矩阵乘法快速幂

一、认识 k 阶常系数线性递推关系

我们熟悉的 Fibonacci 数列：F[n]=F[n-1]+F[n-2]，就是一个 2 阶常系数线性递推关系，由此我们得出 k 阶常系数线性递推关系的一般形式：

$$F_n = a_1 F_{n-1} + a_2 F_{n-2} + \cdots + a_k F_{n-k}$$

其中：a_1、a_2、$\cdots$、a_k，是常数，有 k 项，所以叫着 k 阶常系数线性递推关系；

$$F_n = \sum_{k=1}^{k} a_k \times F_{n-k}$$

二、对矩阵的认识

矩阵就是一个数字阵列，一个 n 行 r 列的矩阵可以表示为：

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1r} \\ a_{21} & a_{22} & \cdots & a_{2r} \\ \cdots & & & \\ a_{n1} & a_{n2} & \cdots & a_{nr} \end{bmatrix}$$

我们称上面的矩阵为$n \times r$矩阵。例如下面一个 2X3 矩阵；

$$\begin{bmatrix} 1 & 4 & 3 \\ 2 & 6 & 6 \end{bmatrix}$$

如果一个行数和列数相等的矩阵，我们叫作方阵。例如下面 3X3 方阵：

$$\begin{bmatrix} 1 & 4 & 3 \\ 2 & 6 & 7 \\ 9 & 7 & 6 \end{bmatrix}$$

其实，矩阵对我们来说，并不陌生，因为它类似于我们程序中的二维数组。

三、矩阵的运算

1、加法，减法

$$\begin{bmatrix}5 & 4 & 3\\ 2 & 1 & 3\end{bmatrix}+\begin{bmatrix}2 & 4 & 3\\ 2 & 5 & 6\end{bmatrix}=\begin{bmatrix}7 & 8 & 6\\ 4 & 6 & 9\end{bmatrix}$$

$$\begin{bmatrix}5 & 4 & 3\\ 2 & 6 & 6\end{bmatrix}-\begin{bmatrix}2 & 4 & 3\\ 2 & 5 & 6\end{bmatrix}=\begin{bmatrix}3 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}$$

2、乘法：

设 A，B 是两个矩阵，令 $C=A\times B$；那么：

（1）A 的行数必须和 B 的列数相等；设 A 是 $n\times r$ 的矩阵，B 是 $r\times m$ 的矩阵；

（2）A 和 B 的乘积 C 是一个 $n\times m$ 的矩阵；

（3）

$$\begin{aligned}c_{i,j}&=a_{i,1}\times b_{1,j}+a_{i,2}\times b_{2,j}+a_{i,3}\times b_{3,j}+\cdots+a_{i,r}\times b_{r,j}\\&=\sum_{k=1}^{r}a_{i,k}\times b_{k,j}\end{aligned}$$

例如：已知：$A=\begin{bmatrix}1 & 4\\ 2 & 5\\ 3 & 6\end{bmatrix}$，$B=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}$，求 $C=A\times B$。

$$A\times B=\begin{bmatrix}1*1+4*4 & 1*2+4*5 & 1*3+4*6\\ 2*1+5*4 & 2*2+5*5 & 2*3+5*6\\ 3*1+6*4 & 3*2+6*5 & 3*3+6*6\end{bmatrix}=\begin{bmatrix}17 & 22 & 27\\ 22 & 29 & 36\\ 27 & 36 & 45\end{bmatrix}$$

由矩阵乘法的运算法则，我们得出下列结论：**矩阵乘法满足结合律，不满足交换律。**

存在一个单位矩阵 I，使得 $I\times A=A\times I=A$，其中 I,A 都是 $n\times n$ 的方阵。

$$\text{单位矩阵 } I = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & \cdots \\ 0 & 0 & \ddots & 0 & \cdots \\ 0 & 0 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & 0 & 1 \end{bmatrix}$$

在 C++中常用结构体来表示一个矩阵并重载这个结构体的乘法为矩阵乘法：

```
struct Matrix {
  int a[N][N];
  inline Matrix operator * (const Matrix &rhs) {
    Matrix c;
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        int sum = 0;
        for (int k = 0; k < n; ++k)
          sum = (a[i][k] * rhs.a[k][j] + sum) % mod;
        c.a[i][j] = sum;
      }
    }
    return c;
  }
};
```

四、方阵乘幂

方阵乘幂是指，A 是一个**方阵**，将 A 连乘 n 次，即：$C = A^n$ 。

若不是方阵则无法进行乘幂运算。

由于矩阵乘法满足律，因此我们可以用二分快速幂的思想来求方阵乘幂。

在 C++中我们也可以将异或操作符（^）重载为矩阵乘幂的运算，但要注意由于（^）的优先级比较低，所以使用的时候注意加上括号。

```
struct Matrix {
  int a[N][N];
  inline void I() {  // 初始化为单位矩阵
    memset(a, 0, sizeof(a));
    for (int i = 0; i < N; ++i) a[i][i] = 1;
  }
```

```
  inline Matrix operator * (const Matrix &rhs) {
    Matrix c;
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        int sum = 0;
        for (int k = 0; k < n; ++k)
          sum = (a[i][k] * rhs.a[k][j] + sum) % mod;
        c.a[i][j] = sum;
      }
    }
    return c;
  }
  inline Matrix operator ^ (int Exp) {
    Matrix res, tmp = *this;
    res.I();
    for (; Exp > 0; Exp >>= 1, tmp = tmp * tmp)
      if (Exp & 1) res = res * tmp;
    return res;
  }
};
```

五、线性递推方程的优化算法

1.Fibonacci 数列

求 Fibonacci 数列的第 n 项，其中 1<=n<=10^9，第一项是 1，第 2 项是 1。输出 Fibonacci 数列的第 n 项 Mod 2017 的值。

问题分析

考虑 1×2 的矩阵[f[n-2],f[n-1]]。根据 fibonacci 数列的递推关系，我们希望通过乘以一个 2×2 的矩阵，得到矩阵[f[n-1],f[n]] = [f[n-1]，f[n-2]+f[n-1]]，很容易构造出这个 2×2 矩阵 A，即：

$$[F[n-1] \quad F[n]]=[F[n-2] \quad F[n-1]]\times\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}$$

所以，有[f[1],f[2]]×A＝[f[2],f[3]]又因为矩阵乘法满足结合律，故有：

$$
\begin{aligned}
&[F[n-1] \quad F[n]] \\
&=[F[n-2] \quad F[n-1]]\times\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix} \\
&=[F[1] \quad F[2]]\times A^{n-1} \\
&=[F[1] \quad F[2]]\times\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{n-1} \\
&=[1 \quad 1]\times\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{n-1}
\end{aligned}
$$

矩阵的第一个元素即为所求。

2. Fibonacci 数列 2

数列 f[n]=f[n-1]+f[n-2]+1,f[1]=f[2]=1，输出该数列的第 n 项 Mod 2017 的值。

问题分析

仿照前例，考虑 1×3 的矩阵 [f[n-2], f[n-1], 1]，希望求得某 3×3 的矩阵 A，使得此 1×3 的矩阵乘以 A 得到矩阵：[f[n-1], f[n], 1] = [f[n-1], f[n-1]+f[n-2]+1, 1].

容易构造出这个 3×3 的矩阵 A，即 $A=\begin{bmatrix}0 & 1 & 0\\ 1 & 1 & 0\\ 0 & 1 & 1\end{bmatrix}$

$$
\begin{aligned}
&[f[n-1], f[n], 1] \\
&=[f[n-1], f[n-1]+f[n-2]+1, 1] \\
&=[f[n-2], f[n-1], 1]\times A \\
&=[f[n-2], f[n-1], 1]\times\begin{bmatrix}0 & 1 & 0\\ 1 & 1 & 0\\ 0 & 1 & 1\end{bmatrix} \\
&=[f[1], f[2], 1]\times\begin{bmatrix}0 & 1 & 0\\ 1 & 1 & 0\\ 0 & 1 & 1\end{bmatrix}^{n-2}
\end{aligned}
$$

3.Fibonacci 数列 3

数列 f[n]=f[n-1]+f[n-2]+n+1, f[1]=f[2]=1，输出该数列的第 n 项 Mod 2017 的值。

问题分析

仿照前例，考虑 1×4 的矩阵[f[n-2], f[n-1], n, 1]，希望求得某 4×4 的矩阵 A，使得此 1

×4 的矩阵乘以 A 得到矩阵：[f[n-1], f[n], n+1, 1] = [f[n-1], f[n-1]+f[n-2]+n+1, n+1, 1]

容易构造出这个 4×4 的矩阵 A，即： $A=\begin{bmatrix}0&1&0&0\\1&1&0&0\\0&1&1&0\\0&1&1&1\end{bmatrix}$

4.Fibonacci 数列 4

数列 f[n]=f[n-1]+f[n-2], f[1]=f[2]=1，输出该数列的前 n 项和 s[n] Mod 2017 的值。

问题分析

仿照之前的思路，考虑 1×3 的矩阵[f[n-2], f[n-1], s[n-2]]，我们希望通过乘以一个 3×3 的矩阵 A，得到 1×3 的矩阵：[f[n-1], f[n], s[n-1]] = [f[n-1], f[n-1]+f[n-2], s[n-2]+f[n-1]]

容易得到这个 3×3 的矩阵是： $A=\begin{bmatrix}0&1&0\\1&1&1\\0&0&1\end{bmatrix}$

5.Fibonacci 数列 5

数列 f[n]=f[n-1]+f[n-2]+n+1,f[1]=f[2]=1，输出该数列的前 n 项和 s[n] Mod 2017 的值。

问题分析

结合前面，考虑 1×5 的矩阵 [f[n-2], f[n-1], s[n-2], n, 1]，我们需要找到一个 5×5 的矩阵 A，使得它乘以 A 得到如下 1×5 的矩阵：

[f[n-1], f[n], s[n-1], n+1, 1] = [f[n-1], f[n-1]+f[n-2]+n+1, s[n-2]+f[n-1], n+1, 1]

容易构造出 A 为： $A=\begin{bmatrix}0&1&0&0&0\\1&1&1&0&0\\0&0&1&0&0\\0&1&0&1&0\\0&1&0&1&1\end{bmatrix}$

总结

一般地，如果有 f[n]= p*f[n-1] + q*f[n-2] + r*n + s,可以构造矩阵 A 为

$$A=\begin{bmatrix}0 & q & 0 & 0 & 0\\ 1 & p & 1 & 0 & 0\\ 0 & 0 & 1 & 0 & 0\\ 0 & r & 0 & 1 & 0\\ 0 & s & 0 & 1 & 1\end{bmatrix}$$

更一般地，对于 m 项递推式，如果记递推式为

$$a_{n+m}=\sum_{i=0}^{m-1}b_i a_{n+i}$$

则可以把递推式写成如下矩阵形式：

$$\begin{bmatrix}a_{n+m}\\ a_{n+m-1}\\ \vdots\\ a_{n+1}\end{bmatrix}=\begin{bmatrix}b_{m-1} & \cdots & b_1 & b_0\\ 1 & \cdots & 0 & 0\\ \vdots & \ddots & \vdots & \vdots\\ 0 & \cdots & 1 & 0\end{bmatrix}\times\begin{bmatrix}a_{n+m-1}\\ a_{n+m}\\ \vdots\\ a_n\end{bmatrix}$$

通过计算这个矩阵的 n 次幂就可以在 $O(m^3\log n)$ 时间内计算出第 n 项的值，若式子含有常数项以及简单的与 n 有关的项，则可以通过在矩阵中加维来计算。

练习题

POJ 3070 ; BZOJ 1009 ; BZOJ 3329 ; BZOJ 1297

1D/1D 动态规划优化初步

一、简介

对于规模为 n 的问题，若一个转移方程其状态数为 O(n^x)，每个状态转移数为 O(n^y)，则称这个问题是 xD/yD 的。

所谓 1D/1D 动态规划，指的是状态数为 O(n)，每一个状态决策量为 O(n)的动态规划方程。直接求解的时间复杂度为 $O(n^2)$，但是，绝大多数这样的方程通过合理的组织与优化都是可以优化到 O(nlogn)乃至 O(n)的时间复杂度的。

下文不会进行过多的证明与推导，主要想说明经典模型的建立、转化与求解方法。

下文中使用两种方式表示一个函数：f(x)与 f[x]，用方括号表示的函数值可以在规划之前全部算出（常量），而用圆括号表示的函数值必须在规划过程中计算得到（变量）。无论是什么函数值一经确定，在以后的计算中就不会更改。

二、经典模型一：$f(x)=\min_{i=1}^{x-1}\{f(i)+w[i,x]\}$

相信这个方程大家一定是不陌生的。另外，肯定也知道一个关于决策单调性的性质：

假如用 k(x)表示状态 x 取到最优值时的决策，则决策单调性表述为：

$\forall i\le j, k(i)\le k(j)$，**当且仅当：**$\forall i\le j, w[i,j]+w[i+1,j+1]\le w[i+1,j]+w[i,j+1]$

对于这个性质的证明读者可以在任意一篇讲述四边形不等式的文章中找到，所以这里不再重复。而且，从实战的角度来看，我们甚至都不需要验证 w 函数的这个性质，最经济也是最可靠的方法是写一个朴素算法打出决策表来观察（反正你总还是要对拍）。当然，有的时候题目要求你做一点准备工作，去掉一些明显不可能的决策，然后再应用决策单调性。这时上述性质也许会有点用处。

正如前文中所述，我们关注的重点是怎样实现决策单调性。有了决策单调性，怎样高效地实现它呢？很容易想到在枚举决策的时候，不需要从 1 开始，只要从 k(x-1)开始就可以了，但这只能降低常数，不可能起到实质性的优化。

另一种想法是从 k(x-1)开始枚举决策更新 f(x)，一旦发现决策 u 不如决策 u+1 来得好，就停止决策过程，选取决策 u 作为 f(x)的最终决策。这样时间是很大提高了，但可惜是不正确的。决策单调性并没有保证 f(j)+w[j,x]有什么好的性质，所以这样做肯定是不对的。

刚才我们总是沿着“f(x)的最优决策是什么”这个思路进行思考，下面我们换一个角度，

思考对于一个已经计算出来的状态 f(j)，“f(j)能够更新的状态有哪些”。这样，每一步过程中某些状态的决策可能不是最优的，但是当算法结束的时候所有状态对应的决策一定是最优的。

一开始，只有 f(1)的函数值被计算出来，于是所有状态的当前最优决策都是 1。

1111111111111111111111111111111111111111111111111111111111111

现在，显然 f(2)的值已经确定了：它的最有决策只能是 1。我们用决策 2 来更新这个决策表。由于决策单调性，我们知道新的决策表只能有这样的形式：

11111111111111111111111111112222222222222222222222222222

这意味着我们可以使用二分法来查找“转折点”，因为如果在一个点 x 上，如果决策 2 更好，则所有比 x 大的状态都是决策 2 更好；如果 x 上决策 1 更好，则所有比 x 小的状态都是决策 1 更好。

现在决策 1 和决策 2 都已经更新完毕，则 f(3)业已确定，现在用决策 3 来更新所有状态。根据决策单调性，现在的决策表只能有以下 2 种类型：

1111111111111111111111111111112222222222222222233333333333

1111111111111111111111113333333333333333333333333333333333

而这样的决策表示绝对不会出现的：

1111111111113333333333333333332222222222222222222222222222。

那么，我们的更新算法就是：

1.考察决策 2 的区间[b,e]的 b 点上是否决策 3 更优，如果是，则全部抛弃决策 2，将此区间划归决策 3；如果否，则在决策 2 的区间[b,e]中二分查找转折点。

2.如果第 1 问的回答是“是”，则用同样的方法考察决策 1。

推演到这一步，相信决策单调性的实现算法已经明了了：使用一个**栈**来维护数据，占中的每一个元素保存一个决策的起始位置与终了位置，显然这些位置相互连接且依次递增。当插入一个新的决策时，从后到前扫描栈，对于每一个老决策来说，做这样两件事：

1.如果在老决策的起点处还是新决策更好，则退栈，全额抛弃老决策，将其区间合并至新决策中，继续扫描下一个决策。

2.如果在老决策的起点处是老决策好，则转折点必然在这个老决策的区间中；二分查找之，然后新决策进栈，结束。

由于一个决策出栈之后再也不会进入，所以均摊时间为 O(1)，但是由于二分查找的存在，

所以整个算法的时间复杂度为 O(nlogn)。

下面我们来看两个例题。

例题 1：玩具装箱。

题目来源：湖南省选 2008。（BZOJ 1010）

题目大意：有 n 个玩具需要装箱，每个玩具的长度为 c[i]，规定在装箱的时候，**必须严格按照给出的顺序进行**，并且同一个箱子中任意两个玩具之间必须且只能间隔一个单位长度，换句话说，如果要在一个箱子中装编号为 i~j 的玩具，则箱子的长度必须且只能是：

$$l = j - i + \sum_{k=i}^{j} c[k]$$

规定每一个长度为 *l* 的箱子的费用是 $P = (l - L)^2$，其中 L 是给定的一个常数。现在要求你使用最少的代价将所有玩具装箱，箱子的个数无关紧要。

分析：本题可以很轻松地列出一个 1D1D 的动态规划方程：

$$f(x) = \min_{i=1}^{x-1}\{f(i) + w[i+1, x]\}$$

其中 $w[i, j] = (j - i + \sum_{k=i}^{j} c[k] - L)^2$.

不难验证这个方程式满足决策单调性的，于是我们可以直接套用上文中的方法进行优化，时间复杂度为 O(nlogn).

例题 2：土地购买

题目来源：USACO Monthly, March, 2008, Gold （BZOJ 1597）

题目大意：有 N 块土地需要购买，每块土地都是长方形的，有特定的长与宽。你可以一次性购买一组土地，价格是这组土地中长的最大值乘以宽的最大值。比方说一块 5*3 的土地和一块 9*2 的土地在一起购买的价格就是 9*3。显然，怎样分组购买土地是一门学问，你的任务就是设计一种方案用最少的钱买下所有的土地。

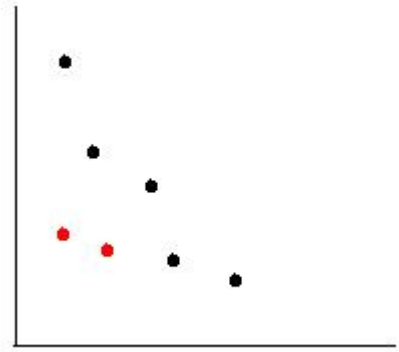
分析：将所有土地按照长度降序排列，依次检索，则当前土地的长度必然在上一块土地之内，我们只需要考虑宽度就可以了。而在宽度的问题上，当前土地的行为只能是这样：和

前面若干块土地绑定；同时这些绑定的土地和他们前后的土地分离。这样很容易得出状态转移方程：

$$f(n)=\min_{k=0}^{n-1}\{(\max_{i=k+1}^{n} w[i])*l[k+1]+f(k)\}$$

这个方程还不能满足决策单调性，下面我们试图再做一下简化。

如果将每一个土地的尺寸看成是一个二维坐标的话，（如下图）



其中不难看出，红色点完全可以忽略，这些点(x,y)必然满足一个性质：存在点(x', y')同时满足x' >= x且y' >= y，这样它就能被一个组完全覆盖。这些被忽略的点可以通过一次线形的扫描得出。

下面，我们着重来看一下不能被忽略的这些点，它们的排布方式必然是单调减。因此状态转移方程可以写成这个样子：

$$f(n)=\min_{k=0}^{n-1}\{x[n]*y[k+1]+f(k)\}$$

这个转移方程就是标准的决策单调性了，读者可以通过w函数的性质直接证明它。然后就用上文中的方法在O(nlogn)时间内求解。

代码：

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;

typedef long long ll;
const int N = 5e4 + 3;
```

```
int n, m, maxv, ql, qr, que[N][3];
ll f[N];
struct node {
    int l, w;
    node(): l(0), w(0){}
    node(const int &_l, const int &_w): l(_l), w(_w){}
    inline bool operator < (const node &b) const {
        return l > b.l || l == b.l && w > b.w;
    }
} a[N], b[N];

inline ll calc(const int &x, const int &y) {
    return f[x] + (ll)b[x + 1].l * b[y].w;
}

char ch;
inline int read() {
    while (ch = getchar(), ch < '0' || ch > '9');
    int res = ch - 48;
    while (ch = getchar(), ch >= '0' && ch <= '9') res = res * 10 + ch - 48;

    return res;
}

int main() {
    n = read(); m = 0;
    for (int i = 1; i <= n; ++i) a[i].l = read(), a[i].w = read();
    sort(a + 1, a + n + 1);
    for (int i = 1; i <= n; ++i)
        if (a[i].w > maxv) {
            maxv = a[i].w;
            b[++m] = a[i];
        }
    ql = qr = 1; que[1][0] = 0;
    que[1][1] = 0; que[1][2] = m;
    for (int i = 1; i <= m; ++i) {
        if (que[ql][1] < que[ql][2]) ++que[ql][1];
        else ++ql;
        f[i] = calc(que[ql][0], i);
        if (calc(i, m) >= calc(que[qr][0], m)) continue;
        while (ql <= qr && calc(que[qr][0], que[qr][1]) >= calc(i, que[qr][1]
)) --qr;
        if (ql > qr) que[++qr][0] = i, que[qr][1] = i, que[qr][2] = m;
        else {
```

```
            int p = que[qr][0], l = que[qr][1], r = que[qr][2], mid;
            while (l <= r) {
                mid = l + r >> 1;
                if (calc(p, mid) < calc(i, mid)) l = mid + 1;
                else    r = mid - 1;
            }
            que[qr][2] = r; ++qr;
            que[qr][0] = i, que[qr][1] = l, que[qr][2] = m;
        }
    }
    cout << f[m] << endl;
    return 0;
}
```

三、另一个模型

以上两个例子都是决策单调性的直接应用。其中第二个例子稍微复杂一些，如果不忽略那些“肯定无用”的决策，不对数据进行有序化，则方程是不满足决策单调性的。这也就提醒我们在做这一类题目的时候不能钻牛角尖死做，还得灵活一点。

另外，决策单调性提供的只是 O(nlogn)的算法，事实上上面两个例题的最佳算法都是 O(n)的，在后文中我们将详细介绍另外一种经典模型，并且试图将这两个规划方程通过数学变换转向另一个模型。

下面我们来看一类特殊的 w 函数：$\forall i \le j < k, w[i,j]+w[j,k]=w[i,k]$，显然，这一类函数都是满足决策单调性的。但是不同的是，由于这一类函数的特殊性，他们可以用一种更加简洁也更加有借鉴意义的方法解决。

由于 w 函数满足 $\forall i \le j < k, w[i,j]+w[j,k]=w[i,k]$，我们总是可以找到一个特定的一元函数 w'[x]，使得 $\forall i \le j, w[i,j]=w'[j]-w'[x]$，这样，假设状态 f(x)的某一个决策是 k，有：

$$\begin{aligned} f(x) &= f(k)+w[k,x]=f(k)+w'[x]-w'[k] \\ &= g(k)+w'[x]-w'[1], \end{aligned}$$

其中 $g(k)=f(k)-w[1,k]$。

这样我们发现：一旦 f(k)被确定，相应地 g(k)也被确定，更加关键的是，**无论 k 值如何，w'[x]-w'[1]总是一个常数。**换句话说，我们可以把方程写成下述形式：

$$f(x)=\min_{k=1}^{x-1}\{g(k)\}+w[1,x]$$

不难发现这个方程是无聊的，因为$\min_{k=1}^{x}\{g(k)\}$我们可以用一个变量“打擂台”直接存储；但是，如果在 k 的下界上加上一个限制，那这个方程就不是很无聊了。于是，我们就得到了另一个经典模型。

经典模型二：$f(x)=\min_{k=b[x]}^{x-1}\{g(k)\}+w[x]$，其中，b[x]随 x 不降。

这个方程怎样求解呢？我们注意到这样一个性质：如果存在两个数 j, k，使得 j <= k，而且 g(k) <= g(j)，则决策 j 是毫无用处的。因为根据 b[x]单调的特性，如果 j 可以作为合法决策，那么 k 一定可以作为合法决策，又因为 k 比 j 要优，（注意：在这个经典模型中，“优”是绝对的，是与当前正在计算的状态无关的），所以说，**如果把待决策表中的决策按照 k 排序的话，则 g(k)必然是不降的。**

这样，就引导我们使用一个**单调队列**来维护决策表。对于每一个状态 f(x)来说，计算过程分为以下几步：

1.队首元素出队，直到队首元素在给定的范围中。

2.此时，队首元素就是状态 f(x)的最优决策。

3.计算 g(x)，并将其插入到单调队列的尾部，同时维持队列的单调性（不断地出队，直到队列单调为止）。

4.重复上述步骤直到所有的函数值均被计算出来。

不难看出这样的算法均摊时间复杂度是 O(1)的。

例题 3：The Sound of Silence

题目来源：Baltic Olympiad in Informatics 2007

题目大意：给出一个 N 项的数列，如果对于一个连续的长度为 M 的片段来说，片段内所有数中最大值与最小值的差不超过一个给定的常数 C，则我们称这样的片段是一个合法的片段。编程求出所有的合法片段的起始位置。

分析：本题不难看出可以分解为两个子问题：求所有片段的最大值以及求所有片段的最小值。而这两个任务实际上是一样的，所以我们只需要求取所有的连续 M 个数的片段中的最小值。

这个任务有很多很多种对数级算法，其中用堆或者用静态最优二叉树都可以做到 O(nlogm)，但是这题的 O(n)算法还是不那么好想的。

事实上，如果用 g[x]表示数列中第 x 个数的值，用 f(x)表示以 x 作为**结尾**的有 M 个数的连续片段的话，显然有：

$$f(x)=\min_{i=x-m+1}^{x}\{g[i]\}$$

直接吻合经典模型二。套用算法，就可以在 O(n)的时间内解决问题。

（当然，本题还有一种别致的“窗口”算法，也漂亮地在 O(n)的时间内解决了问题，详细可以看官方的解题报告。这里引入本题的主要目的是在于佐证上文中讨论到的经典模型二）

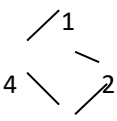
例题 4：Islands

题目来源：IOI2008

题目大意：给出一个具有 N 个顶点的无向加权图，同时这个图中有且恰有 N 条边。现在，对于这个图中的每一个连通分量，求出其最长路径（权值和最大，一个节点在路径上最多只能出现一次）。

分析：当然，这个问题更多的是一个图论题。但是在最后关键问题的处理上还是可以看到经典模型二的影子。首先，用 BFS 找出所有连通分量。然后，对于一个连通分量来说，由于点数与边数相同，因此必然构成**基环+外向树**的结构。我们可以找出基环并确定所有外向树的结构。一条最长路径有两种可能：完整地位于某一棵外向树中；或者位于两棵外向树中，其间通过基环的一段连接。第一种可能可以通过树形 DP 解决，问题就在于第二种可能怎样处理。如果枚举两棵外向树，那就是 $O(n^2)$，就不可以接受了。

我们考虑破环为链，然后将链整体左移，制作一个副本。比方说，如果原来的环是：



，以 1 为首破环，得：1--2--3--4，然后制作副本，得 2--3--4--1--2--3--4，制作副本

的主要目的是使得对于每一个点的方程都有统一的形式，使得环上所有片段都可以对应链上的一个片段。这种情况下，用 g[x]表示 x 点上外向树上的最长下降路的长度，f(x)表示以该点为终点的总最长路径的长度，则有：

$$f(x)=\max_{i=x-n+1}^{x-1}\{g[i]+g[x]+dis[i,x]\}$$

其中 w 函数即 $w[i,j]=dis[i,j]$ 显然满足 $\forall i\le j<k, w[i,j]+w[j,k]=w[i,k]$，通过变换之后就可以变成经典模型二。这样，就在总 O(n)的时间内解决了本题。

如果还嫌以上两个问题不够典型，下面举一个典型到所有 OIer 都耳熟能详的题目。

例题 5：有限背包问题。

题目来源：经典问题。

题目大意：有 N 件物品，每一件物品的价值为 p[k]，重量为 w[k]，最多只能选取 m[k]次；现在给出背包的最大承重量 C，要求在满足重量要求的条件下使得背包中的物品价值总和最大。

分析：如果 m[k] = 1 或者 m[k] = +∞，就都很好做。但现在 m[k]是一个有界值，就比较麻烦了。

我们还是按照背包问题的常见思路，一次枚举每一个物品。设当前枚举的物品编号为 k，用 f(x)表示：为了到达价值 x，背包的重量至少应该是多少；则我们有：

$$f(x)=\min_{i=1}^{m[k]}\{f(x-i*p[k])+i*w[k]\}$$

这个方程很麻烦，因为某一个状态的决策不是连续的，而是间断性的。怎样把决策区间变成一段连续区间呢？很容易想到**等价类**的思想；如果按照模 p[k]对所有的 f(x)划分等价类，那么在同一个等价类中，决策区间就是连续的了，我们不妨把新函数设为 h(x)，则方程变为：

$$h(x)=\min_{i=x-m[k]}^{x-1}\{h(i)+w[k]*(x-i)\}$$

其中，w 函数即 $w[i,j]=w'[k]*(j-i)$ 显然满足 $\forall i\le j<k, w[i,j]+w[j,k]=w[i,k]$，（注意 w'[k]是一个与 i 和 j 无关的常量）经过适当的变化后可以转化为经典模型二。于是有限背包问题可以在 O(NP)的时间内解决，其中 P 是背包可能取到的最大价值。（其实换成重

量也一样），这也就是“背包十讲”中所说的那个单调队列法。

我们注意到，如果 m[k]=1 的话，那么每一个 f(x)的决策量都是 O(1)，这没什么问题；但如果 m[k] = +∞，有意味着什么呢？仔细观察可以发现，这实际上就拿掉了方程中的循环变量的下界，对应的是 $f(x)=\min_{k=1}^{x-1}\{g(k)\}+w[1,x]$ 这样的一个方程，这显然是很简单的，适用单变量打擂台就可以解决了（尽管我们通常并不这样做）。所以说，借助经典模型二，我们在一个更高的高度上统一了 0-1、有限、无限三大背包问题。

下面我们再次来看一下例题 2《土地购买》中的那个方程：

$$f(n)=\min_{k=0}^{n-1}\{f(k)+x[n]*y[k+1]\}$$

我们来仔细地观察这个方程：f(k)是变量，y[k+1]是常量，但不论怎么说，这两个量在以后的计算中都不会变化。而 x[n]是一个比例系数，它与 k 无关，只随着 x 的变化而变化。如果我们建立平面直角坐标系，以 f(k)作为横轴，y[k+1]作为纵轴，则每一个状态 f(k)都可以看作是该坐标系中的一个点。在求解状态 f(n)的过程中，我要求最小化：

$$\min P=x+ky$$

其中 x, y 是我建立的直角坐标系中某一个点的坐标（表示一个决策），k 就是方程中的 x[n]，是只与 n 有关，而与决策无关的一个常量。

这个最小化问题是什么呢？其实就是一个平面上的线性规划。我们把式子改写成：

$$y=-\frac{1}{k}x+\frac{P}{k}$$

就演变成了这样的一个问题：在一个直线簇 $y=-\frac{1}{k}x+C$ 中，选取一条直线，使得这条直线过某个给出的数据点，同时 C 要最小。

既然问题变成了这么有意思的线性规划问题，就有必要进一步的研究，看看是不时有更好的解法，这就导致了我们的另一个经典模型：

四、经典模型三：$f(x)=\min_{i=1}^{x-1}\{a[x]*f(i)+b[x]*g(i)\}$。

注意：这个模型写的比较抽象，其实它的涵盖范围是很广的。首先，a[x], b[x]不一定要是常量，只要他们与决策无关，都可以接受；另外，f(i)和 g(i)不管是常量还是变量都没有关系，只要他们是一个有最优的 f(x)决定的二元组就可以了。

因此，为了方便描述，我们把这个模型写成下面这个形式：

$$f(n)=\min_{i=1}^{n-1}\{a[n]*x(i)+b[n]*y(i)\},$$

其中，x(i)，y(i)都是可以在常数时间内通过 f(i)唯一决定的二元组。

这个经典模型怎样转化求解呢？前文说过，这样的模型的求解与平面上的线性规划有关，我们以 x(i)为横轴，y(i)为纵轴建立平面直角坐标系，这样一个状态 f(i)所决定的二元组就可以用坐标系中的一个点表示。然后，我们的目标是：

$\min P=ax+by$，其中 $a=a[n],b=b[n]$，化成：$y=-\frac{a}{b}x+\frac{P}{b}$，假设 $b>0$（反之亦然），则我们的任务是使得这条直线的纵截距最小。可以想象有一组斜率相同的直线自负无穷向上平移，所碰到的第一个数据点就是最优决策。

这个时候，有一个重要的性质，那就是：**所有最优决策点都在平面点集的凸包上。**基于这个事实，我们可以开发出很多令人满意的算法。

这时，根据直线斜率与数据点分布的特征，可以划分为两种情况：

情况一：决策直线的斜率与二元组的横坐标同时满足单调性。（具体的单调性视最优化目标的性质而定）

这样的模型是比较好处理的，因为这个时候由于斜率变化是单调的，所以决策点必然在凸壳上单调移动。我们只需要维护一个单调队列和一个决策指针，每次决策时作这样几件事：

1.决策指针（也就是队首）后移，直至最佳决策点。

2.进行决策。

3.将进行决策之后的新状态的二元组加入队尾，同时作 Graham-Scan 式的更新操作维护凸壳。（注意此时当前指针所在二元组有可能被抛弃）

算法的时间复杂度为 O(n)

情况二：没有任何限制。

这时问题的解决就比较困难了。显然，决策点还是应该在凸壳上。我们不妨考虑一个单调减的凸壳，这个凸壳上点与点之间的连线必然满足这样的性质：斜率单调减。通过细致的观察我们可以发现，**对于一个给定的斜率 k 来说，对应的直线簇中具有最大纵截距的直线通过的决策点必然满足这样的性质：该点两侧的边的斜率 k1, k2 满足** $k1 \geq k \geq k2$ 。

这样，我们就可以通过二分查找来确定最佳的决策点。

但是，在插入数据点的过程中，我们遇到的麻烦可能更大。首先，肯定是二分查找确定横坐标的插入点，然后对两侧分别进行 Graham 维护凸性。但接下来的问题就严重了：在维护凸形的过程中我们肯定删掉了一些点，怎样重新得到一个完整的凸壳决策表呢？使用 move 是一个折中的办法，但是这与理论的时间复杂度分析根本无益。

完美的解决方法是应用平衡二叉树。我们以横坐标为关键字建立平衡二叉树，这样查找和插入的过程都可以在 O(logn)时间内完成。当我们做 Graham 维护时，首先将新数据点 Splay 到根节点，此时剩下的节点必然分居左子树和右子树。然后，我们以左子树为例，后序遍历依次查找节点，直至查找到一个满足凸形的节点。将这个节点 Splay 到根节点的左孩子，然后删掉这个节点的右孩子。这样的算法的时间复杂度是 O(nlogn)，但是实现起来非常复杂。

另一种方法是利用 CDQ 分治，将每个决策看做一次询问与插入，这样处理完前半部分的询问后，可以将它们看为平面上插入的点，对后半部分的询问进行更新，同时利用归并排序将插入点与询问点排成情况一那样，方便处理。实现起来非常简便。

例题 6：《玩具装箱》的线性算法。

例题 1 中《玩具装箱》的动态规划方程为：

$$f(x)=\min_{i=1}^{x-1}\{f(i)+(x-i-1+\sum_{k=i+1}^{x}c[k]-L)^2$$

下面，我们试图通过数学变换将其变成经典模型三。

为了简化计算，设 $sum[x]=\sum_{i=1}^{x}c[x]$，则：

$$f(x)=\min_{i=1}^{x-1}\{f(i)+(x-i-1-L+sum[x]-sum[i]\}^2\}$$

$$=\min_{i=1}^{x-1}\{f(i)+((sum[x]+x-L-1)-(sum[i]+i))^2\}$$

不妨设 $a[x]=sum[x]+x-l-1, b[i]=sum[i]+i$ 显然这两个量都是常量，则：

$$f(x)=\min_{i=1}^{x-1}\{f(i)+(a[x]-b[i])^2\}$$

$$=\min_{i=1}^{x-1}\{f(i)+a^2[x]+b^2[i]-2a[x]b[i]\}$$
$$=\min_{i=1}^{x-1}\{f(i)+b^2[i]-2a[x]b[i]\}+a^2[x]$$

，

然后问题就明朗了，设平面直角坐标系中 $x(i)=b[i], y(i)=f(i)+b^2[i]$，则问题变成：$\min P=2ax+y$，其对应的线性规划的目标直线为 $y=-2ax+P$。

回顾定义不难看出，a[x]随着 x 的增大而增大，x(i)也随着 i 的增大而增大。因此，问题中直线斜率单调减，数据点横坐标单调增，符合经典模型三种的情形 1。使用单调队列维护凸壳可以在 O(n)的时间内解决本题。

代码：

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;

typedef long long ll;
const int N = 5e4 + 2;
int n, l, ql, qr, que[N];
ll s[N], a[N], f[N], X[N], Y[N];

char ch;
inline int read() {
    while (ch = getchar(), ch < '0' || ch > '9');
    int res = ch - 48;
    while (ch = getchar(), ch >= '0' && ch <= '9')
        res = res * 10 + ch - 48;
    return res;
}

inline bool slope(const int &k, const int &j, const int &i) {
    return (X[j] - X[i]) * (Y[k] - Y[i]) - (Y[j] - Y[i]) * (X[k] - X[i]) >= 0;
}

```

```
inline ll calc(const int &j, const int &i) {
    return f[j] + s[j] * s[j] - a[i] * s[j] * 2;
}

int main() {
    n = read(); l = read() + 1;
    for (int i = 1; i <= n; ++i) {
        s[i] = s[i - 1] + read() + 1;
        a[i] = s[i] - l;
    }
    ql = qr = 1; que[1] = 0;
    for (int i = 1; i <= n; ++i) {
        while (ql < qr && calc(que[ql], i) >= calc(que[ql + 1], i)) ++ql;
        f[i] = calc(que[ql], i) + a[i] * a[i];
        X[i] = s[i]; Y[i] = f[i] + s[i] * s[i];
        while (ql < qr && slope(que[qr - 1], que[qr], i)) --qr;
        que[++qr] = i;
    }
    cout << f[n] << endl;
    return 0;
}
```

例题七：货币兑换（BZOJ 1492）

题目来源：NOI2007

题目大意：有 3 种货币体系：人民币，A 券，B 券，其中 A 券与 B 券的价格在每一天都是不同的。在某一天 D，你可以做 3 件事情：

1.如果你的手头上有 A 券或 B 券，你可以将它们都按照当天的价格换成人民币。

2.如果你的手头上有人民币，你可以将它们按照一个特定的比例 Rate 并以照当天的价格换成 A 券和 B 券（就是说你兑换的 A 券和 B 券的价值比是 Rate）

3.什么也不做。

一开始你有一些人民币，请你通过最佳的操作方式在最后一天结束的时候手头上握有最多的人民币。

分析：试题中的 Hint 已经告诉我们，**如果我们想买进人民币，就必须全额买进；如果我们想卖出人民币，就必须全额卖出。**由于不管是在哪一天，人民币总是越多越好；我们用 f(n)表示到了第 n 天最多可能持有的人民币数量，用 x(i)和 y(i)表示在第 i 天，用最多的钱

能够换成的 A 券和 B 券。（注意：由于 Rate 确定，兑换金额确定，所以 A 券和 B 券的数量是唯一确定的），我们有：

$$f(n)=\max_{i=1}^{n-1}\{a[n]*x(i)+b[n]*y(i)\}$$

其中 a[n]和 b[n]代表 A 券和 B 券在第 n 天的牌价。

这个方程式符合经典模型三中的情形二。所以，我们应该使用一个平衡树来维护凸形。时间复杂度是 O(nlogn)。CDQ 分治也可顺利通过。

CDQ 分治做法代码：

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;

const int N = 1e5 + 2;
int n, q[N], qry[N];
double s, ans, a[N], b[N], rate[N], f[N];
struct point {
    double x, y;
    inline bool operator < (const point &b) const {
        return x < b.x || x == b.x && y < b.y;
    }
} p[N], que[N];

inline bool slope(const point &k, const point &j, const point &i) {
    return (j.x - i.x) * (k.y - i.y) - (j.y - i.y) * (k.x - i.x) <= 0;
}

inline bool cmp(const int &i, const int &j) {
    return a[i] * b[j] < a[j] * b[i];
}

inline double calc(const point &j, const int &i) {
    return j.x * a[i] + j.y * b[i];
}

inline void CdqSolve(const int &l, const int &r) {
    if (l == r) {
        if (f[l - 1] > f[l]) f[l] = f[l - 1];
```

```
        if (f[l] > ans) ans = f[l];
        p[l].y = f[l] / (rate[l] * a[l] + b[l]);
        p[l].x = rate[l] * p[l].y;
        return ;
    }
    int mid = l + r >> 1, idx1 = l, idx2 = mid + 1, ql = 1, qr = 0;
    for (int i = l; i <= r; ++i)
        qry[i] <= mid ? q[idx1++] = qry[i] : q[idx2++] = qry[i];
    for (int i = l; i <= r; ++i) qry[i] = q[i];
    CdqSolve(l, mid);
    for (int i = l; i <= mid; ++i) {
        while (qr > 1 && slope(que[qr - 1], que[qr], p[i])) --qr;
        que[++qr] = p[i];
    }
    for (int i = mid + 1; i <= r; ++i) {
        int j = qry[i];
        while (ql < qr && calc(que[ql], j) <= calc(que[ql + 1], j)) ++ql;
        f[j] = max(f[j], calc(que[ql], j));
    }
    CdqSolve(mid + 1, r);
    if (l == 1 && r == n) return ;
    ql = l; idx1 = l; idx2 = mid + 1;
    while (ql <= r) {
        if (idx2 > r || idx1 <= mid && p[idx1] < p[idx2]) que[ql++] = p[idx1
++];
        else    que[ql++] = p[idx2++];
    }
    for (int i = l; i <= r; ++i) p[i] = que[i];
    return ;
}

int main() {
    scanf("%d%lf", &n, &s);
    for (int i = 1; i <= n; ++i) {
        scanf("%lf%lf%lf", &a[i], &b[i], &rate[i]);
        qry[i] = i;
    }
    f[0] = s;
    sort(qry + 1, qry + n + 1, cmp);
    CdqSolve(1, n);
    printf("%.3lf\n", ans);
    return 0;
}
```

五、总结

上文中，我们着重讨论了这样三类经典模型的建立与求解过程：

经典模型一： $f(x)=\min_{i=1}^{x-1}\{f(i)+w[i,x]\}$，**w[i, x]满足决策单调性。**

经典模型二： $f(x)=\min_{i=b[x]}^{x-1}\{g(i)\}+w[x]$，**其中 b[x]单调增。**

经典模型三： $f(n)=\min_{i=1}^{n-1}\{a[n]*x(i)+b[n]*y(i)\}$，**其中 x(i)，y(i)可以由 f(i) 在常数时间内唯一确定。**

这三类模型都可以在至少 O(nlogn)的时间内解决，从而起到了对 1D/1D 的方程的优化作用。另外，这三种模型并不是孤立存在的，而是可以互相转化的，文中的很多例题就兼具多种模型的特点。

练习题

BZOJ 1597 ; BZOJ 1096 ; BZOJ 1010 ; BZOJ 1911 ; BZOJ 1499 ; BZOJ 1492

其他常见优化技巧

一、滚动数组

在 DP 时经常会发现只有相邻阶段间状态才会有直接联系，在转移方程中的体现就是能更新 i 这一维的前驱状态总是 $i-1$.因此实际上我们每次只用到这维中的两个位置，且用完以后就不会再需要访问。

考虑将这维大小从 n 改为 2，在枚举的时候不断滚动使用这维的两个位置，以做到优化空间的目的，使用时只需要考虑当前 i 的奇偶性就可以知道该使用 0 还是 1.

但注意若统计答案时需要用到前面阶段的 dp 值或是转移时前面许多阶段都可以转移到当前阶段，则滚动数组不适用（也就是中间过程的 dp 值需要存储时）。

常用滚动是大小为 2，实际上大小为 3/4/5 等等都是可以的，看具体问题的转移方程来决定。

滚动数组主要的作用是节约空间。类似地还有用队列存储需要转移的状态。一般使用队列进行 DP 转移时，对应的 DP 状态中无用状态较多，枚举比较耗时，或是理论状态数太大无法全部存下，但实际状态数较少，可以将有用的状态编号，利用队列转移。

二、前缀后缀和

有时转移方程如：$f[i][j]=\sum_{k=1}^{b[i]} f[i-1][k]$

记录一个前缀和数组 $s[i][j]=\sum_{k=1}^{j} f[i][j]$

计算出 $f[i][j]$ 后便可直接更新 $s[i][j], s[i][j]=s[i][j-1]+f[i][j]$，利用 $s[i][j]$ 快速转移，不需要再用一重循环求和。特殊的转移方程可以直接改写递推式。类似的还有前缀最大值，有时需要同时用到前缀后缀和/最值。

前缀后缀合并的技巧在树形 DP 合并子树信息时常用。

三、数据结构优化

考虑如下 DP 方程：

$$f(x)=\min_{1\le i<x\text{且}h(i)<h(x)}\{g(i)+w(i)\}$$

考虑将 h 离散化，则能转移给 $f(x)$ 的 i 是连续一段区间。

用数据结构维护 h 离散后的序列，通常需要支持插入与查询。

时间复杂度 O(n log n).

其他一些类似的转移方程，需要根据具体问题选定用来维护 DP 方程的数据结构，常用的有线段树、平衡树等。