《 女 生 自 用 》

⭐⭐ - 1队

[杂项 3](#_Toc73702977)

[二分 3](#_Toc73702978)

[莫队 3](#_Toc73702979)

[三分 4](#_Toc73702980)

[凸包 4](#_Toc73702981)

[字符串 4](#_Toc73702982)

[KMP 4](#_Toc73702983)

[EXKMP 4](#_Toc73702984)

[字典树 5](#_Toc73702985)

[AC机 5](#_Toc73702986)

[后缀数组 6](#_Toc73702987)

[后缀自动机的各种应用 7](#_Toc73702988)

[回文树 7](#_Toc73702989)

[附：快速IO 7](#_Toc73702990)

[附：没用的优化 8](#_Toc73702991)

[图论 8](#_Toc73702992)

[Floyd 8](#_Toc73702993)

[SPFA（它已经死了 //vector？ 8](#_Toc73702994)

[Dijkstra //vector？ 8](#_Toc73702995)

[Tarjan 9](#_Toc73702996)

[强连通分量 9](#_Toc73702997)

[缩点 10](#_Toc73702998)

[割点 10](#_Toc73702999)

[点双连通分量 11](#_Toc73703000)

[边双连通分量 11](#_Toc73703001)

[桥： 11](#_Toc73703002)

[2-SAT 11](#_Toc73703003)

[找环 12](#_Toc73703004)

[网络流 12](#_Toc73703005)

[最大流-Dinic 12](#_Toc73703006)

[最大流-HLPP+黑魔法优化 13](#_Toc73703007)

[最小费用最大流（MCMF）-EK 14](#_Toc73703008)

[树形DP求树的最小支配集,最小点覆盖,最大独立集 15](#_Toc73703009)

[数据结构 16](#_Toc73703010)

[并查集 16](#_Toc73703011)

[树状数组（动态分配//改成能区间加的区间查的？ 16](#_Toc73703012)

[树状数组（区间查询区间修改） 17](#_Toc73703013)

[线段树//待替换 17](#_Toc73703014)

[主席树（非工程版） 18](#_Toc73703015)

[主席树//待替换 18](#_Toc73703016)

[树链剖分//？ 19](#_Toc73703017)

[Splay 21](#_Toc73703018)

[LCA 22](#_Toc73703019)

[大数 23](#_Toc73703020)

[线性基 25](#_Toc73703021)

[ST表 25](#_Toc73703022)

[带修改整体二分 25](#_Toc73703023)

[计算几何 26](#_Toc73703024)

[必要的头 26](#_Toc73703025)

[分数类 27](#_Toc73703026)

[二维向量 27](#_Toc73703027)

[三维向量 29](#_Toc73703028)

[N维向量 30](#_Toc73703029)

[矩阵 32](#_Toc73703030)

[方阵 34](#_Toc73703031)

[二维直线 34](#_Toc73703032)

[二维有向线段 35](#_Toc73703033)

[二维多边形 35](#_Toc73703034)

[数论 36](#_Toc73703035)

[欧拉筛 36](#_Toc73703036)

[min\_25筛框架 36](#_Toc73703037)

[卢卡斯定理 37](#_Toc73703038)

[EXCRT 37](#_Toc73703039)

[扩欧求逆元 37](#_Toc73703040)

[FFT 37](#_Toc73703041)

[拉格朗日插值 38](#_Toc73703042)

[高斯消元 39](#_Toc73703043)

[公式 39](#_Toc73703044)

[来自bot的球盒问题 39](#_Toc73703045)

[OTHER 39](#_Toc73703046)

[线形基 40](#_Toc73703047)

[Add fhq-Treap 41](#_Toc73703048)

[区间翻转(可以部分替代splay) 41](#_Toc73703049)

[可持久化 42](#_Toc73703050)

[常见博弈 43](#_Toc73703051)

[巴什博弈 43](#_Toc73703052)

[威佐夫博弈 43](#_Toc73703053)

[nim博弈 43](#_Toc73703054)

[anti-nim博弈 43](#_Toc73703055)

[阶梯博弈 43](#_Toc73703056)

## 杂项

### 二分

最大化最小值（二分值最小越小越容易满足条件，求最大值）

区间长度为1时的写法：  
解的范围为

// 计算区间为[lb, rb]  
while( rb > lb ) // 区间长度为1时终止循环  
{  
 // 防止溢出  
 int m = lb + (rb - lb + 1) / 2; // 由于是区间长度为1时终止循环，所以要加1  
 if( ok(m) ) lb = m;  
 else rb = m - 1;  
}  
// 跳出循环时 lb == rb

区间长度为2时的写法：  
解的范围为

while( rb - lb > 1 ) // 区间长度为2时终止循环  
{  
 // 防止溢出  
 int m = lb + (rb - lb) / 2; // 由于是区间长度为2时终止循环，所以不用加1（不会死循环）  
 if( ok(m) ) lb = m;  
 else rb = m;  
}  
// 跳出循环时 lb + 1 == rb  
// 答案为 lb

最小化最大值（二分值越大越容易满足条件，求最小值）

区间长度为1时的写法：  
解的范围为

while( rb > lb )  
{  
 // 防止溢出  
 int m = lb + (rb - lb) / 2; // 这里虽然区间长度为1，但不需要加1（不会死循环）  
 if( ok(m) ) rb = m;  
 else lb = m + 1;  
}  
// 跳出循环时 lb == rb

区间长度为2时的写法：  
解的范围为

while( rb - lb > 1 )  
{  
 // 防止溢出  
 int m = lb + (rb - lb) / 2;  
 if( ok(m) ) rb = m;  
 else lb = m;  
}  
// 跳出循环时 lb + 1 == rb  
// 答案为 rb

浮点数的二分，100次循环可以达到2-100(约为10-30)的精度范围

以最大化最小值为例（即小于该数的解均满足条件）

for( int i = 0; i < 100; ++ i )  
{  
 double m = (lb + rb) / 2;  
 if(check(m)) lb=m;  
 else rb=m;  
}  
// 跳出循环时 lb 与 rb 近似相等，所以都可作为答案

### 莫队

//动态  
struct unit {  
 int l, r, id;  
 bool operator<(const unit &x) const {  
 return l / sq == x.l / sq  
 ? (r == x.r ? 0 : ((l / sq) & 1) ^ (r < x.r))  
 : l < x.l;//莫队奇偶优化  
 }  
};  
  
int totq=0,totxiu=0,sq,ans[N],ans1=0,n,m,a[N],sum[N],l=1,r=0,now=0;  
  
int main()  
{  
 scanf("%d%d",&n,&m);  
 for (int i=1;i<=n;i++) scanf("%d",&a[i]);  
 for (int i=1;i<=m;i++)  
 {  
 char s[2];  
 int x,y;  
 scanf("%s%d%d",&s,&x,&y);  
 if (s[0]=='Q')  
 {  
 totq++;  
 q[totq].l=x; q[totq].r=y;  
 q[totq].x=totxiu;  
 q[totq].id=totq;  
 }  
 else  
 {  
 totxiu++;  
 xiu[totxiu].id=x;  
 xiu[totxiu].cl=y;  
 }  
 }  
 sq=sqrt(n\*1.0);  
 sort(q+1,q+1+totq,cmp);  
 for(int i=1;i<=totq;i++)  
 {  
 while(l>q[i].l)l--,ins(l);  
 while(r<q[i].r)r++,ins(r);  
 while(l<q[i].l)del(l),l++;  
 while(r>q[i].r)del(r),r--;  
   
 while(now<q[i].x)change(now+1),now++;  
 while(now>q[i].x)change(now),now--;  
 ans[q[i].id]=ans1;  
 }  
 for(int i=1;i<=totq;i++)  
 printf("%d\n",ans[i]);  
}

//基本  
struct node {  
 ll l,r,id,A,B;  
}Q[maxn];  
ll be[maxn],a[maxn],sq,num,l=1,r=0,sum[maxn];  
ll ans1=0;  
ll gcd(ll a,ll b)  
 {  
 if(b==0)return a;  
 else return gcd(b,a%b);  
 }  
bool cmp1(const node& A,const node&B) {return be[A.l]==be[B.l]?A.r<B.r:A.l<B.l;}  
bool cmp2(const node& A,const node&B){return A.id<B.id;}  
void ins(int x) {ans1=ans1+2\*sum[x];sum[x]++;}  
void del(int x) {ans1=ans1-2\*sum[x]+2;sum[x]--;}  
int main()  
{  
 scanf("%lld%lld",&n,&m);  
 sq=sqrt(n);  
 for(ll i=1;i<=n;i++)  
 scanf("%d",&a[i]);  
 for(ll i=1;i<=m;++i)  
 {  
 scanf("%d%d",&Q[i].l,&Q[i].r);  
 Q[i].id=i;  
 }  
 for(ll i=1;i<=n;++i){  
 be[i]=i/sq+1;  
 }  
 sort(Q+1,Q+1+m,cmp1);  
 for(int i=1;i<=m;i++){  
 while(l<Q[i].l)del(a[l]),++l;  
 while(l>Q[i].l)--l,ins(a[l]);  
 while(r<Q[i].r)++r,ins(a[r]);  
 while(r>Q[i].r)del(a[r]),--r;  
 if(Q[i].l==Q[i].r){Q[i].A=0;Q[i].B=1;continue;}  
 Q[i].B=ans1;Q[i].A=(Q[i].r-Q[i].l+1)\*(Q[i].r-Q[i].l);  
 }  
 sort(Q+1,Q+1+m,cmp2);  
 for(ll i=1;i<=m;i++){  
 if(Q[i].A==0)printf("0/1\n");  
 else {  
 int g=gcd(Q[i].A,Q[i].B);  
 printf("%lld/%lld\n",Q[i].B/g,Q[i].A/g);  
 }  
 }  
 return 0;  
}

### 三分

// 整数  
while(l+1<r)  
{  
 int lm=(l+r)>>1,rm=(lm+r)>>1;  
 if(judge(lm)>judge(rm))  
 r=rm;  
 else  
 l=lm;  
}  
// double  
  
while(l+eps<r)  
{  
 double lm=(l+r)/2,rm=(lm+r)/2;  
 if(judge(lm)>judge(rm))  
 r=rm;  
 else  
 l=lm;  
}

### 凸包

// stk[] 是整型，存的是下标  
// p[] 存储向量或点  
tp = 0; // 初始化栈  
std::sort(p + 1, p + 1 + n); // 对点进行排序  
stk[++tp] = 1;  
//栈内添加第一个元素，且不更新 used，使得 1 在最后封闭凸包时也对单调栈更新  
for (int i = 2; i <= n; ++i) {  
 while (tp >= 2 // 下一行 \* 被重载为叉积  
 && (p[stk[tp]] - p[stk[tp - 1]]) \* (p[i] - p[stk[tp]]) <= 0)  
 used[stk[tp--]] = 0;  
 used[i] = 1; // used 表示在凸壳上  
 stk[++tp] = i;  
}  
int tmp = tp; // tmp 表示下凸壳大小  
for (int i = n - 1; i > 0; --i)  
if (!used[i]) {  
 // ↓ 求上凸壳时不影响下凸壳  
 while (tp > tmp && (p[stk[tp]] - p[stk[tp - 1]]) \* (p[i] - p[stk[tp]]) <= 0)  
 used[stk[tp--]] = 0;  
 used[i] = 1;  
 stk[++tp] = i;  
}  
for (int i = 1; i <= tp; ++i) // 复制到新数组中去  
h[i] = p[stk[i]];  
int ans = tp - 1;

## 字符串

### KMP

inline void Getnext(LL next[], char t[])  
{  
 LL p1 = 0;  
 LL p2 = next[0] = -1;  
 LL strlen\_t = strlen(t);  
 while (p1 < strlen\_t)  
 {  
 if (p2 == -1 || t[p1] == t[p2])  
 next[++p1] = ++p2;  
 else  
 p2 = next[p2];  
 }  
}  
  
inline void KMP(char string[], char pattern[], LL next[])  
{  
 LL p1 = 0;  
 LL p2 = 0;  
 LL strlen\_string = strlen(string);  
 LL strlen\_pattern = strlen(pattern);  
 while (p1 < strlen\_string)  
 {  
 if (p2 == -1 || string[p1] == pattern[p2])  
 p1++, p2++;  
 else  
 p2 = next[p2];  
 if (p2 == strlen\_pattern)  
 printf("%lld\n", p1 - strlen\_pattern + 1), p2 = next[p2];  
 }  
}

### EXKMP

string pattern;  
string s;  
LL nxt[EXKMPM];  
LL extend[EXKMPM];  
  
void getNEXT(string &pattern, LL next[])  
{  
 LL pLen = pattern.length();  
 LL a = 0, k = 0;  
  
 next[0] = pLen;  
 for (auto i = 1; i < pLen; i++)  
 {  
 if (i >= k || i + next[i - a] >= k)  
 {  
 if (i >= k)  
 k = i;  
 while (k < pLen && pattern[k] == pattern[k - i])  
 k++;  
 next[i] = k - i;  
 a = i;  
 }  
 else  
 {  
 next[i] = next[i - a];  
 }  
 }  
}  
  
void EXKMP(string &s, string &pattern, LL extend[], LL next[]) // string类得配O2不然过不了  
{  
 LL pLen = pattern.length();  
 LL sLen = s.length();  
 LL a = 0, k = 0;  
  
 getNEXT(pattern, next);  
  
 for (auto i = 0; i < sLen; i++)  
 {  
 if (i >= k || i + next[i - a] >= k)  
 {  
 if (i >= k)  
 k = i;  
 while (k < sLen && k - i < pLen && s[k] == pattern[k - i])  
 k++;  
 extend[i] = k - i;  
 a = i;  
 }  
 else  
 {  
 extend[i] = next[i - a];  
 }  
 }  
}

### 字典树

#include <cstdio>  
const int N = 500010;  
char s[60];  
int n, m, ch[N][26], tag[N], tot = 1;  
int main() {  
 scanf("%d", &n);  
 for (int i = 1; i <= n; ++i) {  
 scanf("%s", s + 1);  
 int u = 1;  
 for (int j = 1; s[j]; ++j) {  
 int c = s[j] - 'a';  
 if (!ch[u][c]) ch[u][c] = ++tot;  
 u = ch[u][c];  
 }  
 tag[u] = 1;  
 }  
 scanf("%d", &m);  
 while (m--) {  
 scanf("%s", s + 1);  
 int u = 1;  
 for (int j = 1; s[j]; ++j) {  
 int c = s[j] - 'a';  
 u = ch[u][c];  
 if (!u) break; // 不存在对应字符的出边说明名字不存在  
 }  
 if (tag[u] == 1) {  
 tag[u] = 2;  
 puts("OK");  
 }  
 else if (tag[u] == 2)  
 puts("REPEAT");  
 else  
 puts("WRONG");  
 }  
 return 0;  
}

### AC机

#define Aho\_CorasickAutomaton 2000010  
#define CharacterCount 26  
struct TrieNode  
{  
 TrieNode \*son[CharacterCount], \*fail;  
 // LL word\_count;  
 LL logs;  
  
} T[Aho\_CorasickAutomaton];  
vector<TrieNode \*> FailEdge[Aho\_CorasickAutomaton];  
LL AC\_counter = 0;  
  
vector<TrieNode \*> trieIndex;  
  
TrieNode \*insertWords(string &s)  
{  
 auto root = &T[0];  
 for (auto i : s)  
 {  
 auto nxt = i - 'a';  
 if (root->son[nxt] == NULL)  
 root->son[nxt] = &T[++AC\_counter];  
 root = root->son[nxt];  
 }  
 // word\_count[root]++;  
  
 return root; // 返回含单词的节点号  
} // 用例：trieIndex.push\_back(insertWords(s));  
  
TrieNode \*insertWords(char \*s, LL &sLen)  
{  
 auto root = &T[0];  
 for (auto i = 0; i < sLen; i++)  
 {  
 auto nxt = s[i] - 'a';  
 if (root->son[nxt] == NULL)  
 root->son[nxt] = &T[++AC\_counter];  
 root = root->son[nxt];  
 }  
 // word\_count[root]++;  
  
 return root; // 返回含单词的节点号  
}  
  
void getFail()  
{  
 queue<TrieNode \*> Q; // bfs用  
 for (auto i = 0; i < CharacterCount; i++)  
 {  
 if (T[0].son[i] != NULL)  
 {  
 T[0].son[i]->fail = &T[0];  
 Q.push(T[0].son[i]);  
 }  
 }  
 while (!Q.empty())  
 {  
 auto now = Q.front();  
 Q.pop();  
 now->fail = now->fail == NULL ? &T[0] : now->fail;  
 for (auto i = 0; i < CharacterCount; i++)  
 {  
 if (now->son[i] != NULL)  
 {  
 now->son[i]->fail = now->fail->son[i];  
 Q.push(now->son[i]);  
 }  
 else  
 {  
 now->son[i] = now->fail->son[i];  
 }  
 }  
 }  
} // 先设T[0].fail=0;所有单词插完以后调用一次  
  
LL query(string &s)  
{  
 auto now = &T[0];  
 auto ans = 0;  
 for (auto i : s)  
 {  
 now = now->son[i - 'a'];  
 now = now == NULL ? &T[0] : now;  
 now->logs++;  
 // for (auto j = now; j /\*&& ~word\_count[j]\*/; j = fail[j])  
 // {  
 // // ans += word\_count[j];  
 // // cout << "j:" << j << endl;  
 // // if (word\_count[j])  
 // logs[j]++;  
 // // for (auto k : word\_position[j])  
 // // pattern\_count[k]++;  
 // // word\_count[j] = -1; // 标记已经遍历的节点  
 // }  
 }  
  
 for (auto i = 1; i <= AC\_counter; i++)  
 {  
 FailEdge[T[i].fail - T].push\_back(&T[i]);  
 }  
  
 return ans;  
} // 查询母串，getFail后使用一次  
  
LL query(char \*s, LL &sLen)  
{  
 auto now = &T[0];  
 auto ans = 0;  
 for (auto i = 0; i < sLen; i++)  
 {  
 now = now->son[s[i] - 'a'];  
 now = now == NULL ? &T[0] : now;  
 now->logs++;  
 // for (auto j = now; j /\*&& ~word\_count[j]\*/; j = fail[j])  
 // {  
 // ans += word\_count[j];  
 // cout << "j:" << j << endl;  
 // if (word\_count[j])  
  
 // for (auto k : word\_position[j])  
 // pattern\_count[k]++;  
 // word\_count[j] = -1; // 标记已经遍历的节点  
 // }  
 }  
  
 for (auto i = 1; i <= AC\_counter; i++)  
 {  
 FailEdge[T[i].fail - T].push\_back(&T[i]);  
 }  
  
 return ans;  
}  
  
void AC\_dfs(TrieNode \*u)  
{  
  
 for (auto i : FailEdge[u - T])  
 {  
 AC\_dfs(i);  
 u->logs += i->logs;  
 }  
} // query完后使用，一般搜0号点  
  
// 输出答案使用for(auto i:trieIndex)cout<<i.logs<<endl;这样

### 后缀数组

#include <algorithm>  
#include <cstdio>  
#include <cstring>  
#include <iostream>  
using namespace std;  
const int N = 1000010;  
char s[N];  
int n, sa[N], rk[N << 1], oldrk[N << 1], id[N], cnt[N];  
int main() {  
 int i, m, p, w;  
 scanf("%s", s + 1);  
 n = strlen(s + 1);  
 m = max(n, 300);  
 for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];  
 for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];  
 for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;  
 for (w = 1; w < n; w <<= 1) {  
 memset(cnt, 0, sizeof(cnt));  
 for (i = 1; i <= n; ++i) id[i] = sa[i];  
 for (i = 1; i <= n; ++i) ++cnt[rk[id[i] + w]];  
 for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];  
 for (i = n; i >= 1; --i) sa[cnt[rk[id[i] + w]]--] = id[i];  
 memset(cnt, 0, sizeof(cnt));  
 for (i = 1; i <= n; ++i) id[i] = sa[i];  
 for (i = 1; i <= n; ++i) ++cnt[rk[id[i]]];  
 for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];  
 for (i = n; i >= 1; --i) sa[cnt[rk[id[i]]]--] = id[i];  
 memcpy(oldrk, rk, sizeof(rk));  
 for (p = 0, i = 1; i <= n; ++i) {  
 if (oldrk[sa[i]] == oldrk[sa[i - 1]] &&  
 oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]) {  
 rk[sa[i]] = p;  
 }  
 else {  
 rk[sa[i]] = ++p;  
 }  
 }  
 }

### 后缀自动机的各种应用

struct state {  
 int len, link;  
 std::map<char, int> next;//可以考虑少一个log  
};  
// SAM 本身将会存储在一个 state 结构体数组中。我们记录当前自动机的大小 sz 和变量 last ，当前整个字符串对应的状态。  
const int MAXLEN = 100000;  
state st[MAXLEN \* 2];  
int sz, last;  
//我们定义一个函数来初始化 SAM（创建一个只有初始状态的 SAM）。  
void sam\_init() {  
 st[0].len = 0;  
 st[0].link = -1;  
 sz++;  
 last = 0;  
}  
//最终我们给出主函数的实现：给当前行末增加一个字符，对应地在之前的基础上建造自动机。  
void sam\_extend(char c) {  
 int cur = sz++;  
 st[cur].len = st[last].len + 1;  
 int p = last;  
 while (p != -1 && !st[p].next.count(c)) {  
 st[p].next[c] = cur;  
 p = st[p].link;  
 }  
 if (p == -1) {  
 st[cur].link = 0;  
 }  
 else {  
 int q = st[p].next[c];  
 if (st[p].len + 1 == st[q].len) {  
 st[cur].link = q;  
 }  
 else {  
 int clone = sz++;  
 st[clone].len = st[p].len + 1;  
 st[clone].next = st[q].next;  
 st[clone].link = st[q].link;  
 while (p != -1 && st[p].next[c] == q) {  
 st[p].next[c] = clone;  
 p = st[p].link;  
 }  
 st[q].link = st[cur].link = clone;  
 }  
 }  
 last = cur;  
}

### 回文树

const LL M = 3e5 + 10;  
  
struct PalindromicTreeNode  
{  
 LL son[26];  
 LL suffix;  
 LL curlen;  
 LL cnt;  
 char c;  
} PTN[M];  
// char orginalString[M];  
LL PTNSIZE = 1; // SIZE - 1 actually  
LL last = 0;  
  
void \_\_init\_\_()  
{  
 PTN[0].curlen = 0;  
 PTN[0].suffix = 1;  
 PTN[0].c = '^';  
 PTN[1].c = '#';  
 PTN[1].curlen = -1;  
}  
  
LL \_\_find\_\_(LL pattern)  
{  
 while (PTN[PTNSIZE - PTN[pattern].curlen - 1].c != PTN[PTNSIZE].c)  
 pattern = PTN[pattern].suffix;  
 return pattern;  
}  
  
void \_\_add\_\_(char element)  
{  
 PTNSIZE++;  
 PTN[PTNSIZE].c = element;  
 LL offset = element - 97;  
 LL cur = \_\_find\_\_(last); // 可以加回文的点  
 if (PTN[cur].son[offset] == 0) // 没前向边这条边  
 {  
 PTN[PTNSIZE].suffix = PTN[\_\_find\_\_(PTN[cur].suffix)].son[offset]; // 正在插入的字母的后缀边不可能是cur，所以要用chk往下找合法的  
 PTN[cur].son[offset] = PTNSIZE; // 这才是加前向边  
 PTN[PTNSIZE].curlen = PTN[cur].curlen + 2;  
 }  
 last = PTN[cur].son[offset]; // 加过边以后last就是PTNSIZE  
 PTN[last].cnt++;  
}  
  
LL \_\_count\_\_()  
{  
 LL re = 0;  
 for (LL i = PTNSIZE; i >= 0; i--)  
 {  
 PTN[PTN[i].suffix].cnt += PTN[i].cnt; // 后缀边连接的节点走过次数要加上前面更高级的回文串节点走过次数  
 re = max(re, PTN[i].curlen); // 统计最长回文串长度  
 }  
 return re;  
}  
  
int main()  
{  
 ios::sync\_with\_stdio(false);  
 cin.tie(0);  
 cout.tie(0);  
 \_\_init\_\_();  
 string ss;  
 cin >> ss;  
 for (auto s : ss)  
 \_\_add\_\_(s);  
 \_\_count\_\_();  
 LL ans = 0;  
 for (auto i = 2; i <= PTNSIZE; i++)  
 {  
 ans = max(ans, PTN[i].cnt \* PTN[i].curlen); // 最长回文子串  
 }  
 cout << ans << '\n';  
 return 0;  
}

### 附：快速IO

// char buf[1<<23],\*p1=buf,\*p2=buf,obuf[1<<23],\*O=obuf; // 或者用fread更难调的快读  
// #define getchar() (p1==p2&&(p2=(p1=buf)+fread(buf,1,1<<21,stdin),p1==p2)?EOF:\*p1++)  
  
template <class T>  
void print(T x)  
{  
 if (x < 0)  
 {  
 x = -x;  
 putchar('-');  
 // \*O++ = '-';  
 }  
 if (x > 9)  
 print(x / 10);  
 putchar(x % 10 + '0');  
 // \*O++ = x%10 + '0'  
}  
// fwrite(obuf,O-obuf,1,stdout);  
  
template <class T>  
inline void qr(T &n)  
{  
 n = 0;  
 register char c = getchar();  
 LL sgn = 1;  
  
 while (c > '9' || c < '0')  
 {  
 if (c == '-')  
 sgn = -1;  
 c = getchar();  
 }  
  
 while (c <= '9' && c >= '0')  
 {  
 n = (n << 3) + (n << 1) + (c ^ 0x30);  
 c = getchar();  
 }  
  
 n \*= sgn;  
}  
  
inline char qrc()  
{  
 register char c = getchar();  
 while (c < 'a' || c > 'z')  
 c = getchar();  
 return c;  
}

### 附：没用的优化

#pragma GCC optimize(3)  
#pragma GCC target("avx")

## 图论

### Floyd

for (k = 1; k <= n; k++) {  
 for (i = 1; i <= n; i++) {  
 for (j = 1; j <= n; j++) {  
 f[i][j] = min(f[i][j], f[i][k] + f[k][j]);  
 }  
 }  
}

### SPFA（它已经死了 //vector？

#include <bits/stdc++.h>  
using namespace std;  
const int maxN = 200010;  
struct Edge  
{  
 int to, next, w;  
} e[maxN];  
int n, m, cnt, p[maxN], Dis[maxN];  
int In[maxN];  
bool visited[maxN];  
void Add\_Edge(const int x, const int y, const int z)  
{  
 e[++cnt].to = y;  
 e[cnt].next = p[x];  
 e[cnt].w = z;  
 p[x] = cnt;  
 return;  
}  
bool Spfa(const int S)  
{  
 int i, t, temp;  
 queue<int> Q;  
 memset(visited, 0, sizeof(visited));  
 memset(Dis, 0x3f, sizeof(Dis));  
 memset(In, 0, sizeof(In));  
 Q.push(S);  
 visited[S] = true;  
 Dis[S] = 0;  
 while (!Q.empty())  
 {  
 t = Q.front();  
 Q.pop();  
 visited[t] = false;  
 for (i = p[t]; i; i = e[i].next)  
 {  
 temp = e[i].to;  
 if (Dis[temp] > Dis[t] + e[i].w)  
 {  
 Dis[temp] = Dis[t] + e[i].w;  
 if (!visited[temp])  
 {  
 Q.push(temp);  
 visited[temp] = true;  
 if (++In[temp] > n)  
 return false;  
 }  
 }  
 }  
 }  
 return true;  
}  
int main()  
{  
 int S, T;  
 scanf("%d%d%d%d", &n, &m, &S, &T);  
 for (int i = 1; i <= m; ++i)  
 {  
 int x, y, \_;  
 scanf("%d%d%d", &x, &y, &\_);  
 Add\_Edge(x, y, \_);  
 }  
 if (!Spfa(S))  
 printf("FAIL!\n");  
 else  
 printf("%d\n", Dis[T]);  
 return 0;  
}

### Dijkstra //vector？

#include <cstdio>  
#include <queue>  
#include <vector>  
#define MAXN 200010  
#define INF 0x3fffffff  
using namespace std;  
struct edge {  
 int v, w;  
 edge(int v, int w) :v(v), w(w) {}  
};  
vector <edge> mp[MAXN];  
int dis[MAXN];  
bool vis[MAXN];  
int n, m, s;  
struct node {  
 int v, dis;  
 node(int v, int dis) :v(v), dis(dis) {}  
 const bool operator < (const node& a) const {  
 return a.dis < dis;  
 }  
};  
priority\_queue <node> q;  
  
int read() {  
 char ch; int s = 0;  
 ch = getchar();  
 while (ch < '0' || ch>'9') ch = getchar();  
 while (ch >= '0' && ch <= '9') s = s \* 10 + ch - '0', ch = getchar();  
 return s;  
}  
  
void dj() {  
 for (register int i = 1; i <= n; i++)  
 dis[i] = INF;  
 dis[s] = 0;  
 q.push(node(s, 0));  
 while (!q.empty()) {  
 node cur = q.top();  
 q.pop();  
 if (vis[cur.v]) continue;  
 vis[cur.v] = 1;  
 for (register int i = 0; i < mp[cur.v].size(); i++) {  
 edge to = mp[cur.v][i];  
 if (vis[to.v]) continue;  
 if (dis[to.v] > to.w + dis[cur.v]) {  
 dis[to.v] = to.w + dis[cur.v], q.push(node(to.v, dis[to.v]));  
 }  
 }  
 }  
 for (register int i = 1; i <= n; i++)  
 printf("%d ", dis[i]);  
}  
int main()  
{  
 n = read(), m = read(), s = read();  
 for (register int i = 1; i <= m; i++) {  
 int u, v, w;  
 u = read(), v = read(), w = read();  
 mp[u].push\_back(edge(v, w));  
 }  
 dj();  
 return 0;  
}

### Tarjan

#### 强连通分量

const int MAXN = 1e5 + 10;  
struct Edge{  
 int to, next, dis;  
}edge[MAXN << 1];  
int head[MAXN], cnt, ans;   
bool inStack[MAXN]; //判断是否在栈中   
//dfn 第一次访问到该节点的时间（时间戳）  
//low[i] low[i]能从哪个点（最早时间戳）到达这个点的。   
int dfn[MAXN], low[MAXN], tot;   
stack<int> stc;  
void add\_edge(int u, int v, int dis) {  
 edge[++cnt].to = v;  
 edge[cnt].next = head[u];  
 head[u] = cnt;  
}  
void Tarjan(int x) {  
 dfn[x] = low[x] = ++tot;  
 stc.push(x);  
 inStack[x] = 1;  
 for(int i = head[x]; i; i = edge[i].next) {  
 int to = edge[i].to;  
 if ( !dfn[to] ) {  
 Tarjan(to);  
 low[x] = min(low[x], low[to]);  
 } else if (inStack[to]){  
 low[x] = min(low[x], dfn[to]);  
 }  
 }  
 //cout << x << " " << low[x] << " " << dfn[x] << endl;  
 if(low[x] == dfn[x]) { //发现是整个强连通分量子树里 的最小根。  
 //int cnt = 0;  
 ans++; //强连通分量计数器   
 while(1) {  
 int top = stc.top();  
 stc.pop();  
 //cnt ++;  
 inStack[top] = 0;  
 //cout << top << " "; 每个强连通分量内的点   
 if(top == x) break;  
 }   
 }  
}  
void init() {  
 cnt = 1;  
 tot = 0;  
 ans = 0;  
 memset(inStack, 0, sizeof(inStack));  
 memset(head, 0, sizeof(head));  
 memset(dfn, 0, sizeof(dfn));  
 memset(low, 0, sizeof(low));  
 while(!stc.empty()) stc.pop();  
}  
int main () {  
 std::ios::sync\_with\_stdio(false);  
 cin.tie(0);  
 int n, m;  
 while(cin >> n >> m && (n || m)){  
 init();  
 int x, y;  
 for(int i = 1; i <= m; ++i) {  
 cin >> x >> y;  
 add\_edge(x, y, 0); //有向图求强连通   
 }   
 for(int i = 1; i <= n; ++i) {  
 if( !dfn[i] )  
 Tarjan(i);  
 }   
 }  
 return 0;  
}

#### 缩点

const int MAXN = 5e3 + 20;  
const int MAXM = 1e6 + 10;   
int head[MAXN], cnt, tot, dfn[MAXN], low[MAXN], color[MAXN], col;  
bool vis[MAXN];  
int degree[MAXN];  
stack<int> stc;  
int n, m;  
struct Edge {  
 int to, next, dis;   
}edge[MAXM << 1];  
void add\_edge(int u, int v, int dis) {  
 edge[++cnt].to = v;  
 edge[cnt].next = head[u];  
 head[u] = cnt;   
}  
void Tarjan(int x) {  
 vis[x] = 1;  
 dfn[x] = low[x] = ++tot;  
 stc.push(x);  
 for(int i = head[x]; i; i = edge[i].next) {  
 int to = edge[i].to;  
 if (!dfn[to]) {  
 Tarjan(to);  
 low[x] = min(low[x], low[to]);  
 } else if( vis[to] ) {  
 low[x] = min(low[x], dfn[to]);  
 }  
 }  
 if(dfn[x] == low[x]) {  
 col ++;  
 while(true) {  
 int top = stc.top();  
 stc.pop();  
 color[top] = col; //颜色相同的点缩点   
 vis[top] = 0;  
 // cout << top << " ";  
 if(top == x) break;   
 }  
 //cout << endl;  
 }  
}  
void solve(){  
 for(int i = 1; i <= n; ++i) {  
 if(!dfn[i])  
 Tarjan(i);  
 }  
   
 for(int x = 1; x <= n; ++x) { //遍历 n个节点   
 for(int i = head[x]; i; i = edge[i].next) { //缩点后 每个点的出度   
 int to = edge[i].to;  
 if(color[x] != color[to]) {  
 degree[color[x]] ++;  
 }  
 }  
 }  
}  
void init () {  
 cnt = 1;  
 tot = 0;  
 col = 0;  
 memset(vis, 0, sizeof(vis));  
 memset(head, 0, sizeof(head));  
 memset(dfn, 0, sizeof(dfn));  
 memset(low, 0, sizeof(low));  
 memset(degree, 0, sizeof(degree));  
 memset(color, 0, sizeof(color));  
 while(!stc.empty()) stc.pop();  
}  
int main () {  
 std::ios::sync\_with\_stdio(false);  
 cin.tie(0);  
 while(cin >> n && n) {  
 cin >> m;  
 init();  
 int x, y;  
 for(int i = 1; i <= m; ++i) {  
 cin >> x >> y;  
 add\_edge(x, y, 0);  
 }  
 solve();  
 }   
 return 0;  
}

#### 割点

#include <bits/stdc++.h>  
using namespace std;  
int n, m; // n：点数 m：边数  
int num[100001], low[100001], inde, res;  
// num：记录每个点的时间戳  
// low：能不经过父亲到达最小的编号，inde：时间戳，res：答案数量  
bool vis[100001], flag[100001]; // flag: 答案 vis：标记是否重复  
vector<int> edge[100001]; // 存图用的  
void Tarjan(int u, int father) { // u 当前点的编号，father 自己爸爸的编号  
 vis[u] = true; // 标记  
 low[u] = num[u] = ++inde; // 打上时间戳  
 int child = 0; // 每一个点儿子数量  
 for (auto v : edge[u]) { // 访问这个点的所有邻居 （C++11）  
 if (!vis[v]) {  
 child++; // 多了一个儿子  
 Tarjan(v, u); // 继续  
 low[u] = min(low[u], low[v]); // 更新能到的最小节点编号  
 if (father != u && low[v] >= num[u] &&!flag[u]) // 主要代码  
 // 如果不是自己，且不通过父亲返回的最小点符合割点的要求，并且没有被标记过  
 // 要求即为：删了父亲连不上去了，即为最多连到父亲  
 {  
 flag[u] = true;  
 res++; // 记录答案  
 }  
 }  
 else if (v != father)  
 low[u] =  
 min(low[u], num[v]); // 如果这个点不是自己，更新能到的最小节点编号  
 }  
 if (father == u && child >= 2 &&  
 !flag[u]) { // 主要代码，自己的话需要 2 个儿子才可以  
 flag[u] = true;  
 res++; // 记录答案  
 }  
}  
int main() {  
 cin >> n >> m; // 读入数据  
 for (int i = 1; i <= m; i++) { // 注意点是从 1 开始的  
 int x, y;  
 cin >> x >> y;  
 edge[x].push\_back(y);  
 edge[y].push\_back(x);  
 } // 使用 vector 存图  
 for (int i = 1; i <= n; i++) // 因为 Tarjan 图不一定连通  
 if (!vis[i]) {  
 inde = 0; // 时间戳初始为 0  
 Tarjan(i, i); // 从第 i 个点开始，父亲为自己  
 }  
 cout << res << endl;  
 for (int i = 1; i <= n; i++)  
 if (flag[i]) cout << i << " "; // 输出结果  
 return 0;  
}

#### 点双连通分量

#include<bits/stdc++.h>  
using namespace std;  
typedef long long ll;  
const ll mod=998244353;  
const int maxn=1e6+50;  
const ll inf=0x3f3f3f3f3f3f3f3fLL;  
   
struct Edge{  
 int u,v;  
};  
///割顶 bccno 无意义  
int pre[maxn],iscut[maxn],bccno[maxn],dfs\_clock,bcc\_cut;  
vector<int>G[maxn],bcc[maxn];  
stack<Edge>S;  
int dfs(int u,int fa){  
 int lowu = pre[u] = ++dfs\_clock;  
 int child = 0;  
 for(int i = 0; i < G[u].size(); i++){  
 int v =G[u][i];  
 Edge e = (Edge){u,v};  
 if(!pre[v]){ ///没有访问过  
 S.push(e);  
 child++;  
 int lowv = dfs(v, u);  
 lowu=min(lowu, lowv); ///用后代更新  
 if(lowv >= pre[u]){  
 iscut[u]=true;  
 bcc\_cut++;bcc[bcc\_cut].clear(); ///注意 bcc从1开始  
 for(;;){  
 Edge x=S.top();S.pop();  
 if(bccno[x.u] != bcc\_cut){bcc[bcc\_cut].push\_back(x.u);bccno[x.u]=bcc\_cut;}  
 if(bccno[x.v] != bcc\_cut){bcc[bcc\_cut].push\_back(x.v);bccno[x.v]=bcc\_cut;}  
 if(x.u==u&&x.v==v)break;  
 }  
 }  
 }  
 else if(pre[v] < pre[u] && v !=fa){  
 S.push(e);  
 lowu = min(lowu,pre[v]);  
 }  
 }  
 if(fa < 0 && child == 1) iscut[u] = 0;  
 return lowu;  
}  
void find\_bcc(int n){  
 memset(pre, 0, sizeof(pre));  
 memset(iscut, 0, sizeof(iscut));  
 memset(bccno, 0, sizeof(bccno));  
 dfs\_clock = bcc\_cut = 0;  
 for(int i = 0; i < n;i++)  
 if(!pre[i])dfs(i,-1);  
}

#### 边双连通分量

去除所有桥dfs即可

#### 桥：

int low[MAXN], dfn[MAXN], iscut[MAXN], dfs\_clock;  
bool isbridge[MAXN];  
vector<int> G[MAXN];  
int cnt\_bridge;  
int father[MAXN];  
void tarjan(int u, int fa) {  
 father[u] = fa;  
 low[u] = dfn[u] = ++dfs\_clock;  
 for (int i = 0; i < G[u].size(); i++) {  
 int v = G[u][i];  
 if (!dfn[v]) {  
 tarjan(v, u);  
 low[u] = min(low[u], low[v]);  
 if (low[v] > dfn[u]) {//主要  
 isbridge[v] = true;  
 ++cnt\_bridge;  
 }  
 }  
 else if (dfn[v] < dfn[u] && v != fa) {  
 low[u] = min(low[u], dfn[v]);  
 }  
 }  
}

### 2-SAT

#include<bits/stdc++.h>  
using namespace std;  
#define int long long  
const int maxn=1e6+5;  
int a[maxn<<1];  
vector<int>g[maxn<<1];  
int tot;  
int dfn[maxn<<1],low[maxn<<1];  
stack<int>sta;  
int insta[maxn<<1];  
int scccnt;  
int color[maxn<<1];  
int n,m;  
void tarjan(int u)  
{  
 dfn[u]=low[u]=++tot;  
 sta.push(u);  
 insta[u]=1;  
 for(auto v:g[u])  
 {  
 if(!dfn[v])  
 {  
 tarjan(v);  
 low[u]=min(low[u],low[v]);  
 }  
 else if(insta[v])  
 {  
 low[u]=min(low[u],dfn[v]);  
 }  
 }  
 if(dfn[u]==low[u])  
 {  
 ++scccnt;  
 do{  
 color[u]=scccnt;  
 u=sta.top();sta.pop();  
 insta[u]=0;  
 }while(low[u]!=dfn[u]);  
 }  
   
}  
signed main()  
{  
 ios::sync\_with\_stdio(false);  
   
 cin>>n>>m;  
 for(int i=1;i<=m;i++)  
 {  
 int a,va,b,vb;//a为va或者b为vb  
 cin>>a>>va>>b>>vb;  
 if(va&vb)  
 {  
 g[a+n].push\_back(b);  
 g[b+n].push\_back(a);  
 }  
 else if(!va&vb)  
 {  
 g[a].push\_back(b);  
 g[b+n].push\_back(a+n);  
 }  
 else if(va&!vb)  
 {  
 g[a+m].push\_back(b+n);  
 g[b].push\_back(a);  
 }  
 else if(!va&!vb)  
 {  
 g[a].push\_back(b+n);  
 g[b].push\_back(a+n);  
 }  
 }  
 /\*for (int i = 0; i < m; ++i) {  
 int a = read(), va = read(), b = read(), vb = read();  
 g[a + n \* (va & 1)].push\_back(b + n \* (vb ^ 1));  
 g[b + n \* (vb & 1)].push\_back(a + n \* (va ^ 1));\*/  
 for(int i=1;i<=(n<<1);i++)  
 {  
 if(!dfn[i])  
 tarjan(i);  
 }  
 for(int i=1;i<=n;i++)  
 {  
 if(color[i]==color[i+n])  
 {  
 cout<<"IMPOSSIBLE\n";  
 return 0;  
 }  
 }  
 cout<<"POSSIBLE\n";  
 for(int i=1;i<=n;i++)  
 {  
 int tmp=(color[i]<color[i+n])?1:0;  
 cout<<tmp<<" ";  
 }  
}

### 找环

void tarjan(int u) {  
 low[u] = dfn[u] = ++dfsClock;  
 stk.push(u); ins[u] = true;  
 for (const auto &v : g[u]) {  
 if (!dfn[v]) tarjan(v), low[u] = std::min(low[u], low[v]);  
 else if (ins[v]) low[u] = std::min(low[u], dfn[v]);  
 }  
 if (low[u] == dfn[u]) {  
 ++sccCnt;  
 do {  
 color[u] = sccCnt;  
 u = stk.top(); stk.pop(); ins[u] = false;  
 } while (low[u] != dfn[u]);  
 }  
}  
// 笔者使用了 Tarjan 找环，得到的 color[x] 是 x 所在的 scc 的拓扑逆序。  
for (int i = 1; i <= (n << 1); ++i) if (!dfn[i]) tarjan(i);  
for (int i = 1; i <= n; ++i)  
 if (color[i] == color[i + n]) { // x 与 -x 在同一强连通分量内，一定无解  
 puts("IMPOSSIBLE");  
 exit(0);  
 }  
puts("POSSIBLE");  
for (int i = 1; i <= n; ++i)  
 print((color[i] < color[i + n])), putchar(' '); // 如果不使用 Tarjan 找环，请改成大于号

### 网络流

#### 最大流-Dinic

LL distant[M]; // 用于分层图层次表示  
LL current[M]; // 当前弧优化  
LL n, m, src, dst;  
  
inline LL bfs()  
{  
 for (auto i = 1; i <= n; i++)  
 distant[i] = INF;  
 queue<LL> Q;  
 Q.push(src);  
 distant[src] = 0;  
 current[src] = hds[src];  
 while (!Q.empty())  
 {  
 auto x = Q.front();  
 Q.pop();  
 for (auto i = hds[x]; ~i; i = E[i].next)  
 {  
 auto v = E[i].to;  
 if (E[i].val > 0 and distant[v] == INF)  
 {  
 Q.push(v);  
 current[v] = hds[v];  
 distant[v] = distant[x] + 1;  
 if (v == dst)  
 return 1;  
 }  
 }  
 }  
 return 0;  
}  
  
inline LL dfs(LL x, LL Sum)  
{  
 if (x == dst)  
 return Sum;  
 LL res = 0;  
 for (auto i = current[x]; ~i and Sum; i = E[i].next) // 当前弧优化：改变枚举起点  
 {  
 current[x] = i;  
 auto v = E[i].to;  
 if (E[i].val > 0 and (distant[v] == distant[x] + 1))  
 {  
 LL remain = dfs(v, min(Sum, E[i].val)); // remain:当前最小剩余的流量  
 if (remain == 0)  
 distant[v] = INF; // 去掉增广完毕的点  
 E[i].val -= remain;  
 E[i ^ 1].val += remain;  
 res += remain; // 经过该点的所有流量和  
 Sum -= remain; // 该点剩余流量  
 }  
 }  
 return res;  
}  
  
LL Dinic()  
{  
 LL ans = 0;  
 while (bfs())  
 ans += dfs(src, INF);  
 return ans;  
}

#### 最大流-HLPP+黑魔法优化

/\* 除非卡时不然别用的预流推进桶排序优化黑魔法，用例如下  
signed main()  
{  
 qr(HLPP::n);  
 qr(HLPP::m);  
 qr(HLPP::src);  
 qr(HLPP::dst);  
 while (HLPP::m--)  
 {  
 LL t1, t2, t3;  
 qr(t1);  
 qr(t2);  
 qr(t3);  
 HLPP::add(t1, t2, t3);  
 }  
 cout << HLPP::hlpp(HLPP::n + 1, HLPP::src, HLPP::dst) << endl;  
 return 0;  
}  
\*/  
namespace HLPP  
{  
 const LL INF = 0x3f3f3f3f3f3f;  
 const LL MXn = 1203;  
 const LL maxm = 520010;  
  
 vector<LL> gap;  
 LL n, m, src, dst, now\_height, src\_height;  
  
 struct NODEINFO  
 {  
 LL height = MXn, traffic;  
 LL getIndex();  
 NODEINFO(LL h = 0) : height(h) {}  
 bool operator<(const NODEINFO &a) const { return height < a.height; }  
 } node[MXn];  
  
 LL NODEINFO::getIndex() { return this - node; }  
  
 struct EDGEINFO  
 {  
 LL to;  
 LL flow;  
 LL opposite;  
 EDGEINFO(LL a, LL b, LL c) : to(a), flow(b), opposite(c) {}  
 };  
 std::list<NODEINFO \*> dlist[MXn];  
 vector<std::list<NODEINFO \*>::iterator> iter;  
 vector<NODEINFO \*> list[MXn];  
 vector<EDGEINFO> edge[MXn];  
  
 inline void add(LL u, LL v, LL w = 0)  
 {  
 edge[u].push\_back(EDGEINFO(v, w, (LL)edge[v].size()));  
 edge[v].push\_back(EDGEINFO(u, 0, (LL)edge[u].size() - 1));  
 }  
  
 priority\_queue<NODEINFO> PQ;  
 inline bool prework\_bfs(NODEINFO &src, NODEINFO &dst, LL &n)  
 {  
 gap.assign(n, 0);  
 for (auto i = 0; i <= n; i++)  
 node[i].height = n;  
 dst.height = 0;  
 queue<NODEINFO \*> q;  
 q.push(&dst);  
 while (!q.empty())  
 {  
 NODEINFO &top = \*(q.front());  
 for (auto i : edge[&top - node])  
 {  
 if (node[i.to].height == n and edge[i.to][i.opposite].flow > 0)  
 {  
 gap[node[i.to].height = top.height + 1]++;  
 q.push(&node[i.to]);  
 }  
 }  
 q.pop();  
 }  
  
 return src.height == n;  
 }  
  
 inline void relabel(NODEINFO &src, NODEINFO &dst, LL &n)  
 {  
 prework\_bfs(src, dst, n);  
 for (auto i = 0; i <= n; i++)  
 list[i].clear(), dlist[i].clear();  
  
 for (auto i = 0; i <= n; i++)  
 {  
 NODEINFO &u = node[i];  
 if (u.height < n)  
 {  
 iter[i] = dlist[u.height].insert(dlist[u.height].begin(), &u);  
 if (u.traffic > 0)  
 list[u.height].push\_back(&u);  
 }  
 }  
 now\_height = src\_height = src.height;  
 }  
  
 inline bool push(NODEINFO &u, EDGEINFO &dst) // 从x到y尽可能推流，p是边的编号  
 {  
 NODEINFO &v = node[dst.to];  
 LL w = min(u.traffic, dst.flow);  
 dst.flow -= w;  
 edge[dst.to][dst.opposite].flow += w;  
 u.traffic -= w;  
 v.traffic += w;  
 if (v.traffic > 0 and v.traffic <= w)  
 list[v.height].push\_back(&v);  
 return u.traffic;  
 }  
  
 inline void push(LL n, LL ui)  
 {  
 auto new\_height = n;  
 NODEINFO &u = node[ui];  
 for (auto &i : edge[ui])  
 {  
 if (i.flow)  
 {  
 if (u.height == node[i.to].height + 1)  
 {  
 if (!push(u, i))  
 return;  
 }  
 else  
 new\_height = min(new\_height, node[i.to].height + 1); // 抬到正好流入下一个点  
 }  
 }  
 auto height = u.height;  
 if (gap[height] == 1)  
 {  
 for (auto i = height; i <= src\_height; i++)  
 {  
 for (auto it : dlist[i])  
 {  
 gap[(\*it).height]--;  
 (\*it).height = n;  
 }  
 dlist[i].clear();  
 }  
 src\_height = height - 1;  
 }  
 else  
 {  
 gap[height]--;  
 iter[ui] = dlist[height].erase(iter[ui]);  
 u.height = new\_height;  
 if (new\_height == n)  
 return;  
 gap[new\_height]++;  
 iter[ui] = dlist[new\_height].insert(dlist[new\_height].begin(), &u);  
 src\_height = max(src\_height, now\_height = new\_height);  
 list[new\_height].push\_back(&u);  
 }  
 }  
  
 inline LL hlpp(LL n, LL s, LL t)  
 {  
 if (s == t)  
 return 0;  
 now\_height = src\_height = 0;  
 NODEINFO &src = node[s];  
 NODEINFO &dst = node[t];  
 iter.resize(n);  
 for (auto i = 0; i < n; i++)  
 if (i != s)  
 iter[i] = dlist[node[i].height].insert(dlist[node[i].height].begin(), &node[i]);  
 gap.assign(n, 0);  
 gap[0] = n - 1;  
 src.traffic = INF;  
 dst.traffic = -INF; // 上负是为了防止来自汇点的推流  
 for (auto &i : edge[s])  
 push(src, i);  
 src.traffic = 0;  
 relabel(src, dst, n);  
 for (LL ui; now\_height >= 0;)  
 {  
 if (list[now\_height].empty())  
 {  
 now\_height--;  
 continue;  
 }  
 NODEINFO &u = \*(list[now\_height].back());  
 list[now\_height].pop\_back();  
 push(n, &u - node);  
 }  
 return dst.traffic+INF;  
 }  
}

#### 最小费用最大流（MCMF）-EK

const int maxn = 100010;  
  
bool vis[maxn];  
int n, m, s, t, dis[maxn], pre[maxn], last[maxn], flow[maxn], maxflow, mincost;  
//dis最小花费;pre每个点的前驱；last每个点的所连的前一条边；flow源点到此处的流量  
struct Edge  
{  
 int to, next, flow, dis; //flow流量 dis花费  
} edge[maxn \* 10];  
int head[maxn], num\_edge;  
queue<int> q;  
  
void add\_edge(int from, int to, int flow, int dis)  
{  
 edge[++num\_edge].next = head[from];  
 edge[num\_edge].to = to;  
 edge[num\_edge].flow = flow;  
 edge[num\_edge].dis = dis;  
 head[from] = num\_edge;  
}  
  
bool spfa(int s, int t)  
{  
 memset(dis, 0x7f, sizeof(dis));  
 memset(flow, 0x7f, sizeof(flow));  
 memset(vis, 0, sizeof(vis));  
 q.push(s);  
 vis[s] = 1;  
 dis[s] = 0;  
 pre[t] = -1;  
  
 while (!q.empty())  
 {  
 int now = q.front();  
 q.pop();  
 vis[now] = 0;  
 for (int i = head[now]; i != -1; i = edge[i].next)  
 {  
 if (edge[i].flow > 0 && dis[edge[i].to] > dis[now] + edge[i].dis) //正边  
 {  
 dis[edge[i].to] = dis[now] + edge[i].dis;  
 pre[edge[i].to] = now;  
 last[edge[i].to] = i;  
 flow[edge[i].to] = min(flow[now], edge[i].flow); //  
 if (!vis[edge[i].to])  
 {  
 vis[edge[i].to] = 1;  
 q.push(edge[i].to);  
 }  
 }  
 }  
 }  
 return pre[t] != -1;  
}  
  
void MCMF()  
{  
 while (spfa(s, t))  
 {  
 int now = t;  
 maxflow += flow[t];  
 mincost += flow[t] \* dis[t];  
 while (now != s)  
 { //从源点一直回溯到汇点  
 edge[last[now]].flow -= flow[t]; //flow和dis容易搞混  
 edge[last[now] ^ 1].flow += flow[t];  
 now = pre[now];  
 }  
 }  
}  
  
int main()  
{  
 int tt;  
 scanf("%d", &tt);  
 s = 1000;  
 t = 1001;  
 while (tt--)  
 {  
 memset(head, -1, sizeof(head));  
 memset(pre, 0, sizeof(head));  
 memset(last, 0, sizeof(head));  
 memset(pre, 0, sizeof(head));  
 num\_edge = -1; //初始化  
 maxflow = 0;  
 mincost = 0;  
 scanf("%d", &n);  
 for (auto i = 0; i < n; i++)  
 {  
 scanf("%d", &m);  
 add\_edge(s, m, 1, 0);  
 add\_edge(m, s, 0, 0);  
 for (auto j = 1; j <= 402; j++)  
 {  
 int cost = abs(m - j);  
 add\_edge(m, j + 500, 1, cost);  
 add\_edge(j + 500, m, 0, -cost);  
 }  
 }  
 for (auto j = 1; j <= 402; j++)  
 {  
 add\_edge(j + 500, t, 1, 0);  
 add\_edge(t, j + 500, 0, 0);  
 }  
  
 MCMF();  
 printf("%d\n", mincost);  
 }  
 return 0;  
}

### 树形DP求树的最小支配集,最小点覆盖,最大独立集

**一:最小支配集**

考虑最小支配集,每个点有两种状态,即属于支配集合或者不属于支配集合,其中不属于支配集合时此点还需要被覆盖,被覆盖也有两种状态,即被子节点覆盖或者被父节点覆盖.总结起来就是三种状态,现对这三种状态定义如下:

1):**dp[i] [0]**,表示点 i 属于支配集合,并且以点 i 为根的子树都被覆盖了的情况下支配集中所包含最少点的个数.

2):**dp[i] [1]**,表示点 i 不属于支配集合,且以 i 为根的子树都被覆盖,且 i 被其中不少于一个子节点覆盖的情况下支配集所包含最少点的个数.

3):**dp[i] [2]**,表示点 i 不属于支配集合,且以 i 为根的子树都被覆盖,且 i 没被子节点覆盖的情况下支配集中所包含最少点的个数.即 i 将被父节点覆盖.

**对于第一种状态**,dp[i] [0]含义为点 i 属于支配集合,那么依次取每个儿子节点三种状态中的最小值,再把取得的最小值全部加起来再加 1,就是dp[i] [0]的值了.即只要每个以 i 的儿子为根的子树都被覆盖,再加上当前点 i,所需要的最少的点的个数,DP转移方程如下:

**dp[i] [0] = 1 + ∑(u 取 i 的子节点)min(dp[u] [0], dp[u] [1], dp[u] [2])**

**对于第三种状态**,dp[i] [2]含义为点 i 不属于支配集合,且 i 被其父节点覆盖.那么说明点 i 和点 i 的儿子节点都不属于支配集合,所以点 i 的第三种状态之和其儿子节点的第二种状态有关,方程为:

**dp[i] [2] = ∑(u 取 i 的子节点)dp[u] [1]**

**对于第二种状态**,略有些复杂.首先如果点 i 没有子节点那么 dp[i] [1]应该初始化为 INF;否则为了保证它的每个以 i 的儿子为根的子树被覆盖,那么要取每个儿子节点的前两种状态的最小值之和,因为此时点 i 不属于支配集,不能支配其子节点,所以子节点必须已经被支配,与子节点的第三种状态无关.如果当前所选状态中每个儿子都没被选择进入支配集,即在每个儿子的前两种状态中,第一种状态都不是所需点最小的,那么为了满足第二种状态的定义(因为点 i 的第二种状态必须被其子节点覆盖,即其子节点必须有一个属于支配集,如果此时没有,就必须强制使一个子节点的状态为状态一),需要重新选择点 i 的一个儿子节点为第一种状态,这时取花费最少的一个点,即取min(dp[u] [0] - dp[u] [1])的儿子节点 u,强制取其第一种状态,其他的儿子节点取第二种状态,DP转移方程为:

**if(i 没有子节点) dp[i] [1] = INF**

**else dp[i] [1] = ∑(u 取 i 的子节点)min(dp[u] [0], dp[u] [1]) + inc**

**其中对于 inc 有:**

**if(上面式子中的 ∑(u 取 i 的子节点)min(dp[u] [0], dp[u] [1]) 中包含某个 dp[u] [0], 即存在一个所选的最小值为状态一的儿子节点) inc = 0**

**else inc = min(dp[u] [0] - dp[u] [1]) (其中 u 取点 i 的儿子节点)**

**代码:**

1 void DP(int u, int p) {// p 为 u 的父节点  
 2 dp[u][2] = 0;  
 3 dp[u][0] = 1;  
 4 bool s = false;  
 5 int sum = 0, inc = INF;  
 6 for(int k = head[u]; k != -1; k = edge[k].next) {  
 7 int to = edge[k].to;  
 8 if(to == p) continue;  
 9 DP(to, u);  
10 dp[u][0] += min(dp[to][0], min(dp[to][1], dp[to][2]));  
11 if(dp[to][0] <= dp[to][1]) {  
12 sum += dp[to][0];  
13 s = true;  
14 }  
15 else {  
16 sum += dp[to][1];  
17 inc = min(inc, dp[to][0] - dp[to][1]);  
18 }  
19 if(dp[to][1] != INF && dp[u][2] != INF) dp[u][2] += dp[to][1];  
20 else dp[u][2] = INF;  
21 }  
22 if(inc == INF && !s) dp[u][1] = INF;  
23 else {  
24 dp[u][1] = sum;  
25 if(!s) dp[u][1] += inc;  
26 }  
27 }

**二:最小点覆盖**

对于最小点覆盖,每个点只有两种状态,即属于点覆盖或者不属于点覆盖:

1):**dp[i] [0]**表示点 i 属于点覆盖,并且以点 i 为根的子树中所连接的边都被覆盖的情况下点覆盖集中所包含最少点的个数.

2):**dp[i] [1]**表示点 i 不属于点覆盖,且以点 i 为根的子树中所连接的边都被覆盖的情况下点覆盖集中所包含最少点的个数.

**对于第一种状态**dp[i] [0],等于每个儿子节点的两种状态的最小值之和加 1,DP转移方程如下:

**dp[i] [0] = 1 + ∑(u 取 i 的子节点)min(dp[u] [0], dp[u] [1])**

**对于第二种状态**dp[i] [1],要求所有与 i 连接的边都被覆盖,但是点 i 不属于点覆盖,那么点 i 的所有子节点就必须属于点覆盖,即对于点 i 的第二种状态与所有子节点的第一种状态有关,在数值上等于所有子节点第一种状态的和.DP转移方程如下:

**dp[i] [1] = ∑(u 取 i 的子节点)dp[u] [0]**

**代码：**

1 void DP(int u, int p) {// p 为 u 的父节点  
 2 dp[u][0] = 1;  
 3 dp[u][1] = 0;  
 4 for(int k = head[u]; k != -1; k = edge[k].next) {  
 5 int to = edge[k].to;  
 6 if(to == p) continue;  
 7 DP(to, u);  
 8 dp[u][0] += min(dp[to][0], dp[to][1]);  
 9 dp[u][1] += dp[to][0];  
10 }  
11 }

**三:最大独立集**

对于最大独立集,每个点也只有两种状态,即属于点 i 属于独立集或者不属于独立集两种情况:

1):**dp[i] [0]**表示点 i 属于独立集的情况下,最大独立集中点的个数.

2):**dp[i] [1]**表示点 i 不属于独立集的情况下.最大独立集中点的个数.

**对于第一种状态**dp[i] [0],由于 i 点属于独立集,所以它的子节点都不能属于独立集,所以对于点 i 的第一种状态,只和它的子节点的第二种状态有关.等于其所有子节点的第二种状态的值加 1,DP转移方程如下:

**dp[i] [0] = 1 + ∑(u 取 i 的子节点) dp[u] [1]**

**对于第二种状态**dp[i] [1],由于点 i 不属于独立集,所以子节点可以属于独立解,也可以不属于独立集,所取得子节点状态应该为数值较大的那个状态,DP转移方程:

**dp[i] [1] = ∑(u 取 i 的子节点)max(dp[u] [0], dp[u] [1])**

**代码:**

void DP(int u, int p) {// p 为 u 的父节点  
 dp[u][0] = 1;  
 dp[u][1] = 0;  
 for(int k = head[u]; k != -1; k = edge[k].next) {  
 int to = edge[k].to;  
 if(to == p) continue;  
 DP(to, u);  
 dp[u][0] += dp[to][1];  
 dp[u][1] += max(dp[to][0], dp[to][1]);  
 }  
}

## 数据结构

### 并查集

void init(int n)  
{  
 for(int i=0;i<n;i++)  
 {  
 parent[i]=i;  
 rank[i]=0; // 初始树的高度为0  
 }  
}  
// 合并x和y所属的集合  
void unite(int x,int y)  
{  
 x=find(x);  
 y=find(y);  
 if(x==y) return ;  
 if(rank[x]<rank[y])  
 parent[x]=y; // 合并是从rank小的向rank大的连边  
 else  
 {  
 parent[y]=x;  
 if(rank[x]==rank[y]) rank[x]++;  
 }  
}  
int find(int x) //查找x元素所在的集合,回溯时压缩路径  
{  
 if (x != parent[x])  
 {  
 parent[x] = find(parent[x]); //回溯时的压缩路径  
 } //从x结点搜索到祖先结点所经过的结点都指向该祖先结点  
 return parent[x];  
}

### 树状数组（动态分配//改成能区间加的区间查的？

template <typename T>  
inline T lowbit(T x){return x&-x;}  
  
struct TreeArray  
{  
 LL \*content;  
 LL len;  
 TreeArray(LL len)  
 {  
 this->content = new LL[len];  
 memset(this->content, 0, sizeof(LL) \* len);  
 this->len = len;  
 }  
 LL getsum(LL pos)  
 {  
 LL ans = 0;  
 while (pos > 0)  
 {  
 ans += this->content[pos];  
 pos -= lowbit(pos);  
 }  
 return ans;  
 }  
 void update(LL pos, LL x)  
 {  
 while (pos < this->len)  
 {  
 this->content[pos] += x;  
 pos += lowbit(pos);  
 }  
 }  
};

### 树状数组（区间查询区间修改）

#include<iostream>  
#include<stdio.h>  
#include<string.h>  
#include<algorithm>  
using namespace std;  
const int maxn=1e5+10;  
int sum1[maxn];  
int sum2[maxn];  
int a[maxn];  
int n,m;  
int lowbit(int x){  
 return x&(-x);  
}  
void update(int x,int w){//更新效果：把x位置后面所有的数的值+w  
 for (int i=x;i<=n;i+=lowbit(i)){  
 sum1[i]+=w;//维护前缀和c[i]  
 sum2[i]+=w\*(x-1);//维护前缀和c[i]\*(n-1)  
 }  
}  
void range\_update(int l,int r,int val)//更新效果：把l位置到r位置所有的数的值+w  
{  
 update(l,val);  
 update(r+1,-val);  
}  
int sum(int x){//求1-x的和  
 int ans=0;  
 for (int i=x;i>0;i-=lowbit(i)){  
 ans+=x\*sum1[i]-sum2[i];  
 }  
 return ans;  
}  
int range\_ask(int l,int r){//求l-r的和  
 return sum(r)-sum(l-1);  
}  
int main(){  
 while(~scanf("%d%d",&n,&m)){  
 for (int i=1;i<=n;i++){  
 scanf("%d",&a[i]);  
 update(i,a[i]-a[i-1]);//维护差分数组  
 }  
 }  
 return 0;  
}

### 线段树//待替换

struct segtreeModel  
{  
 LL val;  
 LL lazy\_tag;//Tag一定要本节点做完再打，否则后面的更新不了前面的  
 LL mul\_tag;  
 LL l\_range, r\_range;  
 segtreeModel \*l, \*r;  
} T[M << 2];  
  
void build(const LL &rg1, const LL &rg2, const LL ind) //从0开始的线段树()  
{  
 LL mid = (rg1 + rg2) >> 1;  
 T[ind].lazy\_tag = 0;  
 T[ind].mul\_tag = 1;  
 T[ind].l\_range = rg1;  
 T[ind].r\_range = rg2;  
  
 if (rg1 == rg2)  
 {  
 T[ind].val = Array[rg1];  
 return;  
 }  
 T[ind].l = &T[(ind << 1) + 1];  
 T[ind].r = &T[(ind << 1) + 2];  
 build(rg1, mid, (ind << 1) + 1);  
 build(mid + 1, rg2, (ind << 1) + 2);  
  
 T[ind].val = DSUM(T[ind].l->val, T[ind].r->val);  
 DMO(T[ind].val, mo);  
}  
  
void push\_down(segtreeModel \*ind)  
{  
 segtreeModel \*lson = ind->l;  
 segtreeModel \*rson = ind->r;  
 lson->val = (lson->val \* ind->mul\_tag + ind->lazy\_tag \* (lson->r\_range + 1 - lson->l\_range)) % mo;  
 rson->val = (rson->val \* ind->mul\_tag + ind->lazy\_tag \* (rson->r\_range + 1 - rson->l\_range)) % mo;  
  
 lson->mul\_tag \*= ind->mul\_tag;  
 DMO(lson->mul\_tag, mo);  
 rson->mul\_tag \*= ind->mul\_tag;  
 DMO(rson->mul\_tag, mo);  
  
 lson->lazy\_tag = (ind->lazy\_tag + lson->lazy\_tag \* ind->mul\_tag) % mo;  
 rson->lazy\_tag = (ind->lazy\_tag + rson->lazy\_tag \* ind->mul\_tag) % mo;  
  
 ind->lazy\_tag = 0;  
 ind->mul\_tag = 1;  
}  
  
void modify(const LL &rg1, const LL &rg2, segtreeModel \*ind, const LL v)  
{  
  
 if (rg1 <= ind->l\_range && ind->r\_range <= rg2) //魔改区间内  
 {  
  
 if (mulFlg)  
 {  
  
 ind->mul\_tag \*= v;  
 ind->val \*= v;  
 ind->lazy\_tag \*= v;  
 DMO(ind->lazy\_tag, mo);  
 DMO(ind->val, mo);  
 DMO(ind->mul\_tag, mo);  
 }  
 else  
 {  
 ind->val += (v \* (ind->r\_range - ind->l\_range + 1));  
 ind->lazy\_tag += v;  
 DMO(ind->lazy\_tag, mo);  
 DMO(ind->val, mo);  
 }  
 }  
 else //部分在魔改区间内  
 {  
 push\_down(ind);  
 //LL mid = (ind->l\_range + ind->r\_range) >> 1;  
 if (ind->l->r\_range >= rg1)  
 modify(rg1, rg2, ind->l, v);  
 if (ind->r->l\_range <= rg2)  
 modify(rg1, rg2, ind->r, v);  
 ind->val = DSUM(ind->l->val, ind->r->val) % mo;  
 }  
}  
  
LL query(const LL &rg1, const LL &rg2, segtreeModel \*ind)  
{  
 // if (ind->l\_range > rg2 || ind->r\_range < rg1)  
 // return 0;  
 if (rg1 <= ind->l\_range && ind->r\_range <= rg2)  
 {  
 return ind->val;  
 }  
 else  
 {  
 push\_down(ind);  
 //LL mid = (ind->l\_range + ind->r\_range) >> 1;  
 LL v1 = ind->l->r\_range >= rg1 ? query(rg1, rg2, ind->l) : 0;  
 LL v2 = ind->r->l\_range <= rg2 ? query(rg1, rg2, ind->r) : 0;  
 return DSUM(v1, v2) % mo;  
 }  
}

### 主席树（非工程版）

#include <algorithm>  
#include <cstdio>  
#include <cstring>  
using namespace std;  
const int maxn = 1e5; // 数据范围  
int tot, n, m;  
int sum[(maxn << 5) + 10], rt[maxn + 10], ls[(maxn << 5) + 10],  
 rs[(maxn << 5) + 10];  
int a[maxn + 10], ind[maxn + 10], len;  
inline int getid(const int &val) { // 离散化  
 return lower\_bound(ind + 1, ind + len + 1, val) - ind;  
}  
int build(int l, int r) { // 建树  
 int root = ++tot;  
 if (l == r) return root;  
 int mid = l + r >> 1;  
 ls[root] = build(l, mid);  
 rs[root] = build(mid + 1, r);  
 return root; // 返回该子树的根节点  
}  
int update(int k, int l, int r, int root) { // 插入操作  
 int dir = ++tot;  
 ls[dir] = ls[root], rs[dir] = rs[root], sum[dir] = sum[root] + 1;  
 if (l == r) return dir;  
 int mid = l + r >> 1;  
 if (k <= mid)  
 ls[dir] = update(k, l, mid, ls[dir]);  
 else  
 rs[dir] = update(k, mid + 1, r, rs[dir]);  
 return dir;  
}  
int query(int u, int v, int l, int r, int k) { // 查询操作  
 int mid = l + r >> 1,  
 x = sum[ls[v]] - sum[ls[u]]; // 通过区间减法得到左儿子的信息  
 if (l == r) return l;  
 if (k <= x) // 说明在左儿子中  
 return query(ls[u], ls[v], l, mid, k);  
 else // 说明在右儿子中  
 return query(rs[u], rs[v], mid + 1, r, k - x);  
}  
inline void init() {  
 scanf("%d%d", &n, &m);  
 for (int i = 1; i <= n; ++i) scanf("%d", a + i);  
 memcpy(ind, a, sizeof ind);  
 sort(ind + 1, ind + n + 1);  
 len = unique(ind + 1, ind + n + 1) - ind - 1;  
 rt[0] = build(1, len);  
 for (int i = 1; i <= n; ++i) rt[i] = update(getid(a[i]), 1, len, rt[i - 1]);  
}  
int l, r, k;  
inline void work() {  
 while (m--) {  
 scanf("%d%d%d", &l, &r, &k);  
 printf("%d\n", ind[query(rt[l - 1], rt[r], 1, len, k)]); // 回答询问  
 }  
}  
int main() {  
 init();  
 work();  
 return 0;  
}

### 主席树//待替换

LL Array[M], SORTED[M];  
LL COUNT = 0;  
  
struct segtreeModel  
{  
 LL val;  
 segtreeModel \*l, \*r;  
} T[M << 5];  
  
segtreeModel \*headers[M];  
  
inline segtreeModel \*build(const LL rg1, const LL rg2) //从0开始的主席树()  
{  
 LL tmp = COUNT++;  
 T[tmp].val = 0;  
 if (rg2 > rg1)  
 {  
 T[tmp].l = build(rg1, DMID(rg1, rg2));  
 T[tmp].r = build(DMID(rg1, rg2) + 1, rg2);  
 }  
 return &T[tmp];  
}  
  
inline segtreeModel \*modify(const LL rg1, const LL rg2,  
 const LL &x, const segtreeModel \*pre) //x:排在第几  
{  
 LL tmp = COUNT++;  
 T[tmp].l = pre->l;  
 T[tmp].r = pre->r; //拷贝上一个时间的状态  
 T[tmp].val = pre->val + 1;  
 if (rg2 > rg1)  
 {  
 if (x <= DMID(rg1, rg2))  
 T[tmp].l = modify(rg1, DMID(rg1, rg2), x, pre->l);  
 else  
 T[tmp].r = modify(DMID(rg1, rg2) + 1, rg2, x, pre->r);  
 }  
 return &T[tmp];  
}  
  
inline LL queryPosition(const LL rg1, const LL rg2,  
 segtreeModel \*u, segtreeModel \*v, LL rank)  
{  
 if (rg1 >= rg2)  
 return rg1;  
 LL x = v->l->val - u->l->val;  
 if (x >= rank) //左节点元素个数  
 return query(rg1, DMID(rg1, rg2), u->l, v->l, rank);  
 else  
 return query(DMID(rg1, rg2) + 1, rg2, u->r, v->r, rank - x);  
}  
  
inline LL querySum  
  
LL positions[M];  
LL father[M];  
  
int main()  
{  
 LL n, m, t, \_T, ans;  
 t = qr();  
 F(\_T, t, 1)  
 {  
 printf("Case #%lld:", \_T + 1);  
 n = qr();  
 m = qr();  
 COUNT = ans = 0;  
 mem(father, 0);  
 mem(positions, 0);  
 LL i;  
 F(i, n, 1)  
 {  
 Array[i] = qr();  
 // SORTED[i] = Array[i];  
 }  
 headers[0] = build(0, n - 1);  
 for (i = n - 1; i; i--)  
 {  
 if(!positions[Array[i]])  
 modify(headers[0],)  
 }  
 // std::sort(SORTED, SORTED + n);  
 LL len = std::unique(SORTED, SORTED + n) - SORTED;  
   
 F(i, n, 1)  
 { //lower\_bound:大于等于  
 LL t = std::lower\_bound(SORTED, SORTED + len, Array[i]) - SORTED;  
 headers[i + 1] = modify(0, len - 1, t, headers[i]);  
 }  
  
 while (m--)  
 {  
 LL o1 = qr();  
 LL o2 = qr();  
 LL o3 = qr(); //rank不能减，因为主席树内存的是元素个数，而且这个query查询的时间区间已经从1起排了  
 LL index = query(0, len - 1, headers[o1 - 1], headers[o2], o3);  
 printf("%lld\n", SORTED[index]);  
 }  
 }  
}

### 树链剖分//？

LL ARRAY[M << 1];  
LL CNT = 0;  
LL n;  
LL mod;  
  
struct NodeModel  
{  
 LL size, height, dfs\_order, value;  
 NodeModel \*son, \*fa, \*top;  
 vector<NodeModel \*> E;  
 void d1(NodeModel \*father)  
 {  
 this->size = 1;  
 this->son = NULL;  
 this->fa = father;  
 this->height = father->height + 1;  
 for (auto cur : this->E)  
 {  
 // auto dst = N[E[cur].to];  
 if (cur != father)  
 {  
 cur->d1(this);  
 this->size += cur->size;  
 if (this->son == NULL || cur->size > this->son->size)  
 this->son = cur;  
 }  
 }  
 }  
 void d2(NodeModel \*father, NodeModel \*k)  
 {  
 this->top = k;  
 this->dfs\_order = CNT;  
 ARRAY[CNT++] = this->value;  
 if (this->son != NULL)  
 this->son->d2(this, k);  
 for (auto i : this->E)  
 if (i != father && i != this->son)  
 i->d2(this, i);  
 }  
  
 void add\_edge(NodeModel \*dst)  
 {  
 this->E.push\_back(dst);  
 }  
} N[M << 2];  
  
LL NODECNT = 0;  
struct SegmentTreeNode  
{  
 SegmentTreeNode \*l, \*r;  
 LL value, lazy, max\_value;  
 LL query\_sum(LL rg1, LL rg2, LL operation\_l, LL operation\_r, LL add\_val)  
 {  
 if (operation\_l > operation\_r)  
 swap(operation\_l, operation\_r);  
 if (operation\_l <= rg1 && rg2 <= operation\_r)  
 return (this->value + (LL)(rg2 - rg1 + 1) \* (add\_val % mod)) % mod;  
 else  
 {  
 LL v1 = (DMID(rg1, rg2) >= operation\_l) ? this->l->query\_sum(rg1, DMID(rg1, rg2), operation\_l, operation\_r, (add\_val + this->lazy) % mod) : 0;  
 LL v2 = (DMID(rg1, rg2) + 1 <= operation\_r) ? this->r->query\_sum(DMID(rg1, rg2) + 1, rg2, operation\_l, operation\_r, (add\_val + this->lazy) % mod) : 0;  
 return ((v1 + v2) % mod + mod) % mod;  
 }  
 }  
 LL query\_max(LL rg1, LL rg2, LL operation\_l, LL operation\_r, LL add\_val)  
 {  
 if (operation\_l > operation\_r)  
 swap(operation\_l, operation\_r);  
 if (operation\_l <= rg1 && rg2 <= operation\_r)  
 return this->max\_value /\*+ add\_val\*/;  
 else  
 {  
 LL v1 = (DMID(rg1, rg2) >= operation\_l) ? this->l->query\_max(rg1, DMID(rg1, rg2), operation\_l, operation\_r, add\_val + this->lazy) : -INF;  
 LL v2 = (DMID(rg1, rg2) + 1 <= operation\_r) ? this->r->query\_max(DMID(rg1, rg2) + 1, rg2, operation\_l, operation\_r, add\_val + this->lazy) : -INF;  
 return max(v1, v2);  
 }  
 }  
 void modify(LL rg1, LL rg2, LL operation\_l, LL operation\_r, LL x) // 单点修改  
 {  
 if (operation\_l <= rg1 && rg2 <= operation\_r)  
 {  
 this->value += x \* (rg2 - rg1 + 1);  
 this->value %= mod;  
 this->lazy += x;  
 this->lazy %= mod;  
 return;  
 }  
 if (operation\_l <= DMID(rg1, rg2))  
 this->l->modify(rg1, DMID(rg1, rg2), operation\_l, operation\_r, x);  
 if (DMID(rg1, rg2) + 1 <= operation\_r)  
 this->r->modify(DMID(rg1, rg2) + 1, rg2, operation\_l, operation\_r, x);  
 this->value = this->l->value + this->r->value + this->lazy \* (rg2 - rg1 + 1);  
 this->value %= mod;  
 // this->max\_value = max(this->l->max\_value, this->r->max\_value);  
 }  
  
} T[M];  
SegmentTreeNode \*build(LL rg1, LL rg2)  
{  
 LL tmp = NODECNT++;  
 if (rg2 > rg1)  
 {  
 T[tmp].l = build(rg1, DMID(rg1, rg2));  
 T[tmp].r = build(DMID(rg1, rg2) + 1, rg2);  
 T[tmp].value = (T[tmp].l->value + T[tmp].r->value) % mod;  
 T[tmp].max\_value = max(T[tmp].l->max\_value, T[tmp].r->max\_value);  
 }  
 else  
 {  
 T[tmp].value = T[tmp].max\_value = ARRAY[rg1];  
 }  
 T[tmp].lazy = 0;  
 return &T[tmp];  
}  
  
LL query\_SUM(NodeModel \*u, NodeModel \*v)  
{  
 LL ans = 0;  
 while (u->top != v->top) // top相同应该是在一条链上  
 {  
 if (u->top->height < v->top->height)  
 swap(u, v);  
 ans += T[0].query\_sum(0, n - 1, u->top->dfs\_order, u->dfs\_order, 0);  
 ans %= mod;  
 u = u->top->fa;  
 }  
 if (v->height > u->height)  
 swap(u, v);  
 ans += T[0].query\_sum(0, n - 1, v->dfs\_order, u->dfs\_order, 0);  
 ans %= mod;  
 return (ans + mod) % mod;  
}  
  
void MODIFY(NodeModel \*u, NodeModel \*v, LL x)  
{  
 LL ans = 0;  
 while (u->top != v->top) // top相同应该是在一条链上  
 {  
 if (u->top->height < v->top->height)  
 swap(u, v);  
 T[0].modify(0, n - 1, u->top->dfs\_order, u->dfs\_order, x);  
 u = u->top->fa;  
 }  
 if (v->height > u->height)  
 swap(u, v);  
 T[0].modify(0, n - 1, v->dfs\_order, u->dfs\_order, x);  
}  
  
LL query\_MAX(NodeModel \*u, NodeModel \*v)  
{  
 LL ans = -INF;  
 while (u->top != v->top) // top相同应该是在一条链上  
 {  
 if (u->top->height < v->top->height)  
 swap(u, v);  
 ans = max(ans, T[0].query\_max(0, n - 1, u->top->dfs\_order, u->dfs\_order, 0));  
 u = u->top->fa;  
 }  
 if (v->height > u->height)  
 swap(u, v);  
 ans = max(ans, T[0].query\_max(0, n - 1, v->dfs\_order, u->dfs\_order, 0));  
 return ans;  
}  
char cmd[10];  
signed main()  
{  
 n = qr();  
 LL m = qr();  
 LL root = qr();  
 mod = qr();  
 CNT = NODECNT = 0;  
 mem(T, 0);  
 mem(ARRAY, 0);  
 mem(N, 0);  
  
 for (auto i = 1; i <= n; i++)  
 {  
 N[i].value = qr();  
 }  
 for (auto i = 1; i < n; i++)  
 {  
 LL src = qr();  
 LL dst = qr();  
 N[src].add\_edge(&N[dst]);  
 N[dst].add\_edge(&N[src]);  
 }  
 N[root].d1(&N[0]);  
 N[root].d2(&N[0], &N[root]);  
 build(0, n - 1);  
 LL o1, o2, o3, o4;  
 while (m--)  
 {  
 o1 = qr();  
 switch (o1)  
 {  
 case 1:  
 o2 = qr();  
 o3 = qr();  
 o4 = qr();  
 MODIFY(&N[o2], &N[o3], o4);  
 break;  
 case 2:  
 o2 = qr();  
 o3 = qr();  
 printf("%lld\n", query\_SUM(&N[o2], &N[o3]) % mod);  
 break;  
 case 3:  
 o2 = qr();  
 o3 = qr();  
 T[0].modify(0, n - 1, N[o2].dfs\_order, N[o2].dfs\_order + N[o2].size - 1, o3);  
 break;  
 default:  
 o2 = qr();  
 printf("%lld\n", T[0].query\_sum(0, n - 1, N[o2].dfs\_order, N[o2].dfs\_order + N[o2].size - 1, 0) % mod);  
 break;  
 }  
 }  
 return 0;  
}

### Splay

#include <bits/stdc++.h>  
using namespace std;  
const int N = 100005;  
int rt, tot, fa[N], ch[N][2], val[N], cnt[N], sz[N],mark[N];  
int n, m;  
struct Splay  
{  
 void maintain(int x) { sz[x] = sz[ch[x][0]] + sz[ch[x][1]] + cnt[x]; }  
 bool get(int x) { return x == ch[fa[x]][1]; }  
 void pushdown(int x)  
 {  
 if(mark[x])  
 {  
 mark[ch[x][0]]^=1;  
 mark[ch[x][1]]^=1;  
 mark[x]=0;  
 swap(ch[x][1],ch[x][0]);  
 }  
 }  
 void clear(int x)  
 {  
 ch[x][0] = ch[x][1] = fa[x] = val[x] = sz[x] = cnt[x] = 0;  
 }  
 void write(int u)  
 {  
 pushdown(u);  
 if(ch[u][0])  
 write(ch[u][0]);  
 if(val[u]>1&&val[u]<n+2)  
 cout<<val[u]-1<<" ";  
  
 if(ch[u][1])  
 write(ch[u][1]);  
  
 }  
 void rotate(int x)  
 {  
 int y = fa[x], z = fa[y], chk = get(x);  
 ch[y][chk] = ch[x][chk ^ 1];  
 if (ch[x][chk ^ 1])  
 fa[ch[x][chk ^ 1]] = y;  
 ch[x][chk ^ 1] = y;  
 fa[y] = x;  
 fa[x] = z;  
 if (z)  
 ch[z][y == ch[z][1]] = x;  
 maintain(x);  
 maintain(y);  
 }  
 inline void splay(int x,int goal)  
 {  
 while(fa[x]!=goal)  
 {  
 int y=fa[x];int z=fa[y];  
 if(z!=goal)  
 (ch[z][1]==y)^(ch[y][1]==x)?rotate(x):rotate(y);  
 rotate(x);  
 }  
 if(goal==0)  
 rt=x;  
 }  
 void ins(int k)  
 {  
 if (!rt)  
 {  
 val[++tot] = k;  
 cnt[tot]++;  
 rt = tot;  
 maintain(rt);  
 return;  
 }  
 int cur = rt, f = 0;  
 while (1)  
 {  
 if (val[cur] == k)  
 {  
 cnt[cur]++;  
 maintain(cur);  
 maintain(f);  
 splay(cur,0);  
 break;  
 }  
 f = cur;  
 cur = ch[cur][val[cur] < k];  
 if (!cur)  
 {  
 val[++tot] = k;  
 cnt[tot]++;  
 fa[tot] = f;  
 ch[f][val[f] < k] = tot;  
 maintain(tot);  
 maintain(f);  
 splay(tot,0);  
 break;  
 }  
 }  
 }  
 int kth(int k)  
 {  
 int cur = rt;  
 while (1)  
 {  
 pushdown(cur);  
 if (ch[cur][0] && k <= sz[ch[cur][0]])  
 {  
 cur = ch[cur][0];  
 }  
 else  
 {  
 k -= cnt[cur] + sz[ch[cur][0]];  
 if (k <= 0)  
 {  
 //splay(cur,0);  
 return val[cur];  
 }  
 cur = ch[cur][1];  
 }  
 }  
 }  
 void reverse(int l,int r)  
 {  
 l=kth(l);r=kth(r+2);  
 splay(l,0);  
 splay(r,l);  
 int tmp=ch[rt][1];  
 mark[ch[tmp][0]]^=1;  
 }  
} tree;  
  
int main()  
{  
 ios::sync\_with\_stdio(false);  
 cin >> n >> m;  
 for(int i=1;i<=n+2;i++)  
 tree.ins(i);  
 while(m--)  
 {  
 int l,r;  
 cin>>l>>r;  
 tree.reverse(l,r);  
 }  
 tree.write(rt);  
 return 0;  
}

### LCA

// 倍增方法：  
#include <cstdio>  
#include <cstring>  
#include <iostream>  
#include <vector>  
#define MXN 50007  
using namespace std;  
std::vector<int> v[MXN];  
std::vector<int> w[MXN];  
int fa[MXN][31], cost[MXN][31], dep[MXN];  
int n, m;  
int a, b, c;  
void dfs(int root, int fno) {  
 fa[root][0] = fno;  
 dep[root] = dep[fa[root][0]] + 1;  
 for (int i = 1; i < 31; ++i) {  
 fa[root][i] = fa[fa[root][i - 1]][i - 1];  
 cost[root][i] = cost[fa[root][i - 1]][i - 1] + cost[root][i - 1];  
 }  
 int sz = v[root].size();  
 for (int i = 0; i < sz; ++i) {  
 if (v[root][i] == fno) continue;  
 cost[v[root][i]][0] = w[root][i];  
 dfs(v[root][i], root);  
 }  
}  
int lca(int x, int y) {  
 if (dep[x] > dep[y]) swap(x, y);  
 int tmp = dep[y] - dep[x], ans = 0;  
 for (int j = 0; tmp; ++j, tmp >>= 1)  
 if (tmp & 1) ans += cost[y][j], y = fa[y][j];  
 if (y == x) return ans;  
 for (int j = 30; j >= 0 && y != x; --j) {  
 if (fa[x][j] != fa[y][j]) {  
 ans += cost[x][j] + cost[y][j];  
 x = fa[x][j];  
 y = fa[y][j];  
 }  
 }  
 ans += cost[x][0] + cost[y][0];  
 return ans;  
}  
int main() {  
 memset(fa, 0, sizeof(fa));  
 memset(cost, 0, sizeof(cost));  
 memset(dep, 0, sizeof(dep));  
 scanf("%d", &n);  
 for (int i = 1; i < n; ++i) {  
 scanf("%d %d %d", &a, &b, &c);  
 ++a, ++b;  
 v[a].push\_back(b);  
 v[b].push\_back(a);  
 w[a].push\_back(c);  
 w[b].push\_back(c);  
 }  
 dfs(1, 0);  
 scanf("%d", &m);  
 for (int i = 0; i < m; ++i) {  
 scanf("%d %d", &a, &b);  
 ++a, ++b;  
 printf("%d\n", lca(a, b));  
 }  
 return 0;  
}

// Tarjan方法：  
#include <algorithm>  
#include <iostream>  
using namespace std;  
class Edge {  
public:  
 int toVertex, fromVertex;  
 int next;  
 int LCA;  
 Edge() : toVertex(-1), fromVertex(-1), next(-1), LCA(-1) {};  
 Edge(int u, int v, int n) : fromVertex(u), toVertex(v), next(n), LCA(-1) {};  
};  
const int MAX = 100;  
int head[MAX], queryHead[MAX];  
Edge edge[MAX], queryEdge[MAX];  
int parent[MAX], visited[MAX];  
int vertexCount, edgeCount, queryCount;  
void init() {  
 for (int i = 0; i <= vertexCount; i++) {  
 parent[i] = i;  
 }  
}  
int find(int x) {  
 if (parent[x] == x) {  
 return x;  
 }  
 else {  
 return find(parent[x]);  
 }  
}  
void tarjan(int u) {  
 parent[u] = u;  
 visited[u] = 1;  
 for (int i = head[u]; i != -1; i = edge[i].next) {  
 Edge& e = edge[i];  
 if (!visited[e.toVertex]) {  
 tarjan(e.toVertex);  
 parent[e.toVertex] = u;  
 }  
 }  
 for (int i = queryHead[u]; i != -1; i = queryEdge[i].next) {  
 Edge& e = queryEdge[i];  
 if (visited[e.toVertex]) {  
 queryEdge[i ^ 1].LCA = e.LCA = find(e.toVertex);  
 }  
 }  
}  
int main() {  
 memset(head, 0xff, sizeof(head));  
 memset(queryHead, 0xff, sizeof(queryHead));  
 cin >> vertexCount >> edgeCount >> queryCount;  
 int count = 0;  
 for (int i = 0; i < edgeCount; i++) {  
 int start = 0, end = 0;  
 cin >> start >> end;  
 edge[count] = Edge(start, end, head[start]);  
 head[start] = count;  
 count++;  
 edge[count] = Edge(end, start, head[end]);  
 head[end] = count;  
 count++;  
 }  
 count = 0;  
 for (int i = 0; i < queryCount; i++) {  
 int start = 0, end = 0;  
 cin >> start >> end;  
 queryEdge[count] = Edge(start, end, queryHead[start]);  
 queryHead[start] = count;  
 count++;  
 queryEdge[count] = Edge(end, start, queryHead[end]);  
 queryHead[end] = count;  
 count++;  
 }  
 init();  
 tarjan(1);  
 for (int i = 0; i < queryCount; i++) {  
 Edge& e = queryEdge[i \* 2];  
 cout << "(" << e.fromVertex << "," << e.toVertex << ") " << e.LCA << endl;  
 }  
 return 0;  
}

### 大数

//注：可以直接把BigInt和一样用cin cout都行，就是高精乘为了速度才用了FFT降低了精度，有需要可以自行更改。  
#include <cstdio>  
#include <iostream>  
#include <cmath>  
#include <string>  
#include <cstring>  
#include <vector>  
#include <algorithm>  
using namespace std;  
const double PI = acos(-1.0);  
struct Complex{  
 double x,y;  
 Complex(double \_x = 0.0,double \_y = 0.0){  
 x = \_x;  
 y = \_y;  
 }  
 Complex operator-(const Complex &b)const{  
 return Complex(x - b.x,y - b.y);  
 }  
 Complex operator+(const Complex &b)const{  
 return Complex(x + b.x,y + b.y);  
 }  
 Complex operator\*(const Complex &b)const{  
 return Complex(x\*b.x - y\*b.y,x\*b.y + y\*b.x);  
 }  
};  
void change(Complex y[],int len){  
 int i,j,k;  
 for(int i = 1,j = len/2;i<len-1;i++){  
 if(i < j) swap(y[i],y[j]);  
 k = len/2;  
 while(j >= k){  
 j = j - k;  
 k = k/2;  
 }  
 if(j < k) j+=k;  
 }  
}  
void fft(Complex y[],int len,int on){  
 change(y,len);  
 for(int h = 2;h <= len;h<<=1){  
 Complex wn(cos(on\*2\*PI/h),sin(on\*2\*PI/h));  
 for(int j = 0;j < len;j += h){  
 Complex w(1,0);  
 for(int k = j;k < j + h/2;k++){  
 Complex u = y[k];  
 Complex t = w\*y[k + h/2];  
 y[k] = u + t;  
 y[k + h/2] = u - t;  
 w = w\*wn;  
 }  
 }  
 }  
 if(on == -1){  
 for(int i = 0;i < len;i++){  
 y[i].x /= len;  
 }  
 }  
}  
class BigInt  
{  
#define Value(x, nega) ((nega) ? -(x) : (x))  
#define At(vec, index) ((index) < vec.size() ? vec[(index)] : 0)  
 static int absComp(const BigInt &lhs, const BigInt &rhs)  
 {  
 if (lhs.size() != rhs.size())  
 return lhs.size() < rhs.size() ? -1 : 1;  
 for (int i = lhs.size() - 1; i >= 0; --i)  
 if (lhs[i] != rhs[i])  
 return lhs[i] < rhs[i] ? -1 : 1;  
 return 0;  
 }  
 using Long = long long;  
 const static int Exp = 9;  
 const static Long Mod = 1000000000;  
 mutable std::vector<Long> val;  
 mutable bool nega = false;  
 void trim() const  
 {  
 while (val.size() && val.back() == 0)  
 val.pop\_back();  
 if (val.empty())  
 nega = false;  
 }  
 int size() const { return val.size(); }  
 Long &operator[](int index) const { return val[index]; }  
 Long &back() const { return val.back(); }  
 BigInt(int size, bool nega) : val(size), nega(nega) {}  
 BigInt(const std::vector<Long> &val, bool nega) : val(val), nega(nega) {}  
  
public:  
 friend std::ostream &operator<<(std::ostream &os, const BigInt &n)  
 {  
 if (n.size())  
 {  
 if (n.nega)  
 putchar('-');  
 for (int i = n.size() - 1; i >= 0; --i)  
 {  
 if (i == n.size() - 1)  
 printf("%lld", n[i]);  
 else  
 printf("%0\*lld", n.Exp, n[i]);  
 }  
 }  
 else  
 putchar('0');  
 return os;  
 }  
 friend BigInt operator+(const BigInt &lhs, const BigInt &rhs)  
 {  
 BigInt ret(lhs);  
 return ret += rhs;  
 }  
 friend BigInt operator-(const BigInt &lhs, const BigInt &rhs)  
 {  
 BigInt ret(lhs);  
 return ret -= rhs;  
 }  
 BigInt(Long x = 0)  
 {  
 if (x < 0)  
 x = -x, nega = true;  
 while (x >= Mod)  
 val.push\_back(x % Mod), x /= Mod;  
 if (x)  
 val.push\_back(x);  
 }  
 BigInt(const char \*s)  
 {  
 int bound = 0, pos;  
 if (s[0] == '-')  
 nega = true, bound = 1;  
 Long cur = 0, pow = 1;  
 for (pos = strlen(s) - 1; pos >= Exp + bound - 1; pos -= Exp, val.push\_back(cur), cur = 0, pow = 1)  
 for (int i = pos; i > pos - Exp; --i)  
 cur += (s[i] - '0') \* pow, pow \*= 10;  
 for (cur = 0, pow = 1; pos >= bound; --pos)  
 cur += (s[pos] - '0') \* pow, pow \*= 10;  
 if (cur)  
 val.push\_back(cur);  
 }  
 BigInt &operator=(const char \*s){  
 BigInt n(s);  
 \*this = n;  
 return n;  
 }  
 BigInt &operator=(const Long x){  
 BigInt n(x);  
 \*this = n;  
 return n;  
 }  
 friend std::istream &operator>>(std::istream &is, BigInt &n){  
 string s;  
 is >> s;  
 n=(char\*)s.data();  
 return is;  
 }  
 BigInt &operator+=(const BigInt &rhs)  
 {  
 const int cap = std::max(size(), rhs.size()) + 1;  
 val.resize(cap);  
 int carry = 0;  
 for (int i = 0; i < cap - 1; ++i)  
 {  
 val[i] = Value(val[i], nega) + Value(At(rhs, i), rhs.nega) + carry, carry = 0;  
 if (val[i] >= Mod)  
 val[i] -= Mod, carry = 1;  
 else if (val[i] < 0)  
 val[i] += Mod, carry = -1;  
 }  
 if ((val.back() = carry) == -1) //assert(val.back() == 1 or 0 or -1)  
 {  
 nega = true, val.pop\_back();  
 bool tailZero = true;  
 for (int i = 0; i < cap - 1; ++i)  
 {  
 if (tailZero && val[i])  
 val[i] = Mod - val[i], tailZero = false;  
 else  
 val[i] = Mod - 1 - val[i];  
 }  
 }  
 trim();  
 return \*this;  
 }  
 friend BigInt operator-(const BigInt &rhs)  
 {  
 BigInt ret(rhs);  
 ret.nega ^= 1;  
 return ret;  
 }  
 BigInt &operator-=(const BigInt &rhs)  
 {  
 rhs.nega ^= 1;  
 \*this += rhs;  
 rhs.nega ^= 1;  
 return \*this;  
 }  
 friend BigInt operator\*(const BigInt &lhs, const BigInt &rhs)  
 {  
 int len=1;  
 BigInt ll=lhs,rr=rhs;  
 ll.nega = lhs.nega ^ rhs.nega;  
 while(len<2\*lhs.size()||len<2\*rhs.size())len<<=1;  
 ll.val.resize(len),rr.val.resize(len);  
 Complex x1[len],x2[len];  
 for(int i=0;i<len;i++){  
 Complex nx(ll[i],0.0),ny(rr[i],0.0);  
 x1[i]=nx;  
 x2[i]=ny;  
 }  
 fft(x1,len,1);  
 fft(x2,len,1);  
 for(int i = 0 ; i < len; i++)  
 x1[i] = x1[i] \* x2[i];  
 fft( x1 , len , -1 );  
 for(int i = 0 ; i < len; i++)  
 ll[i] = int( x1[i].x + 0.5 );  
 for(int i = 0 ; i < len; i++){  
 ll[i+1]+=ll[i]/Mod;  
 ll[i]%=Mod;  
 }  
 ll.trim();  
 return ll;  
 }  
 friend BigInt operator\*(const BigInt &lhs, const Long &x){  
 BigInt ret=lhs;  
 bool negat = ( x < 0 );  
 Long xx = (negat) ? -x : x;  
 ret.nega ^= negat;  
 ret.val.push\_back(0);  
 ret.val.push\_back(0);  
 for(int i = 0; i < ret.size(); i++)  
 ret[i]\*=xx;  
 for(int i = 0; i < ret.size(); i++){  
 ret[i+1]+=ret[i]/Mod;  
 ret[i] %= Mod;  
 }  
 ret.trim();  
 return ret;  
 }  
 BigInt &operator\*=(const BigInt &rhs) { return \*this = \*this \* rhs; }  
 BigInt &operator\*=(const Long &x) { return \*this = \*this \* x; }  
 friend BigInt operator/(const BigInt &lhs, const BigInt &rhs)  
 {  
 static std::vector<BigInt> powTwo{BigInt(1)};  
 static std::vector<BigInt> estimate;  
 estimate.clear();  
 if (absComp(lhs, rhs) < 0)  
 return BigInt();  
 BigInt cur = rhs;  
 int cmp;  
 while ((cmp = absComp(cur, lhs)) <= 0)  
 {  
 estimate.push\_back(cur), cur += cur;  
 if (estimate.size() >= powTwo.size())  
 powTwo.push\_back(powTwo.back() + powTwo.back());  
 }  
 if (cmp == 0)  
 return BigInt(powTwo.back().val, lhs.nega ^ rhs.nega);  
 BigInt ret = powTwo[estimate.size() - 1];  
 cur = estimate[estimate.size() - 1];  
 for (int i = estimate.size() - 1; i >= 0 && cmp != 0; --i)  
 if ((cmp = absComp(cur + estimate[i], lhs)) <= 0)  
 cur += estimate[i], ret += powTwo[i];  
 ret.nega = lhs.nega ^ rhs.nega;  
 return ret;  
 }  
 friend BigInt operator/(const BigInt &num,const Long &x){  
 bool negat = ( x < 0 );  
 Long xx = (negat) ? -x : x;  
 BigInt ret;  
 Long k = 0;  
 ret.val.resize( num.size() );  
 ret.nega = (num.nega ^ negat);  
 for(int i = num.size() - 1 ;i >= 0; i--){  
 ret[i] = ( k \* Mod + num[i]) / xx;  
 k = ( k \* Mod + num[i]) % xx;  
 }  
 ret.trim();  
 return ret;  
 }  
 bool operator==(const BigInt &rhs) const  
 {  
 return nega == rhs.nega && val == rhs.val;  
 }  
 bool operator!=(const BigInt &rhs) const { return nega != rhs.nega || val != rhs.val; }  
 bool operator>=(const BigInt &rhs) const { return !(\*this < rhs); }  
 bool operator>(const BigInt &rhs) const { return !(\*this <= rhs); }  
 bool operator<=(const BigInt &rhs) const  
 {  
 if (nega && !rhs.nega)  
 return true;  
 if (!nega && rhs.nega)  
 return false;  
 int cmp = absComp(\*this, rhs);  
 return nega ? cmp >= 0 : cmp <= 0;  
 }  
 bool operator<(const BigInt &rhs) const  
 {  
 if (nega && !rhs.nega)  
 return true;  
 if (!nega && rhs.nega)  
 return false;  
 return (absComp(\*this, rhs) < 0) ^ nega;  
 }  
 void swap(const BigInt &rhs) const  
 {  
 std::swap(val, rhs.val);  
 std::swap(nega, rhs.nega);  
 }  
};  
BigInt ba,bb;  
int main(){  
 cin>>ba>>bb;  
 cout << ba + bb << '\n';//和  
 cout << ba - bb << '\n';//差  
 cout << ba \* bb << '\n';//积  
 BigInt d;  
 cout << (d = ba / bb) << '\n';//商  
 cout << ba - d \* bb << '\n';//余  
 return 0;  
}

### 线性基

void Get\_LB(ll x)  
{  
 for(int i = 62; i >= 0; i--)  
 {  
 if(!(x >> (ll)i))  
 continue;  
 if(!p[i])  
 {  
 p[i] = x;  
 break;  
 }  
 x ^= p[i];  
 }  
}

### ST表

template <typename INTEGER>  
struct STMax  
{  
 // 从0开始  
 std::vector<std::vector<INTEGER>> data;  
 STMax(int siz)  
 {  
 int upper\_pow = clz(siz) + 1;  
 data.resize(upper\_pow);  
 data.assign(upper\_pow, vector<INTEGER>());  
 data[0].assign(siz, 0);  
 }  
 INTEGER &operator[](int where)  
 {  
 return data[0][where];  
 }  
 void generate\_max()  
 {  
 for (auto j : range(1, data.size()))  
 {  
 data[j].assign(data[0].size(), 0);  
 for (int i = 0; i + (1 << j) - 1 < data[0].size(); i++)  
 {  
 data[j][i] = std::max(data[j - 1][i], data[j - 1][i + (1 << (j - 1))]);  
 }  
 }  
 }  
 /\*闭区间[l, r]，注意有效位从0开始\*/  
 INTEGER query\_max(int l, int r)  
 {  
 int k = 31 - \_\_builtin\_clz(r - l + 1);  
 return std::max(data[k][l], data[k][r - (1 << k) + 1]);  
 }  
};

### 带修改整体二分

#include <bits/stdc++.h>  
using namespace std;  
  
const int maxn = 1e5 + 5;  
struct query1  
{  
 int type;  
 int l, r, k;  
 int id;  
 // 对于询问, type = 1, l, r 表示区间左右边界, k 表示询问第 k 小  
 // 对于修改, type = 0, l 表示修改位置, r 表示修改后的值,  
 // k 表示当前操作是插入(1)还是擦除(-1), 更新树状数组时使用.  
 // id 记录每个操作原先的编号, 因二分过程中操作顺序会被打散  
};  
struct num1  
{  
 int p, x;  
};  
vector<int> ans(100000);  
int n, m;  
int t[maxn];  
int a[maxn];  
int sum(int p)  
{  
 int ans = 0;  
 while (p)  
 {  
 ans += t[p];  
 p -= p & (-p);  
 }  
 return ans;  
}  
void add(int p, int x)  
{  
 while (p <= n)  
 {  
 t[p] += x;  
 p += p & (-p);  
 }  
}  
void solve(int l, int r, vector<query1> &q)  
{  
 if (q.size() == 0)  
 return;  
 if (l == r)  
 {  
 for (auto i : q)  
 {  
 if (i.type == 1)  
 ans[i.id] = l;  
 }  
 return;  
 }  
 int mid = (l + r) >> 1;  
 vector<query1> q1;  
 vector<query1> q2;  
 for (auto i : q)  
 {  
 if (i.type == 1)  
 {  
 int t = sum(i.r) - sum(i.l - 1);  
 if (i.k <= t)  
 q1.push\_back(i);  
 else  
 {  
 i.k -= t;  
 q2.push\_back(i);  
 }  
 }  
 else  
 {  
 if (i.r <= mid)  
 {  
 add(i.l, i.k);  
 q1.push\_back(i);  
 }  
 else  
 {  
 q2.push\_back(i);  
 }  
 }  
 }  
 for (auto i : q)  
 if(i.type==0)  
 if (i.r <= mid)  
 {  
 add(i.l, -i.k);  
 }  
 solve(l, mid, q1);  
 solve(mid + 1, r, q2);  
}  
signed main()  
{  
 ios::sync\_with\_stdio(false);  
 int tmp1, tmp2, tmp3;  
 vector<query1> query;  
 cin >> n >> m;  
 for (int i = 1; i <= n; i++)  
 {  
 cin >> a[i];  
 query.push\_back({0, i, a[i], 1, 0});  
 }  
 char op;  
 int l, r, k;  
 int cnt = 0;  
 int x, y;  
 for (int i = 1; i <= m; i++)  
 {  
 cin >> op;  
 if (op == 'Q')  
 {  
 cin >> l >> r >> k;  
 query.push\_back({1, l, r, k, ++cnt});  
 }  
 else if (op == 'C')  
 {  
 cin >> x >> y;  
 query.push\_back({0, x, a[x], -1, 0});  
 a[x]=y;  
 query.push\_back({0, x, y, 1, 0});  
   
 }  
 }  
 solve(0, 1e9, query);  
 for (int i = 1; i <= cnt; i++)  
 cout << ans[i] << "\n";  
}

## 计算几何

### 必要的头

namespace Geometry  
{  
 using FLOAT\_ = double;  
  
 constexpr const FLOAT\_ Infinity = INFINITY;  
 const FLOAT\_ decimal\_round = 1e-8; // 精度参数  
  
 const FLOAT\_ DEC = 1.0 / decimal\_round;  
 const double smooth\_const2 = 0.479999989271164; // 二次项平滑系数  
 const double smooth\_const3 = 0.234999999403954; // 三次项平滑系数  
  
 const FLOAT\_ PI = acos(-1);  
 bool round\_compare(FLOAT\_ a, FLOAT\_ b) { return round(DEC \* a) == round(DEC \* b); }  
 FLOAT\_ Round(FLOAT\_ a) { return round(DEC \* a) / DEC; }  
}

### 分数类

template <typename PrecisionType = long long>  
struct Fraction  
{  
 PrecisionType upper, lower;  
  
 Fraction(PrecisionType u = 0, PrecisionType l = 1)  
 {  
 upper = u;  
 lower = l;  
 }  
 void normalize()  
 {  
 if (upper)  
 {  
 PrecisionType g = abs(std::\_\_gcd(upper, lower));  
 upper /= g;  
 lower /= g;  
 }  
 else  
 lower = 1;  
 if (lower < 0)  
 {  
 lower = -lower;  
 upper = -upper;  
 }  
 }  
 long double ToFloat() { return (long double)upper / (long double)lower; }  
 bool operator==(Fraction b) { return upper \* b.lower == lower \* b.upper; }  
 bool operator>(Fraction b) { return upper \* b.lower > lower \* b.upper; }  
 bool operator<(Fraction b) { return upper \* b.lower < lower \* b.upper; }  
 bool operator<=(Fraction b) { return !(\*this > b); }  
 bool operator>=(Fraction b) { return !(\*this < b); }  
 bool operator!=(Fraction b) { return !(\*this == b); }  
 Fraction operator-() { return Fraction(-upper, lower); }  
 Fraction operator+(Fraction b) { return Fraction(upper \* b.lower + b.upper \* lower, lower \* b.lower); }  
 Fraction operator-(Fraction b) { return (\*this) + (-b); }  
 Fraction operator\*(Fraction b) { return Fraction(upper \* b.upper, lower \* b.lower); }  
 Fraction operator/(Fraction b) { return Fraction(upper \* b.lower, lower \* b.upper); }  
 Fraction &operator+=(Fraction b)  
 {  
 \*this = \*this + b;  
 this->normalize();  
 return \*this;  
 }  
 Fraction &operator-=(Fraction b)  
 {  
 \*this = \*this - b;  
 this->normalize();  
 return \*this;  
 }  
 Fraction &operator\*=(Fraction b)  
 {  
 \*this = \*this \* b;  
 this->normalize();  
 return \*this;  
 }  
 Fraction &operator/=(Fraction b)  
 {  
 \*this = \*this / b;  
 this->normalize();  
 return \*this;  
 }  
 friend Fraction fabs(Fraction a) { return Fraction(abs(a.upper), abs(a.lower)); }  
 std::string to\_string() { return lower == 1 ? std::to\_string(upper) : std::to\_string(upper) + '/' + std::to\_string(lower); }  
 friend std::ostream &operator<<(std::ostream &o, Fraction a)  
 {  
 return o << "Fraction(" << std::to\_string(a.upper) << ", " << std::to\_string(a.lower) << ")";  
 }  
 friend std::istream &operator>>(std::istream &i, Fraction &a)  
 {  
 char slash;  
 return i >> a.upper >> slash >> a.lower;  
 }  
 friend isfinite(Fraction a) { return a.lower != 0; }  
 void set\_value(PrecisionType u, PrecisionType d = 1) { upper = u, lower = d; }  
};

### 二维向量

struct Vector2  
{  
 FLOAT\_ x, y;  
 Vector2(FLOAT\_ \_x, FLOAT\_ \_y) : x(\_x), y(\_y) {}  
 Vector2(FLOAT\_ n) : x(n), y(n) {}  
 Vector2() : x(0.0), y(0.0) {}  
 Vector2 &operator=(Vector2 b)  
 {  
 this->x = b.x;  
 this->y = b.y;  
 return \*this;  
 }  
 bool operator==(Vector2 b) { return round\_compare(this->x, b.x) and round\_compare(this->y, b.y); }  
 bool operator!=(Vector2 b) { return not((\*this) == b); }  
  
 friend std::ostream &operator<<(std::ostream &o, Vector2 v)  
 {  
 o << v.ToString();  
 return o;  
 }  
 friend Vector2 operator\*(FLOAT\_ n, Vector2 v) { return Vector2(v.x \* n, v.y \* n); }  
 friend Vector2 operator/(FLOAT\_ n, Vector2 v) { return Vector2(n / v.x, n / v.y); }  
 Vector2 operator-() { return Vector2(-(this->x), -(this->y)); }  
 Vector2 operator+(Vector2 b) { return Vector2(this->x + b.x, this->y + b.y); }  
 Vector2 operator-(Vector2 b) { return (\*this) + (-b); }  
 Vector2 operator\*(FLOAT\_ n) { return Vector2(this->x \* n, this->y \* n); }  
 Vector2 operator\*(Vector2 b) { return Vector2(this->x \* b.x, this->y \* b.y); }  
 Vector2 operator/(FLOAT\_ n) { return (\*this) \* (FLOAT\_(1) / n); }  
 Vector2 operator/(Vector2 b) { return (\*this) \* (FLOAT\_(1) / b); }  
 Vector2 operator+=(Vector2 b) { return (\*this) = (\*this) + b; }  
  
 bool operator<(Vector2 b) { return this->x < b.x or this->x == b.x and this->y < b.y; }  
  
 /\* 向量的平方模 \*/  
 FLOAT\_ sqrMagnitude() { return pow(this->x, 2) + pow(this->y, 2); }  
 /\* 向量的模 \*/  
 FLOAT\_ magnitude() { return sqrt(this->sqrMagnitude()); }  
 /\*判等\*/  
 bool equals(Vector2 b) { return (\*this) == b; }  
  
 /\*用极坐标换算笛卡尔坐标\*/  
 static Vector2 fromPolarCoordinate(Vector2 v, bool use\_degree = 1)  
 {  
 return v.toCartesianCoordinate(use\_degree);  
 }  
  
 /\*转为笛卡尔坐标\*/  
 Vector2 toCartesianCoordinate(bool use\_degree = 1)  
 {  
 return Vector2(  
 x \* cos(y \* (use\_degree ? PI / 180.0 : 1)),  
 x \* sin(y \* (use\_degree ? PI / 180.0 : 1)));  
 }  
 /\*转为极坐标\*/  
 Vector2 toPolarCoordinate(bool use\_degree = 1)  
 {  
 return Vector2(  
 magnitude(),  
 toPolarAngle(use\_degree));  
 }  
  
 /\*获取极角\*/  
 FLOAT\_ toPolarAngle(bool use\_degree = 1)  
 {  
 return atan2(y, x) \* (use\_degree ? 180.0 / PI : 1);  
 }  
 /\*极坐标排序比较器\*/  
 static std::function<bool(Vector2 &, Vector2 &)> PolarSortCmp = [](Vector2 &a, Vector2 &b) -> bool  
 {  
 return a.toPolarAngle(0) < b.toPolarAngle(0);  
 };  
  
 /\*叉乘排序比较器\*/  
 static std::function<bool(Vector2 &, Vector2 &)> CrossSortCmp = [](Vector2 &a, Vector2 &b) -> bool  
 {  
 return Cross(a, b) > 0;  
 };  
  
 /\*转为极坐标\*/  
 static Vector2 ToPolarCoordinate(Vector2 coordinate, bool use\_degree = 1) { return coordinate.toPolarCoordinate(use\_degree); }  
  
 static bool Equals(Vector2 a, Vector2 b) { return a == b; }  
 /\* 向量单位化 \*/  
 void Normalize()  
 {  
 FLOAT\_ \_m = this->magnitude();  
 this->x /= \_m;  
 this->y /= \_m;  
 }  
 /\*设置值\*/  
 void Set(FLOAT\_ newX, FLOAT\_ newY)  
 {  
 this->x = newX;  
 this->y = newY;  
 }  
 /\*转为字符串\*/  
 std::string ToString()  
 {  
 std::ostringstream ostr;  
 ostr << "Vector2(" << this->x << ", " << this->y << ")";  
 return ostr.str();  
 }  
  
 /\* 返回与该向量方向同向的单位向量 \*/  
 Vector2 normalized()  
 {  
 FLOAT\_ \_m = this->magnitude();  
 return Vector2(this->x / \_m, this->y / \_m);  
 }  
 // FLOAT\_ Distance(Vector2 b) { return ((\*this) - b).magnitude(); }  
 /\* 距离 \*/  
 static FLOAT\_ Distance(Vector2 a, Vector2 b) { return (a - b).magnitude(); }  
  
 /\*向量线性插值\*/  
 static Vector2 LerpUnclamped(Vector2 a, Vector2 b, FLOAT\_ t)  
 {  
 Vector2 c = b - a;  
 return a + c \* t;  
 }  
  
 /\* 拿它的垂直向量（逆时针旋转90°） \*/  
 static Vector2 Perpendicular(Vector2 inDirection)  
 {  
 return Vector2(-inDirection.y, inDirection.x);  
 }  
 /\*根据inNormal法向反射inDirection向量，参考光的平面镜反射，入射光为inDirection，平面镜的法线为inNormal\*/  
 static Vector2 Reflect(Vector2 inDirection, Vector2 inNormal)  
 {  
 return inDirection - 2 \* Vector2::Dot(inDirection, inNormal) \* inNormal;  
 }  
  
 /\* 点积 \*/  
 static FLOAT\_ Dot(Vector2 lhs, Vector2 rhs)  
 {  
 return lhs.x \* rhs.x + lhs.y \* rhs.y;  
 }  
 /\* 叉积 \*/  
 static FLOAT\_ Cross(Vector2 lhs, Vector2 rhs) { return lhs.x \* rhs.y - lhs.y \* rhs.x; }  
  
 /\* 对位相乘罢了 \*/  
 static Vector2 Scale(Vector2 a, Vector2 b) { return Vector2(a.x \* b.x, a.y \* b.y); }  
  
 /\* 对位相乘罢了 \*/  
 Vector2 Scale(Vector2 scale) { return (\*this) \* scale; }  
  
 /\*有符号弧度夹角\*/  
 static FLOAT\_ SignedRad(Vector2 from, Vector2 to) { return atan2(Vector2::Cross(from, to), Vector2::Dot(from, to)); }  
 /\*无符号弧度夹角\*/  
 static FLOAT\_ Rad(Vector2 from, Vector2 to) { return abs(Vector2::SignedRad(from, to)); }  
 /\*有符号角度夹角\*/  
 static FLOAT\_ SignedAngle(Vector2 from, Vector2 to)  
 {  
 // return acos(Vector2::Dot(from, to) / (from.magnitude() \* to.magnitude()));  
 return Vector2::SignedRad(from, to) \* 180.0 / PI;  
 }  
 /\*无符号角度夹角\*/  
 static FLOAT\_ Angle(Vector2 from, Vector2 to) { return abs(Vector2::SignedAngle(from, to)); }  
  
 /\*返回该方向上最大不超过maxLength长度的向量\*/  
 static Vector2 ClampMagnitude(Vector2 vector, FLOAT\_ maxLength)  
 {  
 if (vector.magnitude() <= maxLength)  
 return vector;  
 else  
 return vector.normalized() \* maxLength;  
 }  
 /\*返回俩向量中x的最大值和y的最大值构造而成的向量\*/  
 static Vector2 Max(Vector2 lhs, Vector2 rhs)  
 {  
 return Vector2(max(lhs.x, rhs.x), max(lhs.y, rhs.y));  
 }  
  
 /\*返回俩向量中x的最小值和y的最小值构造而成的向量\*/  
 static Vector2 Min(Vector2 lhs, Vector2 rhs)  
 {  
 return Vector2(min(lhs.x, rhs.x), min(lhs.y, rhs.y));  
 }  
  
 /\*获得vector在onNormal方向的投影\*/  
 static Vector2 Project(Vector2 vector, Vector2 onNormal)  
 {  
 return cos(Rad(vector, onNormal)) \* vector.magnitude() \* onNormal;  
 }  
};

### 三维向量

struct Vector3 // 三维向量  
{  
 FLOAT\_ x, y, z;  
 Vector3(FLOAT\_ \_x, FLOAT\_ \_y, FLOAT\_ \_z) : x(\_x), y(\_y), z(\_z) {}  
 Vector3(FLOAT\_ n) : x(n), y(n), z(n) {}  
 Vector3(Vector2 &b) : x(b.x), y(b.y), z(0.0) {}  
 // Vector3(Vector2 &&b) : x(b.x), y(b.y), z(0.0) {}  
 Vector3() : x(0.0), y(0.0), z(0.0) {}  
 Vector3 &operator=(Vector3 b)  
 {  
 this->x = b.x;  
 this->y = b.y;  
 this->z = b.z;  
 return \*this;  
 }  
 bool operator==(Vector3 b) { return round\_compare(this->x, b.x) and round\_compare(this->y, b.y) and round\_compare(this->z, b.z); }  
 bool operator!=(Vector3 b) { return not((\*this) == b); }  
  
 friend std::ostream &operator<<(std::ostream &o, Vector3 v)  
 {  
 o << v.ToString();  
 return o;  
 }  
 friend Vector3 operator\*(FLOAT\_ n, Vector3 v) { return Vector3(v.x \* n, v.y \* n, v.z \* n); }  
 friend Vector3 operator/(FLOAT\_ n, Vector3 v) { return Vector3(n / v.x, n / v.y, n / v.z); }  
 Vector3 operator-() { return Vector3(-(this->x), -(this->y), -(this->z)); }  
 Vector3 operator+(Vector3 b) { return Vector3(this->x + b.x, this->y + b.y, this->z + b.z); }  
 Vector3 operator-(Vector3 b) { return (\*this) + (-b); }  
 Vector3 operator\*(FLOAT\_ n) { return Vector3(this->x \* n, this->y \* n, this->z \* n); }  
 Vector3 operator\*(Vector3 b) { return Vector3(this->x \* b.x, this->y \* b.y, this->z \* b.z); }  
 Vector3 operator/(FLOAT\_ n) { return (\*this) \* (FLOAT\_(1) / n); }  
 Vector3 operator/(Vector3 b) { return (\*this) \* (FLOAT\_(1) / b); }  
 Vector3 operator-=(Vector3 b) { return (\*this) = (\*this) - b; }  
  
 /\* 向量的平方模 \*/  
 FLOAT\_ sqrMagnitude() { return pow(this->x, 2) + pow(this->y, 2) + pow(this->z, 2); }  
 /\* 向量的模 \*/  
 FLOAT\_ magnitude() { return sqrt(this->sqrMagnitude()); }  
 /\*判等\*/  
 bool equals(Vector3 b) { return (\*this) == b; }  
 static bool Equals(Vector3 a, Vector3 b) { return a == b; }  
 /\* 向量单位化 \*/  
 void Normalize()  
 {  
 FLOAT\_ \_m = this->magnitude();  
 this->x /= \_m;  
 this->y /= \_m;  
 this->z /= \_m;  
 }  
 /\*设置值\*/  
 void Set(FLOAT\_ newX, FLOAT\_ newY, FLOAT\_ newZ)  
 {  
 this->x = newX;  
 this->y = newY;  
 this->z = newZ;  
 }  
 /\*转为字符串\*/  
 std::string ToString()  
 {  
 std::ostringstream ostr;  
 ostr << "Vector3(" << this->x << ", " << this->y << ", " << this->z << ")";  
 return ostr.str();  
 }  
  
 /\* 返回与该向量方向同向的单位向量 \*/  
 Vector3 normalized()  
 {  
 FLOAT\_ \_m = this->magnitude();  
 return Vector3(this->x / \_m, this->y / \_m, this->z / \_m);  
 }  
 // FLOAT\_ Distance(Vector2 b) { return ((\*this) - b).magnitude(); }  
 /\* 距离 \*/  
 static FLOAT\_ Distance(Vector3 a, Vector3 b) { return (a - b).magnitude(); }  
  
 /\*向量线性插值\*/  
 static Vector3 LerpUnclamped(Vector3 a, Vector3 b, FLOAT\_ t) { return a + (b - a) \* t; }  
  
  
 /\* 拿它的垂直向量（逆时针旋转90°） \*/  
 static Vector3 Perpendicular(Vector3 inDirection)  
 {  
 return Vector3(-inDirection.y, inDirection.x, 0);  
 }  
 /\*根据inNormal法向反射inDirection向量，参考光的平面镜反射，入射光为inDirection，平面镜的法线为inNormal\*/  
 static Vector3 Reflect(Vector3 inDirection, Vector3 inNormal)  
 {  
 return inDirection - 2 \* Vector3::Dot(inDirection, inNormal) \* inNormal;  
 }  
  
 /\* 点积 \*/  
 static FLOAT\_ Dot(Vector3 lhs, Vector3 rhs)  
 {  
 return lhs.x \* rhs.x + lhs.y \* rhs.y + lhs.z \* rhs.z;  
 }  
 /\* 叉积 \*/  
 static Vector3 Cross(Vector3 lhs, Vector3 rhs) { return Vector3(lhs.y \* rhs.z - lhs.z \* rhs.y, lhs.z \* rhs.x - lhs.x \* rhs.z, lhs.x \* rhs.y - lhs.y \* rhs.x); }  
  
 /\* 对位相乘罢了 \*/  
 static Vector3 Scale(Vector3 a, Vector3 b) { return a \* b; }  
  
 /\* 对位相乘罢了 \*/  
 Vector3 Scale(Vector3 scale) { return (\*this) \* scale; }  
  
 /\*无符号弧度夹角\*/  
 static FLOAT\_ Rad(Vector3 from, Vector3 to) { return acos(Dot(from, to) / (from.magnitude() \* to.magnitude())); }  
  
 /\*无符号角度夹角\*/  
 static FLOAT\_ Angle(Vector3 from, Vector3 to) { return Rad(from, to) \* 180 / PI; }  
  
 /\*返回该方向上最大不超过maxLength长度的向量\*/  
 static Vector3 ClampMagnitude(Vector3 vector, FLOAT\_ maxLength)  
 {  
 if (vector.magnitude() <= maxLength)  
 return vector;  
 else  
 return vector.normalized() \* maxLength;  
 }  
 /\*返回俩向量中x的最大值和y的最大值构造而成的向量\*/  
 static Vector3 Max(Vector3 lhs, Vector3 rhs)  
 {  
 return Vector3(max(lhs.x, rhs.x), max(lhs.y, rhs.y), max(lhs.z, rhs.z));  
 }  
  
 /\*返回俩向量中x的最小值和y的最小值构造而成的向量\*/  
 static Vector3 Min(Vector3 lhs, Vector3 rhs)  
 {  
 return Vector3(min(lhs.x, rhs.x), min(lhs.y, rhs.y), min(lhs.z, rhs.z));  
 }  
  
 /\*获得vector在onNormal(请自己单位化)方向的投影\*/  
 static Vector3 Project(Vector3 vector, Vector3 onNormal)  
 {  
 return vector.magnitude() \* cos(Rad(vector, onNormal)) \* onNormal;  
 }  
  
 /\*将两个向量单位化，并调整切线位置使之垂直于法向\*/  
 static void OrthoNormalize(Vector3 &normal, Vector3 &tangent)  
 {  
 normal.Normalize();  
 tangent = tangent - Project(tangent, normal);  
 tangent.Normalize();  
 }  
  
 /\*将三个向量单位化，并调整使之两两垂直\*/  
 static void OrthoNormalize(Vector3 &normal, Vector3 &tangent, Vector3 &binormal)  
 {  
 normal.Normalize();  
 tangent = tangent - Project(tangent, normal);  
 tangent.Normalize();  
 binormal -= Project(binormal, normal);  
 binormal -= Project(binormal, tangent);  
 binormal.Normalize();  
 }  
  
 /\*获得vector在以planeNormal为法向量的平面的投影\*/  
 static Vector3 ProjectOnPlane(Vector3 vector, Vector3 planeNormal)  
 {  
 FLOAT\_ mag = vector.magnitude();  
 FLOAT\_ s = Rad(vector, planeNormal);  
 OrthoNormalize(planeNormal, vector);  
 return mag \* sin(s) \* vector;  
 }  
  
 /\*罗德里格旋转公式，获得current绕轴normal(请自己单位化)旋转degree度（默认角度）的向量，右手螺旋意义\*/  
 static Vector3 Rotate(Vector3 current, Vector3 normal, FLOAT\_ degree, bool use\_degree = 1)  
 {  
 FLOAT\_ r = use\_degree ? degree / 180 \* PI : degree;  
 FLOAT\_ c = cos(r);  
 return c \* current + (1.0 - c) \* Dot(normal, current) \* normal + Cross(sin(r) \* normal, current);  
 }  
  
 /\*将current向target转向degree度，如果大于夹角则返回target方向长度为current的向量\*/  
 static Vector3 RotateTo(Vector3 current, Vector3 target, FLOAT\_ degree, bool use\_degree = 1)  
 {  
 FLOAT\_ r = use\_degree ? degree / 180 \* PI : degree;  
 if (r >= Rad(current, target))  
 return current.magnitude() / target.magnitude() \* target;  
 else  
 {  
 // FLOAT\_ mag = current.magnitude();  
 Vector3 nm = Cross(current, target).normalized();  
 return Rotate(current, nm, r);  
 }  
 }  
  
 /\*球面插值\*/  
 static Vector3 SlerpUnclamped(Vector3 a, Vector3 b, FLOAT\_ t)  
 {  
 Vector3 rot = RotateTo(a, b, Rad(a, b) \* t, false);  
 FLOAT\_ l = b.magnitude() \* t + a.magnitude() \* (1 - t);  
 return rot.normalized() \* l;  
 }  
  
 /\*根据经纬，拿一个单位化的三维向量，以北纬和东经为正\*/  
 static Vector3 FromLongitudeAndLatitude(FLOAT\_ longitude, FLOAT\_ latitude)  
 {  
 Vector3 lat = Rotate(Vector3(1, 0, 0), Vector3(0, -1, 0), latitude);  
 return Rotate(lat, Vector3(0, 0, 1), longitude);  
 }  
  
 /\*球坐标转换为xyz型三维向量\*/  
 static Vector3 FromSphericalCoordinate(Vector3 spherical, bool use\_degree = 1) { return FromSphericalCoordinate(spherical.x, spherical.y, spherical.z, use\_degree); }  
 /\*球坐标转换为xyz型三维向量，半径r，theta倾斜角（纬度），phi方位角（经度），默认输出角度\*/  
 static Vector3 FromSphericalCoordinate(FLOAT\_ r, FLOAT\_ theta, FLOAT\_ phi, bool use\_degree = 1)  
 {  
 theta = use\_degree ? theta / 180 \* PI : theta;  
 phi = use\_degree ? phi / 180 \* PI : phi;  
 return Vector3(  
 r \* sin(theta) \* cos(phi),  
 r \* sin(theta) \* sin(phi),  
 r \* cos(theta));  
 }  
 /\*直角坐标转换为球坐标，默认输出角度\*/  
 static Vector3 ToSphericalCoordinate(Vector3 coordinate, bool use\_degree = 1)  
 {  
 FLOAT\_ r = coordinate.magnitude();  
 return Vector3(  
 r,  
 acos(coordinate.z / r) \* (use\_degree ? 180.0 / PI : 1),  
 atan2(coordinate.y, coordinate.x) \* (use\_degree ? 180.0 / PI : 1));  
 }  
 /\*直角坐标转换为球坐标，默认输出角度\*/  
 Vector3 toSphericalCoordinate(bool use\_degree = 1) { return ToSphericalCoordinate(\*this, use\_degree); }  
  
};

### N维向量

template <typename VALUETYPE = FLOAT\_>  
struct VectorN : std::vector<VALUETYPE>  
{  
 void init(int siz, VALUETYPE default\_val = 0)  
 {  
 this->clear();  
 this->assign(siz, default\_val);  
 this->resize(siz);  
 }  
 VectorN(int siz, VALUETYPE default\_val = 0) { init(siz, default\_val); }  
 VectorN() = default;  
  
 bool any()  
 {  
 for (auto &i : \*this)  
 {  
 if (i != 0)  
 return true;  
 else if (!isfinite(i))  
 return false;  
 }  
 return false;  
 }  
 bool is\_all\_nan()  
 {  
 for (auto &i : \*this)  
 {  
 if (!isnan(i))  
 return false;  
 }  
 return true;  
 }  
  
 void rconcat(VectorN &&r)  
 {  
 this->insert(this->end(), r.begin(), r.end());  
 }  
 void rconcat(VectorN &r) { rconcat(std::move(r)); }  
  
 void lerase(int ctr)  
 {  
 assert(this->size() >= ctr);  
 this->erase(this->begin(), this->begin() + ctr);  
 }  
  
   
  
 // 四则运算 和单个数  
  
 VectorN operator\*(VALUETYPE &&operand)  
 {  
 VectorN ret(this->size());  
 for (int i = 0; i < this->size(); i++)  
 ret[i] = (\*this)[i] \* operand;  
 return ret;  
 }  
 friend VectorN operator\*(VALUETYPE &&operand, VectorN &r)  
 {  
 VectorN ret(r.size());  
 for (int i = 0; i < r.size(); i++)  
 ret[i] = operand \* r[i];  
 return ret;  
 }  
  
 VectorN operator/(VALUETYPE &&operand)  
 {  
 VectorN ret(this->size());  
 for (int i = 0; i < this->size(); i++)  
 ret[i] = (\*this)[i] / operand;  
 return ret;  
 }  
 friend VectorN operator/(VALUETYPE &&operand, VectorN &r)  
 {  
 VectorN ret(r.size());  
 for (int i = 0; i < r.size(); i++)  
 ret[i] = operand / r[i];  
 return ret;  
 }  
  
 VectorN operator+(VALUETYPE &&operand)  
 {  
 VectorN ret(this->size());  
 for (int i = 0; i < this->size(); i++)  
 ret[i] = (\*this)[i] + operand;  
 return ret;  
 }  
 friend VectorN operator+(VALUETYPE &&operand, VectorN &r)  
 {  
 VectorN ret(r.size());  
 for (int i = 0; i < r.size(); i++)  
 ret[i] = operand + r[i];  
 return ret;  
 }  
  
 VectorN operator-(VALUETYPE &&operand)  
 {  
 VectorN ret(this->size());  
 for (int i = 0; i < this->size(); i++)  
 ret[i] = (\*this)[i] - operand;  
 return ret;  
 }  
 friend VectorN operator-(VALUETYPE &&operand, VectorN &r)  
 {  
 VectorN ret(r.size());  
 for (int i = 0; i < r.size(); i++)  
 ret[i] = operand - r[i];  
 return ret;  
 }  
  
   
 // 四则运算 和同类  
  
 VectorN operator+(VectorN &&operand)  
 {  
 VectorN ret(this->size());  
 for (int i = 0; i < this->size(); i++)  
 ret[i] = (\*this)[i] + operand[i];  
 return ret;  
 }  
  
 VectorN operator-(VectorN &&operand)  
 {  
 VectorN ret(this->size());  
 for (int i = 0; i < this->size(); i++)  
 ret[i] = (\*this)[i] - operand[i];  
 return ret;  
 }  
  
 VectorN operator\*(VectorN &&operand)  
 {  
 VectorN ret(this->size());  
 for (int i = 0; i < this->size(); i++)  
 ret[i] = (\*this)[i] \* operand[i];  
 return ret;  
 }  
  
 VectorN operator/(VectorN &&operand)  
 {  
 VectorN ret(this->size());  
 for (int i = 0; i < this->size(); i++)  
 ret[i] = (\*this)[i] / operand[i];  
 return ret;  
 }  
  
 // 结束  
  
 std::string ToString()  
 {  
 std::ostringstream ostr;  
 for (int i = 0; i < this->size(); i++)  
 {  
 ostr << std::setw(8) << std::scientific << std::right;  
 ostr << (\*this)[i] << (i == this->size() - 1 ? "\n" : ", ");  
 }  
 return ostr.str();  
 }  
  
 /\* 向量的平方模 \*/  
 VALUETYPE sqrMagnitude()  
 {  
 VALUETYPE res = 0;  
 for (auto &i : (\*this))  
 res += i \* i;  
 return res;  
 }  
  
 /\* 向量的模 \*/  
 VALUETYPE magnitude() { return sqrt(this->sqrMagnitude()); }  
  
 /\* 向量单位化 \*/  
 void Normalize()  
 {  
 VALUETYPE \_m = this->magnitude();  
 for (auto &i : (\*this))  
 i /= \_m;  
 }  
  
 /\* 返回与该向量方向同向的单位向量 \*/  
 VectorN normalized()  
 {  
 VectorN ret(\*this);  
 ret.Normalize();  
 return ret;  
 }  
  
 /\* 距离 \*/  
 static VALUETYPE Distance(VectorN &a, VectorN &b) { return (a - b).magnitude(); }  
  
 /\*向量线性插值\*/  
 static VectorN LerpUnclamped(VectorN &a, VectorN &b, VALUETYPE t) { return a + (b - a) \* t; }  
  
 /\* 点积 \*/  
  
 static VALUETYPE Dot(VectorN &&lhs, VectorN &&rhs)  
 {  
 VALUETYPE ans = 0;  
 for (auto i = 0; i < lhs.\_len; i++)  
 ans += lhs.data[i] \* rhs.data[i];  
 return ans;  
 }  
  
 /\*无符号弧度夹角\*/  
 static VALUETYPE Rad(VectorN &from, VectorN &to) { return acos(VectorN::Dot(from, to) / (from.magnitude() \* to.magnitude())); }  
  
 /\*无符号角度夹角\*/  
 static VALUETYPE Angle(VectorN &from, VectorN &to) { return Rad(from, to) \* 180.0 / PI; }  
  
 /\*返回俩向量中x的最大值和y的最大值构造而成的向量\*/  
 static VectorN Max(VectorN lhs, VectorN &&rhs)  
 {  
 for (auto &&i : range(lhs.\_len))  
 lhs.data[i] = std::max(lhs.data[i], rhs.data[i]);  
 return lhs;  
 }  
  
 /\*返回俩向量中x的最小值和y的最小值构造而成的向量\*/  
 static VectorN Min(VectorN lhs, VectorN &&rhs)  
 {  
 for (auto &&i : range(lhs.\_len))  
 lhs.data[i] = std::min(lhs.data[i], rhs.data[i]);  
 return lhs;  
 }  
  
 /\*获得vector在onNormal方向的投影\*/  
 static VectorN Project(VectorN &vector, VectorN &onNormal) { return cos(Rad(vector, onNormal)) \* vector.magnitude() \* onNormal; }  
  
};

### 矩阵

template <typename VALUETYPE = FLOAT\_>  
struct Matrix : VectorN<VectorN<VALUETYPE>>  
{  
 int ROW, COL;  
  
 std::string ToString()  
 {  
 std::ostringstream ostr;  
 ostr << "Matrix" << ROW << "x" << COL << "[\n";  
 for (auto &i : \*this)  
 ostr << '\t' << i.ToString();  
 ostr << "]";  
 return ostr.str();  
 }  
  
 Matrix(VectorN<VectorN<VALUETYPE>> &&v) : VectorN<VectorN<VALUETYPE>>(v), ROW(v.size()), COL(v.front().size()) {}  
 Matrix(VectorN<VectorN<VALUETYPE>> &v) : VectorN<VectorN<VALUETYPE>>(v), ROW(v.size()), COL(v.front().size()) {}  
  
 Matrix(int r, int c, VALUETYPE default\_val = 0) : ROW(r), COL(c)  
 {  
 this->resize(r);  
 for (r--; r >= 0; r--)  
 (\*this)[r].init(c, default\_val);  
 }  
 Matrix() = default;  
  
 /\*交换两行\*/  
 void swap\_rows(int from, int to)  
 {  
 std::swap((\*this)[from], (\*this)[to]);  
 }  
  
 /\*化为上三角矩阵\*/  
 void triangularify(bool unitriangularify = false)  
 {  
 int mx;  
 int done\_rows = 0;  
 for (int j = 0; j < COL; j++) // 化为上三角  
 {  
 mx = done\_rows;  
 for (int i = done\_rows + 1; i < ROW; i++)  
 {  
 if (fabs((\*this)[i][j]) > fabs((\*this)[mx][j]))  
 mx = i;  
 }  
 if ((\*this)[mx][j] == 0)  
 continue;  
 if (mx != done\_rows)  
 swap\_rows(mx, done\_rows);  
  
 for (int i = done\_rows + 1; i < ROW; i++)  
 {  
 VALUETYPE tmp = (\*this)[i][j] / (\*this)[done\_rows][j];  
 if (tmp != 0)  
 (\*this)[i] -= (\*this)[done\_rows] \* tmp;  
 }  
 if (unitriangularify)  
 {  
 auto tmp = (\*this)[done\_rows][j];  
 (\*this)[done\_rows] /= tmp; // 因为用了引用，这里得拷贝暂存  
 }  
 done\_rows++;  
 if (done\_rows == ROW)  
 break;  
 }  
 }  
  
 /\*化为上三角矩阵，模意义版\*/  
 void triangularify(long long mod, bool unitriangularify = false)  
 {  
 int mx;  
 int done\_rows = 0;  
 for (int j = 0; j < COL; j++) // 化为上三角  
 {  
 mx = done\_rows;  
  
 if ((\*this)[done\_rows][j] < 0)  
 (\*this)[done\_rows][j] = ((\*this)[done\_rows][j] % mod + mod) % mod;  
  
 for (int i = done\_rows + 1; i < ROW; i++)  
 {  
 if ((\*this)[i][j] < 0)  
 (\*this)[i][j] = ((\*this)[i][j] % mod + mod) % mod;  
 if ((\*this)[i][j] > (\*this)[mx][j])  
 mx = i;  
 }  
 if ((\*this)[mx][j] == 0)  
 continue;  
  
 if (mx != done\_rows)  
 swap\_rows(mx, done\_rows);  
  
 for (int i = done\_rows + 1; i < ROW; i++)  
 {  
 VALUETYPE tmp = (\*this)[i][j] \* inv((\*this)[done\_rows][j], mod) % mod;  
 if (tmp != 0)  
 {  
 (\*this)[i] -= (\*this)[done\_rows] \* tmp;  
 (\*this)[i] %= mod;  
 }  
 }  
 if (unitriangularify)  
 {  
 auto tmp = (\*this)[done\_rows][j];  
 (\*this)[done\_rows] \*= inv(tmp, mod);  
 (\*this)[done\_rows] %= mod;  
 }  
 done\_rows++;  
 if (done\_rows == ROW)  
 break;  
 }  
 }  
  
 void row\_echelonify(long long mod = 0)  
 {  
 if (mod)  
 triangularify(mod, true);  
 else  
 triangularify(true);  
 int valid\_pos = 1;  
 for (int i = 1; i < ROW; i++)  
 {  
 while (valid\_pos < COL and (\*this)[i][valid\_pos] == 0)  
 valid\_pos++;  
 if (valid\_pos == COL)  
 break;  
 for (int ii = i - 1; ii >= 0; ii--)  
 {  
 (\*this)[ii] -= (\*this)[i] \* (\*this)[ii][valid\_pos];  
 if (mod)  
 (\*this)[ii] %= mod;  
 }  
 }  
 }  
  
 /\*返回一个自身化为上三角矩阵的拷贝\*/  
 Matrix triangular(bool unitriangularify = false)  
 {  
 Matrix ret(\*this);  
 ret.triangularify(unitriangularify);  
 return ret;  
 }  
  
 /\*求秩，得先上三角化\*/  
 int \_rank()  
 {  
 int res = 0;  
 for (auto &i : (\*this))  
 res += i.any();  
 return res;  
 }  
  
 /\*求秩\*/  
 int rank() { return triangular().\_rank(); }  
  
 /\*高斯消元解方程组\*/  
 bool solve()  
 {  
 if (COL != ROW + 1)  
 throw "dimension error!";  
 triangularify();  
 // cerr << \*this << endl;  
 if (!(\*this).back().any())  
 return false;  
 for (int i = ROW - 1; i >= 0; i--)  
 {  
 for (int j = i + 1; j < ROW; j++)  
 (\*this)[i][COL - 1] -= (\*this)[i][j] \* (\*this)[j][COL - 1];  
 if ((\*this)[i][i] == 0)  
 return false;  
 (\*this)[i][COL - 1] /= (\*this)[i][i];  
 }  
 return true;  
 }  
  
 /\*矩阵连接\*/  
 void rconcat(Matrix &&rhs)  
 {  
 COL += rhs.COL;  
 for (int i = 0; i < ROW; i++)  
 {  
 (\*this)[i].rconcat(rhs[i]);  
 }  
 }  
  
 /\*左截断\*/  
 void lerase(int ctr)  
 {  
 assert(COL >= ctr);  
 COL -= ctr;  
 for (int i = 0; i < ROW; i++)  
 {  
 (\*this)[i].lerase(ctr);  
 }  
 }  
  
 /\*矩阵乘法\*/  
 Matrix dot(Matrix &&rhs, long long mod = 0)  
 {  
 if (this->COL != rhs.ROW)  
 throw "Error at matrix multiply: lhs's column is not equal to rhs's row";  
 Matrix ret(this->ROW, rhs.COL, true);  
 for (int i = 0; i < ret.ROW; ++i)  
 for (int k = 0; k < this->COL; ++k)  
 {  
 VALUETYPE &s = (\*this)[i][k];  
 for (int j = 0; j < ret.COL; ++j)  
 {  
 ret[i][j] += s \* rhs[k][j];  
 if (mod)  
 ret[i][j] %= mod;  
 }  
 }  
 return ret;  
 }  
};

### 方阵

template <typename VALUETYPE = FLOAT\_>  
struct SquareMatrix : Matrix<VALUETYPE>  
{  
 SquareMatrix(int siz, bool is\_reset = false) : Matrix<VALUETYPE>(siz, siz, is\_reset) {}  
 SquareMatrix(Matrix<VALUETYPE> &&x) : Matrix<VALUETYPE>(x)  
 {  
 assert(x.COL == x.ROW);  
 }  
 static SquareMatrix eye(int siz)  
 {  
 SquareMatrix ret(siz, true);  
 for (siz--; siz >= 0; siz--)  
 ret[siz][siz] = 1;  
 return ret;  
 }  
  
 SquareMatrix quick\_power(long long p, long long mod = 0)  
 {  
 SquareMatrix ans = eye(this->ROW);  
 SquareMatrix rhs(\*this);  
 while (p)  
 {  
 if (p & 1)  
 {  
 ans = ans.dot(rhs, mod);  
 }  
 rhs = rhs.dot(rhs, mod);  
 p >>= 1;  
 }  
 return ans;  
 }  
  
 SquareMatrix inv(long long mod = 0)  
 {  
 Matrix<VALUETYPE> ret(\*this);  
 ret.rconcat(eye(this->ROW));  
 ret.row\_echelonify(mod);  
 // cerr << ret << endl;  
 for (int i = 0; i < this->ROW; i++)  
 {  
 if (ret[i][i] != 1)  
 throw "Error at matrix inverse: cannot identify extended matrix";  
 }  
 ret.lerase(this->ROW);  
 return ret;  
 }  
};

### 二维直线

struct Line2  
{  
 FLOAT\_ A, B, C;  
 Line2(Vector2 u, Vector2 v) : A(u.y - v.y), B(v.x - u.x), C(u.y \* (u.x - v.x) - u.x \* (u.y - v.y))  
 {  
 if (u == v)  
 {  
 if (u.x)  
 {  
 A = 1;  
 B = 0;  
 C = -u.x;  
 }  
 else if (u.y)  
 {  
 A = 0;  
 B = 1;  
 C = -u.y;  
 }  
 else  
 {  
 A = 1;  
 B = -1;  
 C = 0;  
 }  
 }  
 }  
 Line2(FLOAT\_ a, FLOAT\_ b, FLOAT\_ c) : A(a), B(b), C(c) {}  
 std::string ToString()  
 {  
 std::ostringstream ostr;  
 ostr << "Line2(" << this->A << ", " << this->B << ", " << this->C << ")";  
 return ostr.str();  
 }  
 friend std::ostream &operator<<(std::ostream &o, Line2 v)  
 {  
 o << v.ToString();  
 return o;  
 }  
 FLOAT\_ k() { return -A / B; }  
 FLOAT\_ b() { return -C / B; }  
 FLOAT\_ x(FLOAT\_ y) { return -(B \* y + C) / A; }  
 FLOAT\_ y(FLOAT\_ x) { return -(A \* x + C) / B; }  
 /\*点到直线的距离\*/  
 FLOAT\_ distToPoint(Vector2 p) { return abs(A \* p.x + B \* p.y + C / sqrt(A \* A + B \* B)); }  
 /\*直线距离公式，使用前先判平行\*/  
 static FLOAT\_ Distance(Line2 a, Line2 b) { return abs(a.C - b.C) / sqrt(a.A \* a.A + a.B \* a.B); }  
 /\*判断平行\*/  
 static bool IsParallel(Line2 u, Line2 v)  
 {  
 bool f1 = round\_compare(u.B, 0.0);  
 bool f2 = round\_compare(v.B, 0.0);  
 if (f1 != f2)  
 return false;  
 return f1 or round\_compare(u.k(), v.k());  
 }  
  
 /\*单位化（？）\*/  
 void normalize()  
 {  
 FLOAT\_ su = sqrt(A \* A + B \* B + C \* C);  
 if (A < 0)  
 su = -su;  
 else if (A == 0 and B < 0)  
 su = -su;  
 A /= su;  
 B /= su;  
 C /= su;  
 }  
 /\*返回单位化后的直线\*/  
 Line2 normalized()  
 {  
 Line2 t(\*this);  
 t.normalize();  
 return t;  
 }  
  
 bool operator==(Line2 v) { return round\_compare(A, v.A) and round\_compare(B, v.B) and round\_compare(C, v.C); }  
 bool operator!=(Line2 v) { return !(\*this == v); }  
  
 static bool IsSame(Line2 u, Line2 v)  
 {  
 return Line2::IsParallel(u, v) and round\_compare(Distance(u.normalized(), v.normalized()), 0.0);  
 }  
  
 /\*计算交点\*/  
 static Vector2 Intersect(Line2 u, Line2 v)  
 {  
 FLOAT\_ tx = (u.B \* v.C - v.B \* u.C) / (v.B \* u.A - u.B \* v.A);  
 FLOAT\_ ty = (u.B != 0.0 ? -u.A \* tx / u.B - u.C / u.B : -v.A \* tx / v.B - v.C / v.B);  
 return Vector2(tx, ty);  
 }  
  
 /\*判断三个点是否共线\*/  
 static bool Collinear(Vector2 a, Vector2 b, Vector2 c)  
 {  
 Line2 l(a, b);  
 return round\_compare((l.A \* c.x + l.B \* c.y + l.C), 0.0);  
 }  
};

### 二维有向线段

struct Segment2 : Line2 // 二维有向线段  
{  
 Vector2 from, to;  
 Segment2(Vector2 a, Vector2 b) : Line2(a, b), from(a), to(b) {}  
 Segment2(FLOAT\_ x, FLOAT\_ y, FLOAT\_ X, FLOAT\_ Y) : Line2(Vector2(x, y), Vector2(X, Y)), from(Vector2(x, y)), to(Vector2(X, Y)) {}  
 bool is\_online(Vector2 poi)  
 {  
 return round\_compare((Vector2::Distance(poi, to) + Vector2::Distance(poi, from)), Vector2::Distance(from, to));  
 }  
 Vector2 &operator[](int i)  
 {  
 switch (i)  
 {  
 case 0:  
 return from;  
 break;  
 case 1:  
 return to;  
 break;  
 default:  
 throw "数组越界";  
 break;  
 }  
 }  
};

### 二维多边形

struct Polygon2  
{  
 std::vector<Vector2> points;  
  
private:  
 Vector2 accordance;  
  
public:  
 Polygon2 ConvexHull()  
 {  
 Polygon2 ret;  
 std::sort(points.begin(), points.end());  
 std::vector<Vector2> &stk = ret.points;  
  
 std::vector<char> used(points.size(), 0);  
 std::vector<int> uid;  
 for (auto &i : points)  
 {  
 while (stk.size() >= 2 and Vector2::Cross(stk.back() - stk[stk.size() - 2], i - stk.back()) <= 0)  
 {  
 // if (stk.size() >= 2)  
 // {  
 // auto c = Vector2::Cross(stk.back() - stk[stk.size() - 2], i - stk.back());  
 // cerr << "c:" << c << endl;  
 // }  
 used[uid.back()] = 0;  
 uid.pop\_back();  
 stk.pop\_back();  
 }  
  
 used[&i - &points.front()] = 1;  
 uid.emplace\_back(&i - &points.front());  
 stk.emplace\_back(i);  
 }  
 used[0] = 0;  
 int ts = stk.size();  
 for (auto ii = ++points.rbegin(); ii != points.rend(); ii++)  
 {  
 Vector2 &i = \*ii;  
 if (!used[&i - &points.front()])  
 {  
 while (stk.size() > ts and Vector2::Cross(stk.back() - stk[stk.size() - 2], i - stk.back()) <= 0)  
 {  
 used[uid.back()] = 0;  
 uid.pop\_back();  
 stk.pop\_back();  
 }  
 used[&i - &points.front()] = 1;  
 uid.emplace\_back(&i - &points.front());  
 stk.emplace\_back(i);  
 }  
 }  
 stk.pop\_back();  
 return ret;  
 }  
  
 /\*凸多边形用逆时针排序\*/  
 void autoanticlockwiselize()  
 {  
 accordance = average();  
 anticlockwiselize();  
 }  
  
 // typedef bool(Polygon2::\*comparator);  
  
 void anticlockwiselize()  
 {  
 // comparator cmp = &Polygon2::anticlock\_comparator;  
 auto anticlock\_comparator = [&](Vector2 &a, Vector2 &b) -> bool {  
 return (a - accordance).toPolarCoordinate(false).y < (b - accordance).toPolarCoordinate(false).y;  
 };  
 std::sort(points.begin(), points.end(), anticlock\_comparator);  
 // for (auto i : points)  
 // {  
 // cerr << (i - accordance).toPolarCoordinate() << "\t" << i << endl;  
 // }  
 }  
  
 Vector2 average()  
 {  
 Vector2 avg(0, 0);  
 for (auto i : points)  
 {  
 avg += i;  
 }  
 return avg / points.size();  
 }  
  
 /\*求周长\*/  
 FLOAT\_ perimeter()  
 {  
 FLOAT\_ ret = Vector2::Distance(points.front(), points.back());  
 for (int i = 1; i < points.size(); i++)  
 ret += Vector2::Distance(points[i], points[i - 1]);  
 return ret;  
 }  
 /\*面积\*/  
 FLOAT\_ area()  
 {  
 FLOAT\_ ret = Vector2::Cross(points.back(), points.front());  
 for (int i = 1; i < points.size(); i++)  
 ret = ret + Vector2::Cross(points[i - 1], points[i]);  
 return ret / 2;  
 }  
 /\*求几何中心（形心、重心）\*/  
 Vector2 center()  
 {  
 Vector2 ret = (points.back() + points.front()) \* Vector2::Cross(points.back(), points.front());  
 for (int i = 1; i < points.size(); i++)  
 ret = ret + (points[i - 1] + points[i]) \* Vector2::Cross(points[i - 1], points[i]);  
 return ret / area() / 6;  
 }  
 /\*求边界整点数\*/  
 long long boundary\_points()  
 {  
 long long b = 0;  
 for (int i = 0; i < points.size() - 1; i++)  
 {  
 b += std::\_\_gcd((long long)abs(points[i + 1].x - points[i].x), (long long)abs(points[i + 1].y - points[i].y));  
 }  
 return b;  
 }  
 /\*Pick定理：多边形面积=内部整点数+边界上的整点数/2-1；求内部整点数\*/  
 long long interior\_points(FLOAT\_ A = -1, long long b = -1)  
 {  
 if (A < 0)  
 A = area();  
 if (b < 0)  
 b = boundary\_points();  
 return (long long)A + 1 - (b / 2);  
 }  
};

## 数论

### 欧拉筛

// #define ORAFM 2333  
int prime[ORAFM + 5], prime\_number = 0, prv[ORAFM + 5];  
// 莫比乌斯函数  
int mobius[ORAFM + 5];  
// 欧拉函数  
LL phi[ORAFM + 5];  
  
bool marked[ORAFM + 5];  
  
void ORAfliter(LL MX)  
{  
 mobius[1] = phi[1] = 1;  
 for (unsigned int i = 2; i <= MX; i++)  
 {  
 if (!marked[i])  
 {  
 prime[++prime\_number] = i;  
 prv[i] = i;  
 phi[i] = i - 1;  
 mobius[i] = -1;  
 }  
 for (unsigned int j = 1; j <= prime\_number && i \* prime[j] <= MX; j++)  
 {  
 marked[i \* prime[j]] = true;  
 prv[i \* prime[j]] = prime[j];  
 if (i % prime[j] == 0)  
 {  
 phi[i \* prime[j]] = prime[j] \* phi[i];  
 break;  
 }  
 phi[i \* prime[j]] = phi[prime[j]] \* phi[i];  
 mobius[i \* prime[j]] = -mobius[i]; // 平方因数不会被处理到，默认是0  
 }  
 }  
 // 这句话是做莫比乌斯函数和欧拉函数的前缀和  
 for (unsigned int i = 2; i <= MX; ++i)  
 {  
 mobius[i] += mobius[i - 1];  
 phi[i] += phi[i - 1];  
 }  
}

### min\_25筛框架

inline void prework(LL n)  
{  
 int tot = 0;  
 for (LL l = 1, r; l <= n; l = r + 1)  
 {  
 r = n / (n / l); // 数论分块？  
 w[++tot] = n / l;  
 // g1[tot] = w[tot] % mo;  
 // g2[tot] = (g1[tot] \* (g1[tot] + 1) >> 1) % mo \* ((g1[tot] << 1) + 1) % mo \* inv3 % mo;  
 // g2[tot]--;  
 // g1[tot] = (g1[tot] \* (g1[tot] + 1) >> 1) % mo - 1;  
 valposition(n / l, n) = tot;  
 g1[tot] = n / l - 1;  
 // g2[tot] = n / l - 1;  
 }  
 for (int i = 1; i <= prime\_number; i++)  
 {  
 for (int j = 1; j <= tot and (LL) prime[i] \* prime[i] <= w[j]; j++)  
 {  
 LL n\_div\_m\_val = w[j] / prime[i];  
 if (n\_div\_m\_val)  
 {  
 int n\_div\_m = valposition(n\_div\_m\_val, n); // m: prime[i]  
 g1[j] -= g1[n\_div\_m] - (i - 1); // 枚举第i个质数，所以可以直接减去i-1，这里无需记录sp  
 }  
 // g1[j] -= (LL)prime[i] \* (g1[k] - sp1[i - 1] + mo) % mo;  
 // g2[j] -= (LL)prime[i] \* prime[i] % mo \* (g2[k] - sp2[i - 1] + mo) % mo;  
 // g1[j] %= mo,  
 // g2[j] %= mo;  
 // if (g1[j] < 0)  
 // g1[j] += mo;  
 // if (g2[j] < 0)  
 // g2[j] += mo;  
 }  
 }  
}  
// 1~x中最小质因子大于y的函数值  
inline LL S\_(LL x, int y)  
{  
 if (prime[y] >= x)  
 return 0;  
 int k = valposition(x, n);  
 // 此处g1、g2代表1、2次项  
 LL ans = (g2[k] - g1[k] + mo - (sp2[y] - sp1[y]) + mo) % mo;  
 // ans = (ans + mo) % mo;  
 for (int i = y + 1; i <= prime\_number and prime[i] \* prime[i] <= x; ++i)  
 {  
 LL pe = prime[i];  
 for (int e = 1; pe <= x; e++, pe \*= prime[i])  
 {  
 LL xx = pe % mo;  
 // 大概这里改ans？原题求p^k\*(p^k-1)  
 ans = (ans + xx \* (xx - 1) % mo \* (S\_(x / pe, i) + (e != 1))) % mo;  
 }  
 }  
 return ans % mo;  
}

### 卢卡斯定理

LL fact[LUCASM];  
  
inline void get\_fact(LL fact[], LL length, LL mo) // 预处理阶乘  
{  
 fact[0] = 1;  
 fact[1] = 1;  
 for (auto i = 2; i < length; i++)  
 fact[i] = fact[i - 1] \* i % mo;  
}  
// 需要先预处理出fact[]，即阶乘  
inline LL C(LL m, LL n, LL p)  
{  
 return m < n ? 0 : fact[m] \* inv(fact[n], p) % p \* inv(fact[m - n], p) % p;  
}  
inline LL lucas(LL m, LL n, LL p) // 求解大数组合数C(m,n) % p,传入依次是下面那个m和上面那个n和模数p（得是质数  
{  
 return n == 0 ? 1 % p : lucas(m / p, n / p, p) \* C(m % p, n % p, p) % p;  
}

### EXCRT

inline LL EXCRT(LL factors[], LL remains[], LL length) // 传入除数表，剩余表和两表的长度，若没有解，返回-1，否则返回合适的最小解  
{  
 bool valid = true;  
 for (auto i = 1; i < length; i++)  
 {  
 LL GCD = gcd(factors[i], factors[i - 1]);  
 LL M1 = factors[i];  
 LL M2 = factors[i - 1];  
 LL C1 = remains[i];  
 LL C2 = remains[i - 1];  
 LL LCM = M1 \* M2 / GCD;  
 if ((C1 - C2) % GCD != 0)  
 {  
 valid = false;  
 break;  
 }  
 factors[i] = LCM;  
 remains[i] = (inv(M2 / GCD, M1 / GCD) \* (C1 - C2) / GCD) % (M1 / GCD) \* M2 + C2; // 对应合并公式  
 remains[i] = (remains[i] % factors[i] + factors[i]) % factors[i]; // 转正  
 }  
 return valid ? remains[length - 1] : -1;  
}

### 扩欧求逆元

inline void exgcd(LL a, LL b, LL &x, LL &y)  
{  
 if (!b)  
 {  
 x = 1;  
 y = 0;  
 return;  
 }  
 exgcd(b, a % b, y, x);  
 y -= a / b \* x;  
}  
inline LL inv(LL a, LL mo)  
{  
 LL x, y;  
 exgcd(a, mo, x, y);  
 return x >= 0 ? x : x + mo;  
}

### FFT

struct Complex  
{  
 double re, im;  
 Complex(double \_re, double \_im)  
 {  
 re = \_re;  
 im = \_im;  
 }  
 Complex(double \_re)  
 {  
 re = \_re;  
 im = 0;  
 }  
 Complex() {}  
};  
Complex operator+(const Complex &c1, const Complex &c2) { return Complex(c1.re + c2.re, c1.im + c2.im); }  
Complex operator-(const Complex &c1, const Complex &c2) { return Complex(c1.re - c2.re, c1.im - c2.im); }  
Complex operator\*(const Complex &c1, const Complex &c2) { return Complex(c1.re \* c2.re - c1.im \* c2.im, c1.re \* c2.im + c1.im \* c2.re); }  
  
void FFT(complex<double> a[], LL n, LL inv, LL rev[]) // FFT系列没有外部依赖数组，不用打ifdef  
{  
 for (auto i = 0; i < n; i++)  
 if (i < rev[i])  
 swap(a[i], a[rev[i]]);  
 for (auto mid = 1; mid < n; mid <<= 1)  
 {  
 complex<double> tmp(cos(pi / mid), inv \* sin(pi / mid));  
 for (auto i = 0; i < n; i += mid << 1)  
 {  
 complex<double> omega(1, 0);  
 for (auto j = 0; j < mid; j++, omega \*= tmp)  
 {  
 auto x = a[i + j], y = omega \* a[i + j + mid];  
 a[i + j] = x + y, a[i + j + mid] = x - y;  
 }  
 }  
 }  
}  
  
LL \*FFTArrayMul(LL A[], LL B[], LL Alen, LL Blen)  
{  
 LL max\_length = max(Alen, Blen);  
 LL limit = 1;  
 LL bit = 0;  
 while (limit < max\_length << 1)  
 bit++, limit <<= 1;  
  
 auto rev = new LL[limit + 5];  
 mem(rev, 0);  
  
 for (auto i = 0; i <= limit; i++)  
 rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (bit - 1));  
  
 // auto a = new complex<double>[limit + 5];  
 // auto b = new complex<double>[limit + 5];  
 complex<double> a[limit + 5], b[limit + 5];  
  
 // for (auto i = 0; i < limit; i++)  
 // {  
 // a[i] = i >= Alen ? 0 : A[Alen - i - 1] ^ 0x30;  
 // b[i] = i >= Blen ? 0 : B[Blen - i - 1] ^ 0x30;  
 // } // 右对齐的输入方式，类似下面的大数乘法板子，答案需要去除前导零  
 for (auto i = 0; i < max\_length; i++) // 左对齐，段错误的坑，得用max\_length  
 {  
 a[i] = A[i];  
 b[i] = B[i];  
 }  
 static LL \*c = new LL[limit + 5];  
 mem(c, 0);  
  
 FFT(a, limit, 1, rev);  
 FFT(b, limit, 1, rev);  
  
 for (auto i = 0; i <= limit; i++)  
 a[i] \*= b[i];  
  
 FFT(a, limit, -1, rev); // 1是FFT变化，-1是逆变换  
 for (auto i = 0; i <= limit; i++)  
 c[i] = (LL)(a[i].real() / limit + 0.5); // +0.5即四舍五入  
 return c; // 左对齐多项式卷积结果有效位数为n+m-1  
}  
  
string FFTBigNumMul(string &A, string &B)  
{  
  
 auto max\_length = max(A.length(), B.length());  
  
 LL limit = 1;  
 LL bit = 0;  
  
 while (limit < max\_length << 1)  
 bit++, limit <<= 1;  
  
 LL rev[limit + 5] = {0};  
  
 for (auto i = 0; i <= limit; i++)  
 rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (bit - 1));  
  
 complex<double> a[limit + 5], b[limit + 5];  
  
 for (auto i = 0; i < limit; i++)  
 {  
 a[i] = i >= A.length() ? 0 : A[A.length() - i - 1] ^ 0x30;  
 b[i] = i >= B.length() ? 0 : B[B.length() - i - 1] ^ 0x30;  
 }  
  
 LL c[limit + 5] = {0};  
  
 FFT(a, limit, 1, rev);  
 FFT(b, limit, 1, rev);  
 for (auto i = 0; i <= limit; i++)  
 a[i] \*= b[i];  
 FFT(a, limit, -1, rev); // 1是FFT变化，-1是逆变换  
  
 for (auto i = 0; i <= limit; i++)  
 c[i] = (LL)(a[i].real() / limit + 0.5); // +0.5即四舍五入  
 bool zerofliter = false;  
 for (auto i = 0; i < limit; i++)  
 {  
 c[i + 1] += c[i] / 10;  
 c[i] %= 10;  
 }  
 char output[limit + 5] = {0};  
 LL outputPtr = 0;  
 // mem(output, 0);  
 for (auto i = limit; i >= 0; i--)  
 {  
 if (c[i] == 0 and zerofliter == 0) // 去前导零  
 continue;  
 zerofliter = true;  
 output[outputPtr++] = c[i] ^ 0x30;  
 }  
 string res(output);  
 if (!res.length())  
 res = string("0");  
 return res;  
}

### 拉格朗日插值

#include <algorithm>  
#include <cstdio>  
#include <cstring>  
const int maxn = 2010;  
using ll = long long;  
ll mod = 998244353;  
ll n, k, x[maxn], y[maxn], ans, s1, s2;  
ll powmod(ll a, ll x) {  
 ll ret = 1ll, nww = a;  
 while (x) {  
 if (x & 1) ret = ret \* nww % mod;  
 nww = nww \* nww % mod;  
 x >>= 1;  
 }  
 return ret;  
}  
ll inv(ll x) { return powmod(x, mod - 2); }  
int main() {  
 scanf("%lld%lld", &n, &k);  
 for (int i = 1; i <= n; i++) scanf("%lld%lld", x + i, y + i);  
 for (int i = 1; i <= n; i++) {  
 s1 = y[i] % mod;  
 s2 = 1ll;  
 for (int j = 1; j <= n; j++)  
 if (i != j)  
 s1 = s1 \* (k - x[j]) % mod, s2 = s2 \* ((x[i] - x[j] % mod) % mod) % mod;  
 ans += s1 \* inv(s2) % mod;  
 ans = (ans + mod) % mod;  
 }  
 printf("%lld\n", ans);  
 return 0;  
}

### 高斯消元

void Gauss() {  
 for(int i = 0; i < n; i ++) {  
 r = i;  
 for(int j = i + 1; j < n; j ++)  
 if(fabs(A[j][i]) > fabs(A[r][i])) r = j;  
 if(r != i) for(int j = 0; j <= n; j ++) std :: swap(A[r][j], A[i][j]);  
  
 for(int j = n; j >= i; j --) {  
 for(int k = i + 1; k < n; k ++)  
 A[k][j] -= A[k][i] / A[i][i] \* A[i][j];  
 }  
 }  
  
 for(int i = n - 1; i >= 0; i --) {  
 for(int j = i + 1; j < n; j ++)  
 A[i][n] -= A[j][n] \* A[i][j];  
 A[i][n] /= A[i][i];  
 }  
}

### 公式

卡特兰数 K(x) = C(2\*x, x) / (x + 1)

### 来自bot的球盒问题

def A072233\_list(n: int, m: int, mod=0) -> list:  
 """n个无差别球塞进m个无差别盒子方案数"""  
 mod = int(mod)  
 f = [[0] \* (m + 1)] \* (n + 1)  
 f[0][0] = 1  
 for i in range(1, n+1):  
 for j in range(1, min(i+1, m+1)): # 只是求到m了话没必要打更大的  
 f[i][j] = f[i-1][j-1] + f[i-j][j]  
 if mod: f[i][j] %= mod  
 return f  
  
def A048993\_list(n: int, m: int, mod=0) -> list:  
 """第二类斯特林数"""  
 mod = int(mod)  
 f = [1] + [0] \* m  
 for i in range(1, n+1):  
 for j in range(min(m, i), 0, -1):  
 f[j] = f[j-1] + f[j] \* j  
 if mod: f[j] %= mod  
 f[0] = 0  
 return f  
  
  
def A000110\_list(m, mod=0):  
 """集合划分方案总和，或者叫贝尔数"""  
 mod = int(mod)  
 A = [0 for i in range(m)]  
 # m -= 1  
 A[0] = 1  
 # R = [1, 1]  
 for n in range(1, m):  
 A[n] = A[0]  
 for k in range(n, 0, -1):  
 A[k-1] += A[k]  
 if mod: A[k-1] %= mod  
 # R.append(A[0])  
 # return R  
 return A[0]  
  
async def 球盒(\*attrs, kwargs={}):  
 """求解把n个球放进m个盒子里面有多少种方案的问题。  
必须指定盒子和球以及允不允许为空三个属性。  
用法：  
 #球盒 <盒子相同？(0/1)><球相同？(0/1)><允许空盒子？(0/1)> n m  
用例：  
 #球盒 110 20 5  
 上述命令求的是盒子相同，球相同，不允许空盒子的情况下将20个球放入5个盒子的方案数。"""  
 # 参考https://www.cnblogs.com/sdfzsyq/p/9838857.html的算法  
 if len(attrs)!=3:  
 return '不是这么用的！请输入#h #球盒'  
 n, m = map(int, attrs[1:3])  
 if attrs[0] == '110':  
 f = A072233\_list(n, m)  
 return f[n][m]  
 elif attrs[0] == '111':  
 f = A072233\_list(n, m)  
 return sum(f[-1])  
 elif attrs[0] == '100':  
 return A048993\_list(n, m)[-1]  
 elif attrs[0] == '101':  
 return sum(A048993\_list(n, m))  
 elif attrs[0] == '010':  
 return comb(n-1, m-1)  
 elif attrs[0] == '011':  
 return comb(n+m-1, m-1)  
 elif attrs[0] == '000': # 求两个集合的满射函数的个数可以用  
 return A048993\_list(n, m)[-1] \* math.factorial(m)  
 elif attrs[0] == '001':  
 return m\*\*n

## OTHER

// v2021.5.22 主席树更面向对象  
  
  
#include <ext/pb\_ds/tree\_policy.hpp>  
#include <ext/pb\_ds/assoc\_container.hpp>  
\_\_gnu\_pbds::tree<int, \_\_gnu\_pbds::null\_type, std::less<int>, \_\_gnu\_pbds::rb\_tree\_tag, \_\_gnu\_pbds::tree\_order\_statistics\_node\_update> TTT;  
  
// 函数不返回值可能会 RE  
// 少码大数据结构，想想复杂度更优的做法  
// 小数 二分/三分 注意break条件  
// 浮点运算 sqrt(a^2-b^2) 可用 sqrt(a+b)\*sqrt(a-b) 代替，避免精度问题  
// long double -> %Lf 别用C11 (C14/16)  
// 控制位数 cout << setprecision(10) << ans;  
// reverse vector 注意判空 不然会re  
// 分块注意维护块上标记 来更新块内数组a[]  
// vector+lower\_bound常数 < map/set/(unordered\_map)  
// map.find不会创建新元素 map[]会 注意空间  
// 别对指针用memset  
// 用位运算表示2^n注意加LL 1LL<<20  
// 注意递归爆栈  
// 注意边界  
// 注意memset 多组会T  
  
// lambda  
  
// sort(p + 1, p + 1 + n,  
// [](const point &x, const point &y) -> bool { return x.x < y.x; });  
  
// append l1 to l2 (l1 unchanged)  
  
// l2.insert(l2.end(),l1.begin(),l1.end());  
  
// append l1 to l2 (elements appended to l2 are removed from l1)  
// (general form ... TG gave form that is actually better suited  
// for your needs)  
  
// l2.splice(l2.end(),l1,l1.begin(),l1.end());  
  
//位运算函数  
//int \_\_builtin\_ffs (unsigned int x最后一位1的是从后向前第几位，1110011001000 返回4  
//int \_\_builtin\_clz (unsigned int x)前导0个数  
//int \_\_builtin\_ctz (unsigned int x)末尾0个数  
//int \_\_builtin\_popcount (unsigned int x) 1的个数  
//此外，这些函数都有相应的usigned long和usigned long long版本，只需要在函数名后面加上l或ll就可以了，比如int \_\_builtin\_clzll。  
  
//java大数  
//import java.io.\*;  
//import java.math.BigInteger;  
//import java.util.\*;  
//public class Main {  
// public static void main(String args[]) throws Exception {  
// Scanner cin=new Scanner(System.in);  
// BigInteger a;  
// BigInteger b;  
// a = cin.nextBigInteger();  
// b = cin.nextBigInteger();  
//  
// System.out.println(a.add(b));  
// }  
//}  
//  
//生成超过32767的随机数  
//unsigned seed = std::chrono::system\_clock::now().time\_since\_epoch().count();  
// mt19937 rand\_num(seed); // 大随机数  
// uniform\_int\_distribution<long long> dist(0, 1000000000); // 给定范围  
// cout << dist(rand\_num) << endl;  
  
double randreal(double begin, double end)  
{  
 static std::default\_random\_engine eng(time(0));  
 std::uniform\_real\_distribution<> skip\_rate(begin, end);  
 return skip\_rate(eng);  
}  
  
int randint(int begin, int end)  
{  
 static std::default\_random\_engine eng(time(0));  
 std::uniform\_int\_distribution<> skip\_rate(begin, end);  
 return skip\_rate(eng);  
}

石子合并 4e4

找到第一个a[i]满足a[i-1]<=a[i+1]，将他俩合并。

从第i位往前找第一个a[k]满足a[k]>刚才的合并结果。

将合并结果放在k位置之后，若无满足条件的k，放在第一个位置。若i不存在，直接合并最后两个数。

#include <bits/stdc++.h>  
using namespace std;  
#define ll long long  
const ll N=41000;  
ll n,a[N],ans,now=1,pro;  
int main()  
{  
 scanf("%lld",&n);  
 for(ll i=1;i<=n;i++) scanf("%lld",&a[i]);  
 while(now<n-1)  
 {  
 for(pro=now;pro<n-1;pro++)  
 {  
 if(a[pro+2]<a[pro]) continue;  
 a[pro+1]+=a[pro];  
 ans+=a[pro+1];ll k;  
 for(k=pro;k>now;k--) a[k]=a[k-1];   
 now++; k=pro+1;  
 while(now<k&&a[k-1]<a[k]) {a[k]^=a[k-1]^=a[k]^=a[k-1];k--;}  
 break;  
 }  
 if(pro==n-1) {a[n-1]+=a[n];ans+=a[n-1];n--;}  
 }  
 if(now==n-1) ans+=(a[n-1]+a[n]);   
 printf("%lld\n",ans);  
 return 0;  
}

### 线形基

#include<bits/stdc++.h>  
using namespace std;  
// #pragma GCC optimize(2)  
#define fi first  
#define se second  
typedef pair<int, int> pii;  
typedef long long ll;  
typedef unsigned long long ull;  
typedef long double ld;  
void io() { ios::sync\_with\_stdio(false); cin.tie(0); cout.tie(0); }  
template<typename T>  
inline void debug(T const& x) { cout << x << "\n"; }  
  
struct LinearBase {  
 const int siz=64;  
 int MN;  
 vector<ll>p, tmp;  
 bool flag = false;  
 LinearBase(){  
 p.resize(siz);  
 tmp.resize(siz);  
 MN=siz-1;  
 }  
  
 void clear()  
 {  
 // siz = MN = 0;  
 p.clear(); tmp.clear();  
 flag = false;  
 }  
  
 void resize(int size)  
 {  
 p.resize(siz);  
 tmp.resize(siz);  
 MN = siz - 1;  
 flag = false;  
 }  
  
 void insert(ll x)  
 {  
 for (int i = MN; ~i; --i) {  
 if (x & (1ll << i)) {  
 if (!p[i]) {  
 p[i] = x;  
 return;  
 }  
 else  
 x ^= p[i];  
 }  
 }  
 flag = true;  
 }  
  
 bool check(ll x)  
 {  
 for (int i = MN; ~i; i--) {  
 if (x & (1ll << i)) {  
 if (!p[i]) return false;  
 else x ^= p[i];  
 }  
 }  
 return true;  
 }  
  
 ll Qmax()  
 {  
 ll res = 0ll;  
 for (int i = MN; ~i; --i) {  
 res = max(res, res ^ p[i]);  
 }  
 return res;  
 }  
  
 ll Qmin()  
 {  
 if (flag) return 0;  
 for (int i = 0; i <= MN; ++i)  
 if (p[i]) p[i];  
 }  
  
 // void rebuild()  
 // {  
 // int cnt=0,top=0;  
 // for(int i=MN;~i)  
 // }  
  
 ll Qnth\_element(ll k)  
 {  
 ll res = 0;  
 int cnt = 0;  
 k -= flag;  
 if (!k) return 0;  
 for (int i = 0; i <= MN; ++i) {  
 for (int j = i - 1; ~j; j--) {  
 if (p[i] & (1ll << j)) p[i] ^= p[j];  
 }  
 if (p[i]) tmp[cnt++] = p[i];  
 }  
 if (k >= (1ll << cnt)) return -1;  
 for (int i = 0; i < cnt; ++i)  
 if (k & (1ll << i))  
 res ^= tmp[i];  
 return res;  
 }  
  
  
};  
  
  
  
int main()  
{  
 io();  
 int n;  
 cin >> n;  
 LinearBase lb;  
 for (int i = 0; i < n; ++i) {  
 ll \_;  
 cin >> \_;  
 lb.insert(\_);  
 }  
 cout << lb.Qmax() << "\n";  
 return 0;  
}

### Add fhq-Treap

#### 区间翻转(可以部分替代splay)

# include<iostream>  
# include<cstdio>  
# include<cstring>  
# include<cstdlib>  
using namespace std;  
const int MAX=1e5+1;  
int n,m,tot,rt;  
struct Treap{  
 int pos[MAX],siz[MAX],w[MAX];  
 int son[MAX][2];  
 bool fl[MAX];  
 void pus(int x)  
 {  
 siz[x]=siz[son[x][0]]+siz[son[x][1]]+1;  
 }  
 int build(int x)  
 {  
 w[++tot]=x,siz[tot]=1,pos[tot]=rand();  
 return tot;  
 }  
 void down(int x)  
 {  
 swap(son[x][0],son[x][1]);  
 if(son[x][0]) fl[son[x][0]]^=1;  
 if(son[x][1]) fl[son[x][1]]^=1;  
 fl[x]=0;  
 }  
 int merge(int x,int y)  
 {  
 if(!x||!y) return x+y;  
 if(pos[x]<pos[y])  
 {  
 if(fl[x]) down(x);  
 son[x][1]=merge(son[x][1],y);  
 pus(x);  
 return x;  
 }  
 if(fl[y]) down(y);  
 son[y][0]=merge(x,son[y][0]);  
 pus(y);  
 return y;  
 }  
 void split(int i,int k,int &x,int &y)  
 {  
 if(!i)  
 {  
 x=y=0;  
 return;  
 }  
 if(fl[i]) down(i);  
 if(siz[son[i][0]]<k)  
 x=i,split(son[i][1],k-siz[son[i][0]]-1,son[i][1],y);  
 else  
 y=i,split(son[i][0],k,x,son[i][0]);  
 pus(i);  
 }  
 void coutt(int i)  
 {  
 if(!i) return;  
 if(fl[i]) down(i);  
 coutt(son[i][0]);  
 printf("%d ",w[i]);  
 coutt(son[i][1]);  
 }  
}Tree;  
int main()  
{  
 scanf("%d%d",&n,&m);  
 for(int i=1;i<=n;i++)  
 rt=Tree.merge(rt,Tree.build(i));  
 for(int i=1;i<=m;i++)  
 {  
 int l,r,a,b,c;  
 scanf("%d%d",&l,&r);  
 Tree.split(rt,l-1,a,b);  
 Tree.split(b,r-l+1,b,c);  
 Tree.fl[b]^=1;  
 rt=Tree.merge(a,Tree.merge(b,c));  
 }  
 Tree.coutt(rt);  
 return 0;  
}

#### 可持久化

#include<cstdio>   
#include<cctype>   
#include<cstring>   
#include<cstdlib>   
#include<ctime>   
#include<utility>   
#include<algorithm>   
using namespace std;   
typedef pair<int,int> Pair;   
int read() {   
 int x=0,f=1;   
 char c=getchar();   
 for (;!isdigit(c);c=getchar()) if (c=='-') f=-1;   
 for (;isdigit(c);c=getchar()) x=x\*10+c-'0';   
 return x\*f;   
}   
const int maxn=5e4+5;   
const int nlogn=1.3e7+5;   
struct node {   
 int x,hp,l,r,sum,size;   
 bool rev;   
 void clear() {   
 x=hp=l=r=sum=size=rev=0;   
 }   
};   
struct TREAP {   
 int pool[nlogn];   
 int pooler;   
 node t[nlogn];   
 int now,all;   
 int root[maxn];   
 TREAP ():now(0),pooler(1) {   
 for (int i=1;i<nlogn;++i) pool[i]=i;   
 root[now]=pool[pooler++];   
 }   
 int newroot() {   
 int ret=pool[pooler++];   
 return ret;   
 }   
 int newnode(int x) {   
 int ret=pool[pooler++];   
 t[ret].hp=rand();   
 t[ret].size=1;   
 t[ret].x=t[ret].sum=x;   
 return ret;   
 }   
 void delnode(int x) {   
 t[x].clear();   
 pool[--pooler]=x;   
 }   
 void next() {   
 root[++all]=newroot();   
 t[root[all]]=t[root[now]];   
 now=all;   
 }   
 void back(int x) {   
 now=x;   
 }   
 void update(int x) {   
 t[x].sum=t[x].x+t[t[x].l].sum+t[t[x].r].sum;   
 t[x].size=t[t[x].l].size+t[t[x].r].size+1;   
 }   
 void pushdown(int x) {   
 if (!t[x].rev) return;   
 if (t[x].l) {   
 int tx=newnode(t[t[x].l].x);   
 t[tx]=t[t[x].l];   
 t[tx].rev^=true;   
 t[x].l=tx;   
 }   
 if (t[x].r) {   
 int tx=newnode(t[t[x].r].x);   
 t[tx]=t[t[x].r];   
 t[tx].rev^=true;   
 t[x].r=tx;   
 }   
 swap(t[x].l,t[x].r);   
 t[x].rev=false;   
 }   
 int merge(int x,int y) {   
 if (!x) return y;   
 if (!y) return x;   
 int now;   
 if (t[x].hp<=t[y].hp) {   
 now=newnode(t[x].x);   
 t[now]=t[x];   
 pushdown(now);   
 t[now].r=merge(t[now].r,y);   
 } else {   
 now=newnode(t[y].x);   
 t[now]=t[y];   
 pushdown(now);   
 t[now].l=merge(x,t[now].l);   
 }   
 update(now);   
 return now;   
 }   
 Pair split(int x,int p) {   
 if (t[x].size==p) return make\_pair(x,0);   
 int now=newnode(t[x].x);   
 t[now]=t[x];   
 pushdown(now);   
 int l=t[now].l,r=t[now].r;   
 if (t[l].size>=p) {   
 t[now].l=0;   
 update(now);   
 Pair g=split(l,p);   
 now=merge(g.second,now);   
 return make\_pair(g.first,now);   
 } else if (t[l].size+1==p) {   
 t[now].r=0;   
 update(now);   
 return make\_pair(now,r);   
 } else {   
 t[now].r=0;   
 update(now);   
 Pair g=split(r,p-t[l].size-1);   
 now=merge(now,g.first);   
 pushdown(now);   
 return make\_pair(now,g.second);   
 }   
 }   
 void rever(int l,int r) {   
 ++l,++r;   
 Pair g=split(root[now],l-1);   
 Pair h=split(g.second,r-l+1);   
 int want=h.first;   
 int here=newnode(t[want].x);   
 t[here]=t[want];   
 t[here].rev^=true;   
 int fi=merge(g.first,here);   
 int se=merge(fi,h.second);   
 root[now]=se;   
 }   
 int query(int l,int r) {   
 ++l,++r;   
 Pair g=split(root[now],l-1);   
 Pair h=split(g.second,r-l+1);   
 int want=h.first;   
 int ret=t[want].sum;   
 int fi=merge(g.first,want);   
 int se=merge(fi,h.second);   
 root[now]=se;   
 return ret;   
 }   
 void insert(int x) {   
 int k=newnode(x);   
 root[now]=merge(root[now],k);   
 }   
} Treap;   
int main() {   
#ifndef ONLINE\_JUDGE   
 freopen("test.in","r",stdin);   
 freopen("my.out","w",stdout);   
#endif   
 srand(time(0));   
 int n=read(),m=read();   
 for (int i=1;i<=n;++i) {   
 int x=read();   
 Treap.insert(x);   
 }   
 while (m--) {   
 int op=read();   
 if (op==1) {   
 Treap.next();   
 int l=read(),r=read();   
 Treap.rever(l,r);   
 } else if (op==2) {   
 int l=read(),r=read();   
 int ans=Treap.query(l,r);   
 printf("%d\n",ans);   
 } else if (op==3) {   
 Treap.back(read());   
 }   
 }   
 return 0;   
}

### 常见博弈

#### 巴什博弈

只有一堆n个物品，两个人轮流从这堆物品中取物，规定每次至少取一个，最多取m个。最后取光者得胜。

n%（m+1）==0必败，否则必胜

#### 威佐夫博弈

有两堆各若干个物品，两个人轮流从任意一堆中取出至少一个或者同时从两堆中取出同样多的物品，规定每次至少取一个，至多不限，最后取光者胜利。设两堆分别为n和m

较小堆\*两堆之差为（黄金分割比+1）时必败，否则比胜

int a=min(n,m);  
int b=max(n,m);  
double r=(sqrt(5.0)+1)/2;  
double c=(double)b-a;  
int temp=(int)(r\*c);  
if(temp==a)  
 败  
else  
 胜

#### nim博弈

有若干堆各若干个物品，两个人轮流从某一堆取任意多的物品，规定每次至少取一个，多者不限，最后取光者得胜。

各堆物品异或为0时必败，否则必胜

#### anti-nim博弈

有若干堆各若干个物品，两个人轮流从某一堆取任意多的物品，规定每次至少取一个，多者不限，最后取光者得败。

先手胜当且仅当 ①所有堆石子数都为1且游戏的SG值为0（即有偶数个孤单堆-每堆只有1个石子数）；②存在某堆石子数大于1且游戏的SG值不为0.

#### 阶梯博弈

地面表示第0号阶梯。每次都可以将一个阶梯上的石子向其左侧移动任意个石子，没有可以移动的空间时（及所有石子都位于地面时）输。

阶梯博弈等效为奇数号阶梯的尼姆博弈