也就是子问题包含公共的“子子问题”**。动态规划算法对每个子子问题只求解一次，并且立刻将其结果保存到一张“表”中，从而避免后面每次遇到子子问题时重复去计算。





## 1.3.2 动态规划的基本概念

### 1. 阶段和阶段变量:

用动态规划求解一个问题时，需要将问题的全过程恰当地分成若干个相互联系的阶段，以便按一定的次序去求解。描述阶段的变量称为阶段变量，通常用 $K$ 表示，阶段的划分一般是根据时间和空间的自然特征来划分，同时阶段的划分要便于把问题转化成多阶段决策过程，如例题1中，可将其划分成4个阶段，即 $K = 1, 2, 3, 4$ 。

### 2. 状态和状态变量:

某一阶段的出发位置称为状态，通常一个阶段包含若干状态。一般地，状态可由变量来描述，用来描述状态的变量称为状态变量。如例题1中， $C3$ 是一个状态变量。





## 1.3.2 动态规划的基本概念

### 3. 决策、决策变量和决策允许集合：

在对问题的处理中作出的**每种选择性的行动就是决策**。即从该阶段的每一个状态出发，通过一次选择性的行动转移至下一阶段的相应状态。在每一个阶段的每一个状态中都需要有一次决策，**决策也可以用变量来描述**，称这种变量为决策变量。在实际问题中，决策变量的取值往往限制在某一个范围之内，此范围称为允许决策集合。

### 4. 策略和最优策略

所有阶段依次排列构成问题的全过程。**全过程中各阶段决策变量所组成的有序总体称为策略**。在实际问题中，从决策允许集合中找出最优效果的策略成为最优策略。





## 1.3.2 动态规划的基本概念

---

### 5. 状态转移方程

前一阶段的终点就是后一阶段的起点，对前一阶段的状态作出某种决策，产生后一阶段的状态，这种关系描述了由 $k$ 阶段到 $k+1$ 阶段状态的演变规律，称为状态转移方程。







## 1.3.3 动态规划的基本特点

---


### 子问题:

国王需要根据两个大臣的答案以及第5座金矿的信息才能判断出最多能够开采出多少金子。为了解决自己面临的问题，他需要给别人制造另外两个问题，这两个问题就是子问题。

### 思考动态规划的第一点——**最优子结构**:

国王相信，只要他的两个大臣能够回答出正确的答案，再加上他的聪明的判断就一定能得到最终的正确答案。我们把这种子问题最优时母问题通过优化选择后一定最优的情况叫做“最优子结构”。

---





## 1.3.2 动态规划的基本特点

思考动态规划的第二点——**子问题重叠**：

实际上国王也好，大臣也好，所有人面对的都是同样的问题，即给你一定数量的人，给你一定数量的金矿，让你求出能够开采出来的最多金子数。我们把这种母问题与子问题本质上是同一个问题的情况称为“子问题重叠”。

思考动态规划的第三点——**边界**：

想想如果不存在前面我们提到的那些底层劳动者的话这个问题能解决吗？永远都不可能！我们把这种子问题在一定时候就不再需要提出子子问题的情况叫做边界，没有边界就会出现死循环。






## 1.3.2 动态规划的基本特点

---

思考动态规划的第四点——**子问题独立**：

要知道，当国王的两个大臣在思考他们自己的问题时他们是不会关心对方是如何计算怎样开采金矿的，因为他们知道，国王只会选择两个人中的一个作为最后方案，另一个人的方案并不会得到实施，因此一个人的决定对另一个人的决定是没有影响的。我们把这种一个母问题在对子问题选择时，当前被选择的子问题两两互不影响的情况叫做“子问题独立”。

---



## 1.3.2 动态规划的基本特点

- 最优子结构
- 子问题重叠
- 边界
- 子问题独立



当你发现你正在思考的问题具备这四个性质的话，那么恭喜你，你基本上已经找到了动态规划的方法。



## 1.4 背包问题

---

背包问题(Knapsack problem)是一种组合优化问题,属于动态规划问题。背包问题问题可以描述为:给定一组物品,每种物品都有自己的重量和价格,在限定的总重量内,我们如何选择,才能使得物品的总价格最高。

- 01背包
- 多重背包
- 完全背包





## 2 [7277] 国王的金矿

国王在他的国家发现了 $n$ 座金矿，为了描述方便，我们给他们从1到 $n$ 编号。对于第 $i$ 个金矿，需要投入 $w(i)$ 个的费用，能挖出来 $V(i)$ 个单位的金子。

现在国王想开挖这些金矿，但是最多只有 $M$ 个人力用于投入，问最多可以挖出来多少单位的金子。

输入第一行两个整数，分别为 $N$ 和 $M$ ；接下来 $N$ 行每行两个整数，第 $i+1$ 行为 $w(i)$ 和 $v(i)$ 。

输出一行一个整数，为最多可以挖出来多少单位的金子。

样例输入

5 10  
3 350  
5 500  
5 400  
4 300  
3 200

样例输出

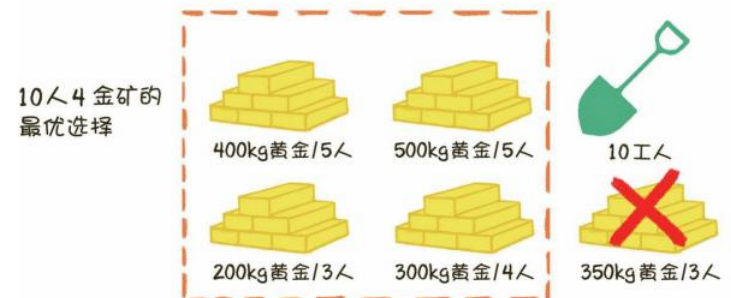
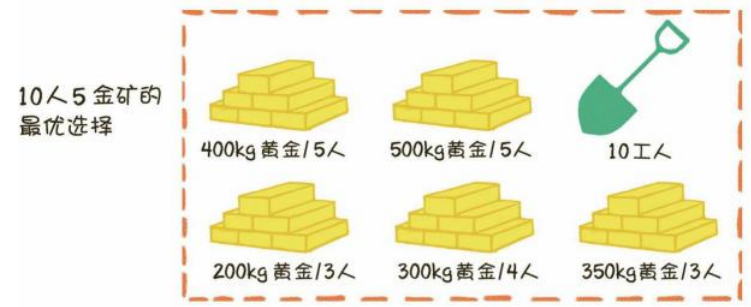
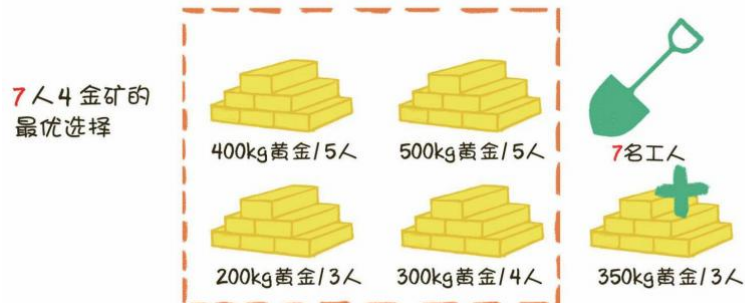
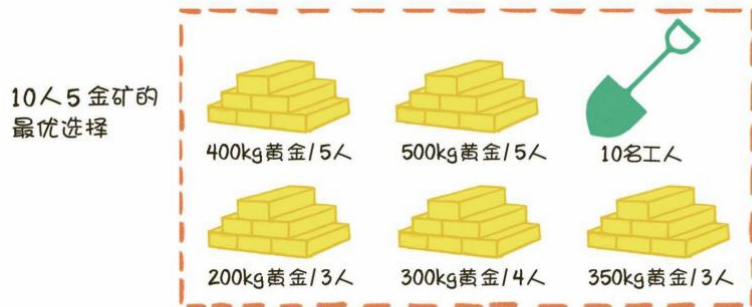
900



# 分析

假设如果最后一个金矿注定不被挖掘，那么问题会转化成10个工人在前4个金矿中做出最优选择；

相应地，假设最后一个金矿一定会被挖掘，那么问题转化成7个工人在前4个金矿中做出最优选择（由于最后一个金矿消耗了3个工人）





# 分析

最后一个金矿到底该不该挖呢？那就要看10个工人在前4个金矿的收益，和7个工人在前4个金矿的收益+最后一个金矿的收益谁大谁小了。

设 $f[i][v]$ 表示前 $i$ 座金矿被 $v$ 个人挖掘后，可以获得的最大价值，那么我们可以很容易分析得出  $f[i][v]$ 的计算方法：

$$f[5][10]=\max(f[4][10],f[4][7]+350);$$

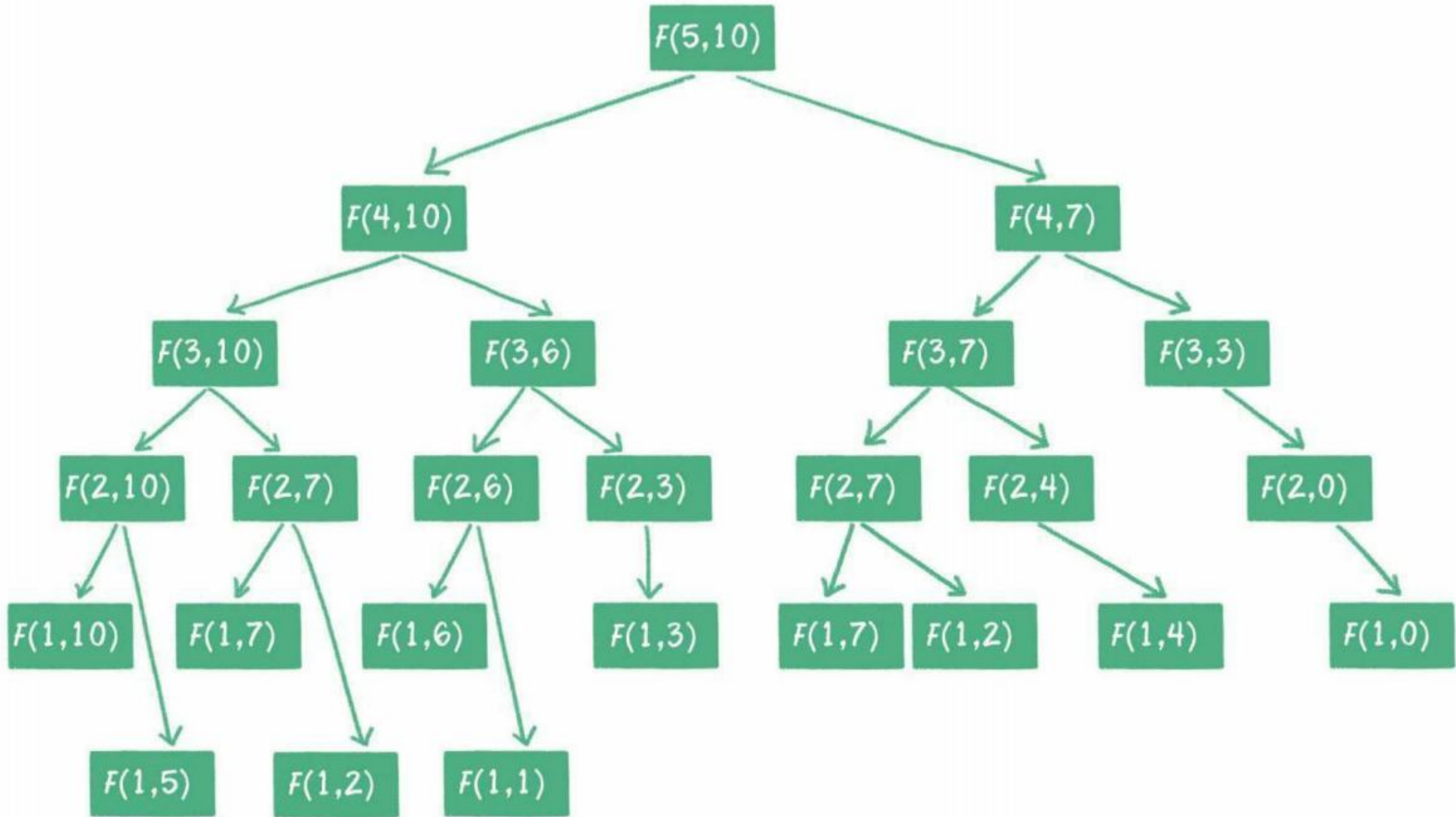
设 $w[i]$ 表示挖掘第 $i$ 座金矿需要的人数，挖掘后能获得的价值是 $c[i]$ ，则

$$f[i][v]=\max\{f[i-1][v],f[i-1][v-w[i]]+c[i]\}。$$





# 分析

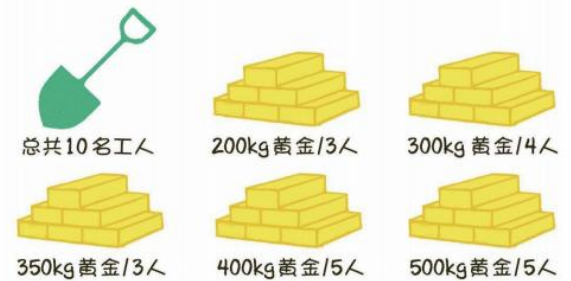
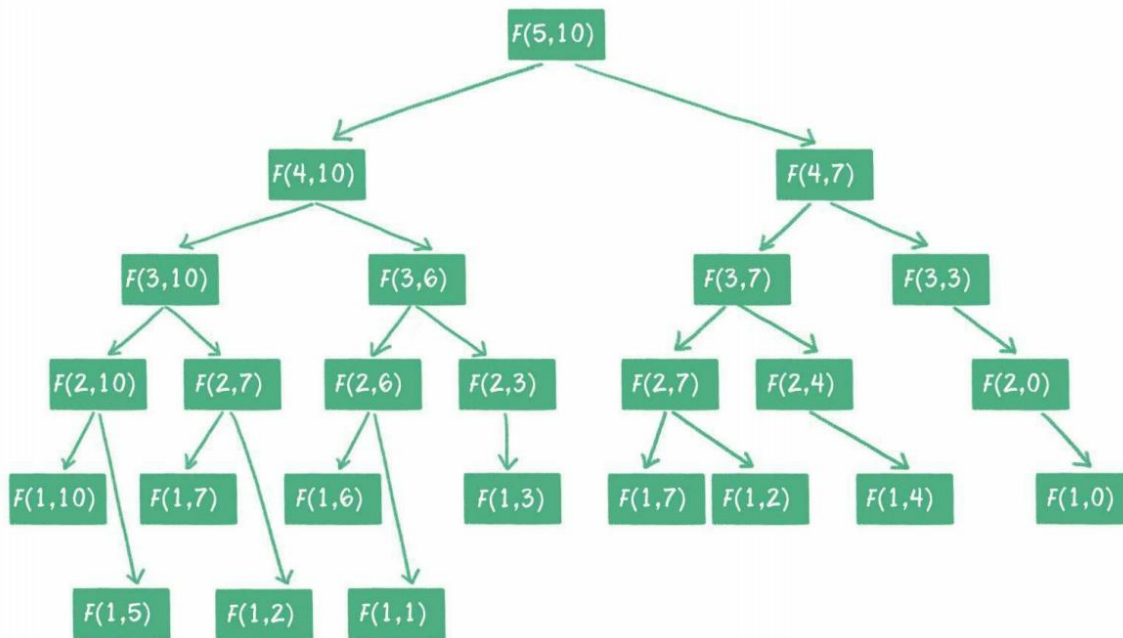


# 分析

同样的道理，对于前4个金矿的选择，我们还可以做进一步简化。

.....

就这样，问题一分为二，二分为四，一直把问题简化成在0个金矿或0个工人时的最优选择，这个收益结果显然是0，也就是问题的边界。





## 1.4.2 01背包问题

**问题描述：**有 $n$ 个物品，它们有各自的重量和价值，现有给定容量的背包，如何让背包里装入的物品具有最大的价值总和？

设有 $n$ 件物品和一个容量为 $V$ 的背包，第 $i$ 件物品的费用 $w[i]$ ，价值是 $c[i]$ ， $f[i][v]$ 表示前 $i$ 件物品放入一个容量为 $v$ 的背包可以获得的最大价值，那么我们可以很容易分析得出  $f[i][v]$ 的计算方法，

(1)  $v < w[i]$ 的情况，这时候背包容量不足以放下第  $i$  件物品，只能选择不拿

$$f[i][v] = f[i-1][v]$$

(2)  $v \geq w[i]$  的情况，这时背包容量可以放下第  $i$  件物品，我们就要考虑拿这件物品是否能获取更大的价值。





## 1.4.2 01背包问题

---

如果拿取， $f[i][v] = f[i-1][v-w[i]] + c[i]$ 。这里的  $f[i-1][v-w[i]] + c[i]$  指的就是考虑了  $i-1$  件物品，背包容量为  $v-w[i]$  时的最大价值，也是相当于为第  $i$  件物品腾出了  $w[i]$  的空间。

如果不拿， $f[i][v] = f[i-1][v]$ ，

究竟是拿还是不拿，自然是比较这两种情况那种价值最大。



## 1.4.2 01背包问题

---

由此可以得到状态转移方程：

**if**( $v \geq w[i]$ )

$f[i][v] = \max(f[i-1][v], f[i-1][v-w[i]] + c[i]);$

**else**

$m[i][v] = f[i-1][v];$

即： $f[i][v] = \max\{f[i-1][v], f[i-1][v-w[i]] + c[i]\}。$





## 2 [7277] 国王的金矿

国王在他的国家发现了 $n$ 座金矿，为了描述方便，我们给他们从1到 $n$ 编号。对于第 $i$ 个金矿，需要投入 $w(i)$ 个的费用，能挖出来 $V(i)$ 个单位的金子。

现在国王想开挖这些金矿，但是最多只有 $M$ 个人力用于投入，问最多可以挖出来多少单位的金子。

输入第一行两个整数，分别为 $N$ 和 $M$ ；接下来 $N$ 行每行两个整数，第 $i+1$ 行为 $w(i)$ 和 $v(i)$ 。

输出一行一个整数，为最多可以挖出来多少单位的金子。

样例输入

5 10  
3 350  
5 500  
5 400  
4 300  
3 200

样例输出

900





# 代码

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int f[205][3005],w[205],c[205];
4  int main(){
5      int n,v;
6      scanf("%d %d",&n,&v);
7      for(int i=1;i<=n;i++){
8          scanf("%d %d",&w[i],&c[i]);
9      }
10     memset(f,0,sizeof(f));
11     for(int i=1;i<=n;i++){
12         for(int j=1;j<=v;j++){
13             if(w[i]<=j)
14                 f[i][j]=max(f[i-1][j],f[i-1][j-w[i]]+c[i]);
15             else
16                 f[i][j]=f[i-1][j];
17         }
18     }
19     printf("%d",f[n][v]);
20     return 0;
21 }
```





# [3894] 大卖场购物车

央视有一个大型娱乐节目—购物街，舞台上模拟超市大卖场，有很多货物，每个嘉宾分配一个购物车，可以尽情地装满购物车，购物车中装的货物价值最高者取胜。假设有  $n$  个物品和 1 个购物车，每个物品  $i$  对应价值为  $v_i$ ，重量  $w_i$ ，购物车的容量为  $W$ （你也可以将重量设定为体积）。每个物品只有 1 件，要么装入，要么不装入，不可拆分。在购物车不超重的情况下，如何选取物品装入购物车，使所装入的物品的总价值最大？最大价值是多少？装入了哪些物品？

输入  $T$ ，表示有  $T$  组数据，输入物品的个数  $n$ ，输入购物车的容量  $W$  ( $1 \leq W \leq 20$ ) 依次输入每个物品的重量  $w$  和价值  $v$ ，用空格分开  
输出装入购物车的最大价值是多少

样例输入

```
1
5
10
2 6 5 3 4 5 2 4 3 6
```

样例输出

```
17
```





# 问题分析

$n$  个物品、物车的容量 $W$ ，每个物品的重量为  $w[i]$ ，价值为  $v[i]$ 。

选若干个物品放入购物车，使价值最大，可表示如下。

$$\text{约束条件: } \begin{cases} \sum_{i=1}^n w_i x_i \leq W \\ x_i \in \{0,1\}, 1 \leq i \leq n \end{cases}$$

$$\text{目标函数: } \max \sum_{i=1}^n v_i x_i$$

该问题就是经典的0-1 背包问题



# 建立最优值的递归式

可以对每个物品依次检查是否放入或者不放入，对于第  $i$  个物品的处理状态：

用  $f[i][j]$  表示前  $i$  件物品放入一个容量为  $j$  的购物车可以获得的最大价值。

- 不放入第  $i$  件物品， $x_i=0$ ，装入购物车的价值不增加。那么问题就转化为“前  $i-1$  件物品放入容量为  $j$  的背包中”，最大价值为  $f[i-1][j]$ 。
- 放入第  $i$  件物品， $x_i=1$ ，装入购物车的价值增加  $v_i$ 。那么问题就转化为“前  $i-1$  件物品放入容量为  $j-w[i]$  的购物车中”，此时能获得的最大价值就是  $f[i-1][j-w[i]]$ ，再加上放入第  $i$  件物品获得的价值  $v[i]$ 。即  $c[i-1][j-w[i]]+ v[i]$ 。



# 建立最优值的递归式

购物车容量不足，肯定不能放入；购物车容量足，我们看放入、不放入哪种情况获得的价值更大。

$$f[i][j] = \begin{cases} f[i-1][j] & , j < w_i \\ \max \{f[i-1][j], f[i-1][j - w[i]] + v[i]\} & , j \geq w_i \end{cases}$$

# 建立最优值的递归式

假设现在有 5 个物品，每个物品的重量为 (2, 5, 4, 2, 3)，价值为 (6, 3, 5, 4, 6)，如图所示。购物车的容量为 10，求在不超过购物车容量的前提下，把哪些物品放入购物车，才能获得最大价值

	1	2	3	4	5		1	2	3	4	5
w[ ]	2	5	4	2	3	v[ ]	6	3	5	4	6

物品的重量和价值

**价值数组** $c = \{8, 10, 6, 3, 7, 2\}$ ,  
**重量数组** $w = \{6, 3, 5, 4, 6\}$ ,  
**背包容量** $j = 10$ 时对应的 $f[i][j]$ 数组

$$f[i][j] = \max\{f[i-1][j], f[i-1][v-w[i]] + c[i]\}.$$

	1	2	3	4	5
w[]	2	5	4	2	3

	1	2	3	4	5
v[]	6	3	5	4	6

初始化： $f[i][j]$ 表示前  $i$  件物品放入一个容量为  $j$  的购物车可以获得的最大价值。初始化  $f[][]$ 数组 0 行 0 列为 0： $c[0][j]=0$ ， $c[i][0]=0$ ，其中  $i=0, 1, 2, \dots, n$ ， $j=0, 1, 2, \dots, W$ 。

$$f[i][j]=\max\{f[i-1][j],f[i-1][v-w[i]]+c[i]\}。$$

f[][]	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	6	6	6	9	9	9	9
3	0	0	6	6	6	6	11	11	11	11	11
4	0	0	6	6	10	10	11	11	15	15	15
5	0	0	6	6	10	12	12	16	16	17	17



# 构造最优解

$f[][]$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	6	6	6	9	9	9	9
3	0	0	6	6	6	6	11	11	11	11	11
4	0	0	6	6	10	10	11	11	15	15	15
5	0	0	6	6	10	12	12	16	16	17	17

$v[]$	1	2	3	4	5
	6	3	5	4	6
$w[]$	1	2	3	4	5
	2	5	4	2	3

首先读取  $f[5][10] > f[4][10]$ , 说明**第 5 个物品装入了购物车**, 即  $j = 10 - w[5] = 7$ ;

去找  $f[4][7] = f[3][7]$ , 说明第 4 个物品没装入购物车;

去找  $f[3][7] > f[2][7]$ , 说明**第 3 个物品装入了购物车**, 即  $j = j - w[3] = 3$ ;

去找  $f[2][3] = f[1][3]$ , 说明第 2 个物品没装入购物车;

去找  $f[1][3] > f[0][3]$ , 说明**第 1 个物品装入了购物车**, 即  $j = j - w[1] = 1$ 。



```
5  int dp[M][maxn]; //dp[i][j] 表示前i个物品放入容量为j购物车
6  int w[M],c[M]; //w[i] 表示第i个物品的重量, v[i] 表示第i个物
7  int main()
8  {
9      int n,v,i,j;
10     cin>>n>>v;
11     for(i=1;i<=n;i++)
12         cin>>w[i]>>c[i];
13     memset(dp,0,sizeof(dp));
14     for(i=1;i<=n;i++)
15         for(j=1;j<=v;j++)
16             if(j<w[i])
17                 dp[i][j]=dp[i-1][j];
18             else
19                 dp[i][j]=max(dp[i-1][j],dp[i-1][j-w[i]]+c[i]);
20
21     cout<<dp[n][v];
22     return 0;
23 }
```

## 4 [2965] 采药

---

辰辰是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”

如果你是辰辰，你能完成这个任务吗？

输入

第一行有两个整数 $T$  ( $1 \leq T \leq 100$ ) 和 $M$  ( $1 \leq M \leq 100$ )，用一个空格隔开， $T$ 代表总共能够用来采药的时间， $M$ 代表山洞里的草药的数目。接下来的 $M$ 行每行包括两个在1到100之间（包括1和100）的整数，分别表示采摘某株草药的时间和这株草药的价值。

输出

包括一行，这一行只包含一个整数，表示在规定的时间内，可以采到的草药的最大总价值。



## 4 [2965] 采药

---

### 【数据规模】

对于30%的数据， $M \leq 10$ ；

对于全部的数据， $M \leq 100$ 。

样例输入

70 3

71 100

69 1

1 2

样例输出

3

经典的01背包问题，将规定时间看作背包容量。采摘每颗草药所花时间可以类比01背包问题中物品的重量，草药价值可以类比01背包中物品的价值。



```
1  #include <iostream>
2  using namespace std;
3  int dp[105][1005]; // 背包结果
4  int v[105]; // 草药价值
5  int w[105]; // 采摘草药所花时间
6  int main()
7  {
8      int t,n;
9      cin>>t>>n;
10     for(int i=1;i<=n;i++)
11         cin>>w[i]>>v[i];
12     for(int i=1;i<=n;i++) // 遍历药品种类
13     {
14         for(int j=1;j<=t;j++) // 将拥有时间从大到小遍历 (类比背包容量)
15             // 如果采摘这颗草药能获得更大收益, 则更新结果
16             if(w[i]<=j)
17                 dp[i][j]=max(dp[i-1][j],dp[i-1][j-w[i]]+v[i]);
18             else
19                 dp[i][j]=dp[i-1][j];
20     }
21     cout<<dp[n][t];
22     return 0;
23 }
24 }
```



# 思考？

---

如何测试数据范围调整为：

$T$  ( $1 \leq T \leq 10000$ ) 和  $M$  ( $1 \leq M \leq 10000$ )

如此定义一个二维数组

`int dp[10000][10000]` 在程序中将内存超限！



# 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。



	0	1	2	3	4	5	6	7	8	9	10
5	0	0	6	6	10	12	12	16	16	17	17

f[][]	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	6	6	6	9	9	9	9
3	0	0	6	6	6	6	11	11	11	11	11
4	0	0	6	6	10	10	11	11	15	15	15
5	0	0	6	6	10	12	12	16	16	17	17

```

7  int main()
8  {
9      int t,n;
10     cin>>t>>n;
11     for(int i=1;i<=n;i++)
12         cin>>w[i]>>v[i];
13     for(int i=1;i<=n;i++) //遍历药品种类
14     {
15         for(int j=t;j>=w[i];j--) //将拥有时间
16             //如果采摘这颗草药能获得更大收益,
17             dp[j]=max(dp[j],dp[j-w[i]]+v[i]);
18     }
19     cout<<dp[t];
20     return 0;
21 }

```

```

1  #include <iostream>
2  #include <algorithm>
3  using namespace std;
4  int dp[1005]; //背包结果
5  int v[105]; //草药价值
6  int w[105]; //采摘草药所花时间

```

# 为什么要倒着推?

f[][]	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	6	6	9	9	9	9	9
3	0	0	6	6	6	6	11	11	11	11	11
4	0	0	6	6	10	10	11	11	15	15	15
5	0	0	6	6	10	12	12	16	16	17	17

The table illustrates the values of a function f over a grid of indices (0 to 5) and (0 to 10). Red arrows point upwards from the value in row i to the value in row i+1 for the same column, indicating that the value in row i+1 depends on the value in row i. Blue arrows point from the value in row i to the value in row i+1 for a different column, indicating that the value in row i+1 also depends on the value in row i from a different column. This demonstrates the need to calculate values from right to left (or bottom to top) to ensure dependencies are resolved.



# 01 背包转换为二维数组详解

价值数组  $C = \{8, 10, 6, 3, 7, 2\}$ ,  
重量数组  $w = \{4, 6, 2, 2, 5, 1\}$ ,

当  $i=1$  时,  $f[11]$  的值, 如下表所示:

$$f[11] = \max\{f[11], f[11-4]+8\} = \max\{f[12], f[7]+8\} = \max\{0, 0+8\} = 8$$

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	0	0	0	0	8	8

... ..

$$f[4] = \max\{f[4], f[4-4]+8\} = \max\{f[4], f[0]+8\} = \max\{0, 0+8\} = 8$$

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	8	8	8	8	8	8	8	8







价值数组  $C = \{8, 10, 6, 3, 7, 2\}$ ,  
重量数组  $w = \{4, 6, 2, 2, 5, 1\}$ ,

当  $i=1$  时,  $f[3]$  的值, 如下表所示:

$v < w[i]$   $f[v]$  不更新, 等于原来的值  $f[3]=0$ , 同理,  $f[2]=f[1]=0$

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	8	8	8	8	8	8	8	8

当  $i=2$  时,  $f[12]$  的值, 如下表所示:

$$\begin{aligned} f[12] &= \max \{f[12], f[12-6]+10\} = \max \{f[12], f[8]+10\} \\ &= \max \{8, 8+10\} = 18 \end{aligned}$$

更新如下:

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	8	8	8	8	8	8	8	<b>18</b>





价值数组  $C = \{8, 10, 6, 3, 7, 2\}$ ,  
重量数组  $w = \{4, 6, 2, 2, 5, 1\}$ ,

当  $i=2$  时,  $f[11]$  的值, 如下表所示:

$$\begin{aligned} f[11] &= \max \{f[11], f[11-6]+10\} = \max \{f[12], f[5]+10\} \\ &= \max \{8, 8+10\} = 18 \end{aligned}$$

更新如下:

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	8	8	8	8	8	8	18	18

$f[10]$  的值, 如下表所示:

$$\begin{aligned} f[10] &= \max \{f[10], f[10-6]+10\} = \max \{f[12], f[4]+10\} \\ &= \max \{8, 8+10\} = 18 \end{aligned}$$

更新如下:

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	8	8	8	8	8	18	18	18





价值数组  $C = \{8, 10, 6, 3, 7, 2\}$ ,  
重量数组  $w = \{4, 6, 2, 2, 5, 1\}$ ,

当  $i=2$  时,  $f[9]$  的值, 如下表所示:

$$\begin{aligned} f[9] &= \max \{f[9], f[9-6]+10\} = \max \{f[9], f[3]+10\} \\ &= \max \{8, 0+10\} = 10 \end{aligned}$$

更新如下:

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	8	8	8	8	<b>10</b>	<b>18</b>	<b>18</b>	<b>18</b>

... ..

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	8	<b>10</b>	<b>10</b>	<b>10</b>	<b>10</b>	<b>18</b>	<b>18</b>	<b>18</b>

$w[i] > 5$   $f[5] = f[5]$ , 不更新, 第二轮结束:

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	8	10	10	10	10	18	18	18





价值数组  $C = \{8, 10, 6, 3, 7, 2\}$ ,  
重量数组  $w = \{4, 6, 2, 2, 5, 1\}$ ,

当  $i=3, 4, 5$  可以得到如下更新的表格:

	1	2	3	4	5	6	7	8	9	10	11	12
5	0	6	6	9	9	14	16	17	17	19	21	24

**当  $i=6$  时**,  $f[12]$  的值, 如下表所示:

$$f[12] = \max\{f[12], f[12-1]+2\} = \max\{f[12], f[11]+2\} = \max\{24, 21+2\} = 24$$

$$f[11] = \max\{f[11], f[11-1]+2\} = \max\{f[11], f[10]+2\} = \max\{21, 19+2\} = 21$$

$$f[10] = \max\{f[10], f[10-1]+2\} = \max\{f[10], f[9]+2\} = \max\{19, 17+2\} = 19$$

$$f[9] = \max\{f[9], f[9-1]+2\} = \max\{f[9], f[8]+2\} = \max\{17, 17+2\} = 19$$

	1	2	3	4	5	6	7	8	9	10	11	12
6	0	6	6	9	9	14	16	17	19	19	21	24





价值数组  $C = \{8, 10, 6, 3, 7, 2\}$ ,  
重量数组  $w = \{4, 6, 2, 2, 5, 1\}$ ,

---

	1	2	3	4	5	6	7	8	9	10	11	12
6	0	6	6	9	9	14	16	17	19	19	21	24

当  $i=6$  时,  $f[3]$  的值, 如下表所示:

$$f[3] = \max\{f[3], f[3-1] + 2\} = \max\{f[3], f[2] + 2\} = \max\{6, 2+2\} = 8$$

	1	2	3	4	5	6	7	8	9	10	11	12
6	0	6	8	9	9	14	16	17	19	19	21	24

$$f[2] = \max\{f[2], f[2-1] + 2\} = \max\{f[2], f[1] + 2\} = \max\{6, 0+2\} = 6$$

$$f[1] = \max\{f[1], f[0] + 2\} = \max\{0, 2\} = 2$$

	1	2	3	4	5	6	7	8	9	10	11	12
6	2	6	8	9	9	14	16	17	19	19	21	24

---





# 考虑顺推的情况

价值数组  $C = \{8, 10, 6, 3, 7, 2\}$ ,  
重量数组  $w = \{4, 6, 2, 2, 5, 1\}$ ,  
背包容量  $V = 12$  时对应的  $f[i][v]$  数组。

初始化 `memset(f, 0, sizeof(f))`, 把  $f[v]$  的每个值初始化为 0.

	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0

当  $v=1, 2, 3$  时,  $v < w[i]$ ,  $f[1]$   $f[2]$   $f[3]$  的值为 0

$$f[4] = \max\{f[4], f[4-4]+8\} = \max\{f[4], f[0]+8\} = \max\{0, 0+8\} = 8$$

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	0	0	0	0	0	0	0	0

# 考虑顺推的情况



价值数组  $C = \{8, 10, 6, 3, 7, 2\}$ ,  
重量数组  $w = \{4, 6, 2, 2, 5, 1\}$ ,  
背包容量  $V = 12$  时对应的  $f[i][V]$  数组。

重复取，  
取了2件

$f[5], f[6], f[7]$  都是8

	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	8	8	8	8	0	0	0	0	0

$$f[8] = \max\{f[8], f[8-4]+8\} = \max\{f[8], f[4]+8\} = \max\{8, 8+8\} = 16$$





# 优化空间复杂度

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。





```

1  #include <iostream>
2  #include <algorithm>
3  using namespace std;
4  int dp[1005]; // 背包结果
5  int v[105]; // 草药价值
6  int w[105]; // 采摘草药所花时间
7  int main()
8  {
9      int t,n;
10     cin>>t>>n;
11     for(int i=1;i<=n;i++)
12         cin>>w[i]>>v[i];
13     for(int i=1;i<=n;i++) // 遍历药品种类
14     {
15         for(int j=t;j>=w[i];j--) // 将拥有时间从大到小遍历 (类比背包容量)
16             // 如果采摘这颗草药能获得更大收益, 则更新结果
17             dp[j]=max(dp[j],dp[j-w[i]]+v[i]);
18     }
19     cout<<dp[t];
20     return 0;
21 }

```

## 5 [3783] 集装箱装载

---

有一批共 $n$ 个集装箱要装上艘载重量为 $c$ 的轮船，其中集装箱 $i$ 的重量为 $w_i$ 。找出一种最优装载方案，将轮船尽可能装满，即在装载体积不受限制的情况下，将尽可能重的集装箱装上轮船。

输入

第一行有2个正整数 $n$ 和 $c$ 。 $n$ 是集装箱数， $c$ 是轮船的载重量。第2行中有 $n$ 个正整数，表示集装箱的重量（ $0 < n < 10000, 0 < c < 32767$ ）。

输出

计算出的最大装载重量输出。

样例输入

5 10

7 2 6 5 4

样例输出

10

# [3539]NASA的食物计划

航天飞机的体积有限,当然如果载过重的物品,燃料会浪费很多钱,每件食品都有各自的体积、质量以及所含卡路里,在告诉你体积和质量的最大值的情况下,请输出能达到的食品方案所含卡路里的最大值,当然每个食品只能使用一次.

输入

第一行两个数体积最大值(<400)和质量最大值(<400)

第二行 一个数 食品总数N(<50).

第三行—第3+N行

每行三个数 体积(<400) 质量(<400) 所含卡路里(<500)

输出

一个数 所能达到的最大卡路里(int范围内)

样例输入

```
320 350 4 160 40 120 80 110 240 220 70 310 40 400 22
```

样例输出

```
550
```

# [3539]NASA的食物计划

---

航天飞机的体积有限,当然如果载过重的物品,燃料会浪费很多钱,每件食品都有各自的**体积、质量**以及所含卡路里,在告诉你**体积和质量的最大值的情况下**,请输出能达到的食品方案所含卡路里的最大值,当然每个食品只能使用一次.

本题是变形的01背包,该题目中体积最大值和质量最大值都属于“背包容量”,即选择物品的花费分为两个方面。结果应用二维数组储存结果,做法上在正常01背包循环中两层循环遍历容量即可。

# [3539]NASA的食物计划

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int f[505][505],c[55],w[55],v[55];
4  int main(){
5      int n,v1,m;
6      cin>>v1>>m>>n; // 体积最大值和质量最大值
7      for(int i=1;i<=n;i++)
8          // 体积(<400) 质量(<400) 所含卡路里(<500)
9          cin>>v[i]>>w[i]>>c[i];
10     for(int i=1;i<=n;i++)
11         for(int j=v1;j>=v[i];j--) // 体积
12             for(int k=m;k>=w[i];k--) // 质量
13                 f[j][k]=max(f[j][k],f[j-v[i]][k-w[i]]+c[i]);
14     cout<<f[v1][m];
15
16     return 0;
17 }
```

## [6922] zb的生日西瓜

今天是阴历七月初五，acm队员zb的生日。zb正在和C小加、never在武汉集训。他想给这两位兄弟买点什么庆祝生日，经过调查，zb发现C小加和never都很喜欢吃西瓜，而且一吃就是一堆的那种，zb立刻下定决心买了一堆西瓜。当他准备把西瓜送给C小加和never的时候，遇到了一个难题，never和C小加不在一块住，只能把西瓜分成两堆给他们，为了对每个人都公平，他想要两堆的重量之差最小。每个西瓜的重量已知，你能帮帮他么？

输入

多组测试数据（ $\leq 1500$ ）。数据以EOF结尾

第一行输入西瓜数量N ( $1 \leq N \leq 20$ )

第二行有N个数， $W_1, \dots, W_n$  ( $1 \leq W_i \leq 100000$ )分别代表每个西瓜的重量

输出

输出分成两堆后的质量差

样例输入

5 5 8 13 27 14

样例输出

---

把两堆西瓜分成差值最小的两堆, 就可以把问题转化为01背包问题。

把背包的最大容量设为西瓜的总质量的二分之一, 然后计算背包最大能装多少。

```
for(int i=1;i<=n;i++){
    for(int j=sum/2;j>=a[i];j--){
        dp[j]=max(dp[j],dp[j-a[i]]+a[i]);
    }
}
```

# [6922] zb的生日西瓜

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int a[10010]; //每个习惯的总量
4  int dp[110010];
5  //dp[v]表示在体积v下能获得的最大西瓜重量
6  int main(){
7      int n, sum=0;
8      while(cin>>n){
9          sum=0;
10         memset(dp,0,sizeof(dp));
11         for(int i=1;i<=n;i++){
12             cin>>a[i]; //计算西瓜的总重量
13             sum=sum+a[i];
14         }
15         for(int i=1;i<=n;i++){
16             for(int j=sum/2;j>=a[i];j--){
17                 dp[j]=max(dp[j],dp[j-a[i]]+a[i]);
18             }
19         }
20         cout<<sum-2*dp[sum/2]<<endl;
21     }
22     return 0;
23 }
```



```

1  #include <iostream>
2  #include <cstdio>
3  #include <string.h>
4  using namespace std;
5  int a[21], v, n, m;
6  void dfs(int i, int ans)
7  {
8      if (ans>v) return; //超过一半退出
9      if (i>n)
10     { //判断完毕
11         if (m<ans) m = ans;
12         return;
13     } //对于每个西瓜两种选择, 取和不取
14     dfs(i+1, ans+a[i]);
15     dfs(i+1, ans);
16 }

```

```

18 int main()
19 {
20     int sum;
21     while (~scanf("%d", &n))
22     {
23         sum = 0;
24         for (int i=1; i<=n; i++)
25         {
26             scanf("%d", &a[i]);
27             sum += a[i];
28         }
29         v = sum/2; //把一半体积作为边界
30         m = 0;
31         dfs(1, 0);
32         cout<<sum-2*m<<endl;
33     }
34     return 0;
35 }

```

## [3785] 贪婪戈尔曼

从前有2只狗，大的叫大狗，小的叫小狗，它们2个合起来就是狗儿们，使用英语的人把它们写作Girلمان，传来传去，到最后大家决定叫它们格尔曼。它们的叫声很特别，但是它们十分吝啬它们的叫声，你为了听到它们的叫声，决定买狗饼干送给它们吃，不同种类的饼干能让它们叫的次数不一样，同一块饼干对于大小 格尔曼的效果也不一样。它们很贪婪，如果你只给其中一只格尔曼吃狗饼干或者给两只格尔曼吃的不一样，有一只就会不高兴，因此你买狗饼干的时候总要两块两块 地买，而且现在每类饼干也只有2块（想要多的也没得）。现在不是流行节约型社会吗？因此你也不能浪费，你要求的是在满足你要听格尔曼叫声次数要求的情况（两只格尔曼实际叫的次数都不小于你的要求即可）下的最小花费是多少。

输入

输入文件的第一行为3个整数 $n$ 、 $s$ 、 $b$ ，分别表示狗饼干的类数、你想听到的小格尔曼的叫声次数和大格尔曼的叫声次数，接下来有 $n$ 行，第 $i+1$ 行有3个整数 $s_i$ 、 $b_i$ 、 $c_i$ ，分别表示第 $i$ 类狗饼干能让小格尔曼叫的次数、能让大格尔曼叫的次数和该类饼干的单价。

## [3785] 贪婪戈尔曼

---

输入

输入文件的第一行为3个整数 $n$ 、 $s$ 、 $b$ ，分别表示狗饼干的类数、你想听到的小格尔曼的叫声次数和大格尔曼的叫声次数，接下来有 $n$ 行，第 $i+1$ 行有3个整数 $s_i$ 、 $b_i$ 、 $c_i$ ，分别表示第 $i$ 类狗饼干能让小格尔曼叫的次数、能让大格尔曼叫的次数和该类饼干的单价。

输出

输出文件只有一个整数，为**满足你的要求情况下的最小花费**。

样例输入

```
5 5 10 (狗饼干的类数、小狗的叫声次数和大狗的叫声次数)
1 2 5 (第i类饼干能让小狗叫的次数、让大狗叫的次数和类饼干的单价)
2 4 10
3 7 8
1 11 36
6 0 18
```

样例输出

```
36
```

---

二维费用的背包问题—01背包，一类饼干最多买两块，但是一买必须买俩，所以是01背包，只不过以前的限制关系都是让在一个最大范围内（给出上限），这次却可以超过上限，只要费用小就可以，这里需要操作一下，超过上限的都让其等于上限，选最小费用的。

5 5 10（狗饼干的种类数、小狗的叫声次数和大狗的叫声次数）  
1 2 5（第i类饼干能让小狗叫的次数、让大狗叫的次数和类饼干的单价）  
2 4 10  
3 7 8  
1 11 36  
6 0 18  
样例输出  
36

```

1  #include<bits/stdc++.h>
2  using namespace std;
3  int si[1010]; //能让小狗叫的次数
4  int bi[1010]; //能让大狗叫的次数
5  int c[1010]; //饼干单价
6  //注意这里要用long long int
7  long long int f[55][55]; //不同想到次数下饼干的最小花费
8  int main ()
9  {
10     int n,s,b;
11     cin>>n>>s>>b;
12     memset(f,0x3f3f3f,sizeof(f)); //初始化花费设置为无限大
13     f[0][0]=0; //初始叫声都为0的情况下花费为0
14     for (int i=1; i<=n; i++)
15         cin>>si[i]>>bi[i]>>c[i];
16     for (int i=1; i<=n; i++) //遍历n种饼干
17         for (int j=s; j>=0; j--) //将想听到小狗的叫声从大到小遍历
18             for (int l=b; l>=0; l--) //将想听到大狗的叫声从大到小遍历
19                 {
20                     //如果当前能得到的叫声(价值)加上饼干能带来的价值如果超过了要求,按要求的算
21                     int x=min(s,j+si[i]); //不能超过s
22                     int y=min(b,l+bi[i]); //不能超过b
23                     if (f[j][l]+c[i]<f[x][y])
24                         f[x][y]=f[j][l]+c[i];
25                 }
26     cout<<2*f[s][b]; //要买两块饼干所以结果要*2
27     return 0;
28 }

```

# [6908] 拔河比赛

一个学校举行拔河比赛，所有的人被分成了两组，每个人必须（且只能够）在其中的一组，要求两个组的人数相差不超过1，且两个组内的所有人体重加起来尽可能地接近。

**【输入格式】** 第一行是一个 $n$ ，表示参加拔河比赛的总人数， $n \leq 100$ ，接下来 $n$ 行表示第 $1 \sim n$ 个人的体重，每个人的体重都是整数（ $1 \leq \text{weight} \leq 450$ ）

**【输出格式】** 包含两个整数，分别是两个组的所有人的体重和，用一个空格隔开，如果这两个数不相等，则把小的放在前面

**【输入样例】**

3  
100  
90  
200

**【输出样例】**

190  
200

---

二维01背包；用 $dp[i][j]$ 代表在选 $j$ 个人时能否凑出体重为 $i$ 的情况，  
=1表示能凑出；=0表示不能凑出  
那么处理出来之后只要从体重和的一半从大到小找到第一个能用一半人凑出的方案即可。

```

1  #include<bits/stdc++.h>
2  using namespace std;
3  const int maxn = 450*105; //最大的体重 100人每人450
4  int dp[105][maxn], a[105], n;
5  //二维01背包; 用dp[i][j]代表在选i个人时能否凑出体重为j的情况,
6  //=1表示能凑出; =0表示不能凑出
7  // 那么处理出来之后只要从体重和的一半从大到小找到
8  //第一个能用一半人凑出的方案即可。
9
10 int main()
11 {
12     scanf("%d", &n);
13     memset(dp, 0, sizeof(dp));
14     int limit1, limit2, sum = 0;
15     for(int i = 1; i <= n; i++)
16         scanf("%d", &a[i]), sum += a[i];
17     limit1 = n/2; // 体积的半数
18     limit2 = n/2 + n%2; //人数的半数

```



```

18     limit2 = n/2 + n%2; //人数的半数
19     dp[0][0] = 1;
20     for(int i = 1; i <= n; i++) // n个物品两个约束条件
21         for(int j = i; j >= 1; j--) //约束条件1, 人数限制
22             for(int v = sum; v >= a[i]; v--) //约束条件2, 重量限制
23                 if(dp[j-1][v-a[i]]) dp[j][v] = 1;
24 //如果在j-1人的情况下能凑出 体重为v-a[i] 则在j人的情况下能凑出 体重为v
25     int temp = 1e9;
26     int k = sum/2+1;
27     while(k--)
28     { //从大到小枚举在两个约束条件下能否凑出体积
29         if(dp[limit1][k] || dp[limit2][k])
30             break;
31     }
32     printf("%d %d\n", k, sum-k);
33     return 0;
34 }

```



## 1.5 完全背包问题

---

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：

$$f[i][v]=\max \{f[i-1][v-k*w[i]]+k*c[i] \mid 0 \leq k*w[i] \leq v\}。$$

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。





# [2820] 完全背包问题

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， $M \leq 200$ ）和 $N$ （物品数量， $N \leq 30$ ）；

第 $2 \dots N+1$ 行：每行二个整数 $W_i, C_i$ ，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】

```
10 4
3 3
4 5
7 9
```

## 【样例输出】

max=12





```
1  #include<bits/stdc++.h>
2  int dp[305],w[35],c[35];
3  int main(){
4      int m,n;
5      cin>>m>>n; // 背包容量m和物品数量n
6      for(int i=1;i<=n;i++){
7          cin>>w[i]>>c[i]; // 每个物品的重量和价值
8      }
9      for(int i=1;i<=n;i++){
10         for(int j=w[i];j<=m;j++){
11             dp[j]=max(dp[j],dp[j-w[i]]+c[i]);
12         }
13     }
14     cout<<"max="<<dp[m];
15     return 0;
16 }
```

# [6917]竞赛总分

---

## 题目描述

学生在我们USACO的竞赛中的得分越多我们越高兴。我们试着设计我们的竞赛以便人们能尽可能的多得分,这需要你的帮助。我们可以从几个种类中选取竞赛的题目,这里的一个"种类"是指一个竞赛题目的集合,解决集合中的题目需要相同多的时间并且能得到相同的分数。你的任务是写一个程序来告诉USACO的职员,应该从每一个种类中选取多少题目,使得解决题目的总耗时在竞赛规定的时间内并且总分最大。输入包括竞赛的时间, $M(1 \leq M \leq 10,000)$ (不要担心,你要到了训练营中才会有长时间的比赛)和 $N$ , "种类"的数目 $1 \leq N \leq 10,000$ 。后面的每一行将包括两个整数来描述一个"种类":第一个整数说明解决这种题目能得的分数( $1 \leq \text{points} \leq 10000$ ),第二整数说明解决这种题目所需的时间( $1 \leq \text{minutes} \leq 10000$ )。你的程序应该确定我们应该从每个"种类"中选多少道题目使得能在竞赛的时间中得到最大的分数。来自任意的"种类"的题目数目可能是任何非负数(0或更多)。计算可能得到的最大分数。



---

输入

第 1 行:  $M, N$ --竞赛的时间和题目"种类"的数目。第 2- $N+1$  行: 两个整数:  
每个"种类"题目的分数和耗时。

输出

单独的一行包括那个在给定的限制里可能得到的最大的分数。

样例输入

300 4

100 60

250 120

120 100

35 20

样例输出

605

---

解题思路：

该题为一道典型的完全背包的题目。与01背包不同，完全背包可以反复的取一件物品。因此在递推的时候，最外层的循环用来控制物品顺序，内层循环用来控制取物品的上限。与01背包不同在于，完全能背包可以反复取一件物品，因此在内层循环上，需要从物品的单件价值开始进行累加，直到背包放不下为止。通过对内层循环对每个物品取一件取两件的状态进行枚举，外层循环控制物品的顺序，最后求得整个问题的最优解

//该题为一道完全背包的题目

```
#include<stdio.h>
```

```
int main()
```

```
{
```

```
    int M,N;
```

```
    int score[10005]={0},time[10005]={0},dp[10005]={0};
```

```
    int i,j;
```

```
    scanf("%d%d",&M,&N);
```

```
    for(i=0;i<N;i++)
```

```
    {
```

```
        scanf("%d%d",&score[i],&time[i]);
```

```
    }
```

```
    for(i=0;i<N;i++)
```

```
    {
```

```
        for(j=time[i];j<=M;j++)//与01背包不同,完全背包可以反复的取一件物品。
```

```
        {
```

```
            if(dp[j]<dp[j-time[i]]+score[i])
```

```
            {
```

```
                dp[j]=dp[j-time[i]]+score[i];
```

```
            }
```

```
        }
```

```
    }
```

```
    printf("%d",dp[M]);
```

```
    return 0;
```

```
}
```

# [2825] 货币系统

---

## 题目描述

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。

输入第一行为 $n$ 和 $m$ 。

输出

样例输入

3 10

1

2

5

样例输出

10

提示

【输入样例】

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

【输出样例】

10 //有10种方案

---

解题思路：

定义 $f[m]$ 为所求的当面值为 $m$ 时的总方案数，遍历所有货币种类然后在循环内遍历当前货币面额到所需面额（可近似理解背包的容量），然后内层不进行判断，因为要求的是有多少种方案，所以直接相加即可。

如果 $m=10$ ，即求 $f[10]$ 。先遍历第一种货币，即求出由第一种货币组成的能够到达 $m=10$ 的种类数。然后遍历第二种货币，即求出包含第二种货币组成的能够到达 $m=10$ 的种类数，两者是累加关系，即

$$f(m) = \sum f_i(m)$$

$i$ 为各种面值的货币

```

3  #include<iostream>
4  using namespace std;
5  long long int f[10000]; // 背包结果
6  long long int a[10000]; // 每种货币的面值
7  int main ()
8  {
9      int n,m;
10     cin>>n>>m;
11     // 初始化结果数组, 当所需面额为0时方案为1
12     f[0]=1;
13     for (int i=1; i<=n; i++)
14         cin>>a[i];
15     for (int i=1; i<=n; i++) // 遍历货币种类
16         for (int j=a[i]; j<=m; j++)
17             f[j]=f[j]+f[j-a[i]];
18     // 从小到大遍历当前货币面额与所需货币面额的差值 (背包容量)
19     // 不同于背包求最大价值, 因为要求有多少种方案
20     // 所以直接相加
21     cout<<f[m];
22     return 0;
23 }

```

# 2335 纪念品

---

小伟突然获得一种超能力，他知道未来  $T$  天  $N$  种纪念品每天的价格。某个纪念品的价格是指购买一个该纪念品所需的金币数量，以及卖出一个该纪念品换回的金币数量。

每天，小伟可以进行以下两种交易无限次：

1. 任选一个纪念品，若手上有足够金币，以当日价格购买该纪念品；
2. 卖出持有的任意一个纪念品，以当日价格换回金币。

每天卖出纪念品换回的金币可以立即用于购买纪念品，当日购买的纪念品也可以当日卖出换回金币。当然，一直持有纪念品也是可以的。

$T$  天之后，小伟的超能力消失。因此他一定会在第  $T$  天卖出所有纪念品换回金币。

小伟现在有  $M$  枚金币，他想要在超能力消失后拥有尽可能多的金币。

---

输入第一行包含三个正整数  $T$ ,  $N$ ,  $M$ , 相邻两数之间以一个空格分开, 分别代表未来天数 $T$ , 纪念品数量  $N$ , 小伟现在拥有的金币数量  $M$ 。

接下来  $T$  行, 每行包含  $N$  个正整数, 相邻两数之间以一个空格分隔。第  $i$  行的 $N$  个正整数分别为  $P_{i,1}, P_{i,2}, \dots, P_{i,N}$ , 其中  $P_{i,j}$  表示第  $i$  天第  $j$  种纪念品的价格。

输出 输出仅一行, 包含一个正整数, 表示小伟在超能力消失后最多能拥有的金币数量。

样例输入

6 1 100

50

20

25

20

25

50

样例输出

305



样例输入

6 1 100 代表未来天数 $T$ ，纪念品数量 $N$ ，小伟现在拥有的金币数量 $M$ 。

50 表示第 $i$ 天第 $j$ 种纪念品的价格。

20

25

20

25

50

样例输出

305

【输入输出样例 1 说明】

最佳策略是：

第二天花光所有 100 枚金币买入 5 个纪念品 1；  
第三天卖出 5 个纪念品 1，获得金币 125 枚；  
第四天买入 6 个纪念品 1，剩余 5 枚金币；  
第六天必须卖出所有纪念品换回 300 枚金币，第  
四天剩余 5 枚金币，共 305 枚金币。  
超能力消失后，小伟最多拥有 305 枚金币。

---

3 3 100  
10 20 15  
15 17 13  
15 25 16  
217

最佳策略是：

第一天花光所有金币买入 10 个纪念品 1；

第二天卖出全部纪念品 1 得到 150 枚金币并买入 8 个纪念品 2 和 1 个纪念品 3，剩余 1 枚金币；

第三天必须卖出所有纪念品换回 216 枚金币，第二天剩余 1 枚金币，共 217 枚金币。

超能力消失后，小伟最多拥有 217 枚金币。

---

这题不太好理解，不过可以断定是DP题，而且是背包题。做如下考虑：

- 1、我不买
- 2、我买完第二天就卖
- 3、我买完过几天卖

若只考虑前两种，我们就可以把题目转换成当天花多少钱，第二天赚了多少钱，即追求每一天的最大升值，也就分成T个完全背包问题。

那如何表示第三种呢，也很简单。比方说我买了3天卖，因为当天价格不变所以，我完全可以分为3天的单独买卖。这样一定会被完全背包所包含。那么这样一来就是个简单的完全背包问题。对每天进行完全背包就行，之后  $m += f[m]$ ，改变背包总容量。

```

1  include<bits/stdc++.h>
2  using namespace std;
3  const int N=1e2+7;
4  int t,n,m,price[N][N];
5  int f[10001],dis[N];
6  void solve(int x)
7  {
23 int main() {
24     cin>>t>>n>>m;
25     for(int i=1;i<=t;i++) {
26         for(int k=1;k<=n;k++)
27             cin>>price[i][k];
28     }
29     for(int i=2;i<=t;i++)
30         solve(i); //得到每天的利润
31     cout<<m<<endl;
32 }

```

```

6 void solve(int x)
7 {
8     memset(f,0,sizeof(f));//f[i]表示背包容量为 i 时的价值
9     int today=x,yesterday=x-1;
10    for(int i=1; i<=n; i++) //得到物品i今天和昨天的价值差
11        dis[i]=price[today][i]-price[yesterday][i];
12    for(int i=1; i<=n; i++) {
13        //对 n 个物品跑一遍完全背包
14        for(int j=price[yesterday][i]; j<=m; j++) {
15            if(f[j-price[yesterday][i]]+dis[i]>=f[j])
16                { //如果入昨天的物品并且在今天卖出能盈利就更新背包
17                    f[j]=f[j-price[yesterday][i]]+dis[i];
18                }
19        }
20    }
21    m+=f[m];//加上今天的利润
22 }

```

## [1120] trs滑雪

trs喜欢滑雪。他来到了一个滑雪场，这个滑雪场是一个矩形，为了简便，我们用 $r$ 行 $c$ 列的矩阵来表示每块地形。为了得到更快的速度，滑行的路线必须向下倾斜。例如样例中的那个矩形，可以从某个点滑向上下左右四个相邻的点之一。例如24-17-16-1，其实25-24-23...3-2-1更长，事实上这是最长的一条。

输入

输入文件 第1行: 两个数字 $r, c$  ( $1 \leq r, c \leq 100$ ), 表示矩阵的行列。第2.. $r+1$ 行: 每行 $c$ 个数, 表示这个矩阵。

输出

输出文件 仅一行: 输出1个整数, 表示可以滑行的最大长度。

样例输入

```
5 5
1 2 3 4 5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9
```

样例输出

```
25
```

---

什么是记忆化搜索呢？搜索的低效在于没有能够很好地处理重叠子问题；动态规划虽然比较好地处理了重叠子问题，但是在有些拓扑关系比较复杂的题目面前，又显得无奈。记忆化搜索正是在这样的情况下产生的，它采用搜索的形式和动态规划中递推的思想将这两种方法有机地综合在一起，扬长避短，简单实用，在信息学中有着重要的作用。

用一个公式简单地说：记忆化搜索=搜索的形式+动态规划的思想。

---

事实上，记忆化搜索与动态规划只是一种优化性能的手段，的确如你所说的那样是 一个是自顶向下，另一个是自底向上。

递推和记忆化搜索并不能相互转化，例如：第一种情况：记忆化搜索可以实现然而普通递推不能实现的问题（或实现起来非常麻烦的问题）最为典型的代表是“滑雪”。



---

我们假设 $f[i][j]$ 表示滑到坐标 $(i, j)$ 所能滑到的最长长度。那么对于状态 $f[i][j]$ 而言，它可以由 $f[i-1][j]$ ,  $f[i][j-1]$ ,  $f[i+1][j]$ ,  $f[i][j+1]$ 四个状态推得，然而我们使用普通的递推（两个for）只能得到上、左两个方向的状态，右、下两个方向的状态却无从得知，因此使用递推就不能满足我们的要求，如果再补上两个for覆盖右、下状态，那么时间复杂度就变为了 $N^4$ ，很明显会TLE。

这题如果采用记忆化搜索，在搜索的过程中发现 $f[i][j]$ 在以前的某个时刻已经被计算过，直接return，可以保证时间复杂度为 $N^2$ 。

---

思路：给出一个二维数组，让你求出最长递减序列长度，可以四个方向行走，起点任意。要对于每个点，都算出到达1的最长路径。典型的动态规划题目，采用记忆化搜索，利用一个数组保存每个点的最大值，每次第一次访问一个点，就记录它到达1的最长路径，当下次访问时，就直接返回记录的值(动态规划的优点，避免重复计算子问题)，对每个点进行上下左右的求解，该点的最大值肯定是从四个方向中最大的+1。即用一个数组dp[110][110]存储每个节点的最长路径，先初始化为0，每当访问一个节点时，就判断dp值是否大于0，大于0则此点的值就是dp的值，直接返回。按照这个思想，就可求解。

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  #define MAXN 110
4  int a[MAXN][MAXN];
5  int dp[MAXN][MAXN]; //用来记录以点(i,j)为最大时的路径长度
6  int r, c;
7  int xx[4] = {1, -1, 0, 0}; //规定方向为下上右左
8  int yy[4] = {0, 0, 1, -1};
9  int dfs(int x,int y){
10     int i,nx,ny;
11     if(dp[x][y]!=0) //该点还未记录过
12         return dp[x][y];
13     int ret=0;
14     for(i=0;i<4;i++){ //进行搜索,寻找比该点小的点
15         nx=x+xx[i];
16         ny=y+yy[i];
17         if(nx>=0&&nx<r&&ny>=0&&ny<c&&a[nx][ny]<a[x][y]){
18             ret=max(ret,dfs(nx,ny));
19         }
20     }
21     return dp[x][y]=ret+1;//记录该点开始的路径长度
22 }

```

```
23 int main(){
24     int ans=0,i,j;
25     scanf("%d%d",&r,&c);
26     for(i=0;i<r;i++)
27         for(j=0;j<c;j++)
28             scanf("%d",&a[i][j]);
29     memset(dp,0,sizeof(dp)); //清空数组dp, 表示所有点都未记录过
30     for(i=0;i<r;i++)
31         for(j=0;j<c;j++)
32             if(dp[i][j]==0)
33                 ans=max(ans,dfs(i,j)); //记录下最长路径
34     printf("%d",ans);
35     return 0;
36 }
```

# 今天的课程结束啦.....

---



下课了...  
同学们**再见!**