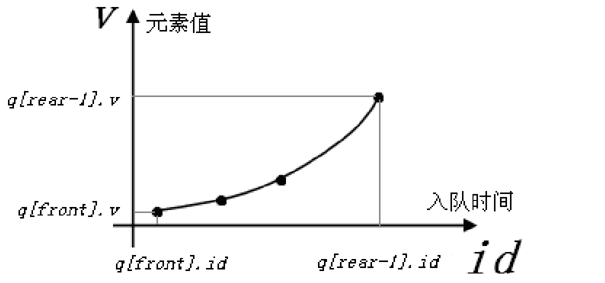
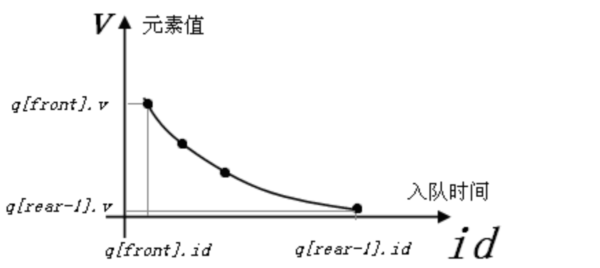
单调队列讲稿

一、单调队列概述

1、单调队列

单调队列是一种可在队尾插入、队首和队尾都可删除、且其内部元素值按照入队时间的先后具有单调性的**双端队列**。

<table>
<tr><td>首小队：单调递增，队首元素是最小的</td><td>首大队：单调递减，队首元素是最大的</td></tr>
<tr><td></td><td></td></tr>
<tr><td>q[front].v<=q[front+1].v<=…<=q[rear-1].v</td><td>q[front].v>=q[front+1].v>=…>=q[rear-1].v</td></tr>
</table>

2、单调队列的入队和出队操作

首小队：队首元素的入队时间最早，且是最小的；

首大队：队首元素的入队时间最早，且是最大的；

所以，在单调队列中选取最小（最大）的时间复杂度是 O(1)。因为单调队列在入队操作中，要保证队列元素的单调性，所以引入“淘汰”机制，如果新加入队尾元素会破坏了队列的单调性，则淘汰当前队尾元素。下面以首小队为例：

入队	出队(维护一段时间内的元素)

while(队尾元素 **大于** 待入队元素) rear--; //淘汰
q[rear++]=待入堆元素； //入队
	while(队首元素过期) front++; //淘汰过期
	v=q[front].v; //取队首元素（最小的）

例如，如果序列：1 3 -1 -3 10，按照从左到右的顺序依次进入队列，在进入队列的过程中，任何时候都要保持队列的单调递增，过程描如下：

元素 1 放入队列中，以初始化队列；

元素 3 要入队，队尾元素 1 比 3 小（虽然老但是很优秀），因此 3 可以直接入队，队列变为 1 3；

元素-1 要入队，淘汰 3、1（又老又不好），再将“-1”入队，队列变为-1；

元素-3 入队后，淘汰-1，队列变为-3；

元素 10 入队后，队列变为-3 10

3、单调栈

单调队列中，如果插入删出元素始终是在堆尾进行，这实质就是栈的操作，所以把这样的队列又称为**单调栈**。

二、单调队列的应用

例 1、滑动窗口（P1511）

给你一个长度为 N 的数组，一个长为 K 的滑动的窗体从最左移至最右端，你只能见到窗口的 K 个数，每次窗体向右移动一位，你的任务是找出窗口在各位置时的最大值和最小值。如下表窗口长度为 3：

窗口位置	最小值	最大值
[1 3 -1] -3 5 3 6 7	-1	3
1 **[3 -1 -3]** 5 3 6 7	-3	3
1 3 **[-1 -3 5]** 3 6 7	-3	5
……		

【样例】

8 3 1 3 -1 -3 5 3 6 7	-1 -3 -3 -3 3 3 3 3 5 5 6 7

【讲解】

上讲中用优先队列的最大队和最小队来维护窗口，维护一次的时间复杂度是 $O(\log_2 n)$

在这里我们用首小队和首大队来维护窗口集合。下面以首小队为例来做介绍：

1、单调队列的定义

```
struct data{ int v,id; };    //v 是元素的值，id 为该元素对应数组的下标（入队时间）
data q[100005];
int front,rear;
```

2、在首小队队尾部插入元素

为了保证首小队的递增性:q[front].v <= q[front-1].v <= … <= q[rear-1].v;

元素 a[i]进入窗口时，不是直接加入队尾，而要与队尾元素比较：当队尾元素 a[i]<q[rear-1].v 时，则淘汰队尾元素(删除)。因为 a[i]后入队，从此时开始，随着窗口向右滑动，若队尾元素 q[rear-1].v 是窗口中的元素，那么 a[i]也一定是窗口中的元素；又因为 a[i] q[rear-1].v 小，新来的不仅年轻而且更优，所以保留新来的 a[i]，而淘汰老的 q[rear-1].v。

```
while(front!=rear && a[i]<q[rear-1].v) rear--;   //淘汰队尾元素
q[rear++]=(data){a[i],i};   //入队
```

3、首小队首元素何时出队

由于我们只需要保存长度为 K 的窗口，如果队首元素的不在窗口之内，则应把它删除。所以当 a[i]进入窗口后，如果 i-q[rear-1].id+1>K，则应删除队首元素。

```
while(front!=rear && i-q[front].id+1>K) front++;  //淘汰过期元素
```

4、首小队实现计算窗口中最小元素的完整代码如下：

```
void solve1()  //首小队：单调递增
{
    front=rear=0;
    for(int i=1;i<=N;i++)  //每次向右延伸(i<=K)或滑动(i>K)
    {
        while(front!=rear && dq[rear-1].v>a[i]) rear--;  //淘汰老而不优的
        q[rear++]=(data){a[i],i};
        while(front!=rear && i-q[front].id+1>K) front++;  //淘汰太老：退休
        if(i>=K) printf("%d ",q[front].v);                //队首是最优秀的
    }
    printf("\n");
}
```

5、滑动窗口单调队列优化的时间复杂度分析

上面的代码看似有两重循环，但从数组元素来看，每个元素只入队一次，最多出队一次，所以这个算法的均摊时间复杂度是 O(n)的。比优先队列维护窗口要效率要高些！

滑动窗口的思想常用于优化一些算法，特别是动态规划！

例 2、连续子序列[2]

给定一个整数序列 A[1],A[2],…,A[n]，完成下面两个任务：

任务 1、如果这个序列是线性的，请计算长度不超过 K 的连续子序列，使得这个序列的和最大。

任务 2、如果这个序列是环形的，环形意味着 A[n]是 A[1]的左邻元素，A[1]是 A[n]的右邻元素。请计算长度不超过 K 的连续子序列，使得这个序列的和最大。

【输出格式】

第一行包含三个整数，表示任务 1 满足条件的连续序列的和、起始下标、结束下标，如果有多个子序列满足条件，则输出起始下标最小的，若仍然有多个，则输出长度最短的。

第二行同样包含三个整数，表示任务 2 的结果，输出要求同任务 1。

【样例】

6 3 //n,K,a,b -1 2 -6 5 -5 6 //A[1],A[2]…	6 3 6 //最大和，起始下标、结束下标 7 6 2

【数据范围】

1<=n<=100000 1<=K<=n 序列元素的绝对值不超过 2^30。

【讲解】

◆对于任务 1：

设前缀和：sum[i]=A[1]+A[2]+…+A[i];

设状态函数：f(i)=以 A[i]为最后一个元素的连续子序列的最大和。

则有：f(i)=max{ sum[i]-sum[j] | i-K<=j<i }

变形为：f(i)=sum[i] - min{ sum[j] | i-K<=j<i }

有上面方程中 min{ sum[j] | i-K<=j<i }实质就是在窗口：[i-K,…i-1]中选择一个最小的 sum。并且随着 i 向右走，窗口不断向右滑动。

请分别用优先队列和单调队列维护窗口元素集合完成这段代码，比较它们的时间效率。

◆对于任务 2：

变成一个环，按照环的处理方法，把两个 A[1]..A[n]首尾相接得到长度为 2*n 的序列：

A[1],A[2],…,A[n],A[1],A[2],…,A[n];

得到： B[1],B[2],…,B[n],B[n+1],…, B[2*n]

然后对线形序列 B[1],…,B[2*n]作任务 1 相同的算法。

只是最后在输出最大连续子序列的起始位置 a,b 时，应对应 A 序列的下表，即：if(b>n) b=b-n;

例 3、百层游戏（P2729）

小沐最近迷上一款称为“100 层”的游戏，其规则如下：

1、最开始角色在第一层第 x 个房间。

2、每一层包含 m 个房间，在同一层上，角色可以向一个方向走，即要么向左，要么向右，但最多经过连续 T 个房间后到达同层一个房间（这里你可理解为连续经过 T+1 个房间）。

3、在每一层也可直接走到上一层，即从第 i 层的第 j 个房间，可以直接上到第 i+1 层的第 j 个房间。

4、角色每经过一个房间和到达一个房间，会获得一定的分数。所以角色最终获得的分数是它经过房间的分数之和。

5、游戏的目标是角色从第 1 层走到最高层要获取最高的分数。

【样例】

3 3 2 1 //n,m,x,T 7 8 1 //第 1 层 m 个房间的分数 4 5 6 1 2 3	29 //能获得的最高分

1<=N<=100 1<=M<=10000 1<=X,T<=M -500<=房间分数<=500

【讲解】

类似 P1419《HB 办证》，动态规划解决：设 a[i][j]表示第 i 层的第 j 个房间的分数

◆状态定义：

f(i,j)=从第一层的第 x 个房间走到第 i 层第 j 个房间所获得的最大分数

◆边界分析：

f(0,x)=0，其它 f(i,j)=-inf

◆最后的答案：

Ans=max{ f(n,j) | 1<=j<=m }

◆状态转移方程：

$$dp(i,j)=\max\begin{Bmatrix}\max\{dp(i-1,k)+sum(j)-sum(k-1)\mid j-T<=k<=j\}\\ \max\{dp(i-1,k)+sum(k)-sum(j-1)\mid j<=k<=j+T\}\end{Bmatrix}$$

这里的 sum(j)表示第 i 层房间的前缀和。

直接实现上述状态转移方程（顺序选择最大值实现决策），时间复杂度为 O(n*m*T)。

下面来看优化决策：

设：　　v1=max{**f(i-1,k)+sum(j)-sum(k-1)** | j-T<=k<=j }

变形为：v1=max{**f(i-1,k)-sum(k-1) |** j-T<=k<=j } **+ sum(j)**

观 察：**max{f(i-1,k) -sum(k-1) | j-T<=k<=j }**

实质就是在窗口[j-T,j]中选择 f(i-1,k)-sum(k-1)最大的元素。

请自行推断第二个选项的方程变形：

部分代码如下：

```
void solve100()
{
   初始化 f[0][j]的边界;
   for(int i=1;i<=n;i++)
   {
      front=rear=0;  //清空队列 ;
      sum[0]=0;       //前缀和;
      for(int j=1;j<=m;j++)
      {
         sum[j]=sum[j-1]+a[i][j];        //第 i 行 1..j 的前缀和
         int t=f[i-1][j]-sum[j-1];      // 计算要入队的元素
         while(front!=rear && q[rear-1].v<t) rear--;   //入队（首大队）操作
         q[rear++]=(data){t,j};
         while(front!=rear && j-q[front].id>T) front++;
         f[i][j]=q[front].v+sum[j];     //队首元素为最大
      }

      实现：v2=max{f(i-1,k)+sum[k]-sum[j-1] | j<=k<=j+T }
      提示：最好转换为后缀和，然后 for(int j=m;j>0;j--)的顺序来做，即窗口向左滑动.
      ……
}
```

例 4、龙珠（P1867）

你得到了一个龙珠雷达，它会告诉你龙珠出现的时间和地点。

龙珠雷达的画面是一条水平的数轴，每一个窗口时间，数轴的某些点上会出现同一种龙珠，每当你获得其中一颗龙珠，其它龙珠就会消失。下一个窗口时间，数轴上又会出现另一种龙珠。总共有 n 个窗口时间，也就是总共有 n 种龙珠。

假设你会瞬间移动，你从数轴的 x 点移动到 y 点，耗时 0 秒，但是需要耗费|x-y|的体力。同时，挖出一颗龙珠也需要耗费一定的体力。请问，最少耗费多少体力，就可以收集齐所有种类的龙珠。

【输入格式】

第一行，三个整数 n,m,x_0,表示共有 n 个窗口时间，每个窗口时间会出现 m 个龙珠，x_0 是一开始你所处的位置。接下来有两个 n*m 的矩阵：

对于第一个矩阵，坐标为(i,j)的数字表示第 i 个窗口时间，第 j 个龙珠的位置。

对于第二个矩阵，坐标为(i,j)的数字表示第 i 个窗口时间，挖取第 j 个龙珠所需的体力。

【样例】

3 2 5 2 3 4 1 1 3 1 1 1 3 4 2	8 //所需最小体力

【数据范围】

数轴范围在 0 到 30000，挖一颗龙珠所需体力不超过 30000，结果保证在 int 范围

对于 50%的数据，1<=n<=50,1<=m<=500。

对于 100%的数据，1<=n<=50,1<=m<=5000。

【讲解】

第 i 个窗口时间可以选择出现的 m 个龙珠的某一个，后面窗口时间不会影响当前的选择，所以符合动态规划的无后效性原则。

设结构体成员：p[i][j].x=表示第 i 个时间窗口出现的第 j 个龙珠的位置。

结构体成员：：p[i][j].c=表示第 i 个时间窗口挖第 j 个龙珠需要耗费的体力。

把每个时间窗口的龙珠信息按照 x 由小到大排序。

◆状态定义：

f(i,j)= 前 i 个时间窗口，第 i 窗口收取的是第 j 个龙珠，并停留在这个位置的情况下，所耗费的最小体力。

◆边界分析：

f(1,j)= |p[1][j].x-x_0|+ p[1][j].c //在第 1 个时间窗口收集第 j 个龙珠需要的体力

◆最后的答案：

ans=min{ f(n,j) | 1<=j<=m }

◆状态转移方程：

f(i,j)=min{f(i-1,k)+|p[i][j].x-p[i-1][k].x| | 1<=k<=m } + p[i][j].c

用顺序插查找实现状态转移方程需要穷举 k，所以时间复杂度为：O(n*m*m)。

下面来看怎样优化决策：对方程变形，主要是把绝对值去掉：

●当 p[i-1][k].x<=p[i][j].x 时（从 p[i][j].x 的左边过来），方程变为：

f(i,j)=min{f(i-1,k)-p[i-1][k].x|1<=k<=m 且 p[i-1][k].x<=p[i][j].x }
+ p[i][j].x+p[i][j].c

实质是在 p[i][j].x 的左边选择一个最小的 f(i-1,k)-p[i-1][k].x;

由此可以用滑动窗口维护，只是这个窗口不断地向右延长，即 p[i-1][1]一直在窗口中。

●当 p[i-1][k].x>p[i][j].x 时（从 p[i][j].x 的右边过来），方程变为：

f(i,j)= min{f(i-1,k)+p[i-1][k].x | 1<=k<=m 且 p[i][k].x>p[i-1][j].x}
- p[i][j].x+ p[i][j].c

实质是在 p[i][j].x 的右边选择一个最小的 f(i-1,k)-p[i-1][k].x;

由此可以用滑动窗口维护，只是这个窗口不断地向左延长，即 p[i-1][m]一直在窗口中。

实现的部分代码如下：

```
int q[5005],front,rear;  //队列定义，这了不需要存元素的下标，为什么？
void solve100()
{
    边界;
    for(int i=2;i<=n;i++)
    {
       int k=1;
       front=rear=0;
       q[rear++]=oo;
       for(int j=1;j<=m;j++)  //a[i][j].x>=a[i-1][k].x
       {
           while(k<=m && a[i-1][k].x<=a[i][j].x)  //扫描(i,j)左边还未进队的点
           {
               int t=f[i-1][k]-a[i-1][k].x;    //计算要进队元素
               while(front!=rear && q[rear-1]>t) rear--;   //保持首小队的单调性
               q[rear++]=t;                          //进队
               k++;
           }
           f[i][j]=q[front]+a[i][j].c+a[i][j].x;  //实现决策
       }
      ……从右向左计算 f[i][j]的状态值。
    }
   选择并输出答案;
}
```

注意到本题滑动窗口的特点：窗口不断地向右延长，而左边不变。所以队列可以简化为一个变量：

minv= min{f(i-1,k)-p[i-1][k].x|1<=k<=m 且 p[i-1][k].x<=p[i][j].x }

所以代码可以简化为：

```
minv=oo;
for(int j=1;j<=m;j++)  //a[i][j].x>=a[i-1][k].x
{
     while(k<=m && a[i-1][k].x<=a[i][j].x)  //扫描(i,j)左边还未进队的点
     {
             int t=f[i-1][k]-a[i-1][k].x;    //计算要进队元素
             minv=min(minv,t);
             k++;
     }
     f[i][j]=minv+a[i][j].c+a[i][j].x;  //实现决策
}
```

前面讲过多次的“最大连续子序列”问题，用 O(n)的算法实现：

f(i) = sum[i] - min{ sum[j] | 0<=j<i }

其实质也是维护的左边不变，右边不断向右延伸的窗口！

单调队列除了优化动态规划外（滑动窗口的思想）；还有一类典型运用——区间扩展类的问题。例如下面的例题：

例题 5、序列的 M 数

n 个数排成一排。一个数的 M 数是指的在这个数的左边且比它小的数中最靠近（即最靠右）它的那个数，如果左边不存小于这个数的数，则它的 M 数上 0。 数学描述 A[i]的 M 数：max{ j | 0 < j < i 且 A[j] < A[i] }，若集合为空，则 A[i]的 M 数为 0。依次给出这 n 个数，请求出每个数相对应的 M 数。

【输入格式】

数据的第一行是一个正整数 n，表示一共有多少个数。

第二行有 n 个正整数，它们从左至右给出了数列中的 n 个数。这些数的绝对值保证小于 2^31。

【输出格式】

输出 n 行，第 i 行输出 A[i]的 M 数和 M 数在序列中的下标。

如果没有 M 数（即它左边的数都不比它小），请输出 0 0。

【输入样例】

7 3 1 2 7 6 7 4	0 0 0 0 1 2 2 3 2 3 6 5 2 3

【数据范围】

对于 50%的数据，n<=1000；

对于 100%的数据，n<=500 000。

【讲解】

令 L[i]=A[i]的 M 数的下标（左边比 A[i]且最靠右的元素的下标）。问题转换为计算 L[i]。

算法 1、无脑的暴力查找，时间复杂度为 $O(n^2)$

计算 L[i]，依次从 i-1 开始向左查找第一个比 A[i]小的元素。

```
void solve50()
{
    int i,j;
    for(i=1;i<=n;i++)
    {
        for(j=i-1;j>0;j--)    //顺序查找
          if(A[j]<A[i]) break;
        L[i]=j;
    }
    输出答案;
}
```

算法 2、有脑的记忆化查找，时间复杂度为 O(Kn)，其中 k 是一个常数

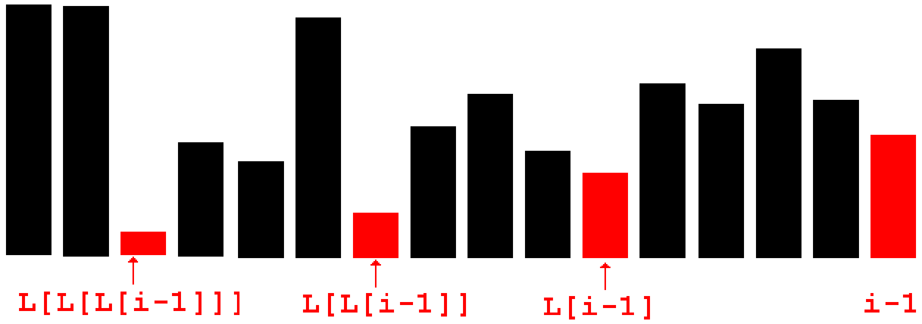
在暴力查找中，j 从 i-1 一步一步向左跳，但仔细分析不难发现，如果 A[j]>=A[i]，那么下次 j 应向左跳到 L[j]。因为 L[j]是小于 A[j]的最靠右元素的下标，说明 A[L[j]+1]..A[j-1]都不比 A[j]小，又因为 A[j]>=A[i]，那么 A[L[j]+1]..A[j-1]也不小于 A[i]，因此可直接跳过这些元素。

暴力算法中的查找中 j 的循环修改为：for(j=i-1;j>0; **j=L[j]**)。

算法 3、脑洞大开的单调性分析，时间复杂度为 O(n)

在算法 2 中，计算了 L[i-1]之后，有如下的关系：

…… **A[L[L[L[i-1]]]]** < **A[L[L[i-1]]]** < **A[L[i-1]]** < **A[i-1]**如下图。



把这样一个序列组成一个单调递增队列——首小队 struct{int v,id;}q[80008];

当前有：q[front].v < q[front+1].v < … <q[rear-1].v

计算 L[i]时，需要淘汰队尾大于等于 A[i]的，淘汰完后的队尾元素 q[rear-1]就是 A[i]左边第一个比 A[i]小的元素，然后把 A[i]进入队尾，保持队列单调递增：

```
while(front!=rear && A[i]<=q[rear-1].v)rear--;   //淘汰队尾元素
if(front!=rear) L[i]=q[rear-1].id; else L[i]=0;  //当前队尾元素是第一个小于 A[i]的
q[rear++]=A[i];   //A[i]插入队尾
```

完整的代码为：

```
void solve100x()
{
    front=rear=0;
    for(int i=1;i<=n;i++)
    {
        while(front!=rear && q[rear-1].v>=A[i]) rear--;
       if(front!=rear) L[i]=q[rear-1].id; else L[i]=0;
       q[rear++]=(data){A[i],i};
    }
    for(int i=1;i<=n;i++)
      printf("%d %d\n",A[L[i]],L[i]);
}
```

时间复杂度分析：因为每个元素进队列一次，最多出队列一次，所有时间复杂度为 O(n)

注意到本题维护单调队列的特点：元素只在队尾插入和删除，也只是读取队尾元素的值，因此其实质就是栈的操作。所以把这样的单调队列通常称为**“单调栈”**。

```
struct{int v,id;}st[80008];
int top;
void solve100x()
{
    top=0;
    for(int i=1;i<=n;i++)
    {
       while(top>0 && st[top].v>=A[i]) top--;
       if(top>0) L[i]=st[top].id; else L[i]=0;
       st[++top]=(data){A[i],i};
    }
    for(int i=1;i<=n;i++)
    printf("%d %d\n",A[L[i]],L[i]);
}
```

例 6、损坏的牛棚

FJ 买了一个矩形的 n 行 m 列的牧场。不幸的是，他发现某些 1 x 1 的区域被损坏了，所以它不可能在把整个牧场建造成牛棚了。 FJ 数了一下，发现有 p 个 1 x 1 的损坏区域。

现请你帮助他找到不包含损坏区域的面积最大的矩形的牛棚。

【输入格式】

第 1 行：三个空格隔开的整数 n, m, 和 P.

第 2..p+1 行：每行包含两个空格隔开的整数：x y，给出一个损坏区域的行号和列号。对同一个损坏区域可能有多次描述。

【输出格式】

牛棚的最大可能面积

【样例】

3 4 2 1 3 2 1	6

【数据范围】

测试点 1：n=5, m=6, p=5
测试点 2：n=10, m=10, p=10
测试点 3：n=20, m=15, p=17
测试点 4：n=50, m=50, p=100
测试点 5：n=98, m=100, p=1089
测试点 6：n=50, m=300, p=3000
测试点 7：n=1000, m=1000, p=5000
测试点 8：n=2000, m=1500, p=25000
测试点 9：n=3000, m=2500, p=30000
测试点 10：n=3000, m=3000, p=100000

【讲解】

设二维数组 int A[maxn][maxn]表示给定的 n*m 的牧场，其中坏格子表示为 A[x][y]=1，好格子表示为 A[x][y]=0;

算法 1、无脑暴力枚举，时间复杂度$O(n^3 \times m^3)$

枚举矩形的左上角(x,y)和右下角(xx,yy)，然后检查这个矩形中是否有坏格子。

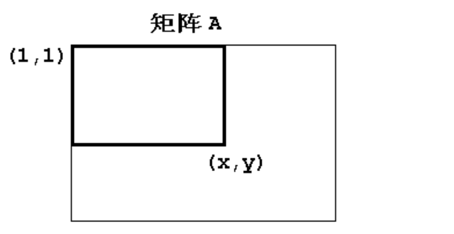
```
void solve20()
{
    ans=0;
    for(int x=1;x<=n;x++)   //枚举左上角
    for(int y=1;y<=m;y++)
    for(int xx=x;xx<=n;xx++)  //枚举右下角
    for(int yy=y;yy<=m;yy++)
      if(check(x,y,xx,yy)) //如果不包含坏格子
        ans=max(ans,(xx-x+1)*(yy-y+1));
    }
    printf("%d\n",ans);
}
```

```
//检查矩形(x,y)->(xx,yy)是否包含坏格子
bool check(int x,int y,int xx,int yy)
{
    for(int i=x;i<=xx;i++)
    for(int j=y;j<=yy;j++)
      if(A[i][j]) return false;
    return true;
}
```

算法 2、有脑稍加分析的枚举，时间复杂度$O(n^2 \times m^2)$

改造一下 A[x][y]，当(x,y)是坏格子时，A[x][y]=-inf（请体会妙处），好格子时，A[x][y]=0。设前缀和 sum[x][y]=以 a[1][1]为左上角，a[x][y]为右下角的矩形区域中所有格子的和。即：

```
sum[x][y]= a[1][1] + a[1][2] + … + a[1][y]
          +a[2][1] + a[2][2] + … + a[2][y]
          ……
          +a[x][1] + a[x][2] + … + a[x][y]
```



sum[x][y]的计算可以用递推：

$$sum[x][y] = sum[x-1][y] + sum[x][y-1] - sum[x-1][y-1] + A[i][j]$$

那么此次对于子矩阵：左上角(x,y)->右下角(xx,yy)的和为：

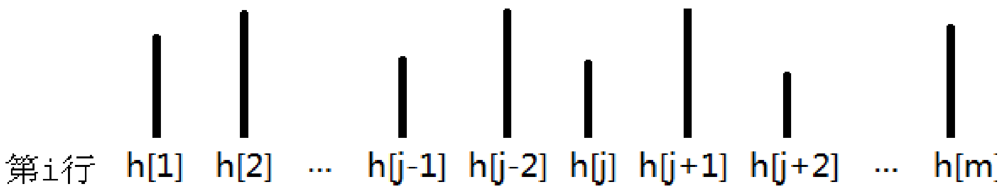
$$T = sum[xx][yy] - sum[x-1][yy] - sum[xx][y-1] + sum[x-1][y-1]$$

显然，如果 T<0 则，该子矩阵含有坏格子，否则不含坏格子。根据这个条件，可以把算法 1 的 check 函数的时间复杂度降为 O(1)。

```
int sum[maxn][maxn],inf=3000*3000+1000;
void solve40()
{
    memset(sum,0,sizeof(sum));
    for(int i=1;i<=n;i++)
    for(int j=1;j<=m;j++)
    {
          sum[i][j]=sum[i-1][j]+sum[i][j-1]-sum[i-1][j-1];
          if(A[i][j]) sum[i][j]+=-inf;
    }
    for(int x=1;x<=n;x++)
    for(int y=1;y<=m;y++)
    for(int xx=x;xx<=n;xx++)
    for(int yy=y;yy<=m;yy++)
    {
         int T=sum[xx][yy]-sum[x-1][yy]-sum[xx][y-1]+sum[x-1][y-1];
         if(T>=0) ans=max(ans,(xx-x+1)*(yy-y+1));
    }
    printf("%d\n",ans);
}
```

算法 3、脑洞大开的枚举，时间复杂度$O(n \times m)$

在这个算法中把坏格子看成一个无法逾越的障碍，然后设 h[j]表示格子(i,j)能向上能走到的最大高度高度，如果(i,j)本身就是一个障碍，则 h[j]=0。设第 i 行的 h[1]..h[m]对应的图形如下：



如果以格子(i,j)为下边线上的一点，h[j]为高的子矩阵的最大宽度是多少呢？这个问题就是我们前面学习过的《广告印刷》一题。

设 L[j]表示 h[j]左边小于 h[j]且最靠右的元素下标（例题 5 列的 M 数）

设 R[j]表示 h[j]右边小于 h[j]且最靠左的元素下标

那么，以格子(i,j)为下边线的点，h[j]为高的子矩阵的最大宽度就是 R[j]-L[j]-1。

计算 L[j]和 R[j]的快速方法在例题 6 都讲到了。

```
void solve100()
{
    ans=0;
    memset(h,0,sizeof(h));
    for(int i=1;i<=n;i++)
    {
       for(int j=1;j<=m;j++)   //计算第 i 行的 h[1]..h[m]
         if(A[i][j]==1) h[j]=0; else h[j]++;

       for(int j=1;j<=m;j++) //计算 L[j]
       {
            int k;
            for(k=j-1;k>0;k=L[k])
                if(h[k]<h[j]) break;
            L[j]=k;
       }

       for(int j=m;j>0;j--) ////计算 R[j]
       {
            int k;
            for(k=j+1;k<=m;k=R[k])
               if(h[k]<h[j]) break;
            R[j]=k;
       }

       for(int j=1;j<=m;j++)    //以 h[j]为高的最大子矩阵
          ans=max(ans,h[j]*(R[j]-L[j]-1));
      }
      printf("%d\n",ans);
}
```

这个算法的时间复杂度为：$O(n \times k * m)$，其中 k 是一个很小的常数。
类似例 5，把上面计算 L[j]和 R[j]改成单调队列（栈），则平均时间复杂度为：$O(n \times m)$
请自行把上面的算法改成单调队列的！