ACM模版集（随缘更新中）

基础板子

```
#include<iostream>
#include<vector>
#include<cmath>
#include<set>
#include<bits/stdc++.h>
#define ll long long
#define ld long double
using namespace std;
#define ET '\n'
#define FOR(i,a,b) for(ll i=(a);i<=(b);i++)
#define rFOR(i,a,b) for(ll i=(a);i>=(b);i--)
//趋向for循环，一般情况别用，规定方向时会跳出的循环在这里不会跳出
#define rep(i,a,b) for(ll i=(a);i!=(b)+2*(a<b)-1;i+=2*(a<b)-1)
#define CIN(a) ll a;cin>>a
#define DCIN(a) ld a;cin>>a
#define SCIN(a) string a;cin>>a
#define CA cout<<ans<<ET
#define CY cout<<"YES"<<ET
#define CN cout<<"NO"<<ET
#define max(a,b) ((a>b)?(a):(b))
#define min(a,b) ((a<b)?(a):(b))
std::mt19937 myrand(time(0));
ll rnd(ll l, ll r) {return std::uniform_int_distribution<ll>(l, r)(myrand);}
//没有C++20就不能用ranges了//CF又可以用C++20了，封印解除
#define PP(l,r,CK) *ranges::partition_point(ranges::iota_view((l),(r)+1),(CK))
int M=1e9+7;
inline ll MO(ll x){return (x%M+M)%M;}
string ANS[2]={"No\n","Yes\n"};

void solve(){

}

int main(){
    ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
    pre();
    CIN(t);while(t--)
    solve();
    return 0;
}
```

STL

multiset

好用的东西，可以用来写大顶堆或小顶堆，其可O(logn)查询，删除，插入元素

```
#include<set>
multiset<ll,greater<ll>> a;//大顶堆
multiset<ll,less<ll>> b;//小顶堆

a.size();//返回集合大小
a.empty();//判断集合是否为空
a.begin();//指向堆顶的迭代器
a.end();//指向正向迭代的最后一个元素下一个位置
a.rbegin();//指向堆底
a.rend();//指向逆向迭代的最后一个元素的下一个元素
a.insert(123);//插入元素，会自动排序
a.find(1234);//返回一个迭代器，指向第一个被查询的元素，如果没有，则指向end()
//a.find()!=a.end()//判断集合内是否存在元素
a.count(143);//统计元素数量
a.erase(it);//按迭代器删除元素，常用搭配：a.erase(a.find(x));//注意得先判断集合内有该元素
```

算法

并查集

```
int f[N];
int find(int x){//溯源
    while(x!=f[x])x=f[x]=f[f[x]];//路径压缩
    return x;
}
void join(int u,int v){//连边
    int a1=find(u),a2=find(v);
    if(a1!=a2)f[a1]=a2;
}
```

Kruskal

```
struct node{//点类
    int u,v,w;
    bool operator<(const &node a){
        return a.w<b.w;
    }
}edge[200005];

int f[5005];
int find(int x){
    while(x!=f[x])x=f[x]=f[f[x]];
    return x;
}//并查集与路径压缩
```

```
void kruskal(){
    sort(edge,edge+m);
    int cnt=0;
    for(int i=0;i<m;i++){//多理解一下
        int u=find(edge[i].u),v=find(edge[i].v);
        if(u==v)continue;
        ans+=edge[i].w;
        f[v]=u;
        if(++cnt==n-1)break;
    }
}
```

极简二分板子（FROM jiangly）

若数列已经按某规则划分成了左右两组，则该STL可快速返回其分界点（右边那组最靠左的元素）。

注意事项：若ck函数在给定范围内均成立，则返回的是r+1的值，如果有必要，这点记得特判一下。

```
//所需库：algorithm, ranges（C++20才有的，还是得记一下标准二分板子，这个只是偷懒用的）
//返回第一个使ck(x)为假的x
//（l,r+1）因为是左开右闭，所以右端应+1
int ans = *ranges::partition_point(ranges::iota_view(l, r+1),
    [&](int v) {
        return ck(v);
    });

//使用例：
#define PP(r,l,CK) *ranges::partition_point(ranges::iota_view(l, r+1),CK)

ll ans=PP(1,n,[&](int x){
    ...
    return 1;
    ...
    return 0;//ck函数直接写这里就行
});
//注意，如果你使用#define的调用方法，那么CK块里面不能出现逗号
//不然会报错。如果你真的需要在ck函数里写逗号，请auto ck=[&](int x){...};
//再PP(1,n,ck);
```

最基础的两个线性二分STL：

```
//返回指向第一个大于v的元素的迭代器
auto mid=upper_bound(a.begin(),a.end(),v);
if(mid==a.end())//此时数组中不存在满足条件的数
else int ans=mid-a.begin();

//返回指向第一个小于v的元素的迭代器
auto mid=lower_bound(a.begin(),a.end(),v);
```

单调队列

```
struct P{
    ll i,v;
    bool operator<(const P&a)const{
        return v<a.v;//排序方式
    }
}t;
priority_queue<P> q;
FOR(i,1,n){
    cin>>a[i];
}
FOR(i,1,n){
    t.i=i;t.v=a[i];
    q.push(t);/*滑动窗口的条件↓，窗口大小为m*/
    while(!q.empty()&&q.top().i<i-m+1)q.pop();//理解一下
    if(i>=m)cout<<q.top().v<<" ";
}
```

字符串

Manacher算法

给定一个字符串，该算法可以在 $O(n)$ 的复杂度下处理出该字符串每点的最大回文半径。

```
vector<int> manacher(string s) {//返回的R数组每项减1才是答案
    string t="#";
    for(auto c:s){t+=c;t+='#';}
    int n=t.size();
    vector<int> r(n);
    for(int i=0,j=0;i<n;i++){
        if(2*j-i>=0&&j+r[j]>i)r[i]=min(r[2*j-i],j+r[j]-i);
        while(i-r[i]>=0&&i+r[i]<n&&t[i-r[i]]==t[i+r[i]])r[i]+=1;
        if(i+r[i]>j+r[j])j=i;
    }
    return r;
}
```

数据结构

树状数组

实现单点修改，区间求和

```
template<typename T>class BIT{
public:
```

```
    T tree[N];//开到最大
    inline ll lowbit(ll x){return x&-x;}
    void add(ll x,ll v){//第x个数加上v
        for(ll i=x;i<=n;i+=lowbit(i))tree[i]+=v;
    }
    T sum(ll x){//求前x个数的和
        T ans=0;
        for(ll i=x;i;i-=lowbit(i))ans+=tree[i];
        return ans;
    }
}
//使用方法：
//BIT bit;
//bit.add(i,v);
//bit.sum(i)-bit.sum(j-1);
```

线段树

区间修改$O(logN)$，区间查询$O(logN)$。

```
class SGT{
public:
    ll M;ll* tree;ll* tag;
    SGT(int n,ll M_){
        M=M_;
        tree=new ll[(n<<2)+5];
        tag=new ll[(n<<2)+5];
    }
    //线代树所维护的函数
    void modify(int p,int l,int r,ll k){//当符合枚举区间时对区间的操作
            tree[p]=k;
            //常用的还有（记得最后取模）：
            //tree[p]+=(r-l+1)*k;//区间加
            //add[p]+=k;//lazytag更新，如果有的话

            //tree[p]*=k;//区间乘
            //mul[p]*=k;

            //tree[p]=(tree[p]*km+(r-l+1)*ka)%M;//区间加和乘(要对lazytag先乘后加)
            //mul[p]*=km;mul[p]%=M;//对乘tag直接乘即可
            //add[p]*=km;add[p]%=M;//对加tag要先乘
            //add[p]+=ka'add[p]%=M;//再加

            //tree[p]=k;//区间替换
            //tag[p]=k;
    }
    void collect(int p){//向上传递子节点的结果，这里用的是区间最大值的模版
        tree[p]=max(tree[p<<1],tree[p<<1|1]);
        //常用的传递还有：
        //tree[p]=tree[p<<1]+tree[p<<1|1];//区间和
        //tree[p]=tree[p<<1]^tree[p<<1|1];//区间异或
```

```
    }

    ll *base;
    void init(int p,int l,int r){//build建树，可省（？存疑）
        tag[p]=0;//lazytag重置
        if(l==r){tree[p]=base[p];return;}
        int mid=((l+r)>>1);
        init(p<<1,l,mid);
        init(p<<1|1,mid+1,r);
        collect(p);
    }

    int L,R;
    void range(int L_,int R_){L=L_;R=R_;}

    //===================================LZ=====================================//
    void clear(int p,int l,int r){//清空区间lazytag，如果有lazytag就需要这个函数
        int mid=((l+r)>>1);
        modify(p<<1,l,mid,tag[p]);
        modify(p<<1|1,mid+1,r,tag[p]);
        tag[p]=0;//lazytag重置

}//===================================LZ=====================================//

    void update(int p,int l,int r,ll k){
        if(L<=l&&r<=R){
            modify(int p,int l,int r,ll k);
            return;
        }
        //clear(p,l,r);//如果有lazytag，就得加这句，没有lazytag就不用
        ll mid=((l+r)>>1);
        if(L<=mid)update(p<<1,l,mid,k);
        if(mid<R)update(p<<1|1,mid+1,r,k);
        collect(p);
    }
    ll query(int p,int l,int r){
        if(L<=l&&r<=R)return tree[p];
        int mid=((l+r)>>1);
        ll a=-inf,b=-inf;//这里视情况而定，设为和lazytag一样的默认值
        if(L<=mid)a=query(p<<1,l,mid);
        if(mid<R)b=query(p<<1|1,mid+1,r);
        return max(a,b);//一般是(a+b)%M，(a^b)%M这种
    }
};

//使用例：
//定义线段树类
int main(){
    ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
    CIN(m);CIN(M);ll _=m;

    int cnt=0;ll t=0;
    SGT sgt={200000,M};//以大小和模数声明一个线段树对象
    while(_--){
```

```
        char op;cin>>op;
        CIN(x);
        if(op=='A'){
            sgt.range(cnt+1,cnt+1);//设定操作范围
            sgt.update(1,1,m,(x+t)%M);//对设定好的操作范围进行更新
            cnt++;
            //cout<<cnt<<"]";
        }else{
            sgt.range(cnt-x+1,cnt);
            //FOR(i,0,cnt)cout<<sgt.tree[i]<<"]]";
            t=sgt.query(1,1,m);
            cout<<t<<ET;
        }
    }
    return 0;
}
```

经典线段树：

```
#include<iostream>
#include<cstdio>
#define ll long long
using namespace std;

#define MAXN 1000001
unsigned ll n,m,a[MAXN],ans[MAXN<<2],tag[MAXN<<2];
inline ll ls(ll x){return x<<1;}
inline ll rs(ll x){return x<<1|1;}
void scan(){
    cin>>n>>m;
    for(ll i=1;i<=n;i++)
    scanf("%lld",&a[i]);
}
inline void push_up(ll p)
{
    ans[p]=ans[ls(p)]+ans[rs(p)];
}
void build(ll p,ll l,ll r)
{
    tag[p]=0;
    if(l==r){ans[p]=a[l];return ;}
    ll mid=(l+r)>>1;
    build(ls(p),l,mid);
    build(rs(p),mid+1,r);
    push_up(p);
}
inline void f(ll p,ll l,ll r,ll k)
{
    tag[p]=tag[p]+k;
    ans[p]=ans[p]+k*(r-l+1);
}
inline void push_down(ll p,ll l,ll r)
```

```
{
    ll mid=(l+r)>>1;
    f(ls(p),l,mid,tag[p]);
    f(rs(p),mid+1,r,tag[p]);
    tag[p]=0;
}
inline void update(ll nl,ll nr,ll l,ll r,ll p,ll k)
{
    if(nl<=l&&r<=nr)
    {
        ans[p]+=k*(r-l+1);
        tag[p]+=k;
        return ;
    }
    push_down(p,l,r);
    ll mid=(l+r)>>1;
    if(nl<=mid)update(nl,nr,l,mid,ls(p),k);
    if(nr>mid) update(nl,nr,mid+1,r,rs(p),k);
    push_up(p);
}
ll query(ll q_x,ll q_y,ll l,ll r,ll p)
{
    ll res=0;
    if(q_x<=l&&r<=q_y)return ans[p];
    ll mid=(l+r)>>1;
    push_down(p,l,r);
    if(q_x<=mid)res+=query(q_x,q_y,l,mid,ls(p));
    if(q_y>mid) res+=query(q_x,q_y,mid+1,r,rs(p));
    return res;
}
int main()
{
    ll a1,b,c,d,e,f;
    scan();
    build(1,1,n);
    while(m--)
    {
        scanf("%lld",&a1);
        switch(a1)
        {
            case 1:{
                scanf("%lld%lld%lld",&b,&c,&d);
                update(b,c,1,n,1,d);
                break;
            }
            case 2:{
                scanf("%lld%lld",&e,&f);
                printf("%lld\n",query(e,f,1,n,1));
                break;
            }
        }
    }
    return 0;
}
```

```
```

珂朵莉树

当数据随机时，用于快速处理与“区间推平”有关的题。

```cpp
#define IT set<Node>::iterator
struct Node{//中规中矩的珂朵莉树节点定义
    ll l,r;
    mutable ll v;
    Node(ll l,ll r=0,ll v=0):l(l),r(r),v(v){}
    bool operator<(const Node &a)const{
        return l<a.l;
    }
};

set<Node> s;

IT split(int pos){//珂朵莉树的区间分割
    IT it=s.lower_bound(Node(pos));
    if(it!=s.end()&&it->l==pos){
        return it;
    }
    it--;
    if(it->r<pos)return s.end();
    ll l=it->l;
    ll r=it->r;
    ll v=it->v;
    s.erase(it);
    s.insert(Node(l,pos-1,v));
    return s.insert(Node(pos,r,v)).first;
}

void add(ll l,ll r,ll x){//珂朵莉树的暴力区间加
    IT itr=split(r+1),itl=split(l);//这两句是珂朵莉树经常用到的区间遍历操作
    for(IT it=itl;it!=itr;++it){
        it->v+=x;//这里可以任意更改，修改成其他的区间操作
    }
}

void assign(ll l,ll r,ll x){//珂朵莉树的暴力区间推平
    IT itr=split(r+1),itl=split(l);
    s.erase(itl,itr);
    s.insert(Node(l,r,x));
}

ll kth(ll l,ll r,ll k){//珂朵莉树求区间第k大
    vector<pair<ll,ll>> a;
    auto itr=split(r+1),itl=split(l);//先遍历一遍把区间段全取出来，然后排序
    for(IT it=itl;it!=itr;it++){
        a.push_back({it->v,it->r-it->l+1});
    }
    sort(a.begin(),a.end());
```

```
    for(auto it=a.begin();it!=a.end();it++){//再暴力求区间第k大即可
        k-=it->second;
        if(k<=0)return it->first;
    }
    return -1;
}

ll powp(ll x,ll n,ll p){//快速幂，和珂朵莉树无关
    ll ans=1;
    x%=p;
    while(n){
        if(n&1){ans*=x;ans%=p;}
        x*=x;x%=p;n>>=1;
    }
    return ans;
}

ll cal(ll l,ll r,ll x,ll y){//珂朵莉树区间幂求和（其实我个人觉得这个换成更一般的操作也行，思路都是分成几块来管理
    IT itr=split(r+1),itl=split(l);
    ll ans=0;
    for(IT it=itl;it!=itr;it++){
        ans=(ans+powp(it->v,x,y)*(it->r-it->l+1)%y)%y;
    }
    return ans;
}
//=====================你可以使用的接口如下：
s.insert(Node(i,i,a[i]));//初始化珂朵莉树，为每个节点赋初始值
add(l,r,x);//区间加
assign(l,r,x);//区间推平
kth(l,r,x);//区间第k大
cal(l,r,x,y);//区间求f(x)和
```

FHQ Treap

快速求数在数组的前驱，后继，排名（第几大），以及数组中第k大的数。同时可随意添加元素。

```
const int N=100005;
struct Treap
{
    const int INF;
    int Root,cnt;
    deque<int>del_list;
    struct Node
    {
        int ch[2],v,rnd,siz;
    }node[N];
    int newNode(int x)//申请新节点
    {
        int tmp;
```

```
        if(del_list.empty()) tmp=++cnt;
        else tmp=del_list.front(),del_list.pop_front();

node[tmp].rnd=rand(),node[tmp].v=x,node[tmp].siz=1,node[tmp].ch[0]=node[tmp].ch[1]
=0;
        return tmp;
    }
    void update(int x)//更新信息
    {
        node[x].siz=node[node[x].ch[0]].siz+node[node[x].ch[1]].siz+1;
    }
    void vsplit(int pos,int v,int &x,int &y)//按权值分裂
    {
        if(!pos)
        {
            x=y=0;
            return;
        }
        if(node[pos].v<=v) x=pos,vsplit(node[pos].ch[1],v,node[pos].ch[1],y);
        else y=pos,vsplit(node[pos].ch[0],v,x,node[pos].ch[0]);
        update(pos);
    }
    void ssplit(int pos,int k,int &x,int &y)//按size分裂
    {
        if(!pos)
        {
            x=y=0;
            return;
        }
        if(k>node[node[pos].ch[0]].siz)
            x=pos,ssplit(node[pos].ch[1],k-node[node[pos].ch[0]].siz-
1,node[pos].ch[1],y);
        else y=pos,ssplit(node[pos].ch[0],k,x,node[pos].ch[0]);
        update(pos);
    }
    int merge(int x,int y)//合并
    {
        if(!x||!y) return x+y;
        if(node[x].rnd<node[y].rnd)
        {
            node[x].ch[1]=merge(node[x].ch[1],y);
            update(x);
            return x;
        }
        node[y].ch[0]=merge(x,node[y].ch[0]);
        update(y);
        return y;
    }
    void insert(int v)//插入
    {
        int x,y;
        vsplit(Root,v,x,y);
        Root=merge(merge(x,newNode(v)),y);
    }
```

```
    void erase(int v)//删除
    {
        int x,y,z;
        vsplit(Root,v,x,z);
        vsplit(x,v-1,x,y);
        del_list.push_back(y);
        y=merge(node[y].ch[0],node[y].ch[1]);
        Root=merge(merge(x,y),z);
    }
    int pre(int v)//前驱
    {
        int x,y,cur;
        vsplit(Root,v-1,x,y);
        cur=x;
        while(node[cur].ch[1]) cur=node[cur].ch[1];
        merge(x,y);
        return node[cur].v;
    }
    int nxt(int v)//后继
    {
        int x,y,cur;
        vsplit(Root,v,x,y);
        cur=y;
        while(node[cur].ch[0]) cur=node[cur].ch[0];
        merge(x,y);
        return node[cur].v;
    }
    int get_rank(int v)//查排名
    {
        int x,y,ans;
        vsplit(Root,v-1,x,y);
        ans=node[x].siz;
        merge(x,y);
        return ans;
    }
    int kth(int k)//查排名为k的数
    {
        ++k;
        int x,y,cur;
        ssplit(Root,k,x,y);
        cur=x;
        while(node[cur].ch[1]) cur=node[cur].ch[1];
        merge(x,y);
        return node[cur].v;
    }
    Treap():INF(2147483647)//构造函数初始化
    {
        Root=cnt=0;
        insert(-INF),insert(INF);
    }
};
//使用例：
int main(){
    ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
```

```
    Treap T;//声明对象
    int n,t,x;
    cin>>n;
    while(n--){
        cin>>t>>x;
        if(t==1)cout<<T.get_rank(x)<<ET;//排名
        else if(t==2)cout<<T.kth(x)<<ET;//第x大的数
        else if(t==3)cout<<T.pre(x)<<ET;//前驱
        else if(t==4)cout<<T.nxt(x)<<ET;//后继
        else T.insert(x);//插入
    }
    return 0;
}
```

归并排序

排序，求逆序数

```
template<typename T>class MSORT{
public:
    ll nxs=0;//逆序数
    vector<T> r;//排序结果
    void reset(){
        nxs=0;
    }
    void sort(T* a,int s,int e){
        r.resize(e+1);
        msort(a,s,e);
    }
    void msort(T* a,int s,int e){
        if(s>=e)return;
        int mid=((s+e)>>1);
        msort(a,s,mid);msort(a,mid+1,e);
        int i=s,j=mid+1,k=s;
        while(i<=mid&&j<=e){
            if(a[i]<=a[j]){
                r[k]=a[i];k++;i++;
            }else{
                r[k]=a[j];k++;j++;
                nxs+=mid-i+1;
            }
        }
        while(i<=mid)r[k++]=a[i++];
        while(j<=e)r[k++]=a[j++];
        for(int i=s;i<=e;i++)a[i]=r[i];
    }
};
//使用例：
int a[100000];
...//对a数组完成输入
MSORT<int> S;
```

```
S.sort(a,1,n);
cout<<S.nxs<<ET;//输出逆序数
for(int &i:S.r)cout<<i<<" ";//输出排序结果
```

数论

分解质因数

```
vector<ll> d;
ll tmp=n;
for(ll i=2;i*i<=n;i++){
    if(tmp%i==0){
        d.push_back(i);
        while(tmp%i==0)tmp/=i;
    }
}
if(tmp>1)d.push_back(tmp);//别忘了判一下剩的tmp，通常情况下其不为1
```

容斥定理

非常简单的概念，集合学过吧，$A\bigcup B\bigcup C=A+B+C-(A\bigcap B)-(B\bigcap C)-(A\bigcap C)+(A\bigcap B\bigcap C)$，差不多就是这样的原理。

常用位运算进行集合操作，下面给出的是利用容斥定理求n以内与n不互质的数的个数的算法。$O(2^n)$（n为质因数个数）

```
ll cnt=d.size();//分解质因数的结果，不含1
ll ans=0;
for(ll i=1;i<(1<<cnt);i++){//
    ll mul=1,num=0;
    FOR(j,0,cnt-1){//
        if(i&(1<<j)){//
            num++;mul*=d[j];
        }
    }
    if(num&1)ans+=n/mul;//
    else ans-=n/mul;//
}
```

快速幂

```
ll INF = (((ull)~0)>>2);
ll powp(ll a,ll n,ll M=INF){
    ll ans=1;
    while(n){
        if(n&1){ans*=a;ans%=M;}
```

```
        a*=a;a%=M;
        n>>=1;
    }
    return ans;
}
```

模p运算

$(a+b)mod~p=(a~mod~p+b~mod~p)~mod~p$

$(a-b)mod~p=(a~mod~p-b~mod~p+p)~mod~p$（特别注意要保证结果不为负数

$(ab)mod~p=((a~mod~p)(b~mod~p))mod~p$

$(a/b)mod~p=(a*\textrm{inv}(b))mod~p$ 其中 $\textrm{inv}(b)$ 为 b 在模 p 运算下的乘法逆元。

```
ll inv(ll x){//利用费马小定理求逆元，注意，模数p不是质数的话就不存在逆元了
    return powp(b,p-2);
}
```

矩阵快速幂

其实就是照抄快速幂的逻辑，核心还是手写矩阵运算

```
ll M=10000;
class Matrix{
public:
    int n,m;
    vector<vector<ll> > a;
    Matrix(int n_,int m_){
        n=n_;m=m_;
        a=vector<vector<ll> >(n+1,vector<ll>(m+1,0));
    }
    Matrix(vector<vector<ll> > tmp){//快速创建矩阵（为什么C++98用不了brace-init啊）
        n=tmp.size();m=tmp[0].size();
        a=vector<vector<ll> >(n+1,vector<ll>(m+1,0));
        FOR(i,1,n)FOR(j,1,m)a[i][j]=tmp[i-1][j-1];
    }
    vector<ll>& operator[](ll i){//重载运算符
        return a[i];
    }
    Matrix operator*(Matrix b){//矩阵求积
        Matrix tmp(n,b.m);
        FOR(i,1,n)FOR(j,1,b.m)FOR(k,1,m){
            tmp[i][j]+=a[i][k]*b[k][j];
            tmp[i][j]%=M;
        }
        return tmp;
    }
```

```
    Matrix operator^(ll n){//快速幂
        Matrix a=*this,ans=a;ans=1;
        while(n){
            if(n&1)ans=ans*a;
            a=a*a;
            n>>=1;
        }
        return ans;
    }
    void operator=(ll n){//单位矩阵
        FOR(i,1,n)FOR(j,1,m)a[i][j]=(i==j)*n;
    }
    //====接下来是一些和快速幂无关的函数
    Matrix T(){//矩阵求转置
        Matrix tmp(m,n);
        FOR(i,1,n)FOR(j,1,m)tmp[j][i]=a[i][j];
        return tmp;
    }
    void print(){//测试输出用，可以不写
        FOR(i,1,n)FOR(j,1,m)cout<<a[i][j]<<" \n"[j==m];
    }
};
//用法举例：
Matrix a(2,2);
a[1][1]=1;a[1][2]=1;
a[2][1]=1;a[2][2]=0;
cout<<(a^n)[1][2];//求斐波那契第n项（模M)

Matrix b({  {1,2,3},
            {4,5,6} });
(b*(b.T())).print();//矩阵乘法
```

青春gcd少年不会遇到lcm学姐

求两数间最大公约数 $\gcd(x,y)$

```
ll gcd(ll a, ll b){
    if(b == 0)return a;
    return gcd(b, a % b);
}
//当然你也可以直接#include<algorithm>，然后用：
__gcd(x,y);
```

求两数间最小公倍数 $\textrm{lcm}(x,y)$

gcd 与 lcm 的关系如下：

- $ab=\gcd(a,b)\textrm{lcm}(a,b)$

所以lcm的函数可由gcd函数直接推得：

```
ll lcm(ll x,ll y){
    return x/__gcd(x,y)*y;//这里先除后乘，防止溢出
}
```

exgcd，扩展欧几里得

先了解裴蜀(贝祖)定理

裴蜀(贝祖)定理：若 a,b 是整数,且 $\gcd(a,b)=d$，那么对于任意的整数 x、y, $ax+by$ 都一定是 d 的倍数，特别地，一定存在整数 x,y，使 $ax+by=d$ 成立。 同时可知对于一般方程 $ax+by=c$ 有解的条件为 $\color{#ff6666}{c\equiv0~(mod~\gcd(a,b))}$。

拓展gcd即是要求 $ax+by=\gcd(a,b)$ 中 x、y 的整数解

设有 $ax+by=\gcd(a,b)$，显然，当 $b = 0$ 时，$\gcd(a,b) = a$, $x = 1$, $y = 任何数$。为了方便这里使 $y = 0$。当 $b \ne 0$ 时

```
ll ex_gcd(ll a, ll b, ll &x, ll &y) {
    if(!b) {
        x = 1;
        y = 0;
        return a;
    }
    ll g = ex_gcd(b, a%b, x, y);
    ll t = x;
    x = y;
    y = t - a / b * y;
    return g;//这里返回的是gcd(a,b)
}
```

组合数预处理

```
const int MAXN=2e5+5;
ll fac[MAXN],invfac[MAXN];
ll C(ll m,ll n){return (m<0||m>n)?0:MO(MO(fac[n]*invfac[m])*invfac[n-m]);}
ll powp(ll x,ll n){
    ll ans=1;
    while(n){
        if(n&1)ans=MO(ans*x);
        x=MO(x*x);n>>=1;
    }
    return ans;
}
void pre(){
    fac[0]=fac[1]=invfac[0]=invfac[1]=1;
    FOR(i,2,MAXN-1){
        fac[i]=MO(fac[i-1]*i);
        invfac[i]=powp(fac[i],M-2);
```

```
    }
}
```

TIPS&WA

这里记录收集到的一些小技巧和注意事项（大部分和STL相关）。

[TIP] 获取一个整数二进制位数

正常做法是写循环位运算，但是如果直接用log2函数，写法上会更简洁更快

```
int msb(ll x){//获取二进制位数（记得#include<cmath>）
    return x>0?log2l(x)+1:0;
}
//log这里不需要写int，因为返回值自动设为int了
```

[TIP/WA] 位运算相关

- 按位左移，正数负数最左边补符号位（除开符号位的最左边）；按位右移，正数负数最右边都补0；无符号数当正数处理；
- 含有对高达10^{18}的数进行位运算的题，在对表达式进行左移时记得开ll，比如

  ```
  1ll<<n
  ```

- 使用~进行按位取反操作，包括符号位。

  可以用~构造INF：

  ```
  ll INF=(((ull)~0)>>1);
  ```

[TIP/WA] 获取带空格的整行输入

洛谷的字符串题总是喜欢带一堆连续空格，还要求你不能忽略这些空格，这时候就需要用到$$getline(cin,t)$$了。 (什么？不喜欢iostream？那去这里选一个你喜欢的款式：C/C++如何整行输入，我就不一一列举了)

[WA] sort函数与string数组

两个string之间的比较也是要花时间的，若数组大小为 n，string长度为 m，则sort(a,a+n)的时间复杂的是 $O(nmlogn)$

[WA] for(auto &i:a)问题

在for(auto &i:a)循环体中对i修改直接反映于原迭代体a，但是不能在这种循环体中往a继续添加元素，不然会报错。

[TIP] nth_element()与sort()

二者都可以求数组中第 k 大，sort得先对数据进行 $O(n\log n)$ 的排序，再a[k]调用，而nth_element()则只需花 $O(n)$ 的时间进行部分排序，即可使得数组中第 k 大的数就位，且其左边的数都比它小，右边的数都比它大

[TIP] 和python类似的多行字符串输入

使用R"(多行内容)"，即可完成，多行字符串的存储。和python的 '''多行内容'''差不多。

[WA] 取模问题

有时候可能会遇到一个式子内的取模符号太多，导致很容易丢失重要信息，比如该模的式子没有取模：

```
cout<<(((pre[n]-(ans>0)*ans)%M+(ans>0)*(ans*powp(2,k)%M)%M)%M+M)%M<<ET;
//某场div2上，有个ans忘记取模了，导致一直WA2，怎么也调不出问题，赛后补上这个%M就AC了：
cout<<(((pre[n]-(ans>0)*ans)%M+(ans>0)*((ans%M)*powp(2,k)%M)%M)%M+M)%M<<ET;
```

这时候可以考虑开一个函数来处理取模，化繁为简：

```
inline ll MO(ll x){return (x%M+M)%M;}
cout<<MO(MO(pre[n]-(ans>0)*ans)+(ans>0)*MO(MO(ans)*MO(powp(2,k))))<<ET;
//虽然看着好像差不了多少的样子，但是起码不用再受到%M+M)%M的折磨了
```

> 前面的区域以后再来探索吧！