区间动规 2

本节来讲区间动规的区间合并类型。

区间合并类型，我们往往用 f[i][j]表示区间 i 和 j 的最优值，而为了获得这个最优值，往往需要两个子区间进行合并得来。为了划分这两个更小的区间，我们则需用用一个循环变量 k 来枚举，即将 i 和 j 的区间划分为 i 到 k 的区间与 k 到 j 这两个区间，状态转移方程为：

f[i][j]=max/min(f[i][j],f[i][k]+f[k][j]+区间合并代价)

代码实现的模板如下：

```
1.      for(int j=2;j<=n;j++){  //j 循环放外层
2.          for(int i=j-1;i>0;i--){  //注意，这里 i 用倒序循环，必须的，为什么？
3.              for(int k=i+1;k<j;k++){
4.                  f[i][j]=min(f[i][j],f[i][k]+f[k][j]+合并代价);
5.              }
6.          }
7.      }
```

观察以上代码，外循环为什么要用逆序循环呢？如果用顺序循环，对于状态 f[i][j]，需要有 f[i][k]和 f[k][j]来更新，因为 k 大于 i，而 f[k][j]这个状态是还没求解出来的，所以更新会出错。

以递推的方式求解区间合并动规，有时会较难，为了简化操作，我们也会用记忆化递归的方式来实现，如下：

```
1.  int dp(int lf,int rt){  //获得区间 lf 到 rt 的最优值
2.      if(rt-lf==1)return 0;  //边界处理
3.      if(f[lf][rt]!=f[0][0])return f[lf][rt];  //记忆化处理
4.      for(int i=lf+1;i<rt;i++){  //枚举区间断点
5.          f[lf][rt]=min(f[lf][rt],dp(lf,i)+dp(i,rt)+合并代价);
6.      }
7.      return f[lf][rt];
8.  }
```

在处理区间合并问题时，我们还可能经常性遇到环形区间合并问题。对于环形区间，我们的处理办法是将环形复制一次展开成线性序列，然后再枚举每一个起点求最优值，如下所示：

<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>1（6）</td><td>2（7）</td><td>3（8）</td><td>4（9）</td><td>5（10）</td></tr>
<tr><td colspan="5">求 f[1][5]</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td colspan="5">求 f[2][6]，相当于从 2 起点环绕到 1</td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td colspan="5">求 f[3][7]，相当于从 3 起点环绕到 2</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td colspan="5">求 f[4][8]，相当于从 4 起点环绕到 3</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td colspan="5">求 f[5][9]，相当于从 5 起点环绕到 4</td><td></td></tr>
</table>

乘法游戏

【题目描述】
乘法游戏是在一行牌上进行的。每一张牌包括了一个正整数。在每一个移动中，玩家拿出一张牌，得分是用它的数字乘以它左边和右边的数，所以不允许拿第 1 张和最后 1 张牌。最后一次移动后，这里只剩下两张牌。
你的目标是使得分的和最小。
例如，如果数是 10 1 50 20 5，依次拿 1、 20、 50，总分是
10*1*50+50*20*5+10*50*5=8000
而拿 50、 20、 1，总分是 1*50*20+1*20*5+10*1*5=1150。
【输入】
输入文件的第一行包括牌数(3≤n≤100)，第二行包括 N 个 1-100 的整数，用空格分开。
输出
输出文件只有一个数字：最小得分
【样例输入】
6
10 1 50 50 20 5
【样例输出】
3650
【分析】
本题为区间合并动规模板题。设 f[i][j]表示移动 i 到 j 区间的牌的最小得分，假设 i 到 j 区间存在最后一张牌 k，那么将 i 到 k 之间的牌移走的最小得分为 f[i][k]，将 k 到 j 之间的牌移走的最小得分为 f[k][j]，移走 k 这张牌的得分为 a[i]*a[k]*a[j]，所以，状态转移方程为：
f[i][j]=f[i][k]+f[k][j]+a[i]*a[k]*a[j]|i<k<j
边界为相邻两张牌的得分为 0，即 f[i][i+1]=0。
【代码 1：递推版】

```
#include<iostream>
#include<cstring>
using namespace std;
int f[101][101],a[101],n;
int main(){
    cin>>n;
    for(int i=1;i<=n;i++)cin>>a[i];
    memset(f,127,sizeof(f));
    for(int i=1;i<n;i++)f[i][i+1]=0;
    for(int i=n;i>=1;i--){
        for(int j=i+1;j<=n;j++){
            for(int k=i+1;k<j;k++){
                f[i][j]=min(f[i][j],f[i][k]+f[k][j]+a[i]*a[k]*a[j]);
            }
        }
    }
    cout<<f[1][n];
```

```
25. }
```

【代码 2：递归版】

```
1.  #include<iostream>
2.  #include<cstring>
3.  using namespace std;
4.  int f[101][101],a[101],n,ans;
5.  int dp(int lf,int rt){
6.      if(rt-lf==1)return 0;
7.      if(f[lf][rt]!=f[0][0])return f[lf][rt];
8.      for(int i=lf+1;i<rt;i++){
9.          f[lf][rt]=min(f[lf][rt],dp(lf,i)+dp(i,rt)+a[lf]*a[i]*a[rt]);
10.     }
11.     return f[lf][rt];
12. }
13. int main(){
14.     memset(f,127,sizeof(f));
15.     cin>>n;
16.     for(int i=1;i<=n;i++)cin>>a[i];
17.     cout<<dp(1,n);
18. }
```

沙子合并

【题目描述】
设有 N 堆沙子排成一排，其编号为 1，2，3，…，N（N<=300）。每堆沙子有一定的数量，可以用一个整数来描述，现在要将这 N 堆沙子合并成为一堆，每次只能合并相邻的两堆，合并的代价为这两堆沙子的数量之和，合并后与这两堆沙子相邻的沙子将和新堆相邻，合并时由于选择的顺序不同，合并的总代价也不相同，如有 4 堆沙子分别为 1 3 5 2 我们可以先合并 1、2 堆，代价为 4，得到 4 5 2 又合并 1，2 堆，代价为 9，得到 9 2 ，再合并得到 11，总代价为 4+9+11=24，如果第二步是先合并 2，3 堆，则代价为 7，得到 4 7，最后一次合并代价为 11，总代价为 4+7+11=22；问题是：找出一种合理的方法，使总的代价最小。输出最小代价。
【输入】
第一行一个数 N 表示沙子的堆数 N。
第二行 N 个数，表示每堆沙子的质量 a[i](a[i]<=1000)。
【输出】
合并的最小代价
【样例输入】
4
1 3 5 2
【样例输出】
22

【分析】

这道题也是区间合并的模板，即每次进行合并的时候就是合并一个区间。我们可以设 f[i][j] 为合并 i-j 区间的石子的最优方案即最小花费，假设在我们进行 i 和 j 的状态转移的时候必然存在更小的区间已经求出了最优的方案数， 用 k 来划分 i 和 j 区间的最后一次合并，则状态转移方程为：

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

sum[j]-sum[i-1]是用前缀和的方式求出合并 i 到 j 这些沙子的代价。

边界:f[i][i]=0，自己和自己合并的代价为 0。

【代码 1：递推版】

```
#include<iostream>
using namespace std;
int f[1001][1001],a[1001],n,sum[1001];
int main(){
    memset(f,0x7f,sizeof(f));
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>a[i];
        sum[i]=sum[i-1]+a[i];  //前缀和
        f[i][i]=0;    //边界初始化
    }
    for(int j=2;j<=n;j++){
        for(int i=j-1;i>0;i--){
            for(int k=i;k<j;k++){
                f[i][j]=min(f[i][j],f[i][k]+f[k+1][j]+sum[j]-sum[i-1]);
            }
        }
    }
    cout<<f[1][n];
}
```

【代码 2：递归版】

```
#include<iostream>
using namespace std;
int f[1001][1001],a[1001],n,sum[1001];
int dp(int lf,int rt){
    if(lf==rt)return 0;
    if(f[lf][rt]!=f[0][0])return f[lf][rt];
    for(int i=lf;i<rt;i++){
        f[lf][rt]=min(f[lf][rt],dp(lf,i)+dp(i+1,rt)+sum[rt]-sum[lf-1]);
    }
    return f[lf][rt];
}
int main(){
```

```
    memset(f,0x7f,sizeof(f));
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>a[i];
        sum[i]=sum[i-1]+a[i];
    }
    cout<<dp(1,n);
}
```

关路灯

【题目描述】
某一村庄在一条路线上安装了 n 盏路灯，每盏灯的功率有大有小（即同一段时间内消耗的电量有多有少）。老张就住在这条路中间某一路灯旁，他有一项工作就是每天早上天亮时一盏一盏地关掉这些路灯。
为了给村里节省电费，老张记录下了每盏路灯的位置和功率，他每次关灯时也都是尽快地去关，但是老张不知道怎样去关灯才能够最节省电。他每天都是在天亮时首先关掉自己所处位置的路灯，然后可以向左也可以向右去关灯。开始他以为先算一下左边路灯的总功率再算一下右边路灯的总功率，然后选择先关掉功率大的一边，再回过头来关掉另一边的路灯，而事实并非如此，因为在关的过程中适当地调头有可能会更省一些。
现在已知老张走的速度为 1m/s，每个路灯的位置（是一个整数，即距路线起点的距离，单位：m）、功率（W），老张关灯所用的时间很短而可以忽略不计。
请你为老张编一程序来安排关灯的顺序，使从老张开始关灯时刻算起所有灯消耗电最少（灯关掉后便不再消耗电了）。
【输入】
文件第一行是两个数字 n(0<n<50，表示路灯的总数)和 c(1<＝c<=n 老张所处位置的路灯号)；接下来 n 行，每行两个数据，表示第 1 盏到第 n 盏路灯的位置和功率。
【输出】
一个数据，即最少的功耗(单位：J，1J＝1W/s)。
【样例输入】
5 3
2 10
3 20
5 20
6 30
8 10
【样例输出】
270　{此时关灯顺序为 3 4 2 1 5，不必输出这个关灯顺序}
【分析】
因为关灯的范围是一个区间，从起点出发，可以不断发范围扩大，这个一个区间动规问题。我们可以设定状态变量为 f[i][j][0]，表示关闭了 i 到 j 这个区间的灯且人在 i 位置时的最少功耗，对应的 f[i][j][1]表示关闭区间 i 和 j 的灯且人在 j 位置时的最少功耗。怎么推出状态转移方程呢？
对于 f[i][j][0]，因为人最后在 i 位置，所以应该先关闭 i+1 到 j 之间的灯，其最少功耗为 f[i+1][j][0]

和 f[i+1][j][1]。如果从 i+1 位置走过去关第 i 盏灯，时间为 a[i+1]-a[i]，在这段时间内，左边有 1 到 i 盏灯亮着，右边有 n-j 盏 灯亮着，利用前缀和数组记录，这些灯的功耗为 (a[i+1]-a[i])*(sum[i]+sum[n]-sum[j])，总功耗为：

f[i+1][j][0]+(a[i+1]-a[i])*(sum[i]+sum[n]-sum[j])

同理，如果关掉 i+1 到 j 这个区间的灯后人在 j 的位置，那么人从 j 走到 i 的时间为 a[j]-a[i],这期间依然左边有 1 到 i 盏灯亮着，右边有 n-j 盏 灯亮着，这些灯的功耗为 (a[j]-a[i])*(sum[i]+sum[n]-sum[j])，总功耗为：

f[i+1][j][1]+(a[j]-a[i])*(sum[i]+sum[n]-sum[j])

在这两种方式中，取功耗最小的方案：

f[i][j][0]=min(f[i+1][j][0]+(a[i+1]-a[i])*(sum[i]+sum[n]-sum[j]),
f[i+1][j][1]+(a[j]-a[i])*(sum[i]+sum[n]-sum[j]))

那么，对于 f[i][j][1]的分析，同理可得：

f[i][j][1]=min(f[i][j-1][0]+(a[j]-a[i])*(sum[i-1]+sum[n]-sum[j-1]),
f[i][j-1][1]+(a[j]-a[j-1])*(sum[i-1]+sum[n]-sum[j-1]))

最后，答案在 f[1][n][0]和 f[1][n][1]中取最小值。

因为状态表示比较复杂，建议用递归写法求解。

【代码】

```
#include<iostream>
using namespace std;
int n,c,a[51],w[51],sum[51],f[51][51][2];
int dp(int l,int r,int k){
    if(l==r){
        if(l==c)return 0;  //关闭起点的灯，时间为 0
        else return 1<<30;  //其他点，赋值为最大
    }
    if(f[l][r][k])return f[l][r][k];  //记忆化
    if(k==0){  //人在左端点的情况
        return f[l][r][0]=min(dp(l+1,r,0)+(a[l+1]-a[l])*(sum[l]+sum[n]-sum[r
]),dp(l+1,r,1)+(a[r]-a[l])*(sum[l]+sum[n]-sum[r]));
    }else{  //人在右端点的情况
        return f[l][r][1]=min(dp(l,r-1,0)+(a[r]-a[l])*(sum[l-1]+sum[n]-sum[r
-1]),dp(l,r-1,1)+(a[r]-a[r-1])*(sum[l-1]+sum[n]-sum[r-1]));
    }
}
int main(){
    cin>>n>>c;
    for(int i=1;i<=n;i++){
        cin>>a[i]>>w[i];
        sum[i]=sum[i-1]+w[i];
    }
    cout<<min(dp(1,n,0),dp(1,n,1));
```

```
23. }
```

能量项链

【题目描述】
在 Mars 星球上，每个 Mars 人都随身佩带着一串能量项链。在项链上有 N 颗能量珠。能量珠是一颗有头标记与尾标记的珠子，这些标记对应着某个正整数。并且，对于相邻的两颗珠子，前一颗珠子的尾标记一定等于后一颗珠子的头标记。因为只有这样，通过吸盘（吸盘是 Mars 人吸收能量的一种器官）的作用，这两颗珠子才能聚合成一颗珠子，同时释放出可以被吸盘吸收的能量。如果前一颗能量珠的头标记为 m，尾标记为 r，后一颗能量珠的头标记为 r，尾标记为 n，则聚合后释放的能量为（Mars 单位），新产生的珠子的头标记为 m，尾标记为 n。
需要时，Mars 人就用吸盘夹住相邻的两颗珠子，通过聚合得到能量，直到项链上只剩下一颗珠子为止。显然，不同的聚合顺序得到的总能量是不同的，请你设计一个聚合顺序，使一串项链释放出的总能量最大。
例如：设 N=4，4 颗珠子的头标记与尾标记依次为(2，3) (3，5) (5，10) (10，2)。我们用记号⊕表示两颗珠子的聚合操作，(j⊕k)表示第 j，k 两颗珠子聚合后所释放的能量。则第 4、1 两颗珠子聚合后释放的能量为：
(4⊕1)=10*2*3=60。
这一串项链可以得到最优值的一个聚合顺序所释放的总能量为
((4⊕1)⊕2)⊕3）=10*2*3+10*3*5+10*5*10=710。
【输入】
输入文件 energy.in 的第一行是一个正整数 N（4≤N≤100），表示项链上珠子的个数。第二行是 N 个用空格隔开的正整数，所有的数均不超过 1000。第 i 个数为第 i 颗珠子的头标记（1≤i≤N），当 i<N< span>时，第 i 颗珠子的尾标记应该等于第 i+1 颗珠子的头标记。第 N 颗珠子的尾标记应该等于第 1 颗珠子的头标记。
至于珠子的顺序，你可以这样确定：将项链放到桌面上，不要出现交叉，随意指定第一颗珠子，然后按顺时针方向确定其他珠子的顺序。
【输出】
输出文件 energy.out 只有一行，是一个正整数 E（E≤2.1*109），为一个最优聚合顺序所释放的总能量。
【样例输入】
4
2 3 5 10
【样例输出】
710
【分析】
本题为环形区间合并问题。定义 f[i][j]为第 i 颗珠子到第 j 颗珠子聚合产生的能量，那么：
F[i][j]=max(f[i][k]+f[k+1][j]+a[i]*a[k+1]*a[j+1])。
对于环形问题，我们将珠子复制一份在后面展开，依次求 f[1][n],f[2][n+1]……f[n][n+n-1]中的最大值即可。
【代码 1：递推版】

```
1. #include<iostream>
```

oiClass

```
using namespace std;
int n,a[300],f[300][300],maxn=-1;
int main(){
    int i,j,k;
    cin>>n;
    for(i=1;i<=n;i++){
        cin>>a[i];
        a[i+n]=a[i];  //复制展开
    }
    for(j=2;j<2*n;j++){
        for(i=j-1;j-i<n&&i>=1;i--){
            for(k=i;k<j;k++){
                f[i][j]=max(f[i][j],f[i][k]+f[k+1][j]+a[i]*a[k+1]*a[j+1]);
            }
            if(f[i][j]>maxn)maxn=f[i][j];
        }
    }
    cout<<maxn;
    return 0;
}
```

【代码 2：递归版】

```
#include<iostream>
using namespace std;
int f[202][202],a[202],n,ans;

int dp(int l,int r){
    if(l==r)return 0;
    if(f[l][r])return f[l][r];
    int maxn=0;
    for(int i=l;i<r;i++){
        maxn=max(maxn,dp(l,i)+dp(i+1,r)+a[l]*a[i+1]*a[r+1]);
    }
    return f[l][r]=maxn;
}
int main(){
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>a[i];
        a[i+n]=a[i];
    }
    for(int i=1;i<=n;i++){
        ans=max(ans,dp(i,i+n-1));
```

```
    }
    cout<<ans;
}
```

决斗

【题目描述】
Michel 最近迷上了买彩票。现在，某赌场就一轮决斗的结果开设了赌局。这个赌局同样被 Michel 盯上了，他决定购买这个彩票。
当然，身为有教养有文化的人， Michel 买彩票并不是胡乱买的。他在买之前进行了详尽的市场调查，并拿到了任意两个选手对决后的胜败情况。可以假定正式比赛的时候决斗后果也是一样的。
同时决斗的规则是这样的：
首先，选手们围成一个圈。每一回合随机抽出一个选手的号码，让他和他右边的选手决斗。开始时， 1 号右边的是 2 号， 2 号右边的是 3 号，依此类推，值得注意的是 n 号右边的是 1 号。
战败的选手则退出战场。例如： 2 号战败，则 1 号右边的就变成了 3 号。
现在，他找到了你，希望你能告诉他哪些选手可能赢。
【输入】
输入数据的第一行为一个整数 n，表示有 n 个选手。
接下来 n 行，每行 n 个整数，第 I+1 行第 J 列表示第 I 个选手与第 J 个选手对决后的胜败情况，0 表示选手 I 失败， 1 表示选手 I 获胜。
【输出】
输出数据的第一行为一个整数 k，表示有多少选手可能赢。
接下来 k 行，每行一个整数，从小到大输出这些选手的编号。
【样例输入】
2
0 0
1 0
【样例输出】
1
2
【数据规模】
对于 100%的数据： 1≤n≤500；
【分析】
题目有点难懂，但还是可以理解到这是一个区间合并类问题。我们设 f[i][j]表示 i 和 j 能否经过一系列决斗后相遇，如果可以，该值为 true，否则该值为 false，那么能够相遇的条件为：假设存在关键人物 k 在 i 和 j 之间，i 和 k 相遇，即 f[i][k]为 true，其 k 和 j 也能相遇，即 f[k][j]为 true，那么现在 i 和 j 能否相遇的关键是能否打败 k，如果 i 能打败 k，则 i 能与 j 相遇，或者 j 打败 k，j 也能和 i 相遇，状态转移方程为：
`f[i][j]=(f[i][k]&&f[k][j]&&(a[i][k]||a[j][k]))`
`a[i][k]`表示 i 与 k 决定的结果，赢为 true，输为 false。
因为存在多这个 k，只有有一种情况成立，那么 i 和 j 就能相遇，即：
`f[i][j]|=(f[i][k]&&f[k][j]&&(a[i][k]||a[j][k]))|i<k<j`

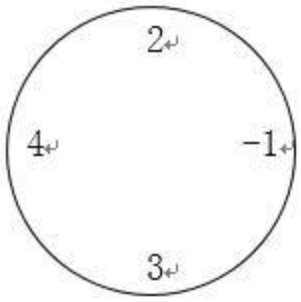
oiClass

因为题目是一个环形合并问题，将数据复制一份展开。
本题细节较多，详见代码。
【代码】

```
#include<iostream>
using namespace std;
bool a[1001][1001],f[1001][1001];
int n,x,ans;
int main(){
    cin>>n;
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++){
            cin>>x;  //以下四行表示将人数复制一份之后的关系
            a[i][j]=x;
            a[i+n][j]=x;
            a[i][j+n]=x;
            a[i+n][j+n]=x;
        }
    }
    for(int i=1;i<=2*n-1;i++)f[i][i+1]=true; //相邻的人肯定能遇到，故值为 true
    for(int j=1;j<=2*n;j++){
        for(int i=j-1;i>=1;i--){
            if(j-i>n)break;  //超过一个环，跳过
            for(int k=i+1;k<j;k++){
                f[i][j]|=(f[i][k]&&f[k][j]&&(a[i][k]||a[j][k]));
            }
        }
    }
    for(int i=1;i<=n+n;i++){
        if(f[i][i+n])ans++;  //自己能遇到自己，即该人赢了
    }
    cout<<ans<<endl;
    for(int i=1;i<=n;i++)if(f[i][i+n])cout<<i<<endl;  //输出谁赢了
}
```

数字游戏

【题目描述】
丁丁最近沉迷于一个数字游戏之中。这个游戏看似简单，但丁丁在研究了许多天之后却发觉原来在简单的规则下想要赢得这个游戏并不那么容易。游戏是这样的，在你面前有一圈整数（一共 n 个），你要按顺序将其分为 m 个部分，各部分内的数字相加，相加所得的 m 个结果对 10 取模后再相乘，最终得到一个数 k。游戏的要求是使你所得的 k 最大或者最小。
例如，对于下面这圈数字（n=4，m=2）：



当要求最小值时，((2-1) mod 10)×((4+3) mod 10)=1×7=7，要求最大值时，为((2+4+3) mod 10)×(-1 mod 10)=9×9=81。特别值得注意的是，无论是负数还是正数，对 10 取模的结果均为非负值。

丁丁请你编写程序帮他赢得这个游戏。

【输入】

输入文件第一行有两个整数，n（1≤n≤50）和 m（1≤m≤9）。以下 n 行每行有个整数，其绝对值不大于 10^4，按顺序给出圈中的数字，首尾相接。

【输出】

输出文件有两行，各包含一个非负整数。第一行是你程序得到的最小值，第二行是最大值。

【样例输入】

4 2
4
3
-1
2

【样例输出】

7
81

【分析】

本题是要求将环形区间分成 k 部分所获得的最大值和最小值。为了解决环形问题，我们将数据复制一份展开，最后枚举区间 i 到 i+n-1 即可解决环形问题。问题在于怎么解决将区间分成 k 份。我们设变量 f[k][i][j]表示将区间 i 到 j 分成 k 份所能获得的最大值，那么这个对应着区间分割问题，存在最后一个分割 t，i 到 t 分割为 k-1 份，t+1 到 j 独立一份，那么状态转移方程为：

f[k][i][j]=max(f[k-1][i][t]+sum[j]-sum[t])。

同理，定义变量 g[k][i][j]表示将区间 i 到 j 分成 k 份所能获得的最小值，状态转移方程为：

g[k][i][j]=min(g[k-1][i][t]+sum[j]-sum[t])。

本题需要特别注意取余的用法。

【代码】

```
#include<iostream>
using namespace std;
int n,m,a[110],sum[110],f[10][110][110],g[10][110][110],maxn,minn=1<<20;
int mod(int x){  //特别注意取余的函数，取余后结果必须为整数。
    return (x%10+10)%10;
}
```

oiClass

```
int main(){
    cin>>n>>m;
    for(int i=1;i<=n;i++){
        cin>>a[i];
        a[i+n]=a[i];  //复制数据展开
    }
    for(int i=1;i<=2*n;i++)sum[i]=sum[i-1]+a[i];  //求前缀和
    for(int i=1;i<=2*n;i++){  //边界初始化，分成一份时结果为区间和
        for(int j=i;j<=2*n;j++){
            f[1][i][j]=g[1][i][j]=mod(sum[j]-sum[i-1]);
        }
    }
    for(int k=2;k<=m;k++){
        for(int i=1;i<=2*n;i++){
            for(int j=1;j<=2*n;j++){
                g[k][i][j]=1<<20;  //最小值初始化为最大
                for(int t=i;t<j;t++){
                    f[k][i][j]=max(f[k][i][j],f[k-1][i][t]*mod(sum[j]-sum[t]
));
                    g[k][i][j]=min(g[k][i][j],g[k-1][i][t]*mod(sum[j]-sum[t]
));
                }
            }
        }
    }
    for(int i=1;i<=n;i++){  //枚举环形区间的最小值和最大值
        maxn=max(maxn,f[m][i][i+n-1]);
        minn=min(minn,g[m][i][i+n-1]);
    }
    cout<<minn<<endl<<maxn<<endl;
}
```