【例 1 锯木场选址】(**CEOI2004**)

从山顶上到山底下沿着一条直线种植了 n 棵树。当地的政府决定把他们砍下来。为了不浪费任何一棵木材，树被砍倒后要运送到锯木厂。木材只能按照一个方向运输：朝山下运。山脚下有一个锯木厂。另外两个锯木厂将新修建在山路上。你必须决定在哪里修建两个锯木厂，使得传输的费用总和最小。假定运输每公斤木材每米需要一分钱。

任务

你的任务是写一个程序：

从标准输入读入树的个数和他们的重量与位置

计算最小运输费用

将计算结果输出到标准输出

输入

输入的第一行为一个正整数 n——树的个数（$2 \le n \le 20\ 000$）。树从山顶到山脚按照 1，2……n 标号。接下来 n 行，每行有两个正整数（用空格分开）。第 $i+1$ 行含有：v_i——第 i 棵树的重量（公斤为单位）和 d_i——第 i 棵树和第 $i+1$ 棵树之间的距离，$1 \le v_i \le 10\ 000$，$0 \le d_i \le 10\ 000$。最后一个数 d_n，表示第 n 棵树到山脚的锯木厂的距离。保证所有树运到山脚的锯木厂所需要的费用小于 2000 000 000 分。

输出

输出只有一行一个数：最小的运输费用。

样例输入

```
9
1 2
2 1
3 3
1 1
3 2
1 6
2 1
1 2
1 1
```

样例输出

```
26
```

在解决这一问题时，首先我们要明确，将锯木厂建立在相邻两棵树之间是没有任何意义的，否则我们可以将这样的锯木厂上移到最近的一棵树处，此时运送上方树木的费用减少，运送下方树木的费用没有变化，总费用降低。

为了方便讨论，我们先作如下定义：

假设山脚锯木场处也有一棵树，编号为 $n+1$，并且 v[n+1]=d[n+1]=0。

$sumw[i]$ 表示第 1 棵树到第 i 棵树的质量和，即 $sumw[i]=\sum_{j=1}^{i} w[j]$。

$sumd[i]$ 表示第 1 棵树到第 i 棵树的距离，即 $sumd[i]=\sum_{j=1}^{i-1} d[j]$。特别的，有 $sumd[1]=0$，表示第 1 棵树到自己的距离为 0。

c[i]表示在第 i 棵树处建一个锯木厂，并且将第 1 到第 i 棵树全部运往这个伐木场所需的费用。显然有 c[i]=c[i-1]+sumw[i-1]*d[i-1]。特别的，有 c[1]=0。

w[j,i]表示在第 i 棵树处建一个锯木场，并且将第 j 到第 i 棵树全部运往这个锯木场所需的费用。则 w[j,i]=c[i]-c[j-1]-sumw[j-1]*(sumd[i]-sumd[j-1])。特别的，当 i<=j 时 w[j,i]=0。

综上可知，求出所有 sumw[i],sumd[i]与 c[i]的时间复杂度为 O(n)，此后求任意 w[j,i]的时间复杂度都为 O(1)。

设 f[i] 表 示 在 第 i 棵 树 处 建 立 第 二 个 锯 木 场 的 最 小 费 用 ， 则 有 $f[i]=\min\limits_{1\le j\le i-1}\{c[j]+w[j+1,i]+w[i+1,n+1]\}$ 。直接用这个式子计算的时间复杂度为 $O(n^2)$，由于问题规模太大，直接使用这一算法必然超时，因此我们必须对算法进行优化。在讨论如何进行优化以前我们首先证明下面这一猜想。

[猜想]

如果在位置 i 建设第二个锯木厂,第一个锯木厂的位置是 j 时最优。那么如果在位置 i+1 建设第二个锯木厂,第一个锯木厂的最佳位置不会小于 j。

证明：假设第 1 个锯木厂建立在第 j 个处时,f[i]取得最优值,且此时 j 为 f[i]取得最优值时的最小值,如下图所示.此时 f[i]=c[j]+w[j+1,i]+w[i+1,n+1]

则对于任意的 k<j,

c[k]+w[k+1,i]+w[i+1,n+1]>c[j]+w[j+1,i]+w[i+1,n+1]

$\Longleftrightarrow$ c[k]+w[k+1,i]>c[j]+w[j+1,i]

$$\Longleftrightarrow c[k]+w[k+1,i]+(\overbrace{sumw[i]-sumw[k]}^{\text{第 k+1 至第 i 的总重量}})*\overset{\text{第 i 至第 i+1 的距离}}{d[i]}>c[j]+w[j+1,i]+(sumw[i]-sumw[k])*d[i]$$

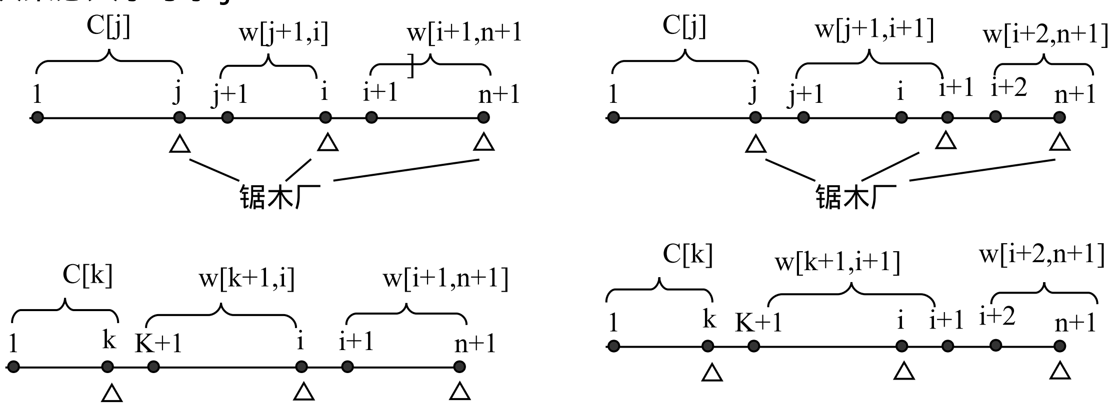
$$\Longrightarrow c[k]+w[k+1,i]+(sumw[i]-sumw[k])*d[i]>c[j]+w[j+1,i]+(sum\overbrace{w[i]-sumw[j]}^{\textbf{第 j+1 至第 i 的总重量, 必小于第 k+1 至第 i 的总重量}})*d[i]$$

$\Longleftrightarrow$ c[k]+w[k+1,i+1]>c[j]+w[j+1,i+1]

$\Longleftrightarrow$ c[k]+w[k+1,i+1]+w[i+2,n+1]>c[j]+w[j+1,i+1]+w[i+2,n+1]

即求 f[i+1]时，决策 k 比决策 j 来得差！因此，当 f[i]的第一个最佳决策为 j 时，f[i+1]的第一个最优决策必大于等于 j!

证毕！



【算法的改进】

令 s[k,i]表示决策变量取 k 时 f[i]的值，即 s[k,i]=c[k]+w[k+1,i]+w[i+1,n+1]。

设 k1<k2,则有 s[k1,i]-s[k2,i]

=(c[k1]+w[k1+1,i]+w[i+1,n+1])-(c[k2]+w[k2+1,i]+w[i+1,n+1])

=(c[k1]+c[i]-c[k1]-sumw[k1]*(sumd[i]-sumd[k1])

- (c[k2]+c[i]-c[k2]-sumw[k2]*(sumd[i]-sumd[k2])

=sumw[k2]*(sumd[i]-sumd[k2])- sumw[k1]*(sumd[i]-sumd[k1])

=sumd[i](sumw[k2]-sumw[k1])-(sumw[k2]*sumd[k2]-sumw[k1]*sumd[k1])

若 s[k1,i]-s[k2,i]<0，则有：

sumd[i](sumw[k2]-sumw[k1])<(sumw[k2]*sumd[k2]-sumw[k1]*sumd[k1])

即(sumw[k2]*sumd[k2]-sumw[k1]*sumd[k1])/ (sumw[k2]-sumw[k1])> sumd[i]

或(sumw[k1]*sumd[k1]-sumw[k2]*sumd[k2])/ (sumw[k1]-sumw[k2])> sumd[i]

我们令 g[k1,k2]=不等式左边，当 g[k1,k2]>sumd[i]时 s[k1,i]-s[k2,i]<0。

由上面已经证明的猜想，说明决策变量 j 是单调的，(即 f[i+1]的决策 j2 必大于等于 f[i]时的决策 j1)此时问题就好解决了。我们可以维护一个特殊的队列 k，这个队列只能从队尾插入，但是可以从两端删除。这个队列满足 k1<k2<k3<…<k_p 以及 g[k_1,k_2]<g[k_2,k_3]<…<g[k_{p-1},k_p]。

1．计算状态 f[i]前，若 g[k_1,k_2]<sumd[i]，表示 s[k1,i]-s[k2,i]>0，即决策 k1 没有 k2 优，应当删除队首,将 k1,k2,指针后移，直至 g[k_1,k_2]>sumd[i]。此时, s[k1,i]-s[k2,i]<0, 即决策 k1 比 k2 优。

2．计算状态 f[i]时，直接使用决策 k1,f[i]=c[k1]+w[k1+1,i]+w[i+1,n+1]。O(1)代价！

（由当前的决策序列：sumd[i]<g[k1,k2]<g[k2,k3]<…<g[k_{p-1},kp]知，决策 k1 优于 k2,k2 优于 k3,…,k_{p-1}优于 k_p。）

3．计算状态 f[i]后向队列插入新的决策 i。

若 g[k_{p-1},k_p]>g[k_p,i]，此时,如果求某个 f[r]时选择 K_p 为决策,则 k_{p-1} 决策必然没有 k_p 好,即 s[k_{p-1},r]-s[k_p,r]>0 => g[k_{p-1},k_p]<sumd[r]=> g[k_p,i]<sumd[r]

=> s[k_p,r]-s[i,r]>0,此时,说明决策 i 必然比 k_p 来得优

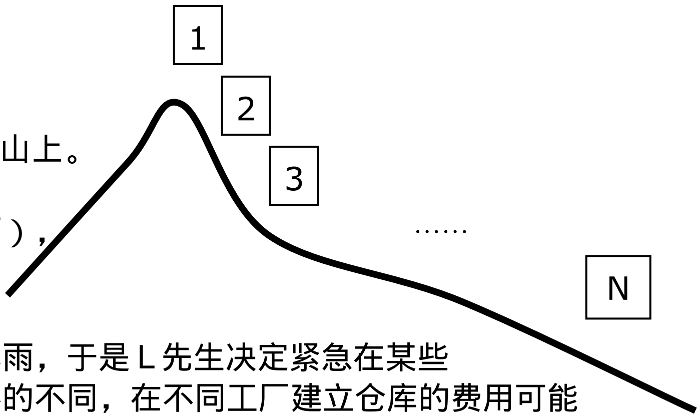
表示在 k_p 比 k_{p-1} 优之前 i 就将比 k_p 优，k_p 没必要保存。并且，根据前面的猜想，对于以后的状态，决策 i 也会比 k_p 优。

因此删除 k_p，将指针 p 前移，直至 g[k_{p-1},k_p]< g[k_p,i]。

队列中的元素只会入队一次，出队一次，维护队列 k 的总复杂度为 O(n)，因此每个状态 f[i] 计算的均摊复杂度都为 O(1)，整个算法的时间复杂度为 O(n)。问题得到解决！

注意：上例的决策单调性的条件是，每棵树的重量均>0,这样，重量前缀数组 sumw 满足单调性,才会满足第 j+1 至第 i 的总重量,必小于第 k+1 至第 i 的总重量(k<j)。

【例 **2** 仓库建设】(浙江 **2007** 年省选）



L 公司有 N 个工厂，由高到底分布在一座山上。如右图所示，工厂 1 在山顶，工厂 N 在山脚。

由于这座山处于高原内陆地区（干燥少雨），L 公司一般把产品直接堆放在露天，以节省费用。突然有一天，L 公司的总裁 L 先生接到气象部门的电话，被告知三天之后将有一场暴雨，于是 L 先生决定紧急在某些工厂建立一些仓库以免产品被淋坏。由于地形的不同，在不同工厂建立仓库的费用可能是不同的。第 i 个工厂目前已有成品 Pi 件，在第 i 个工厂位置建立仓库的费用是 Ci。对于没有建立仓库的工厂，其产品应被运往其他的仓库进行储藏，而由于 L 公司产品的对外销售处设置在山脚的工厂 N，故产品只能往山下运（即只能运往编号更大的工厂的仓库），当然运送产品也是需要费用的，假设一件产品运送 1 个单位距离的费用是 1。假设建立的仓库容量都都是足够大的，可以容下所有的产品。你将得到以下数据：

工厂 i 距离工厂 1 的距离 Xi（其中 X1=0）;工厂 i 目前已有成品数量 Pi;在工厂 i 建立仓库的费用 Ci;

请你帮助 L 公司寻找一个仓库建设的方案，使得总的费用（建造费用+运输费用）最小。

【输入文件】

输入文件 storage.in 第一行包含一个整数 N，表示工厂的个数。

接下来 N 行每行包含两个整数 Xi, Pi, Ci, 意义如题中所述。

【输出文件】

输出文件 storage.out 仅包含一个整数，为可以找到最优方案的费用。

【样例输入】

3

0 5 10

【数据规模】

对于 20%的数据，N ≤500;

对于 40%的数据，N ≤10000;

对于 100%的数据，N ≤1000000。

所有的 Xi, Pi, Ci 均在 32 位带符号整数以内，

保证中间计算结果不超过 64 位带符号整数。

5 3 100
9 6 10
【样例输出】
32
【样例说明】
在工厂 1 和工厂 3 建立仓库，建立费用为 10+10=20，运输费用为(9-5)*3 = 12，总费用 32。如果仅在工厂 3 建立仓库，建立费用为 10，运输费用为(9-0)*5+(9-5)*3=57，总费用 67，不如前者优。

【初步分析】

1.状态表示

以 sump[i]表示第 1 个工厂至第 i 个工厂中的总的产品数量；

以 sumw[i]表示将第 1 个工厂至第 i 个工厂的所有产中运至第 i 个工厂的费用；

以 w[i,j]表示将第 i 个工厂至第 j 个工厂的所有产中运至第 j 个工厂的费用。

W[i,j]=sumw[j]-sumw[i-1]-sump[i-1]*(x[j]-x[i-1])

所有的 sump[1..n]、sumw[1..n]可在 O(n)的总次数内算出，每个 w[i,j]可以 O(1)算出。

以 f[i]表示在第 i 个工厂建立一个仓库，前 i 个工厂需要的最小费用；所求为 f[n]。

2.状态转移方程

$$f[i] = \mathop{Min}_{0 \le k \le i-1} \{ f[k] + w[k+1, i] + c[i] \}$$

$$f[0] = 0$$

时间复杂度为 O(n^2)。

下面使用例 1 相似的方法，将算法的时间复杂度降为 O(n)。

【猜想】

求 f[i]函数时的决策 k 关于 i 单调。

证明：

假设求得 f[i]时的最优决策为 k,对于任意的 j<k,我们来证明求 f[i+1]时决策 k 必优于决策 j。

$$f[k] + w[k+1,i] + c[i] < f[j] + w[j+1,i] + c[i]$$

$$\rightleftarrows f[k] + w[k+1,i] < f[j] + w[j+1,i]$$

$$\rightleftarrows f[k] + sumw[i] - sumw[k] + sump[k] \times (x[i] - x[k]) < f[j] + sumw[i] - sumw[j] + sump[j] \times (x[i] - x[j])$$

$$\rightleftarrows f[k] - sumw[k] + sump[k] \times (x[i] - x[k]) < f[j] - sumw[j] + sump[j] \times (x[i] - x[j])$$

$$\rightleftarrows f[k] - sumw[k] - sump[k] \times x[k] < f[j] - sumw[j] - sump[j] \times x[j] + x[i] \times (sump[k] - sump[j])$$

$$\Rightarrow f[k] - sumw[k] - sump[k] \times x[k] < f[j] - sumw[j] - sump[j] \times x[j] + x[i+1] \times (sump[k] - sump[j])$$

$$\rightleftarrows f[k] - sumw[k] + sump[k] \times (x[i+1] - x[k]) < f[j] - sumw[j] + sump[j] \times (x[i+1] - x[j])$$

$$\rightleftarrows f[k] + sumw[i+1] - sumw[k] + sump[k] \times (x[i+1] - x[k])$$

$$< f[j] + sumw[i+1] - sumw[j] + sump[j] \times (x[i+1] - x[j])$$

$$\rightleftarrows f[k] + w[k+1, i+1] < f[j] + w[j+1, i+1]$$

$$\rightleftarrows f[k] + w[k+1, i] + c[i+1] < f[j] + w[j+1, i] + c[i+1]$$

证毕！

【算法的优化】

令 s[k,i]表示当决策为 k 时求 f[i]的表达式的值,即 s[k,i]=f[k]+w[k+1,i]+c[i]。

对于 k1<k2，

$$
\begin{aligned}
& s[k1,i]-s[k2,i] \\
& =f[k1]+w[k1+1,i]+c[i]-(f[k2]+w[k2+1,i]+c[i]) \\
& =f[k1]+sumw[i]-sumw[k1]-sump[k1]\times(x[i]-x[k1]) \\
& -(f[k2]+sumw[i]-sumw[k2]-sump[k2]\times(x[i]-x[k2])) \\
& =(f[k1]+sumw[k1]+sump[k1]\times x[k1])-(f[k2]+sumw[k2]+sump[k2]\times x[k2]) \\
& -x[i]\times(sump[k1]-sump[k2])
\end{aligned}
$$

当 $s[k1,i]-s[k2,i]<0$ 时，

$$
\begin{aligned}
& (f[k1]+sumw[k1]+sump[k1]\times x[k1])-(f[k2]+sumw[k2]+sump[k2]\times x[k2]) \\
& < x[i]\times(sump[k1]-sump[k2]) \\
& \Leftrightarrow \frac{(f[k1]+sumw[k1]+sump[k1]\times x[k1])-(f[k2]+sumw[k2]+sump[k2]\times x[k2])}{(sump[k1]-sump[k2])} > x[i]
\end{aligned}
$$

令 g[k1,k2]=不等式左边的值,它只与 k1 和 k2 有关，与 i 无关。

瑰丽华尔兹 NOI2005 day1

给定一个 N 行 M 列的矩阵，矩阵中的某些方格上有障碍物。有一个人从矩阵中的某个方格开始滑行。每次滑行都是向一个方向最多连续前进 c 格（也可以原地不动）（两次滑行的 c 值不一定相同）。但是这个人在滑行中不能碰到障碍物。现按顺序给出 K 次滑行的方向（东、南、西、北中的一个）以及对应的 c，试求这个人能够滑行的最长距离（即格子数）。

数据范围：1≤N,M≤200，K≤200，$\sum_{i=1}^{k} c_i$ ≤40000

分析：

本题是一个求最值的问题。根据题目中 K 次滑行的有序性以及数据范围，可以很容易设计出这样一种动态规划算法：

令 $f(k,x,y)$=此人 k 次滑行后到达(x,y)方格时已经滑行的最长距离。动态规划的状态转移方程如下（以下仅给出向东滑行的状态转移方程，其他 3 个方向上的转移方程可以类似地推出）：

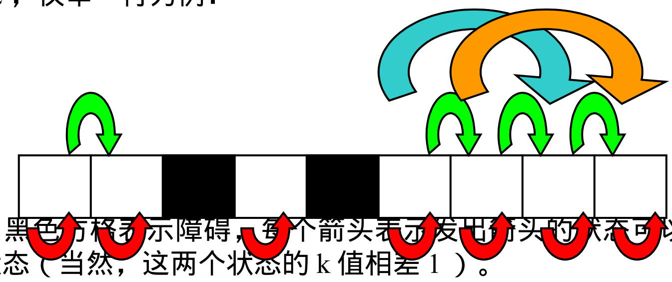
$f(0,startx,starty)=0$

$f(k,x,y)=\max\{f(k-1,x,y),f(k-1,x,y-1)+1,f(k-1,x,y-2)+2,\ldots\ldots,f(k-1,x,y')+y-y'\}$

（其中 y'为满足 y=1 或(x,y'-1)上有障碍或 $y'=y-c_k$ 的最大值）

从图 14 中可以很清楚地看出动态规划转移的条件：

令 $c_k=2$，仅举一行为例：



其中，黑色方格表示障碍，每个箭头表示发出箭头的状态可以去改进被箭头指向的状态（当然，这两个状态的 k 值相差 1 ）。

图 14

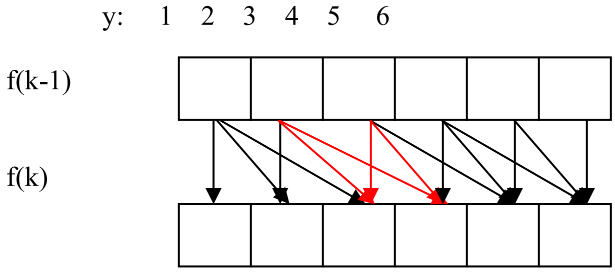
现在来分析这个动态规划算法的时间复杂度。

显然，状态总数为 O(KMN)，而每次状态转移在最坏情况下的时间复杂度为 O(max{M,N})，因此总的时间复杂度为 $O(KMN*\max\{M,N\})=O(1.6*10^9)$，难以承受。因此，我们需要对这个动态规划算法进行优化。

先不考虑障碍，可以看出，在求 f(k,x,y)与 f(k,x,y+1)时，很多状态我们都重复考虑了。以图 15 为例，求 f(k,1,3)与 f(k,1,4)时都用到了 f(k-1,1,2)和 f(k-1,1,3). 在先前介绍的动态规划算法中，这样的重复大量出现，导致算法的低效。那么，如何才能有效地防止这类重复计算的发生呢？让我们先来研究动态规划方程：

$f(k,x,y)=\max\{f(k-1,x,y),f(k-1,x,y-1)+1,f(k-1,x,y-2)+2,\ldots\ldots,f(k-1,x,y')+y-y'\}$

令 $c_k=2$， x=1



对于一个具体的例子 k=2，x=1，$c_2=2$ 可以列出如下等式：

$f(2,1,1)=\max\{f(1,1,1)\}$

$f(2,1,2)=\max\{f(1,1,1)+1,f(1,1,2)\}$

$f(2,1,3)=\max\{f(1,1,1)+2,f(1,1,2)+1,f(1,1,3)\}$

$f(2,1,4)=\max\{f(1,1,2)+2,f(1,1,3)+1,f(1,1,4)\}$

……

如果我们定义一个序列 a，使得 $a_i=f(1,1,i)-i+1$，则以上等式可以写成：

$f(2,1,1)=\max\{a_1\}=\max\{a_1\}$

$f(2,1,2)=\max\{a_1+1,a_2+1\}=\max\{a_1,a_2\}+1$

$f(2,1,3)=\max\{a_1+2,a_2+2,a_3+2\}=\max\{a_1,a_2,a_3\}+2$

$f(2,1,4)=\max\{a_1+3,a_2+3,a_3+3,a_4+3\}=\max\{a_2,a_3,a_4\}+3$

……

显然，在应用了 a 序列之后，我们就可以只关注 a 序列而不必为每个 a_i 加上一个不同的值，从而简化了操作。

现在，我们可以加入对障碍物的考虑。例如对于图 15 中的状态，我们依次要求 $\max\{a_1\}$,$\max\{a_1,a_2\}$, $\max\{a_4\}$, $\max\{a_6\}$, $\max\{a_6,a_7\}$, $\max\{a_6,a_7,a_8\}$, $\max\{a_7,a_8,a_9\}$. 考虑 max 函数中的序列，可以发现，每次都是在序列的尾部添上一个 a 值（遇到障碍物除外），并有时在头部删去一些 a 值（如果区间长度超过 c_k+1，就删去 1 个 a 值；如果遇到一个障碍物，则清空整个序列），而且这个序列中 a 的下标一定是连续的。有了这些条件与限制，就可以运用一种专门计算区间最值的数据结构——线段树[1]。每次根据 a 值建立一棵线段树，然后对于需要求最大值的区间，直接在线段树中查找。对于一行来说，建立线段树的时间复杂度为 O(M)，每次查找的时间复杂度为 $O(\log_2 M)$，而对于前文所说的区间处理，由于每个 a 值最多进入区间一次，被删除一次，所以维护区间的总的时间复杂度为 O(M). 这样，整个算法的时间复杂度降为 $O(KMN\log_2(\max\{m,n\}))$.

然而，线段树的编程复杂度较高，初始化、插入、查找分别需要一个子过程，容易出错。并且 $O(KMN\log_2(\max\{m,n\}))$的复杂度再乘以线段树操作中的系数，还是比较慢，不能令人满意。实际上，有一种基本数据结构——队列可以非常好地解决这个问题。

前面已经对于序列的插入与删除进行过讨论。其中只在一端插入，另一端删除的特性恰好符合队列的性质。但是，这里是要求队列中所有数的最大值，普通的队列可以胜任这个操作吗？让我们首先来分析一下如何存储队列中的数。

如图 16 所示，对于已经出现在队列中的 a_2 与 a_3，如果 $a_2 \le a_3$，则 a_2 是没有必要出现在队列中的。因为根据队列的插入与删除原则可以推导出，如果队列中已经出现 a_3 了，则在 a_2 被删除之前，a_3 是一定不会被删除的。因此，a_2 与 a_3 会一直同时出现在队列中，直至 a_2 被删除。但是 $a_2 \le a_3$，因此队列中的最大值永远不会是 a_2，也就没有必要存储 a_2.

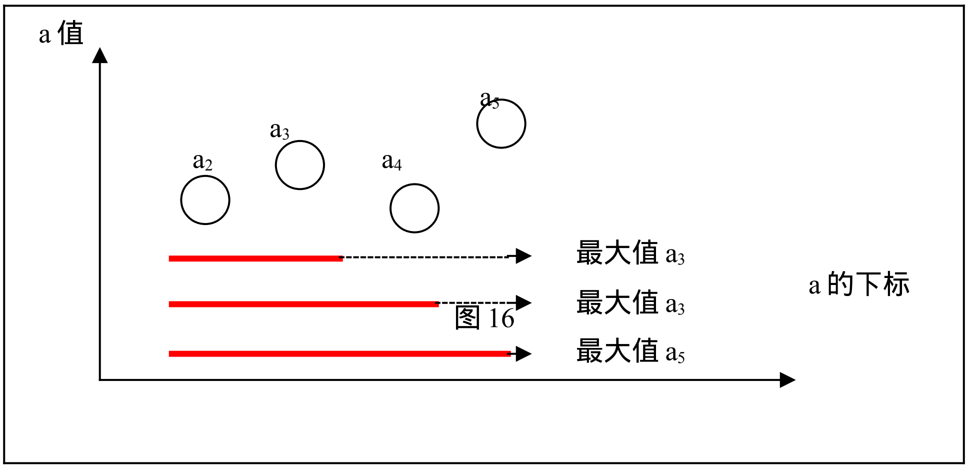


图 16

根据这一条重要的性质，可以立刻推导出"有必要"存储在队列中的 a 值的大小关系——严格递减。也就是说，存储在队列中的 a 值是依次减小的，而队头元素的值为最大值，也就是当前队列中所有数字（无论是否存储在队列中）的最大值。这样，每次可以取出位于队头的 a 值作为最大值。但是一个新的问题摆在我们面前——如何实时维护队列，即如何正确地插入或删除元素。

首先研究删除操作。很显然，如果队头 a 值的下标对应的方格与当前处理的方格之间的距离已经大于 c_k，则直接将它从队列中删除，即队头指针加一。

接着是插入操作。根据前文中对于队列中元素大小关系的讨论可以得知，插入一个元素 a_i 后，队列中不能有元素 a_j 满足 $a_j \le a_i$. 于是，我们可以从队尾开始，依次删除掉不大于 a_i 的 a 值，直到队列中剩下的元素都大于 a_i. 此时就可以将 a_i 插入队尾。（插入与删除过程见图 17）。

很显然，每次求队列中所有元素的最大值可以直接查看队头元素。根据此队列性质，队头元素一定是队列中所有元素里最大的一个。

由于在一行中，一个元素最多被插入一次，删除一次，所以插入与删除的总时间复杂度为 $O(\max\{M,N\})$，而每次询问最大值的时间复杂度仅为 O(1). 综上所述，此算法的时间复杂度为 O(KMN)，是

[1] 线段树是一棵以线段作为基本单位的二叉树，在线段树中进行的区间插入、删除以及询问的时间复杂度均为 $O(\log_2 N)$. 关于线段树的介绍详见参考书目[2].

一个十分优秀的算法。

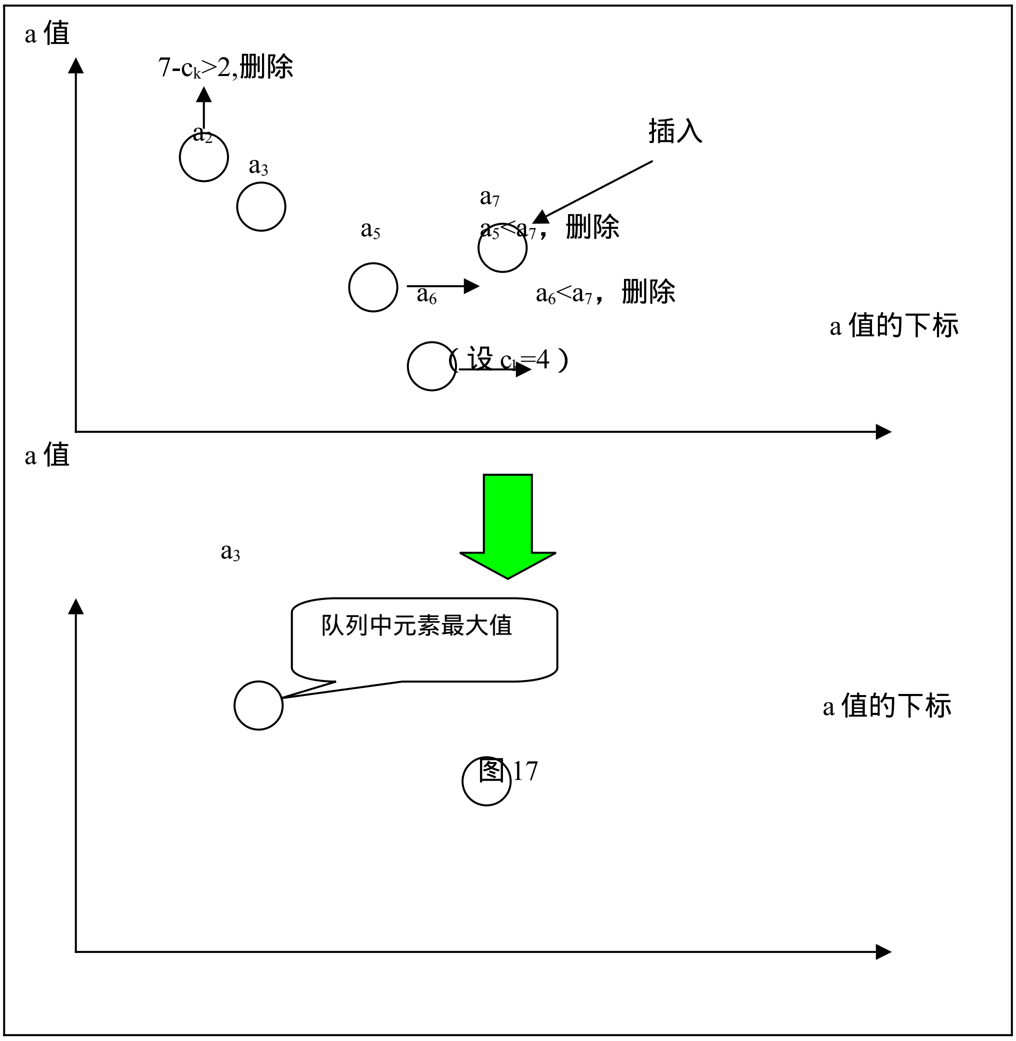


图 17

相应的伪代码（见下页）：

```
(以下仅考虑向东滑行的情况，向其他方向滑行的情况可以类似推出)
for (x=1; x<=n; x++)
 for (y=1; y<=m; y++)
  f[0][x][y]=-∞;
f[0][startx][starty]=0;
for (k=1; k<=slidenum; k++)
  for (x=1; x<=n; x++)
  {
    head=tail=0;
    for (y=1; y<=m; y++)
    {
      if (方格(x,y)上有障碍)
        {f[k][x][y]=-∞;  head=tail=0;  continue;}
   if (queue[head]<y-c[k])
        head++;
   while ((tail>head)&&(f[k-1][x][queue[tail].num]-queue[tail].num+1<=f[k-1][x][y]-y+1))
        tail--;
   queue[++tail]=y;
   f[k][x][y]=f[k-1][x][queue[head].num]+y-queue[head].num;
    }
  }
return max{f[slidenum][1][1], f[slidenum][1][2],……,f[slidenum][n][m]};
```

算法 3

回顾整个过程，我们首先找到了一个动态规划的解法。但是由于每次转移的时间复杂度太高，使得我们必须减少冗余运算。文中提到的线段树是一个不错的解决方案但是其编程复杂度与时间复杂度都还不能令人十分满意。而经常被我们所忽略的基本数据结构——队列却在这里发挥了巨大的作用。由前文可以看出，运用队列的解法巧妙而简洁，不但降低了时间复杂度，还大大减少了由于程序复杂而导致编程错误的可能性，很好地解决了这个问题。这再一次说明，在新趋势下的信息学竞赛中，基本数据结构的作用没有减小，而是变得更加重要了。在我们找到一种好算法而畏于其复杂的程序实现时，不妨转换思路，尝试用基本数据结构加以解决，说不定会有事半功倍的效果。