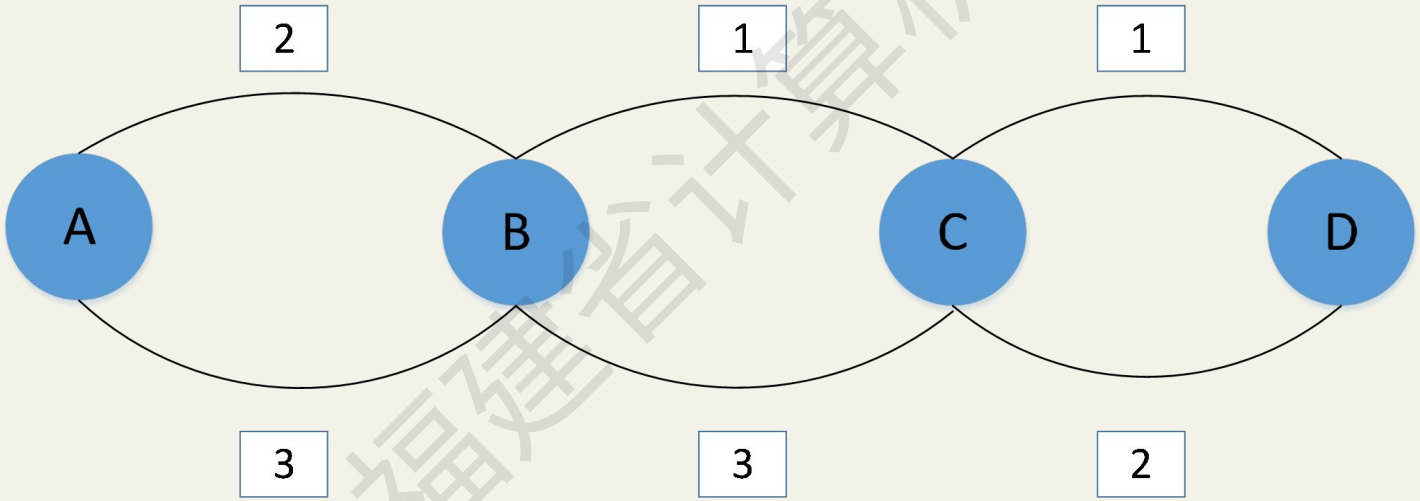
动态规划（二）

基础回顾

例题：给出一个图，相邻两点之间有路径长度，求从起点到终点的路径长度对k求余的最小值，点数小于等于100.如下图所示，该A~D路径对4求余最小值为0.



分析说明：

类似求最短路算法，采用动态规划思想。

基础回顾

【分析】

设d[i]表示走到点i时，该路径mod k的最小值，我们进行松弛操作，即状态转移方程：

$$d[i]=\min\{(d[j]+a[i][j])\bmod k\}\text{其中点}j\text{到点}i\text{有道路}$$

我们来求解一下样例：

$$d[A]=0$$
$$d[B]=\min\{(d[A]+2)\bmod 4,(d[A]+3)\bmod 4\}=2$$
$$d[C]=\min\{(d[B]+1)\bmod 4,(d[B]+3)\bmod 4\}=1$$
$$d[D]=\min\{(d[C]+1)\bmod 4,(d[C]+2)\bmod 4\}=2$$

显然答案错误，那么造成这种错误的原因是什么呢？

基础回顾

【分析】

仔细分析可以看出，每次由方程 $d[i]=\min\{(d[j]+a[i][j])\bmod k\}$ 得出的阶段最优值，并不能决定下一个阶段的最优值的发展。换句话说，按这个方程推导出的当前最优值不能推导出整个问题的最优值。这种状态的描述不具有最优性原理。

所谓最优性原理是指“多阶段决策过程的最优决策序列具有这样的性质：不论初始状态和初始决策如何，对于前面决策造成的某一状态而言，其后各阶段的决策序列必须构成最优策略”。

因此我们将该问题的状态重新描述，对每个阶段的状态并不仅仅只保留mod k 的最小值，而需要保留mod k 的所有余数。

基础回顾

【分析】

设一个数组can[i][x]表示走到点i路径长度mod k 余数能否为x，若能则返回true，否则返回false，则有：

$$can[i][x] = can[j][y]\&\&((x + a[i][j])\%k == y)$$

下面我们再看样例：

$$can[A][0] = true$$

$$can[B][2] = true, can[\mathrm{B}][3] = \mathrm{true}$$

$$\mathrm{can}[\mathrm{C}][0] = \mathrm{true}, \mathrm{can}[\mathrm{C}][1] = \mathrm{true}, \mathrm{can}[\mathrm{C}][2] = \mathrm{tru\,e}$$

$$can[D][0] = true, can[D][1] = true, can[D][2] = true, can[D][3] = true$$

答案为can[D][0]，所以最小值为0.

基础回顾

【分析】

上述状态设计之所以能成功推导结论，是由于每个阶段都记录了所有的最优结果，因此符合最优性原理。同时，我们可以看到，每一个阶段的状态推导下一个阶段的状态后，当前阶段的状态值以后不会再用，可以删除。换一句话来说：阶段i中的状态只能通过阶段i+1中的状态通过状态转移方程得来，与其他状态没有关系，特别是与未来发生的状态没有关系。这个性质称为无后效性。

01背包问题

题目描述

有n个重量和价值分别为wi，vi的物品。从这些物品中挑选出总重量不超过W的物品，求所有挑选方案中价值总和最大值。

输入

输入的第一行为一个整数n(1<=n<=100)；第二行到第n+1行，每行两个整数，第i+1分别表示第i个物品重量wi与价值vi（1<=wi,vi<=100)；最后一行有一个整数W（1<=W<=10000)，表示挑选出的物品总重量限制。

输出

输出只有一个整数，为所求的最大总价值。

样例输入

4 2 3 1 2 3 4 2 2 5

样例输出

7

分析说明**:**

显然这个题可用深度优先方法对每件物品进行枚举**(选或不选用 0,1** 控制**).** 程序简单**,但是当 n** 的值很大的时候不能满足时间要求，时间复杂度为 **O（2^n）**。

01背包问题

时间/空间复杂度
均为：O(n*W)

算法分析：

由于n个物品中每个物品要么选择，要么不选择。先选后选没有区别。所以我们可以按1到n这样顺序来做决策（要或不要），即可以按1到n划分阶段。现在我们用f(n,W)表示前n个物品中装入W重量背包中获得最大的价值。因此容易得到

$$f(n,W)=\begin{cases} f(n-1,W) & ,w[n]>W \\ \max\{f(n-1,W),f(n-1,W-w[n])+v[n]\} & ,\text{其它} \end{cases}$$

一般地，

$$f(i,j)=\begin{cases} f(i-1,j) & ,w[i]>j \\ \max\{f(i-1,j),f(i-1,j-w[i])+v[i]\} & ,\text{其它} \end{cases} \qquad i=1,2,\ldots,n \quad j=1,2,\ldots W$$

$$f(0,j)=0,f(i,0)=0;$$

f（n，m）即为最优解

01背包问题

```cpp
#include<iostream>
#include<cstring>
#include<cstdio>
using namespace std;
const int MaxN=100;
const int MaxW=10000;
int dp[MaxN+1][MaxW+1];
int n,w[MaxN+1],v[MaxN+1],W;

int main(){
    //输入数据及初始化
    cin>>n;
    for(int i=1;i<=n;i++) cin>>w[i]>>v[i];
    cin>>W;
    memset(dp,0,sizeof(dp));
    //
    for(int i=1;i<=n;i++)
      for(int j=1;j<=W;j++){
        if(j<w[i]) dp[i][j]=dp[i-1][j];//如果w[i]>j,第i个物品一定不能选择
        //否则在第i个物品选与不选中取个大的方案
        else dp[i][j]=max(dp[i-1][j],dp[i-1][j-w[i]]+v[i]);
      }
    cout<<dp[n][W]<<endl;//输出答案
}
```

01背包问题（空间优化）

```cpp
#include<iostream>
#include<cstring>
#include<cstdio>
using namespace std;
const int MaxN=100;
const int MaxW=10000;
int dp[MaxW+1];
int n,w[MaxN+1],v[MaxN+1],W;

int main(){
    //输入数据及初始化
    cin>>n;
    for(int i=1;i<=n;i++) cin>>w[i]>>v[i];
    cin>>W;
    memset(dp,0,sizeof(dp));
    //
    for(int i=1;i<=n;i++){
      for(int j=W;j>=w[i];j--){
        dp[j]=max(dp[j],dp[j-w[i]]+v[i]);
      }
    }
    cout<<dp[W]<<endl;//输出答案
}
```

完全背包问题

题目描述

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

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,Ui，表示每个物品的重量和价值。

输出

仅一行，一个数，表示最大总价值，格式如样例。

样例输入

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

样例输出

```
max=12
```

完全背包问题

与**0/1**背包类似，我们可以按**1**至**n**顺序划分出阶段。这题同一种类的物品可以选择多件了。我们试着写出状态递推方程。

$$f(i,j)=\max\{f(i-1,j-k*w[i])+k*v[i]\mid 0\le k\}$$

$$f(0,j)=0,f(i,0)=0$$

完全背包问题

```
#include<cstring>
#include<cstdio>
using namespace std;
const int MaxN=100;
const int MaxW=10000;
int dp[MaxN+1][MaxW+1];
int n,w[MaxN+1],v[MaxN+1],W;

int main(){
    //输入数据及初始化
    cin>>W>>n;
    for(int i=1;i<=n;i++) cin>>w[i]>>v[i];
    memset(dp,0,sizeof(dp));
    //**********************
    for(int i=1;i<=n;i++)//动态规划计算
      for(int j=1;j<=W;j++)
        for(int k=0;k*w[i]<=j;k++)
          dp[i][j]=max(dp[i][j],dp[i-1][j-k*w[i]]+k*v[i]);
    cout<<"max="<<dp[n][W]<<endl;//输出答案
}
```

空间复杂度均为：O(n*W)，时间复杂度为O（n*W*W)

完全背包（时间优化）

$$dp(i,j)=max(dp(i-1,j),dp(i-1,j-w[i])+v[i],\ dp(i-1,j-2*w[i])+2*v[i],\ldots,dp(i-1,k*w[i])+k*v[i])\}$$

$$dp(i,j-w[i])=max(dp(i-1,j-w[i]),dp(i-1,j-2*w[i])+v[i],,\ldots,dp(i-1,j-k*w[i])+(k-1)*v[i]\}$$

$$dp(i,j)=max(dp(i,j-w[i])+v[i],dp(i-1,j))$$

完全背包问题（改进时间）

```cpp
#include<cstring>
#include<cstdio>
using namespace std;
const int MaxN=100;
const int MaxW=10000;
int dp[MaxN+1][MaxW+1];
int n,w[MaxN+1],v[MaxN+1],W;

int main(){
    //输入数据及初始化
    cin>>W>>n;
    for(int i=1;i<=n;i++) cin>>w[i]>>v[i];
    memset(dp,0,sizeof(dp));
    //**********************
    for(int i=1;i<=n;i++)//动态规划计算
      for(int j=1;j<=W;j++)
        if(j<w[i]) dp[i][j]=dp[i-1][j];
        else  dp[i][j]=max(dp[i-1][j],dp[i][j-w[i]]+v[i]);
    cout<<"max="<<dp[n][W]<<endl;//输出答案
}
```

空间复杂度均为：O(n*W)，时间复杂度为O（n*W)

完全背包问题（改进空间）

```
#include<iostream>
#include<cstring>
#include<cstdio>
using namespace std;
const int MaxN=100;
const int MaxW=10000;
int dp[MaxW+1];
int n,w[MaxN+1],v[MaxN+1],W;

int main(){
    //输入数据及初始化
    cin>>W>>n;
    for(int i=1;i<=n;i++) cin>>w[i]>>v[i];
    memset(dp,0,sizeof(dp));
    //**********************
    for(int i=1;i<=n;i++)//动态规划计算
      for(int j=1;j<=W;j++)
        if(j>=w[i]) dp[j]=max(dp[j],dp[j-w[i]]+v[i]);
    cout<<"max="<<dp[W]<<endl;//输出答案
}
```

空间复杂度均为：O(W)，时间复杂度为O（n*W)

优化原理

先看一下二维数组的遍历

f[i][j] 表示前i件物品，在容量不超过j的情况下的最大权值

n 表示种类， t 表示 容量
w[MAXN]表示重量，v[MAXN]表示价值

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

01背包的关键两点： 1. 选与不选。

2. 前 i 件物品的 dp 情况，是由前 i-1 件物品的dp推出来的，不受本i件物品的dp的影响

再看一维的遍历

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

第二层循环的遍历就是为了满足上述的第二个条件。 我们先设 j1 > j2 ,更新dp[j1] 肯定在 更新dp[j2] 的前面，就算 j1- w[i] = j2 , 也不会受影响。 故在 第 i 层的循环下，可以保证各个dp[j] 是由 前 i -1 层 dp 推出来的，而不会受本次循环其他dp更新的影响。

也可以用反证法，当第二层循环为递增循环时

```
for(i=1;i<=n;i++)                    // 完全背包
  for(j=w[i];j<=t;j++)
    dp[j]=max(dp[j],dp[j-w[i]]+v[i]);
```

也先设 j1 > j2 ， 如果 j1 - w[i] == j2 . 而更新 dp[j2] 在 dp [j1] 前面，故 dp[j1] 会受到 dp[j2] 的影响。这与第二个条件不符。

不过这与完全背包的条件相符

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

完全背包的关键两点： 1. 可选有穷个。

2. 前 i 件物品的 dp 情况，是由前 i 件物品的dp推出来的，受本i件物品的dp的影响

线性动规

所谓线性动规，就是该问题的模型是线性的，数据结构表现为线性表的形式，对于这种数据类型的动态规划问题，我们看一下如何思考。

【问题】

小蒟的工作。小蒟为了计算一天能干多少事，他将一天划分为n个单位时间，并告诉你这天他可以干m项工作，其中第i项工作需要从第Si时刻开始连续做到第Ei个时刻后才能完成，同时完成这项工作能获得Pi的收入。小蒟在某一个时刻只能干一项工作，并且一旦选择某项工作，必须不间断一次性做完，问他一天的工作中能获得最多的收入为多少，请你帮他进行工作的选取。

输入格式：

第一行，一个整数n（n<=5000).

第二行，一个整数m（m<=5000).

接下来m行，描述m件事情，每行包含3个整数Si，Ei，Pi。

输入格式：

仅包含一行，为小蒟在这一天内所能获得最大收入。

线性动规

【分析】

将每一项工作看成一个点，若将m项工作按开始时间排序，那么本题一天内的m个工作就是一条线上的m个点。小翡在一个时间内只能做一项工作，小翡选取哪些工作就相当于在这条线上如何选取哪些点的问题。

设f[i]表示到第i个单位时刻为止小翡当前能获得的最大收入。现在考虑决策，对于时刻i+1，要么闲着，要么干一件以i+1开始的事情（也就是上一最优情况以时刻i结束）

状态转移方程为：

$$f[i] = \max\{f[i-1], f[S[j]] + p[i](E[j] = i)\}, f[0] = 0$$

样例代码

```
#include<iostream>
#include<vector>
using namespace std;
int n,m,f[5200]={0};
struct Node{
    int to,val;
}tmp;
vector<Node>vec[5200];
int main(){
    cin>>n>>m;  //输入n个单位时间，m项工作
    for(int x,y,z;m--;){//初始化m件事请（开始时间，结束时间，价值）
        cin>>x>>y>>z;
        if(y>n||x<0)continue;
        vec[x].push_back((Node){y,z});
    }
    for(int i=0;i<n;i++){    //线性动态规划，进行状态转移
        for(int j=0;j<(int)vec[i+1].size();j++){
            tmp=vec[i+1][j];
            f[tmp.to]=max(f[tmp.to],f[i]+tmp.val);
        }
        f[i+1]=max(f[i+1],f[i]);
    }
    cout<<f[n];
}
```

```
24 3
2 8 10
7 12 7
14 15 8
18
--------------------------------
```

vector容器

①定义与初始化

```
vector<int> vec1;   //默认初始化，vec1为空
vector<int> vec2(vec1);  //使用vec1初始化vec2
vector<int> vec3(vec1.begin(),vec1.end());//使用vec1初始化vec2
vector<int> vec4(10);   //10个值为0的元素
vector<int> vec5(10,4);  //10个值为4的元素
vector<string> vec6(10,“null”);   //10个值为null的元素
vector<string> vec7(10,“hello”);  //10个值为hello的元素
```

vector容器

②常用的操作方法

```
vec1.push_back(100);          //添加元素
int size = vec1.size();       //元素个数
bool isEmpty = vec1.empty();  //判断是否为空   cout<<vec1[0]<<endl;      //取得第一个元素
vec1.insert(vec1.end(),5,3);  //从vec1.back位置插入5个值为3的元素
vec1.pop_back();              //删除末尾元素   vec1.erase(vec1.begin(),vec1.end());//删除之间的元素，其他元素前移
cout<<(vec1==vec2)?true:false; //判断是否相等==、！=、>=、<=...
vector<int>::iterator iter = vec1.begin();   //获取迭代器首地址
vector<int>::const_iterator c_iter = vec1.begin();  //获取const类型迭代器
vec1.clear();                 //清空元素
```

线性动规

【问题】 给定一个长度为n的序列a[1],a[2]...a[n-1],a[n]，求一个连续的子序列a[i],a[i+1]...a[j-1],a[j]，使得a[i]+a[i+1]...a[j-1]+a[j]最大。

【分析】

步骤 1：令状态 dp[i] 表示以 A[i] 作为末尾的连续序列的最大和（这里是说 A[i] 必须作为连续序列的末尾）。

步骤 2：因为 dp[i] 要求是必须以 A[i] 结尾的连续序列，那么只有两种情况：

- 这个最大和的连续序列只有一个元素，即以 A[i] 开始，以 A[i] 结尾。
- 这个最大和的连续序列有多个元素，即从前面某处 A[p] 开始 (p<i)，一直到 A[i] 结尾。

对第一种情况，最大和就是 A[i] 本身，即 dp[i]=A[i];

对第二种情况，最大和是 dp[i-1]+A[i]，即 dp[i]=dp[i-1]+A[i]；

于是得到状态转移方程： dp[i] = max{A[i], dp[i-1]+A[i]}

这个式子只和 i 与 i 之前的元素有关，且边界为 dp[0] = A[0]。

最大连续序列和（DP）

```cpp
#include<cstdio>
#include<algorithm>
#include<string>
#include<iostream>
using namespace std;
int dp[200];
int main(){
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> dp[i];
    int m = dp[1];
    for (int i = 2; i <= n; i++){
        dp[i] = max(dp[i], dp[i] + dp[i - 1]);    //dp[i]代表前i个元素中连续最大序列和
        if (dp[i] > m)
            m = dp[i];
    }
    cout << m << endl << endl;
}
```

最大连续序列和（解二）

```cpp
#include<iostream>
using namespace std;
int a[100001];
int n,i,ans,len,tmp,beg;
int main()
{
    cin>>n;
    for(i=0;i<n;i++)
        cin>>a[i];
    tmp=0;ans=0;len=0;
    beg=-1;
    for(i=0;i<n;i++){
        if(tmp+a[i]>ans)
        {
          ans=tmp+a[i];
          len=i-beg;
        }
        else if(tmp+a[i]==ans&&i-beg>len)
           len=i-beg;
        if(tmp+a[i]<0)
        {
          beg=i;
          tmp=0;
        }
        else
        tmp+=a[i];
    }
    cout<<ans<<endl;
    return 0;
}
```

1141: 抄书问题copybook

时间限制: 1 Sec 内存限制: 128 MB
提交: 288 解决: 87
[提交][状态][讨论版]

题目描述

印刷术发明之前，复制一本书是非常困难的。书本上所有内容必须通过抄写员用手工重写。抄写员被提供给一本书，然后几个月以后他完成了复制。在15世纪有一个很著名的抄写作坊，他的名字为Xerox。然而这个工作是烦人和无趣的。提高复制速度的唯一方法是雇用更多的人。

后来，有一个戏剧团要上演一部闻名遐迩的古代悲剧。这些戏剧的剧本分别在若干本书上，当然演员们需要更多的备份。于是他们就雇了许多抄写员来复制这些书。假设你有M本书（编号为1，2…M），分别为不同数目的页数（P1，P2，…，Pm），而你要将每本书复制一份。你的任务是将这些书分给K个抄写员（K<=M）。每本书只能被指定给唯一的一个抄写员。则每个抄写员也只能得到编号连续的几本书。这意味着，存在一个增加的序列数0 = b0 < b1 <b2, ... < bk-1 <= bk = m，这样第I个抄写员得到的书本编号为bi-1+1 到 bi。这次要复制所有书本一份的时间由指定给最多书本的那个抄写员决定。因此我们的目标是对单一的某个抄写员总的抄写页数要尽量小。你的任务是找出这个安排方案。

输入

输入的第1行包行两个正整数m和k（均不大于500）。第2行给出p1，p2，…，pm，用空格分开。这些值都小于10000000。

输出

输出仅1行，将p1，p2，…，pm分成k个部分用k-1个'/'隔开。
如果不止一种解答，输出其中这样的一种，指定给第一个抄写员的工作最少，然后是第二个等等。但每个抄写员至少要指定抄写一本书。
每个序列中的数之间或数与"/"号之间有一个空格，最后一个数的后面有一个空格。

样例输入

```
9 3
100 200 300 400 500 600 700 800 900
```

样例输出

```
100 200 300 400 500 / 600 700 / 800 900
```

提示

抄书问题

阶段划分：由于“每本书只能被指定给唯一的一个抄写员。则每个抄写员也只能得到编号连续的几本书。”，因此我们可以将这m本书，划分为k个部分。问题的阶段数为K。

最优子问题性质：有了以上的阶段划分后，我们可假设某一划分｛$b_1, b_2, b_3, \ldots, b_k$，其中$b_i$为划分中每一部分的长度｝是原问题的一个最优划分（这种划分具有最优值）。则｛$b_1, b_2, b_3, \ldots, b_{k-1}$｝必定是子问题(m-$b_k$,k-1)（前m-$b_k$本书由k-1人来抄）的最优划分。可用反证法证明。例如：对于问题：m=9,k=3，每本书的页数分别为：“100 200 300 400 500 600 700 800 900”，一种最优划分为：“ 100 200 300 400 500 / 600 700 / 800 900”，即将前m本书划分为{5,2,2}这3个部分，则对于划分{5,2}（ “ 100 200 300 400 500 / 600 700 ”）必定是对应子问题的一种最优划分。

状态的选择与递归方程：可设f(m,k)表示前m本书由k个抄写员来抄写最少需要的时间，则可根据以上的最优子问题的性质得出递归方程为：

$$\begin{cases} f(i,j)=\min\limits_{p=1}^{i-(j-1)}\{\max(f(i-p,j-1),b[i]-b[i-p])\}, i>=j, j=2,3,\ldots,k \\ f(i,1)=b[i], i=1,2,\ldots,m-(k-1) \\ f(i,j)\text{表示前}i\text{本书由}j\text{个人来抄，抄最多的人最少需要抄的页数} \\ b[i]\text{为前i本书的页数总和} \end{cases}$$

时空复杂度均为O(m^2)

抄书问题（实例手算）

$$\begin{cases} f(i,j)=\min_{p=1}^{i-(j-1)}\{\max(f(i-p,j-1),b[i]-b[i-p])\}, i>=j, j=2,3,...,k \\ f(i,1)=b[i], i=1,2,...,m-(k-1) \\ f(i,j)\text{表示前}i\text{本书由}j\text{个人来抄，抄最多的人最少需要抄的页数} \\ b[i]\text{为前i本书的页数总和} \end{cases}$$

Sample input

6 3

10 20 15 30 25 10

Sample ouput

10 20/15 30/25 10

i,j	1	2	3
1	10	/	/
2	30	20	/
3	45	30	/
4	75	45	/
5	/	55	/
6	/	/	45

最优值数组f(i,j)

i,j	1	2	3
1	1	/	/
2	2	1	/
3	3	1	/
4	4	2	/
5	/	2	/
6	/	/	2

决策数组e(i,j),存放当f(i,j)取最优值时，p的值

抄书问题

```
#include<cmath>
#include<cstring>
const int INF=0x7fffffff;
using namespace std;
int e[501][501],f[501][501],a[501],b[501],m,k;
//f[i,j]表示前i本书由前j个人抄，抄最多的人最小要抄的页数
//e[i,j]表示当f[i,j]取最优值时最后一人抄的书本数
void init(){//输入数据
    cin>>m>>k;
    for(int i=1;i<=m;i++){cin>>b[i];a[i]=b[i];if(i>1)b[i]+=b[i-1];}
}
void dymic(){//动态规划算法主函数
    for(int i=1;i<=m-k+1;i++){//设置边界条件f[i,1]=b[i]
        f[i][1]=b[i];e[i][1]=i;
    }
    for(int j=2;j<=k;j++){//以抄写员编号为阶段递推，求f[i,j]
        for(int i=(j==k?m:j);i<=m-(k-j);i++){
          int mi=INF;
          for(int p=1;p<=i-(j-1)/*前面还有j-1人*/;p++){
            if(b[i]-b[i-p]>mi)break;
            int t=max(f[i-p][j-1],b[i]-b[i-p]);//前i本书中第j个人抄p本，前j-1个人抄i-p本
            if(t<=mi){mi=t;e[i][j]=p;}
          }
          f[i][j]=mi;
        }
    }
}
```

抄书问题

```
void print(){//输出最优方案
    int path[501],x=0,y,mm=m,kk=k;//从e[m,k]开始往前推
    do{
        //将e[mm,kk]值保存入path数组，mm=mm-y;人数kk-1
        y=e[mm][kk];path[++x]=y;mm-=y;kk--;
    }while(kk!=0);//直到kk=0
    y=0;
    for(mm=x;mm>=1;mm--){//path数组倒着保存每个抄写员抄写的书本数量
        for(kk=1;kk<=path[mm];kk++){//输出某抄写员抄写那几本的页数
            cout<<a[++y]<<" ";
        }
        if(mm!=1)cout<<"/ ";//不是最后一人时，输出/
    }
}
main(){
  init();dymic();print();return 0;
}
```

区间动态规划

最小中间和

给定一个正整数序列a1,a2,...,an，不改变序列中的每个元素在序列中的位置，把它们相加，并用括号记每次加法所得的和，称为中间和。

编程：找到一种方法，添上n-1对括号，加法运算依括号顺序进行，得到n-1个中间和，使得求出使中间和最少。

例如给出的序列是4,1，2,3。

- 第一种添加括号方法：（（4＋1）＋（2＋3））＝（（5）＋（5））＝（10），有三个中间和是5,5，10，它们之和为5＋5＋10＝20;
- 第二种添括号方法：（4＋（（1＋2）＋3））＝（4＋（（3）＋3）＝（4＋（6））＝（10），中间和是3,6，10，它们之和为19。

输入：
第一行为N（N<=100)，第二行依次为a1,a2,...,an(ai<=10000)

输出：
输出第一行为S（最小的中间和）

最小中间和

算法分析：

这道题很容易想到，选相邻两个数的和小的进行相加。这是一种贪心做法，正确性较难证明。由于**N**很小，我们就想尝试用动态规划算法解决这个问题。

分析样例： **4,1**，**2,3**的任意一个合并方案。我们会发现，最后一次相加的代价都是所有整数的和，如果最后一次相加是在第**k**个整数后面且这种方案最优，则第**1**至第**k**个整数必须合并成一个整数，第**k+1**至第**n**个整数也必须先合并成一个整数，那么它们肯定必须是最优的合并。因此，我们可以容易得到动态规划方程发下：

这个问题中，状态**[i,j]**是原问题中一个连续整数的区间。以上方程的边界为**f[i,i+1]=a[i]+a[j](i=1,2,...,n-1)**。

最小中间和

$$f[i,j] = \min_{k=i}^{j-1}(f[i,k] + f[k+1,j] + sum(i,j))$$

接下来，可以先求出所有区间长度1（连接2个整数）所有状态最优值。然后再依次求出区间长度为2、3、...、n-1的所有状态的最优值。则f[1,n]就是本题的答案。

这种以区间为状态，以区间长度从小到大递推求解各个区间状态最优值的算法，我们就把它称为区间动态规划。

最小中间和

```
#include<iostream>
#include<cstring>
using namespace std;
int a[101],b[101],f[101][101];
int main()
{
    memset(f,0,sizeof(f));
    int n,i,j,k,l,mi;
    cin>>n;
    for(i=1;i<=n;i++){
        //输入n个整数至a[i]，并将它们前缀和保存至b[i]
        cin>>a[i];b[i]=b[i-1]+a[i];
    }
    //设置边界条件，求区间长度为1的状态最优值
    for(i=1;i<n;i++) f[i][i+1]=a[i]+a[i+1];
    for(l=2;l<n;l++)//以区间长度为2到n-1递推求出每个状态
        for(i=1;i<=n-l;i++){      //枚举区间的左边界
          j=l+i;                  //计算出区间右边界
          mi=999999;              //根据状态方程求f[i][j]
          for(k=i;k<=j-1;k++)
            if(f[i][k]+f[k+1][j]+b[j]-b[i-1]<mi) mi=f[i][k]+f[k+1][j]+b[j]-b[i-1];
          f[i][j]=mi;
        }
    cout<<f[1][n];                //输出答案
    return 0;
}
```

涂色

假设你有一条长度为 5 的木版，初始时没有涂过任何颜色。你希望把它的 5 个单位长度分别涂上红、绿、蓝、绿、红色，用一个长度为 5 的字符串表示这个目标：`RGBGR`。

每次你可以把一段连续的木版涂成一个给定的颜色，后涂的颜色覆盖先涂的颜色。例如第一次把木版涂成 `RRRRR`，第二次涂成 `RGGGR`，第三次涂成 `RGBGR`，达到目标。

用尽量少的涂色次数达到目标。

输入格式

输入仅一行，包含一个长度为 n 的字符串，即涂色目标。字符串中的每个字符都是一个大写字母，不同的字母代表不同颜色，相同的字母代表相同颜色。

输出格式

仅一行，包含一个数，即最少的涂色次数。

涂色

【分析】

题目意思是对字符串的最少染色次数，设f[i][j]为字符串的字串s[i]~s[j]的最少染色次数，我们分析一下：

当i==j时，子串明显只需要涂色一次，于是f[i][j]=1.

当i!=j且s[i]==s[j]时，可以想到只需要在首次涂色时多涂一格即可，于是

f[i][j]=min(f[i][j-1],f[i+1][j])

当i!=j且s[i]!=s[j]时，我们需要考虑将子串断成两部分来涂色，于是需要枚举子串的断点，设断点为k，那么

f[i][j]=min(f[i][j], f[i][k]+f[k+1][j])

因此：

$$f_{i,j} = 1 \ (i == j)$$

$$f_{i,j} = \min(f_{i,j-1}, f_{i+1,j}) \ (i! = j, \ s_i == s_j)$$

$$f_{i,j} = \min(f_{i,j}, f_{i,k} + f_{k+1,j}) \ (i! = j, \ s_i! = s_j, \ i \leq k < j)$$

由此可知，f[1][n]即为答案。

涂色

【标程】

```
#include<bits/stdc++.h>
using namespace std;
char s[52];
int f[52][52];
int main() {
    int n;
    scanf("%s",s+1);
    n=strlen(s+1);
    memset(f,0x7F,sizeof(f));          //由于求最小，于是应初始化为大数
    for(int i=1;i<=n;++i)
        f[i][i]=1;
    for(int l=1;l<n;++l)
        for(int i=1,j=1+l;j<=n;++i,++j) {
            if(s[i]==s[j])
                f[i][j]=min(f[i+1][j],f[i][j-1]);
            else
                for(int k=i;k<j;++k)
                    f[i][j]=min(f[i][j],f[i][k]+f[k+1][j]);
        }
    printf("%d",f[1][n]);
    return 0;
}
```