## 优化模板

```
ll read()
{
    ll x=0,f=1;char ch=getchar();
    while(ch<&apos;0&apos;||ch>&apos;9&apos;){if(ch==&apos;-&apos;)f=-
1;ch=getchar();}
    while(ch<=&apos;9&apos;&&ch>=&apos;0&apos;){x=x*10+ch-
&apos;0&apos;;ch=getchar();}
    return f*x;
}
// 快读
ios::sync_with_stdio(false), cin.tie(nullptr), cout.tie(nullptr);
// 二分
lower_bound();     // 返回范围内第一个不小于 val 的位置
upper_bound();     // 返回范围内第一个大于 val 的位置
// 如果该序列内没有 val，那么上面二者是相等的
// 这两个函数只能在已经排好序的序列中使用
```

## 拓扑排序

```
int in[101];  // 描述入度
int n;
int a[101];  // 用来存拓扑序
vector<int> g[101];
bool bfs()
{
    int tot = 0;
    queue<int> q;
    for(int u=1;u<=n;u++)
        if(!in[u])  q.push(u);   // 入度为 0 的点入队
    while(!q.empty())
    {
        int u = q.front();
        q.pop();
        a[++tot] = u;  // 一般以出队序为拓扑序
        for(auto v:g[u])
        {
            in[v]--;
            if(!in[v])  q.push(v);
        }
    }
    // 如果一定为 DAG，则下面的代码不需要
    // 用于判定是否是 DAG
    if(tot==n)
        return true;   // 是 DAG
    else    return false;   // 不是 DAG
}
```

## 最短路

Dijkstra 算法

```
struct node{
    ll dis;    // 存点 u 以及他的最短路长度
    int u;
    bool operator>(const node& a) const { return dis > a.dis; };  // 重载运算符
};
priority_queue<node,vector<node>,greater<node>> q;
bool flag[1000005];    // 初始化为 false，起初全在 T 集合中
void dijkstra(int s)
{
    memset(dis,0x3f,sizeof(dis));
    dis[s] = 0;     // 初始化
    q.push({0,s});
```

```
    while(!q.empty())
    {
        int u = q.top().u;
        q.pop();
        if(flag[u]) continue;   // 点u已在集合S中了
        flag[u] = true;   // 先放到集合S中
        // 再对相邻点进行松弛
        for(auto ed:e[u])
        {
            int v=ed.to,w=ed.w;
            if(dis[v]>dis[u]+w)
            {
                dis[v] = dis[u] + w;
                q.push({dis[v],v});   // 没变小千万不要放
            }
        }
    }
}
```

SPFA

```
struct edge {
  int v, w;
};

vector<edge> e[maxn];
int dis[maxn], cnt[maxn], vis[maxn];
queue<int> q;

bool spfa(int n, int s) {
  memset(dis, 63, sizeof(dis));
  dis[s] = 0, vis[s] = 1;
  q.push(s);
  while (!q.empty()) {
    int u = q.front();
    q.pop(), vis[u] = 0;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        cnt[v] = cnt[u] + 1;  // 记录最短路经过的边数
        if (cnt[v] >= 2*n) return true;    // 为什么用2*n?  在有0环的情况下，可能会出现长度大于n的最短路，适当改大一点可以通过大部分数据
        // 在不经过负环的情况下，最短路至多经过 n - 1 条边
        // 因此如果经过了多于 n 条边，一定说明经过了负环
        if (!vis[v]) q.push(v), vis[v] = 1;
      }
    }
  }
  return false;
}
```

Floyd

```
void floyd()
{
    for(int k=1;k<=n;k++)
        for(int x=1;x<=n;x++)
            for(int y=1;y<=n;y++)
                dp[x][y] = min(dp[x][y],dp[x][k]+dp[k][y]);
}
```

**最多经过 k 条路径的最短路**

设起点为 s，终点为 t，这个问题可以简单用 DP 解决：记 f(u, k) 表示从结点 u 出发，经过不超过 k 条边到达终点 t 的最短路径长度。转移时枚举 u 的出边 (u, v)，得到 f(u, k) = min{ f(v, k

- 1) + w(u, v) }。边界是 f(t, k) = 0，f(u, 0) = +∞（u ≠ t），答案为 f(s, k)
时间复杂度 O( |E|·k )

**LCA 朴素算法**

```
void bfs(int rt)   // 预处理dep,根节点的深度为0, O(n)
{
    queue<int> q;
    q.push(rt);
    dep[rt]=0;
    while(!q.empty())
    {
        int u=q.front();q.pop();
        for(auto v:tr[u])
        {
            if(v!=fa[u])
            {
                q.push(v);
                dep[v]=dep[u]+1;
            }
        }
    }
    return;
}
// 朴素的lca , dep要用bfs预处理
int lca(int a,int b)
{
    if(dep[a]<dep[b])   swap(a,b);
    while(dep[a]>dep[b])    a=fa[a];  // 直到dep[a]=dep[b]
    while(a!=b){a=fa[a];b=fa[b];}
    return a;
}
```

## 数论相关

**拓展欧几里得**

以下代码可以求不定方程 a ∗ x+b ∗ y=1a*x+b*y=1a ∗ x+b ∗ y=1 的一组特解 (如果有解的话)

```
void Exgcd(ll a, ll b, ll &x, ll &y) {
        if (!b) x = 1, y = 0;
        else Exgcd(b, a % b, y, x), y -= a / b * x;
}
// 调用实例
int main(void)
{
    cin>>a>>b;
    ll x,y;     // 用来存答案
    Exgcd(a,b,x,y);
    cout<<x<<y<<endl;
    return 0;
}
```

对于不定方程 a ∗ x+b ∗ y=gcd(a,b)a*x+b*y=gcd(a,b)a ∗ x+b ∗ y=gcd(a,b) 用 a′=agcd(a,b),b′=bgcd(a,b)a'=\frac{a}{gcd(a,b)},b'=\frac{b}{gcd(a,b)}a′=gcd(a,b)a,b′=gcd(a,b)b 代换即可转换为以上方程

**逆元和快速幂**

```
ll fpm(ll x,ll power,ll mod)    // 求逆元函数,费马小定理+快速幂
{
    x=x%mod;
    ll ans = 1;
    for (; power; power >>= 1, x=x*x%mod)
        if(power & 1) ans=ans*x%mod;
    return ans;
```

```
}
int main()
{
        ll x = fpm(a, p - 2, p); //x为a在mod p意义下的逆元
}
```

线性区间的逆元：

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn=3e6+5;
ll n,p;
// inverse--逆元//
ll inv[maxn];
int main(void)
{
    cin>>n>>p;
    inv[1]=1;
    for(ll i=2;i<=n;i++)
        inv[i]=((p-p/i)*inv[p-(p/i)*i])%p;
    for(ll i=1;i<=n;i++)
        printf("%d\n",inv[i]);
    return 0;
}
```

**Lucas 定理**

```
// Lucas 定理 //
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const ll maxn=1e5+4;
ll n,m,T,p;
// 阶乘 factory //
ll fac[maxn];
void pre()   // 预处理阶乘
{
    fac[0]=1;   // 一定要从0开始，血的教训
    for(ll i=1;i<=p;i++)    fac[i]=fac[i-1]*i%p;   // p-1之后都是0
    return;
}
ll fpm(ll x,ll power,ll mod)    // 求逆元函数,费马小定理+快速幂
{
    x=x%mod;
    ll ans = 1;
    for (; power; power >>= 1, x=x*x%mod)
        if(power & 1) ans=ans*x%mod;
    return ans;
}
ll C(ll n,ll m){return n<m?0:fac[n]*fpm(fac[m],p-2,p)%p*fpm(fac[n-m],p-2,p)%p;}  // n个物品取m个
ll Lucas(ll n,ll m,ll p){return !m?1:C(n%p,m%p)*Lucas(n/p,m/p,p)%p;}
int main(void)
{
    scanf("%lld",&T);
    while(T--)
    {
        scanf("%lld %lld %lld",&n,&m,&p);
        pre();
        printf("%lld\n",Lucas(n+m,n,p));
    }
    return 0;
}
```

**递归实现 FFT(常数较大)**

```
typedef long long ll;
typedef complex<double> CP;
const ll maxn=1<<20;
const CP I(0,1);     // 虚数单位
const double PI=acos(-1);   // 常数PI
// FFT //
// 时间复杂度O(nlogn) //
// n=2^k ,若不足则补齐，否则该算法不成立 //
CP tmp[maxn];
void _FFT(CP* f,ll n,ll rev)
{
    if(n==1)  return;   // 长度为1，无需操作，直接返回
    for(ll i=0;i<n;++i) tmp[i]=f[i];
    // 偶数放左边，奇数放右边 //
    for(ll i=0;i<n;++i)
    {
        if(i&1) f[n/2+i/2]=tmp[i];
        else    f[i/2]=tmp[i];
    }
    // 递归DFT
    _FFT(f,n/2,rev);_FFT(f+n/2,n/2,rev);
    // cur当前的乘数因子 , step为本原单位根
    CP cur(1,0),step(cos(2*PI/n),rev*sin(2*PI/n));
    for(ll k=0;k<n/2;++k)
    {
        tmp[k]=f[k]+f[n/2+k]*cur;
        tmp[k+n/2]=f[k]-f[n/2+k]*cur;
        cur*=step;
    }
    for(ll i=0;i<n;i++)    f[i]=tmp[i];
    return;
}
// rev=1;DFT & rev=-1;IDFT//
// n=2^k ,若不足则补齐，否则该算法不成立 //
void FFT(CP* f,ll n,ll rev)
{
    _FFT(f,n,rev);
    if(rev==-1) for(ll i=0;i<n;i++) f[i]*=(CP)(1.0/n);
    return;
}
// 该算法的辅助函数 2^k严格大于n，n为f的最高次数 //
ll log2ceil(ll n){ll cnt=0;for(ll i=1;i<=n;i=i<<1)++cnt;return cnt;}

// 调用方式 ,求g*h并存到f中 //
CP f[maxn],g[maxn],h[maxn];
ll dg,dh;
void mul(CP *f,CP* g,CP* h,ll dg,ll dh)
{
    ll n=1<<log2ceil(dg+dh);
    FFT(g,n,1);FFT(h,n,1);
    for(ll i=0;i<n;i++) f[i]=g[i]*h[i];
    FFT(f,n,-1);
}
mul(f,g,h,dg,dh);
```

## 数据结构

**ST 表**

预处理 + 查询 （无法更新）ST 表 的实现（递推） -> O(nlogn)

ST 表 每次查询 -> O(1)

f[i][j]表示区间[i,i+2j−1]的最大值。f[i][j]表示区间[i,i+2^j-1]的最大值。f[i][j]表示区间[i,i+2j−1]的最大值。 转移方程：f[i][j]=max(f[i][j−1],f[i+2j−1][j−1])转移方程：f[i][j]=max(f[i][j-1],f[i+2^{j-1}][j-1])转移方程：f[i][j]=max(f[i][j−1],f[i+2j−1][j−1])

```
#include <bits/stdc++.h>
using namespace std;
const int logn = 21;
const int maxn = 2000001;
int f[maxn][logn + 1], Logn[maxn + 1];

int read() {  // 快读
  char c = getchar();
  int x = 0, f = 1;
  while (c < &apos;0&apos; || c > &apos;9&apos;) {
    if (c == &apos;-&apos;) f = -1;
    c = getchar();
  }
  while (c >= &apos;0&apos; && c <= &apos;9&apos;) {
    x = x * 10 + c - &apos;0&apos;;
    c = getchar();
  }
  return x * f;
}

void pre() {  // 准备工作，初始化
  Logn[1] = 0;
  Logn[2] = 1;
  for (int i = 3; i < maxn; i++) {
    Logn[i] = Logn[i / 2] + 1;
  }
}

int main() {
  int n = read(), m = read();
  for (int i = 1; i <= n; i++) f[i][0] = read();
  pre();
// 实现 ST 表
  for (int j = 1; j <= logn; j++)
    for (int i = 1; i + (1 << j) - 1 <= n; i++)
      f[i][j] = max(f[i][j - 1], f[i + (1 << (j - 1))][j - 1]);  // ST 表具体实现，递推
// 查询
  for (int i = 1; i <= m; i++) {
    int x = read(), y = read();
    int s = Logn[y - x + 1];
    printf("%d\n", max(f[x][s], f[y - (1 << s) + 1][s]));
  }
  return 0;
}
```

**左偏树**

```
#include<bits/stdc++.h>
using namespace std;
const int maxn=1e7+5;
int n;
// 左偏树 leftist tree / leftist heap //
struct lheap{
    int val;  // 权值
    int fa;   // father，根节点的父节点为 0
    int ls,rs;   // left son & right son ->  没有即为 0(若用指针表示就是 NULL)
    int dist;   // 距离，0 号节点初始化为-1，其余初始化为 0 或不初始化均可
}tr[maxn];
// 合并操作
int Merge(int x,int y)
{
    if(!x||!y)  return x+y;  // 返回 x，y 中的非零者或 0
    if(tr[x].val>tr[y].val) swap(x,y);   // 以小根堆举例，此处保证 x 节点的权值比 y 节点的权值小，然后把 y 插入 x
    int &ur=tr[x].rs,&ul=tr[x].ls;
    ur=Merge(ur,y);    // 为什么要与柚子树合并？ 因为 dist[rs]<=dist[ls],这是左偏树的性质,该性质保证了合并时的复杂度
```

```
    tr[ur].fa=x;   // 不能忘记，因为合并的时候，可能交换过左右子树
    // 合并完了，看看是否还符合左偏树的结构，调整结构
    if(tr[ur].dist>tr[ul].dist) swap(ur,ul);
    tr[x].dist=tr[ur].dist+1;   // 更新该节点的距离
    return x;  // 返回该节点
}
// 插入操作
// 一个节点x 插入以root为根的左偏树，可以把单节点x看做一棵左偏树，然后合并
void Insert(int root,int x)
{
    Merge(root,x);
    return;
}
// 删除根节点
int Erase(int x)
{
    int ans=tr[x].val;
    int ur=tr[x].rs,ul=tr[x].ls;
    tr[x].val=-1;   // 用一个值表式该点未初始化，或单独用flag数组表示也可以
    tr[x].ls=0,tr[x].rs=0;
    int r=Merge(ur,ul);   // 返回新的根节点
    tr[r].fa=0;
    return ans;
}
// 删除任意节点
void Delete(int x)
{
    int fa=tr[x].fa;
    int temp=Merge(tr[x].rs,tr[x].ls);
    tr[x].val=-1,tr[x].rs=0,tr[x].ls=0;
    tr[temp].fa=fa;
    int &ur=tr[fa].rs,&ul=tr[fa].ls;
    (x==ur)?ur=temp:ul=temp;    // 看看x是左儿子还是右儿子
    tr[fa].dist=tr[tr[fa].rs].dist+1;
    // 向上维护左偏性质，直到根节点或左偏性质不再被破坏 //
    while(fa&&tr[tr[fa].rs].dist<=tr[tr[fa].ls].dist)   // 当前节点不是根节点（注：根节点的父节点为0）
    {
        swap(tr[fa].rs,tr[fa].ls);
        tr[fa].dist=tr[tr[fa].rs].dist+1;  // 更新dist
        // 向上维护
        fa=tr[fa].fa;
    }
    return;
}
// 建树操作:暴力插入 复杂度O(nlogn) //
// 前置条件，tr[1]~tr[n]的权值已经初始化完毕，只是没有连接起来 //
void Build(int n)   // 参数也可以改成一个数组或其他容器
{
    int root=1;
    for(int i=2;i<=n;i++)
        root=Merge(root,i);
    return;
}
```

**线段树**

```
// 下放lazy tag给子节点
void pushdown(int cur,int cl,int cr,int mid)
{
    tree[2*cur].tag += tree[cur].tag;     // 左子树更新
    tree[2*cur].sum += (mid-cl+1)*tree[cur].tag;
    tree[2*cur+1].tag += tree[cur].tag;   // 右子树更新
    tree[2*cur+1].sum += (cr-mid)*tree[cur].tag;
    tree[cur].tag = 0;       // cur标签更新
    return;
}
```

```
// 区间更新(对区间进行同一种操作) -> 类似于区间查询
// 此处以 全部加k 举例
void update(int cl,int cr,int cur,int vl,int vr,int k)
{
    if(cr<vl||cl>vr)    return;   // 区间无交集，直接返回
    if(vl<=cl&&cr<=vr)   // 如果当前区间被包含在修改区间，进行修改
    {
        tree[cur].sum += (cr-cl+1)*k;
        tree[cur].tag += k;
        return;
    }
    // 如果当前区间与修改区间有交集且不被包含于修改区间内
    int mid = ((cr-cl)>>1)+cr;
    if(tree[cur].tag && cl!=cr) pushdown(cur);    // 如果当前节点 懒标签不为空 且 不是树叶
-> 下放懒节点

    if(vl<=mid) update(cl,mid,2*cur,vl,vr,k);
    if(vr>mid) update(mid+1,cr,2*cur+1,vl,vr,k);
    tree[cur].sum = tree[2*cur].sum + tree[2*cur+1].sum;   // 懒节点不在这一层，这一层的
更新必不可少
    return;
}
// 带pushdown的查询
int query(int cur,int cl,int cr,int vl,int vr)
{
    if(vl>cr||vr<cl)    return 0;
    if(vl<=cl&&cr<=vr)  return tree[cur].sum;
    int mid = cl + ((cr-cl)>>1);
    if(tree[cur].tag && cl!=cr) pushdown(cur,cl,cr,mid);
    return query(2*cur,cl,mid,vl,vr)+query(2*cur+1,mid+1,cr,vl,vr);
}

void tree_build(int l,int r,int root)
{
    if(l==r)
    {
        tree[root].sum = a[l];
        return;
    }
    int mid=l+((r-l)>>1);
    tree_build(l,mid,root*2);   // 左子树
    tree_build(mid+1,r,root*2+1);  // 右子树
    tree[root].sum = tree[root*2].sum + tree[root*2+1].sum;
    return;
}
```

## 决策单调 DP

```
// 分治法 示例代码 //
int s[N];
void solve(int l,int r,int L,int R)   // [l,r]是待测区间 , [L,R]是搜索区间
{
    if(l>r) return;
    int mid = l +(r-l>>1);
    int id = -1;
    for(int i=L;i<=min(R,mid-1);i++)
        if(id==-1||(double)a[i]+sqr[mid-i]<(double)a[id]+sqr[mid-id])  id=i;   // 找
到mid的最佳决策点
    s[mid] = id;
    solve(l,mid-1,L,id);
    solve(mid+1,r,id,R);
}
```