例题一：任务安排

题面描述：

> n个任务排成一个序列在一台机器上等待完成（顺序不得改变），这n个任务被分成若干批，每批包含相邻的若干任务。
> 从零时刻开始，这些任务被分批加工，第i个任务单独完成所需的时间为t_i。在每批任务开始前，机器需要启动时间s，而完成这批任务所需的时间是各个任务需要时间的总和（同一批任务将在同一时刻完成）。
> 每个任务的费用是它的完成时刻乘以一个费用系数f_i。请确定一个分组方案，使得总费用最小。

样例输入

```
5
1
1 3
3 2
4 3
2 3
1 4
```

样例输出

```
153
```

数据范围

$$对于100\%的数据，1 \le n \le 5000, 0 \le s \le 50, 1 \le t_i, f_i \le 100$$

- ## 初步分析

> dp题意味明显，发现时间/空间要求为N^2，计算一段区间的代价还受到前面分组数的影响，即如果设计一维状态，则会出现后效性。

- ## 设计状态

$$f[i][j]表示前i个任务，分为j组的最小费用$$
$$fs[i]表示f[i]的前缀和，ts[i]表示t[i]的前缀和$$

- ## 转移

$$f[i][j] = min\{f[k][j-1] + (ts[i] + j * s) * (fs[i] - fs[k])\}$$

- **代码**

```
#include<bits/stdc++.h>
using namespace std;
const int N = 5010;
const int S = 60;
const int inf = 0x3f3f3f3f;
int n,s,ts[N],fs[N],f[N][N],ans=inf;
int main()
{
    freopen("2365.in","r",stdin);
    scanf("%d%d",&n,&s);
    int T,F;
    for(int i = 1;i <= n;i++){
        scanf("%d%d",&T,&F);
        ts[i] = ts[i-1] + T;fs[i] = fs[i-1] + F;
    }
    memset(f,inf,sizeof(f));
    memset(f[0],0,sizeof(f[0]));
    for(int i = 1;i <= n;i++){
        for(int j = 1;j <= i;j++){
            for(int k = j-1;k <= i;k++){
                f[i][j] = min(f[i][j],f[k][j-1]+(ts[i]+j*s)*(fs[i]-fs[k]));
            }
        }
    }
    for(int i = 1;i <= n;i++)ans = min(ans,f[n][i]);
    printf("%d",ans);
    return 0;
}
```

- **复杂度分析**

时间复杂度达到了O(N^3),仍不满足限制
空间复杂度达到了O(N^2)，满足限制

- **状态优化**

可以发现，此题对于分组个数并没有限制，回顾一下以上的转移方程

$$f[i][j] = min\{f[k][j-1] + (ts[i] + j * s) * (fs[i] - fs[k])\}$$

即j的作用仅仅是计算要加多少个启动时间，于是想到“费用提前(计算)”的思想，正向处理后效性，把多出一个分组的后续所有代价直接加入f[i]。
具体的，对于f[i]从f[j]转移过来，多出一个分组的后续所有代价为(fs[n]-fs[j])*s，将其直接加入f[i]

- **转移**

$$f[i] = min(f[i], f[j] + ts[i] * (fs[i] - fs[j]) + s * (fs[n] - fs[j]))$$

- **代码**

```
#include<bits/stdc++.h>
using namespace std;
const int N = 5010;
const int S = 60;
const int inf = 0x3f3f3f3f;
int n,s,ts[N],fs[N],f[N];
int main()
{
    freopen("2365.in","r",stdin);
    scanf("%d%d",&n,&s);
    int T,F;
    for(int i = 1;i <= n;i++){
        scanf("%d%d",&T,&F);
        ts[i] = ts[i-1] + T;fs[i] = fs[i-1] + F;
    }
    memset(f,inf,sizeof(f));f[0] = 0;
    for(int i = 1;i <= n;i++){
        for(int j = 0;j < i;j++){
            f[i] = min(f[i],f[j]+ts[i]*(fs[i]-fs[j])+s*(fs[n]-fs[j]));
        }
    }
    printf("%d",f[n]);
    return 0;
}
```

- **复杂度分析**

时间复杂度缩至O(N^2)，满足限制
空间复杂度缩至O(N),进一步满足限制
此解法为满分解法

- **联想解法**

考虑可以正向处理后效性，实际上“反向处理后效性”也可以，思路与正向处理一致，此处的反向dp不再称为”提前计算“，这里直接给出代码

- **代码**

```
#include<bits/stdc++.h>
using namespace std;
const int N = 5100;
const int inf = 0x3f3f3f3f;
int n,s,t[N],f[N],fs[N],dp[N],ts[N];
int main()
{
    freopen("2365.in","r",stdin);
    scanf("%d%d",&n,&s);
    for(int i = 1;i <= n;i++)scanf("%d%d",&t[i],&f[i]);
    for(int i = n;i >= 1;i--){
        ts[i] = ts[i+1] + t[i];
        fs[i] = fs[i+1] + f[i];
```

```
    }
    memset(dp,inf,sizeof(dp));dp[n+1] = 0;
    for(int i = n;i >= 1;i--){
        for(int j = i + 1;j <= n + 1;j++){
            int cost = fs[i]*(s + ts[i] - ts[j]);
            dp[i] = min(dp[i],dp[j] + cost);
        }
    }
    printf("%d",dp[1]);
    return 0;
}
```

• 进一步优化

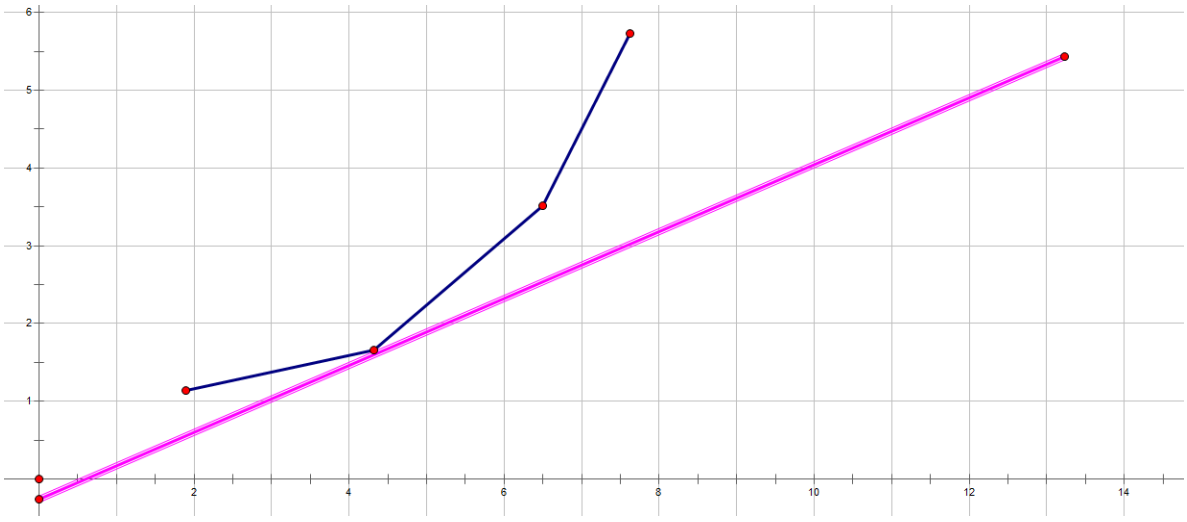
> 在此特别感谢洛谷用户63727*ButterflyDew*的题解&图片

$$f[i] = min(f[i], f[j] + ts[i] * (fs[i] - fs[j]) + s * (fs[n] - fs[j])) \cdots\cdots ①$$
$$f[j] = (s + ts[i]) * fs[j] + f[i] - ts[i] * fs[i] - s * fs[n] \cdots\cdots ②$$

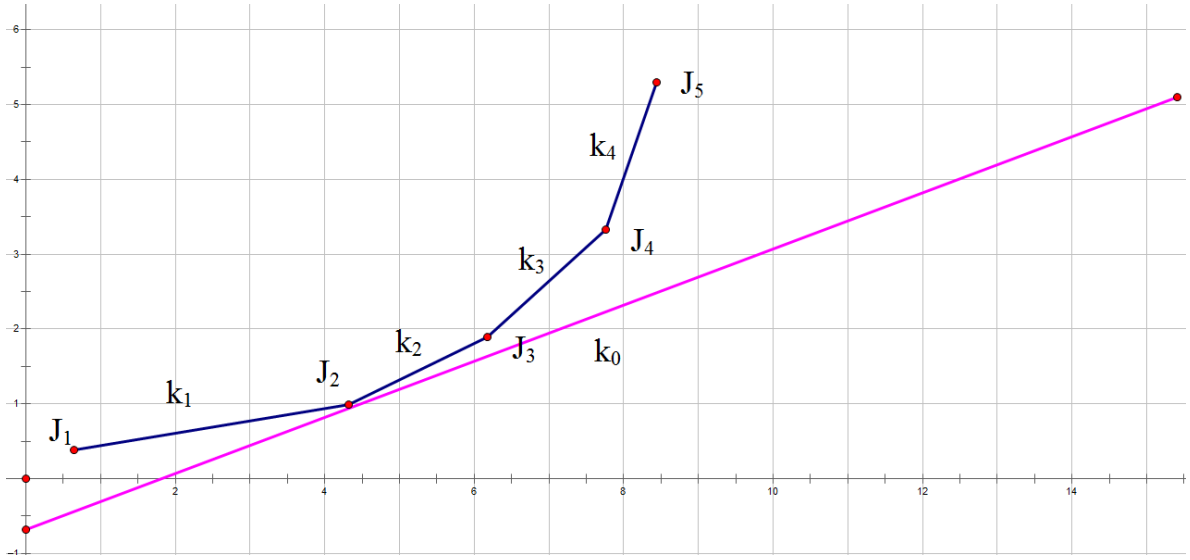
对于①式子，把min去掉，进行整理。
对于j的取值集合所映射的f[j]、fs[j],把f[j]想象为f(x),fs[j]想象为x，即为一条直线

$$这条直线的斜率 k = s + ts[i], 截距 b = f[i] - ts[i] * fs[i] - s * fs[n]$$

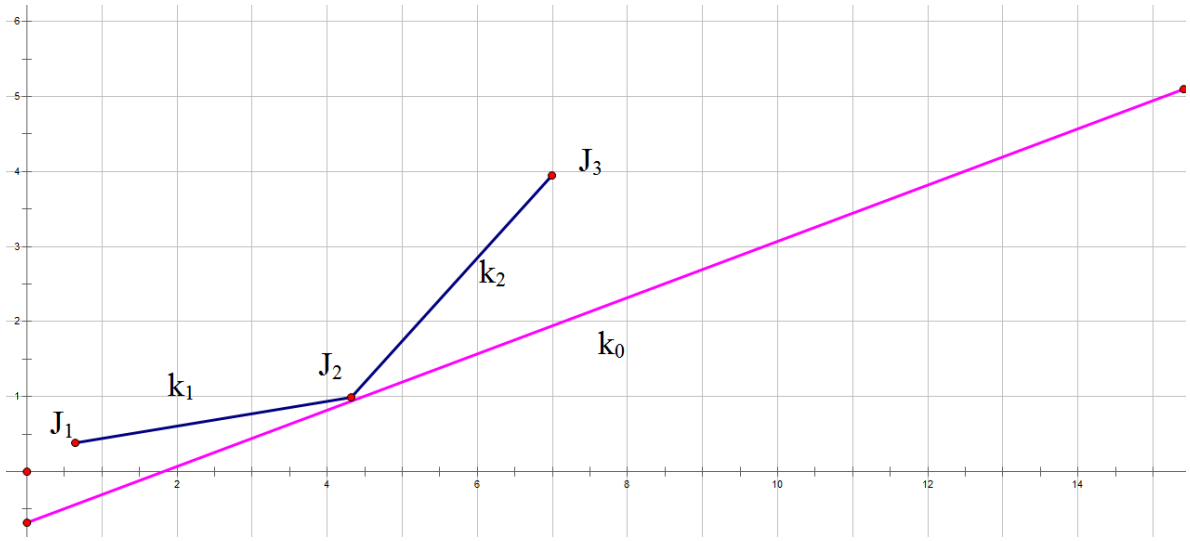
想要使得f[i]最小，等价于b最小。在二维平面上，拿一个已知斜率的直线向上滑动，第一次遇到取值集合内的点(fs[j],f[j])，就取到了截距b的最小值。

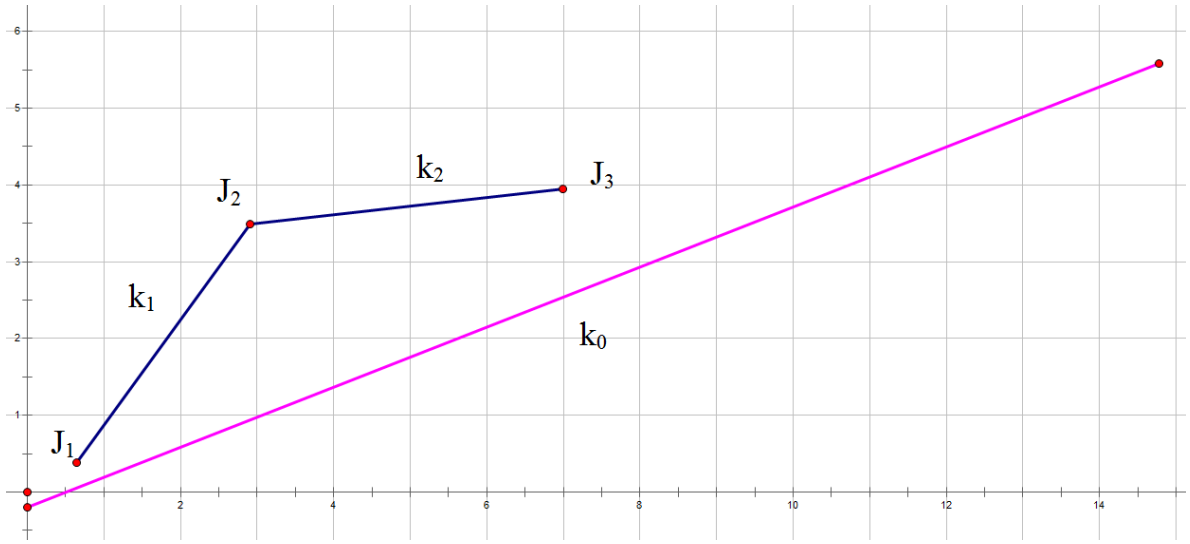


考虑什么时候取到最小值：当直线斜率k0满足ki<k0<ki+1时取到最小值



考虑什么时候可能取到最小值：图一（下凸）可以，图二（上凸）不可





于是使用单调队列维护点集(fs[j],f[j]),使得相邻点的斜率k递增(由于fs[j]递增，我们把fs[j]当做x（第一维）)
当已知一个直线的斜率时，二分求得第一次遇到的点即可得到答案
优化计算答案：
由于每条直线的斜率为s+ts[i]递增，所以可以直接维护队首，如果当前斜率大于队首与第二个点的斜率，不断出队。
更新队列：
用(fs[i],f[i])更新队尾的元素，后将它放进去。

- **代码**

```
#include<bits/stdc++.h>
using namespace std;
const int N = 5010;
const int S = 60;
const int inf = 0x3f3f3f3f;
int n,s,ts[N],fs[N],f[N],q[N<<2];
int main()
{
    freopen("2365.in","r",stdin);
    scanf("%d%d",&n,&s);
    int T,F;
    for(int i = 1;i <= n;i++){
        scanf("%d%d",&T,&F);
        ts[i] = ts[i-1] + T;fs[i] = fs[i-1] + F;
    }
    memset(f,inf,sizeof(f));f[0] = 0;
    int l = 1,r = 1;//提前加入(0,0)
    for(int i = 1;i <= n;i++){
        while(r>l && f[q[l+1]]-f[q[l]] <= (s+ts[i])*(fs[q[l+1]]-fs[q[l]]))++l;
        f[i] = f[q[l]] + ts[i] * (fs[i]-fs[q[l]]) + s * (fs[n] - fs[q[l]]);
        while(r>l && (f[i]-f[q[r]])*(fs[q[r]]-fs[q[r-1]])<=(f[q[r]]-f[q[r-1]])*
(fs[i]-fs[q[r]]))--r;
        q[++r] = i;
    }
    printf("%d",f[n]);
    return 0;
}
```

- **复杂度分析**

时间复杂度：O(n)，已到达极限
空间复杂度：O(n)，已到达极限
满分解法，最优解法

小结

针对本题，从一开始的设计二维状态，转移O(N^3)的时间。利用“费用提前”的思想优化时间&空间，优化到O(N)的空间与O(N^2)的时间。再进一步利用”斜率优化“的思想，达到O(N)的时间空间极限复杂度。

例题二：玩具装箱

题面描述

> P 教授要去看奥运，但是他舍不下他的玩具，于是他决定把所有的玩具运到北京。他使用自己的压缩器进行压缩，其可以将任意物品变成一堆，再放到一种特殊的一维容器中。
>
> P 教授有编号为 1 · · ·*n*的*n* 件玩具，第*i* 件玩具经过压缩后的一维长度为C_i。
>
> 为了方便整理，P教授要求：
>
> 1. 在一个一维容器中的玩具编号是连续的。
> 2. 同时如果一个一维容器中有多个玩具，那么两件玩具之间要加入一个单位长度的填充物。形式地说，如果将第 *i* 件玩具到第 *j* 个玩具放到一个容器中，那么容器的长度将为 $x = j - i + \sum_{k=i}^{j} C_k$。
>
>   制作容器的费用与容器的长度有关，根据教授研究，如果容器长度为*x*，其制作费用为 $(x - L)^2$。其中 *L* 是一个常量。P 教授不关心容器的数目，他可以制作出任意长度的容器，甚至超过 *L*。但他希望所有容器的总费用最小。

输入格式

> 第一行有两个整数，用一个空格隔开，分别代表 *n* 和 *L*。
>
> 第 2 到 第 (*n*+1) 行，每行一个整数，第 (*i*+1) 行的整数代表第 *i* 件玩具的长度 C_i。

输出格式

> 输出一行一个整数，代表所有容器的总费用最小是多少。

样例输入

```
5 4
3
4
2
1
4
```

样例输出

```
1
```

状态设计&斜率优化的运用

$$f[i]=min\{f[j]+(i-j-1+\sum_{k=j+1}^{i}C_k-L)^2\}$$

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

$$令i+sum[i]=q[i]$$

$$f[i]=min\{f[j]+(q[i]-q[j]-1-L)^2\}$$

$$如果存在j1<j2<i,且dp[i]从j2转移更优(对于本题来说，也就是dp[i]从j2转移取得的值更小)$$

$$f[j1]+(q[i]-q[j1]-1-L)^2>=f[j2]+(q[i]-q[j2]-1-L)^2$$

$$f[j1]-f[j2]>=(2q[i]-(q[j2]+q[j1])-2-2L)*(q[j1]-q[j2])$$

$$f[j1]-f[j2]+(q[j1]^2-q[j2]^2)+2(1+L)(q[j1]-q[j2])>=2q[i]*(q[j1]-q[j2])$$

$$\frac{f[j1]-f[j2]+(q[j1]^2-q[j2]^2)+2(1+L)(q[j1]-q[j2])}{2(q[j1]-q[j2])}>=q[i]$$

$$令g[i]=(q[i]+L+1)^2$$

$$\frac{f[j1]-f[j2]+(q[j1]+L+1)^2-(q[j2]+L+1)^2}{2(q[j1]-q[j2])}>=q[i]$$

$$\frac{f[j1]-f[j2]+g[j1]-g[j2]}{2(q[j1]-q[j2])}>=q[i]$$

$$\frac{f[j2]+g[j2]-(f[j1]+g[j1])}{q[j2]-q[j1]}>=2q[i]$$

$$x->q[i],y->f[i]+g[i]$$

总结：如果$dp[i]=max\{a[i]b[j]+c[j]\}+d[i]+C$的形式，就可以使用斜率优化

代码

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
const int N = 50010;
int n,L,sum[N],dp[N],g[N],f[N],q[N];
inline void init(){
    scanf("%lld%lld",&n,&L);
    for(int i = 1;i <= n;i++){
        scanf("%lld",&sum[i]);
        sum[i] += sum[i-1];
        f[i] = i + sum[i];
        g[i] = (f[i]+L+1)*(f[i]+L+1);
    }
}
inline double slope(int i,int j){
    return 1.0*(dp[i]+g[i]-dp[j]-g[j])/(f[i]-f[j]);
}
signed main()
{
    init();
    int l = 1, r = 1;
    g[0] = (L+1)*(L+1);
    for(int i = 1;i <= n;i++){
```

```
        while(l<r && slope(q[l+1],q[l]) <= 2.0*f[i])++l;
        dp[i] = dp[q[l]] + (f[i]-f[q[l]]-1-L)*(f[i]-f[q[l]]-1-L);
        while(l<r && slope(i,q[r]) <= slope(q[r],q[r-1]))--r;
        q[++r] = i;
    }
    printf("%lld",dp[n]);
    return 0;
}
```

例题三：序列分割

题面描述

> 你正在玩一个关于长度为n的非负整数序列的游戏。这个游戏中你需要把序列分成k+1个非空的块。为了得到k+1块，你需要重复下面的操作k次：
> 选择一个有超过一个元素的块（初始时你只有一块，即整个序列）
> 选择两个相邻元素把这个块从中间分开，得到两个非空的块。
> 每次操作后你将获得那两个新产生的块的元素和的乘积的分数。你想要最大化最后的总得分。

输入格式

> 第一行包含两个整数n和k。保证$k+1 \le n$。
>
> 第二行包含n个非负整数$a_1, a_2, \cdots, a_n (0 \le a_i \le 104)$，表示前文所述的序列。

输出格式

> 第一行输出你能获得的最大总得分。
>
> 第二行输出k个介于1到$n-1$之间的整数，表示为了使得总得分最大，你每次操作中分开两个块的位置。第i个整数s_i 表示第i次操作将在s_i和 s_{i+1}之间把块分开。
>
> 如果有多种方案使得总得分最大，输出任意一种方案即可。

样例输入

```
7 3
4 1 3 4 0 2 3
```

样例输出

```
108
1 3 5
```

数据范围

```
2 ≤ n ≤ 100000, 1 ≤ k ≤ min(n-1,200)
```

初步分析

一眼看上去是区间dp，但是看到n的范围(1e5)之后再仔细思索，能够发现分段的顺序并不影响结果，所以先列基本转移方程。

转移方程

$$f[i][k]\text{表示前}i\text{个元素，分为}k\text{段的最大得分}$$

$$f[i][k] = max\{f[j][k-1] + sum[j] * (sum[i] - sum[j]) \quad \text{其中}sum[i]\text{为}a[i]\text{的前缀和}\}$$

就这么做?

```
#include <bits/stdc++.h>
#define int long long   //好评
using namespace std;
const int N = 1e5+10;
const int K = 210;
int n,k,sum[N],f[N][K],to[N][K];
signed main()
{
//    freopen("data.in","r",stdin);
    scanf("%lld%lld",&n,&k);
    for(int i = 1;i <= n;i++){
        scanf("%lld",&sum[i]);sum[i] += sum[i-1];
    }
    for(int i = 1;i <= n;i++){
        for(int j = 1;j <= n;j++){
            for(int p = 1;p <= k;p++){
                int now = f[j][p-1]+sum[j]*(sum[i]-sum[j]);
                if(now > f[i][p]){
                    f[i][p] = now;
                    to[i][p] = j;
                }
            }
        }
    }
    printf("%lld\n",f[n][k]);
    int i = n;
    for(int j = k;j >= 1;j--){
        i = to[i][j];
        printf("%lld ",i);
    }
    return 0;
}
```

复杂度分析

时间复杂度：O(K*N^2)，不满足
空间复杂度：O(NK)，还可以更好

开始优化:

基础操作

压维：$f[j]=max\{g[j]+sum[j]*(sum[i]-sum[j])\}$

进阶操作

还记得例1中提到的斜率优化的方法吗?
.
.
.
.
.
.
.
.
.
.
.
.
.
.
很可惜这里用不了,因为这个方程不是“关于(sum[j],f[j])的一次函数”

这里我们使用更常用的一种方法：“作差法”
任取j,k满足 0 <= k < j < i 如果j比k更优，那么有如下：

$$g[j]+s[j](s[i]-s[j])\geq g[k]+s[k](s[i]-s[k])$$

$$\frac{(g[j]-s[j]^2)-(g[k]-s[k]^2)}{s[k]-s[j]}\leq s[i]$$

把$g[i]-s[i]^2$看作y,把$s[i]$看作x

即，如果相邻两点的斜率满足上式，那么j一定比k更优

斜率优化具体的做法就是：维护一个斜率单调上升的点集单调队列，每次在队首查询第一个满足式子的点

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cstdio>
#define ll long long
using namespace std;
const int N = 1e5+5;
const int K = 205;
int n,k,q[N],to[N][K];
ll sum[N],g[N],f[N];
double slope(int i,int j) {
    if(sum[i]==sum[j]) return -1e18;
    return 1.0*((g[i]-sum[i]*sum[i])-(g[j]-sum[j]*sum[j]))/(sum[j]-sum[i]);
}
int main()
{
//    freopen("data.in","r",stdin);
    scanf("%d%d",&n,&k);
    for(int i = 1;i <= n;i++){
        scanf("%lld",&sum[i]);sum[i] += sum[i-1];
    }
```

```
    for(int t = 1;t <= k;t++){
        int l = 1, r = 1;
        for(int i = 1;i <= n;i++){
            while(l<r && slope(q[l],q[l+1]) <= sum[i])++l;
            f[i] = g[q[l]] + sum[q[l]] * (sum[i]-sum[q[l]]);
            to[i][t] = q[l];
            while(l<r && slope(q[r-1],q[r])>=slope(q[r],i))--r;
            q[++r] = i;
        }
        memcpy(g,f,sizeof(f));
    }
    printf("%lld\n",f[n]);
    for(int x=n,i=k;i>=1;--i) x=to[x][i],printf("%d%c",x," \n"[i==1]);
    return 0;
}
```

例题四：Corn Fields G

题目描述：

> 在n行m列的矩阵上，有些格子不能选择，其他格子可选可不选，且要求所有选到的格子两两没有公共边，求方案数
>
> 给定n，m，以及一个01矩阵，其中1的位置可选，0的位置不可选

样例输入：

```
2 3
1 1 1
0 1 0
```

样例输出：

```
9
```

数据范围

$n \leq 12$，$m \leq 12$

题目分析：

既然是求方案数，那我们可以枚举出所有的方案再计数，对于每个不被限制的格子，有“选”和“不选”两种状态，我们可以直接枚举每个格子后判断合法，也可以深搜的时候通过观察周围的进行剪枝，二者复杂度都在$O(2^{n\times m})$

初步优化：

可以发现某个位置是否选择，只会受到影响，或影响到相邻的一圈格子。如果只有一列的话，可以直接记录上一行的状态转移这一行。同理，由于列数很小，我们可以状压记录上一行的所有状态来转移本行

设计状态：

$$dp[i][s]表示第i行状态为s的情况下的方案数$$

转移：

$$dp[i][s]+=dp[i-1][s']，其中s和s'保证没有相邻被选的$$

代码：

```
#include <iostream>
#include <cstdio>
#define LL long long
using namespace std;
const int MAX=15;
const LL mod=100000000;
LL dp[MAX<<1][1<<MAX];
bool map[MAX<<1][MAX<<1];
int h;
int n,m;
LL ans;
//dfs暴力枚举本层可能的所有状态(土地，上一层，旁边的)
void dfs(int now,int state_last,int state_now){     //now是正在做 1<<now
    if(now==m+1){
        //dp[i+1][s0]+=dp[i][s]
        dp[h+1][state_now]+=dp[h][state_last];
        dp[h+1][state_now]%=mod;
        return ;
    }
    if(map[h+1][now] && !(state_last & (1<<now)) && !(state_now & (1<<(now-1)))
)dfs(now+1,state_last,state_now|(1<<now));
    dfs(now+1,state_last,state_now);
}
int main() {
    scanf("%d %d",&n,&m);
    for(int i=1;i<=n;i++){
        for(int j=1;j<=m;j++)cin>>map[i][j];
    }
    dp[0][0]=1;
    dfs(1,0,0);
    for(h=1;h<n;h++){     //已经做完h
        for(int j=0;j<=(1<<(m+1));j++){    //枚举状态
            if(dp[h][j])dfs(1,j,0);
        }
    }
    for(int i=0;i<=(1<<(m+1));i++)ans+=dp[n][i],ans%=mod;//,cout<<dp[1][i]<<" ";
    printf("%lld",ans);
    return 0;
}
```

复杂度分析

时间复杂度$O(n\times 2^{2\times m})$

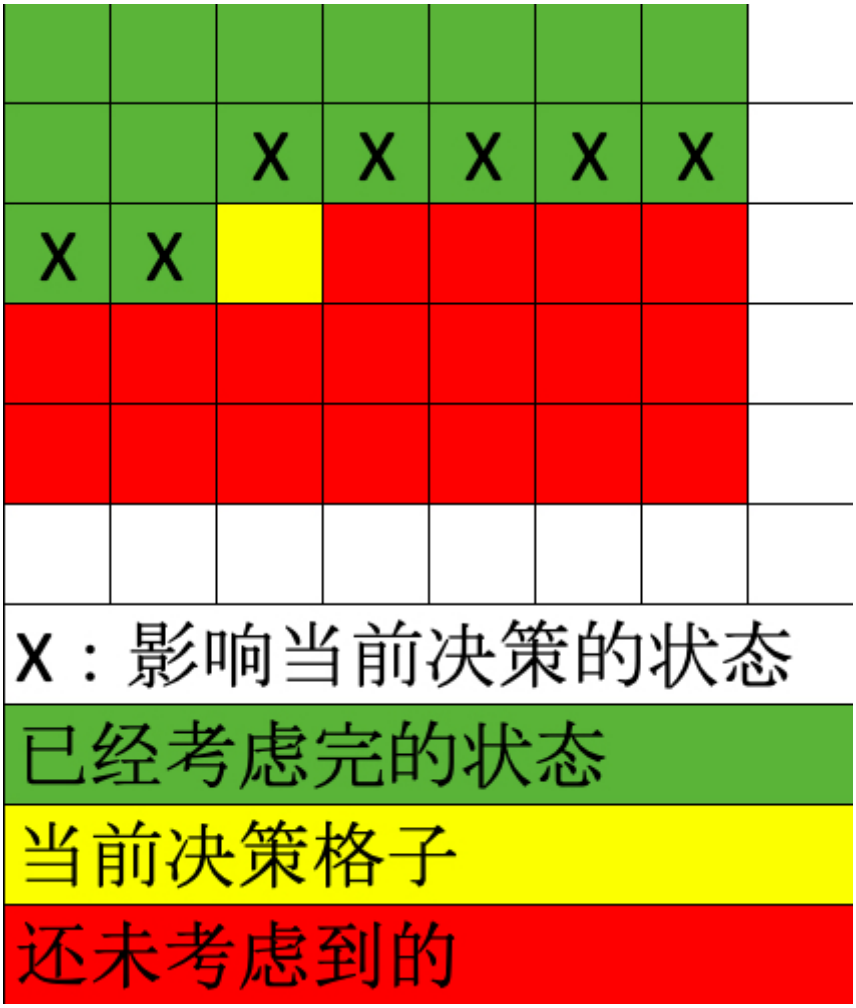
空间复杂度$O(n\times 2^m)$

进一步优化：

发现上一种方法的问题主要是每次需要枚举当前一整行的状态，这样和上一行验证的时候就会出现很多矛盾而无法转移的状态。所以我们还是每次转移一个位置，由上面分析可知转移位置(i,j)时只与$(i-1,1...m)$，$(i,1...j-1)$有关：

这样要状压$2\times m$个位置的状态，总的空间复杂度就是$n\times m\times 2^{2\times m}$无法接受

实际上还可以更少一点，由于转移是顺序的，所以上图的$X1,X2$两个状态也是不必要的：



所以我们只需要状压m个位置的状态，每列一个，也就是轮廓线的状态

设计状态：

$$dp[i][j][s]表示转移完(i,j)，轮廓线状态为s的情况下的方案数$$

转移：

$$本列选择的时候dp[i][j][s]+=dp[i][j-1][s']$$
$$本列不选的时候dp[i][j][s]+=dp[i][j-1][s']+dp[i][j-1][s'']$$

代码：

```
#include <iostream>
#include <cstdio>
#define LL long long
using namespace std;
const int MAX=15;
const LL mod=100000000;
LL dp[MAX][MAX][1<<MAX];     //正在做i行j列且轮廓线为s的方案数
bool map[MAX<<1][MAX<<1];
int n,m;
LL ans;
int main() {
    scanf("%d %d",&n,&m);
    for(int i=1;i<=n;i++){
        for(int j=1;j<=m;j++)cin>>map[i][j];
    }
```

```
    if(map[1][1])dp[1][1][1<<0]=1;
    dp[1][1][0]=1;
    for(int i=1;i<=n;i++){
        for(int j=1;j<=m;j++){
            if(i==1 && j==1)continue;
            for(int k=0;k<(1<<m);k++){    //考虑完的状态
                if(j==1){
                    if(map[i][j] && (k&(1<<(j-1))))dp[i][j][k]+=dp[i-1][m][k-
(1<<(j-1))];
                    if(!(k&(1<<(j-1))))dp[i][j][k]+=dp[i-1][m][k]+dp[i-1][m][k|
(1<<(j-1))];
                }else{
                    if(map[i][j] && (k&(1<<(j-1))) && !(k&(1<<(j-2)))){
                        //if(i==1 && j==2)cout<<k<<" ";
                        dp[i][j][k]+=dp[i][j-1][k-(1<<(j-1))];
                    }
                    if(!(k&(1<<(j-1))))dp[i][j][k]+=dp[i][j-1][k]+dp[i][j-1][k|
(1<<(j-1))];
                }
                dp[i][j][k]%=mod;
                //cout<<dp[i][j][k]<<" ";
            }
            //cout<<endl;
        }
    }
    for(int i=0;i<(1<<m);i++)ans+=dp[n][m][i],ans%=mod;
    cout<<ans;
    return 0;
}
```

复杂度分析

时间复杂度$O(n \times m \times 2^m)$

空间复杂度$O(n \times m \times 2^m)$

小结

设计状态的时候要充分利用题目的性质，比如解法1就先简化地只考虑一列，发现当前行只受到上一行的影响，所以只需要状态压缩上一行进行逐行的转移；解法2就更进一步的讨论对于一个格子的相关状态，从$2 \times m$个状态优化到m个状态，复杂度更优秀

例题五：可乐

题目描述：

> 给定一n点m边的图，第0秒时你在1号点，此后每一秒可以选择留在原地，去下一个相邻的点，自爆，求t秒后你的行为方案数
>
> 第一行n，m，接下来m行每行两个数u, v表示一条双向边，最后一行一个数t

样例输入：

```
3 2
1 2
2 3
2
```

样例输出：

```
8
```

数据范围

$n \leq 100$，$m \leq 100$，$t \leq 10^9$

题目分析：

既然是求方案数，那我们仍然可以深搜走出每一个方案再计数，对于当前位置，可以自爆计数直接返回，或者走向相邻点，或者原地不动，复杂度都在$O(n^t)$

初步优化：

可以发现某个位置是否选择，只会受到影响，或影响到相邻的一圈格子。如果只有一列的话，可以直接记录上一行的状态转移这一行。同理，由于列数很小，我们可以状压记录上一行的所有状态来转移本行

设计状态：

$$dp[i][j]表示第i秒在j点的方案数$$

转移：

$$dp[i][j]+=dp[i-1][j']，其中j和j'有边相连$$

代码：

```
#include <iostream>
#include <cstdio>
using namespace std;
const int MAXt=1e6+2;
const int MAXn=32;
const int MAXm=102;
const int mod=2017;
struct edge{
    int next,to;
}h[(MAXm+MAXn+5)<<1];
int head[MAXn],cnt;
void add(int x,int y){
    h[++cnt]=edge{head[x],y};
    head[x]=cnt;
}
int dp[MAXt][MAXn];     //时间i到j点的方案数
int n,m,t,ans;
int main(){
    scanf("%d %d",&n,&m);
    for(int i=1;i<=n;i++)add(i,0),add(i,i);          //自爆，有去无回
```

```
    add(0,0);
    for(int x,y,i=1;i<=m;i++)scanf("%d %d",&x,&y),add(x,y),add(y,x);
    scanf("%d",&t);
    dp[0][1]=1;
    for(int i=1;i<=t;i++){
        for(int j=0;j<=n;j++){
            if(!dp[i-1][j])continue;
            for(int aim,u=head[j];u;u=h[u].next){
                aim=h[u].to;
                dp[i][aim]=(dp[i][aim]+dp[i-1][j])%mod;
            }
        }
    }
    for(int i=0;i<=n;i++)ans=(ans+dp[t][i])%mod;
    printf("%d",ans);
    return 0;
}
```

复杂度分析

时间复杂度$O(m \times t)$

空间复杂度$O(n \times t)$

进一步优化：

这道题$t \leq 1e9$，要么是找规律，要么把复杂度降到$log\ t$；由于是给定的图所以不是很好找规律，另一方面n又很小，所以可以考虑用邻接矩阵转移，先考虑走一步时从i到j的转移：
$dp[i][j] = \sum dp[i][k] \times dp[k][j]$(乘法原理)，发现这就是一个很经典的矩阵乘法的模型，走t步就是$dp[i][j]$的t次方；还有自爆和停留操作，分别对应着一条从从i到0的单向边(有去无回)和自环

设计状态：

$$dp[i][j]表示从i到j的方案数$$

转移：

$$(dp[i][j])^k表示走了k步后从i到j的方案数$$

代码：

```
#include <iostream>
#include <cstdio>
using namespace std;
const int MAX=35;
const int mod=2017;
struct node{
    int a[MAX][MAX];
}map,ret,rret;
int n,m,t,ans;
node mul(node x,node y){
    for(int i=0;i<=n;i++){
        for(int j=0;j<=n;j++){
            rret.a[i][j]=0;
```

```
            for(int k=0;k<=n;k++){
                rret.a[i][j]=(rret.a[i][j]+(x.a[i][k]*y.a[k][j])%mod)%mod;
            }
        }
    }
    return rret;
}
void quick(){
    for(int i=0;i<=n;i++)ret.a[i][i]=1;
    while(t){
        if(t&1){
            ret=mul(ret,map);
        }
        t>>=1;
        map=mul(map,map);
    }
}
int main(){
    scanf("%d %d",&n,&m);
    for(int i=0;i<=n;i++)map.a[i][i]=1,map.a[i][0]=1;        //原地不动就是自环，爆炸
就指向0且回不来
    for(int x,y,i=1;i<=m;i++)scanf("%d %d",&x,&y),map.a[x][y]=map.a[y][x]=1;
    scanf("%d",&t);
    quick();
    for(int i=0;i<=n;i++)ans=(ans+ret.a[1][i])%mod;
    printf("%d",ans);
    return 0;
}
```

复杂度分析

时间复杂度$O(n^3 \times log_2\ t)$

空间复杂度$O(n^2)$

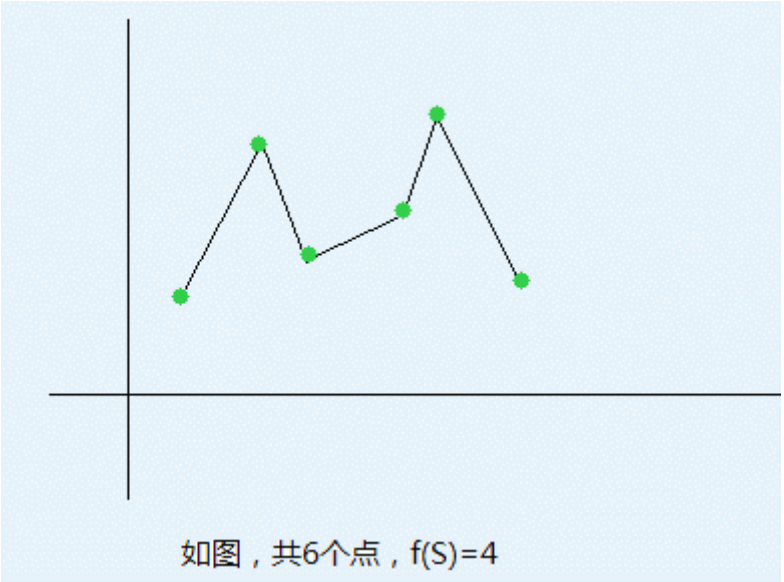
小结

设计状态的时候要注意观察题目的数据范围，$t \leq 1e9$的范围很明显不能一个一个时间走，要么就是找规律，要么就是把复杂度降到$log\ t$，通常可以考虑用倍增，图论可以考虑应用邻接矩阵乘法的意义，将多步移动做成快速幂

例题六：折线统计

题面描述

> 二维平面上有n个点(x_i, y_i)，现在这些点中取若干点构成一个集合S，对它们按照x坐标排序，顺次连接，将会构成一些连续上升、下降的折线，设其数量为f(S)。如下图中，1->2,2->3,3->5,5->6（数字为下图中从左到右的点编号），将折线分为了4部分，每部分连续上升、下降。



> 现给定k，求满足f(S) = k的S集合个数。

输入格式

> 第一行两个整数n和k，以下n行每行两个数(x_i, y_i)表示第i个点的坐标。所有点的坐标值都在[1, 100000]内，且不存在两个点，x坐标值相等或y坐标值相等

输出格式

> 输出满足要求的方案总数 mod 100007的结果

样例输入

```
5 1
5 5
3 2
4 4
2 3
1 1
```

样例输出

```
19
```

数据范围

```
对于100%的数据，n <= 50000，0 < k <= 10
```

初步分析&状态设计

```
dp痕迹非常明显，根据范围判断复杂度应到达O(nklog(?)) ?表示还不确定，初步分析不能确定是n(5e4)还是纵坐标的范围1e5.
```

下面设计状态：

$f[i][j][0/1]$表示前i个点，分为j段，最后一段是上升/下降的方案数

得到转移方程：

$$f[i][j][0/1] = \begin{cases} f[k][j][0] + f[k][j-1][1], & y[k] < y[i] \\ f[k][j][1] + f[k][j-1][0], & y[k] >= y[i] \end{cases}$$

初始(边界)条件为 f[i][0][0/1] = 1

转移优化

考虑上述方程 **依靠** 点的纵坐标值进行转移，使用树状数组以点的纵坐标值为键值，维护前缀方案数

$c[i][j][0/1]$维护纵坐标值小于等于i，分为j段，上升/下降的前缀方案数

代码

```
#include<bits/stdc++.h>
using namespace std;
const int mod = 1e5+7;
const int Y = 1e5+10;
const int N = 5e4+10;
const int K = 12;
int n,k,f[N][K][2],c[Y][K][2];
struct node{
    int x,y;
    friend bool operator < (const node A,const node B){
        return A.x < B.x;
    }
}a[N];
inline int lowbit(int x){
    return x & (-x);
}
inline void add(int y,int j,int dir,int v){
    for(int i = y;i < Y;i += lowbit(i)){
        c[i][j][dir] = (c[i][j][dir] + v)%mod;
    }
}
inline int query(int y,int j,int dir){
    int sum = 0;
    for(int i = y;i;i -= lowbit(i)){
        sum = (sum + c[i][j][dir])%mod;
    }
    return sum;
}
inline void init(){
    scanf("%d%d",&n,&k);
    for(int i = 1;i <= n;i++)scanf("%d%d",&a[i].x,&a[i].y);
    sort(a+1,a+1+n);
}
int main()
{
    freopen("line.in","r",stdin);
    init();
    for(int i = 1;i <= n;i++){
```

```
        f[i][0][0] = f[i][0][1] = 1;
        add(a[i].y, 0, 0, 1); add(a[i].y, 0, 1, 1);
        for(int j = 1;j <= k;j++){
            f[i][j][0] = (f[i][j][0] + query(a[i].y-1, j, 0) + query(a[i].y-1,
j-1, 1))%mod;
            f[i][j][1] = (f[i][j][1] + (query(Y-1, j, 1)-query(a[i].y, j, 1) +
query(Y-1, j-1, 0)-query(a[i].y, j-1, 0))%mod+mod)%mod;
            add(a[i].y, j, 0, f[i][j][0]); add(a[i].y, j, 1, f[i][j][1]);
        }
    }
    int ans = 0;
    for(int i = 1;i <= n;i++){
        for(int j = 0;j < 2;j++){
            ans = (ans + f[i][k][j])%mod;
            cout<<f[i][k][j]<<" ";
        }
        cout<<endl;
    }
    printf("%d",ans);
    return 0;
}
```

复杂度分析

时间复杂度达到O(nklog(1e5)) 1e5即为纵坐标范围

小结

此类动态规划的问题，通过二维偏序的概念运用树状数组进行优化，类似的还有经典问题：树状数组求逆序对，本质上也是树状数组通过二维偏序优化动态规划。通常方程的转移都依靠一个键值（分类），而运用树状数组的下标对应键值的方法，达到时间上的优化，一般为一个数量级的优化(O(kn)->O(klogn))。

例题七：子序列问题

题目描述

> 给定一个长度为 *n* 的正整数序列 $A_1, A_2, \cdots, A_n$。定义一个函数 *f(l,r)* 表示：序列中下标在 [*l*,*r*] 范围内的子区间中，不同的整数个数。换句话说，f(l,r) 就是集合 $\{A_l, A_{l+1}, \cdots, A_r\}$ 的大小，这里的集合是不可重集，即集合中的元素互不相等。
>
> 现在，请你求出 $\sum_{l=1}^{n} \sum_{r=l}^{n} ((f(l,r))^2$。由于答案可能很大，请输出答案对 10^9 +7 取模的结果。

输入格式

> 第一行一个正整数 *n*，表示序列的长度。
>
> 第二行 *n* 个正整数，相邻两个正整数用空格隔开，表示序列 $A_1, A_2, \cdots, A_n$。

输出格式

> 仅一行一个非负整数，表示答案对 10^9+7 取模的结果

样例输入

```
4
2 1 3 2
```

样例输出

```
43
```

数据范围

> 对于 100% 的数据，满足 $1\le n\le 10^6$，集合中每个数的范围是 $[1,10^9]$。

分析&设计状态&优化

设 $g(r)=\sum\limits_{l=1}^{r} f(l,r)^2$，那么本题要求的答案为 $\sum\limits_{r=1}^{n} g(r)$。

思路很简单的，循环枚举 r，然后用数据结构在 $O(\log n)$ 的时间内求出 $g(r)$（当然得利用 $g(r-1)$ 的值），这样总时间复杂度为 $O(n\log n)$。

首先预处理一个 `last` 数组：`last[i]` 表示上一个等于 a_i 的位置，不存在就为 0。用 `map` 可能会卡常，建议离散化。

下面考虑 $r\Rightarrow r+1$ 的变化：

此时新增了一个数 a_{r+1}，在 $l\in[last_{r+1}+1,r+1]$ 的范围内的 $f(l,r+1)=f(l,r)+1$，因为这部分原本没有 a_{r+1} 这个数，而其余部分是不变的，对 g 也毫无影响。

那么 $g(r+1)-g(r)=\sum\limits_{l=last_{r+1}+1}^{r+1} f(l,r+1)^2-f(l,r)^2=\sum\limits_{l=last_{r+1}+1}^{r+1}(2f(l,r)+1)=2\sum\limits_{l=last_{r+1}+1}^{r+1} f(l,r)+r+1-last_{r+1}$

那么对于当前固定的 r，我们的数据结构需要能查询 $f(l,r)$ 的一段区间和，其中 $l\in[last_{r+1}+1,r+1]$，并且让这段区间加 1。

采用树状数组/线段树均可，目的是实现区间修改+区间查询

代码

```
#include <bits/stdc++.h>
#define ls (id<<1)
#define rs ((id<<1)|1)
#define mid ((l+r)>>1)
#define ll long long
using namespace std;
const int N = 1e6+10;
const int mod = 1e9+7;
int n,A[N],last[N];
ll ans,g[N];
struct segmentTree{
    ll c[N<<2],lazy[N<<2];
```

```
    inline void pushup(int id){
        c[id] = (c[ls] + c[rs])%mod;
    }
    inline void pushdown(int id,int l,int r){
        if(lazy[id]){
            lazy[ls] = (lazy[ls] + lazy[id])%mod;
            lazy[rs] = (lazy[rs] + lazy[id])%mod;
            c[ls] = (c[ls] + lazy[id]*(mid-l+1))%mod;
            c[rs] = (c[rs] + lazy[id]*(r-mid))%mod;
            lazy[id] = 0;
        }
    }
    void update(int id,int l,int r,int x,int y){//区间加
        if(x<=l&&r<=y){
            lazy[id] = (lazy[id] + 1)%mod;
            c[id] = (c[id] + r - l + 1)%mod;
            return ;
        }
        pushdown(id,l,r);
        if(x<=mid){
            update(ls,l,mid,x,y);
        }
        if(y>mid){
            update(rs,mid+1,r,x,y);
        }
        pushup(id);
    }
    ll query(int id,int l,int r,int x,int y){//区间和
        if(x<=l&&r<=y){
            return c[id];
        }
        pushdown(id,l,r);
        ll ans = 0;
        if(x<=mid){
            ans = (ans + query(ls,l,mid,x,y))%mod;
        }
        if(y>mid){
            ans = (ans + query(rs,mid+1,r,x,y))%mod;
        }
        return ans;
    }
}tree;
inline void LSH(){
    vector<int>q;
    for(int i = 1;i <= n;i++)q.push_back(A[i]);
    sort(q.begin(),q.end());
    unique(q.begin(),q.end());
    for(int i = 1;i <= n;i++){
        A[i] = (int)(lower_bound(q.begin(),q.end(),A[i])-q.begin()+1);
    }
}
int main()
{
    scanf("%d",&n);
    for(int i = 1;i <= n;i++)scanf("%d",&A[i]);
    LSH();
    //g[1] = 1;last[A[1]] = 1;ans = 1;
    for(int r = 1;r <= n;r++){
```

```
        //根据g[r-1]求出g[r]
        g[r] = ((g[r-1] + (2LL*tree.query(1,1,n,last[A[r]]+1,r))%mod)%mod+r-
last[A[r]])%mod;
        ans = (ans + g[r])%mod;
        tree.update(1,1,n,last[A[r]]+1,r);
        last[A[r]] = r;
    }
    printf("%lld",ans);
    return 0;
}
```

复杂度分析

时间复杂度到达了O(nlogn)

总结

> 动态规划的优化有很多种，不同的思想、转化、数据结构分别对应不同类型的题目，包括“费用提前”的思想解决转移后效性、对转移方程转化后使用斜率优化、通过树状数组、线段树等数据结构加速转移等。在拿到动态规划题后，我们首先可以通过数据范围大致确定所需的时间复杂度，再结合具体方程和转移方式，选择恰当的优化方法。同时需要注意的是，不要局限于优化的思路，对于某些题目通过例如贪心的方法也可以解决，并达到更好的时间复杂度。