Misc 杂项算法

莫队

普通莫队

将询问离线，并按照左端点块号为第一关键字，右端点为第二关键字进行排序。

设区间长度 n，询问次数 q，块长 b，块数 $\frac{n}{b}$，那么左指针移动的次数最多为 bq，右指针移动次数最多为 $\frac{n}{b}n$，令 $bq=n\frac{n}{b}$ 时最优，即 $b=\sqrt{\frac{n^2}{q}}$。

时间复杂度 $\mathcal{O}(n\sqrt{q}+q\log q)$。

```
namespace mo {
    constexpr int N=1e5+10,Q=1e5+10,block=320;
    using Query=tuple<int,int,int>;
    vector<Query> query;
    int ans[Q];

    void solve() {
        auto getid=[](int x) {
            return x/block;
        };

        sort(query.begin(),query.end(),[&](const Query &x,const Query &y) {
            const auto &[l,r,_]=x;
            const auto &[L,R,__]=y;
            if(getid(l)!=getid(L)) return getid(l)<getid(L);
            return getid(l)&1?r<R:r>R;
        });

        int l=1,r=0,res=0;

        auto add=[&](int idx) {

        };

        auto del=[&](int idx) {

        };

        for(const auto &[L,R,id]:query) {
            while(l>L) add(--l);
            while(r<R) add(++r);
            while(l<L) del(l++);
            while(r>R) del(r--);
            // TODO ans[id]=res;
        }
    }
}
```

trick

可以用奇偶化排序优化莫队的指针移动次数，即左端点所在块为奇数时右端点排升序，反之按降序排序。这样指针在处理奇数块时大概率滚到较右边，然后处理偶数块时刚好能从右侧一直滚回左侧。这个优化较为玄学，有一定概率能优化常数（也可能变慢）。不能用于回滚莫队。

```
return getid(l)&1?r<R:r>R;
```

关于 l, r 指针移动顺序，应先拓展区间，然后收缩区间，防止维护的区间长度变为负数。

回滚莫队

也叫只增莫队，用来对付只可加不可减的信息维护，比如 max。

算法流程：

- 如果当前询问的左端点块号与之前一次不同，设当前块右端点为 rbd，暴力将 l 拉到 $rbd+1$，将 r 拉到 rbd，将维护的信息重置到初始状态。注意，因为已经排好序，左端点必定会在 $rbd+1$ 左侧，但 r 则不一定。
- 如果当前询问仅在一个块之内
  - 备份答案（实际上，此时答案必定为初始状态）
  - 暴力统计区间信息，并计算答案
  - 暴力删除区间信息，回滚答案（需要保证答案恢复到初始状态）
- 如果询问跨块
  - 首先滚右端点
  - 备份答案，然后滚左端点，计算答案
  - 将左端点回滚至 $rbd+1$，回滚答案。

复杂度同普通莫队，但常数更大，$\mathcal{O}(n\sqrt{q}+q\log q)$。

```
namespace mo {
    constexpr int N=1e5+10,Q=1e5+10,block=320;
    using Query=tuple<int,int,int>;
    vector<Query> query;
    int ans[Q];

    void solve(int n) {
        auto getid=[](int x) {
            return x/block;
        };
        auto getr=[&](int id) {
            return min(id*block+block-1,n);
        };

        sort(query.begin(),query.end(),[&](const Query &x,const Query &y) {
            const auto &[l,r,_]=x;
            const auto &[L,R,__]=y;
            if(getid(l)!=getid(L)) return getid(l)<getid(L);
            return r<R;
```

```
        });

        int l=1,r=0,lastid=-1,rbd=0;

        auto add=[&](int idx) {

        };

        auto del=[&](int idx) {

        };

        auto reset=[&]() {
            while(r<rbd) add(++r);
            while(r>rbd) del(r--);
            while(l<=rbd) del(l++);
            // res=bak=0;
        };

        for(const auto &[L,R,id]:query) {
            if(getid(L)!=lastid) {
                lastid=getid(L);
                rbd=getr(lastid);
                reset();
            }

            if(getid(L)==getid(R)) {
                // bak=res;
                for(int i=L;i<=R;i++) add(i);
                // ans[id]=res;
                for(int i=L;i<=R;i++) del(i);
                // res=bak;
            }
            else {
                while(r<R) add(++r);
                // bak=res;
                while(l>L) add(--l);
                // ans[id]=res;
                while(l<=rbd) del(l++);
                // res=bak;
            }
        }
    }
}
```

带修莫队

通过增加一维时间维，就可以让莫队处理带修改的问题。

将时间设定为进行修改的次数，初始时间设为0，即0次修改。然后将所有询问按照左端点块号为第一关键字，右端点块号为第二关键字，时间为第三关键字排序。

算法流程和普通莫队类似：

- 首先滚左右端点
- 再对齐时间：
  - 若当前修改不在询问区间内，则仅需修改
  - 否则，修改的同时更新维护的区间信息

设区间长度为 n，询问数为 q，修改数为 m，块长为 b，块数为 $\frac{n}{b}$。左右指针移动次数均为 bq，时间指针移动次数为 $(\frac{n}{b})^2m$，当块长取 $\frac{n^{\frac{2}{3}}m^{\frac{1}{3}}}{q^{\frac{1}{3}}}$ 时得到最优复杂度 $O(n^{\frac{2}{3}}q^{\frac{2}{3}}m^{\frac{1}{3}}+q\log q)$ 。若令 n,q,m 同阶，则 $b=n^{\frac{2}{3}}$ ，时间复杂度 $O(n^{\frac{5}{3}}+q\log q)$ 。

```
namespace mo {
    constexpr int N=1e5+10,Q=1e5+10,block=2155;
    using Query=tuple<int,int,int,int>;
    vector<Query> query;
    int ans[Q];

    void solve() {
        auto getid=[](int x) {
            return x/block;
        };

        sort(query.begin(),query.end(),[&](const Query &x,const Query &y) {
            const auto &[l,r,t,_]=x;
            const auto &[L,R,T,__]=y;
            if(getid(l)!=getid(L)) return getid(l)<getid(L);
            if(getid(r)!=getid(R)) return getid(r)<getid(R);
            return t<T;
        });

        int l=1,r=0,tm=0;

        auto add=[&](int idx) {

        };

        auto del=[&](int idx) {

        };

        auto modify=[&](int t) {
            // int idx=pos[t];
            // if(idx>=l&&idx<=r) del(idx);
            // swap(w[idx],nw[t]);
            // if(idx>=l&&idx<=r) add(idx);
        };

        auto rollback=[&](int t) {
            modify(t);
        };
```

```
        for(const auto &[L,R,T,id]:query) {
            while(l>L) add(--l);
            while(r<R) add(++r);
            while(l<L) del(l++);
            while(r>R) del(r--);
            while(tm<T) modify(++tm);
            while(tm>T) rollback(tm--);
            // TODO ans[id]=res;
        }
    }
}
```

trick

在进行修改后，可以将修改值置成反效果，这样在撤销修改时就可以复用修改的代码。例如，如果是单点赋值的修改，我们可以swap数组的值和修改的值，之后`rollback`直接调用`modify`即可。

```
auto modify=[&](int t) {
    int idx=pos[t];
    if(idx>=l&&idx<=r) del(idx);
    swap(w[idx],nw[t]);
    if(idx>=l&&idx<=r) add(idx);
};

auto rollback=[&](int t) {
    modify(t);
};
```

树上莫队

通过 dfs 序将一棵树拍平成一个序列，我们就可以用莫队来解决树上路径询问的问题，序列长度为节点数x2。

用 $first, last$ 表示 u 在 dfs 序出现的前后两个位置。

设 $first_u < first_v$，则 u, v 间路径可以转化为：

- 若 $lca(u,v)=u$，那么询问区间为 $[first_u, first_v]$。
- 反之询问区间为 $[last_u, first_v]$，并加上父节点 $lca(u,v)$。

用 odd 表示节点出现次数的奇偶性，若一个点出现在树上路径上，则 $odd=1$，反之 $odd=0$。

之后使用普通莫队的做法即可，块长取 $b=2\sqrt{\frac{n^2}{q}}$ 最优。

复杂度 $\mathcal{O}(n\sqrt{q}+q\log q)$，由于 n 翻倍且需要求 lca，常数较大。

```
namespace mo {
    constexpr int N=5e4+10,Q=1e5+10,block=320;
    using Query=tuple<int,int,int,int>;
```

```
    vector<Query> query;
    vector<int> adj[N];
    int uid[N<<1],first[N],last[N],idx;
    bool odd[N];
    int ans[Q];

    namespace hpd {
        int dep[N],sz[N],top[N],p[N],hch[N];

        void dfs1(int u,int fa,int d) {
            dep[u]=d,p[u]=fa,sz[u]=1;
            for(int v:adj[u]) {
                if(v==fa) continue;
                dfs1(v,u,d+1);
                sz[u]+=sz[v];
                if(sz[hch[u]]<sz[v]) hch[u]=v;
            }
        }

        void dfs2(int u,int t) {
            top[u]=t;
            if(!hch[u]) return;
            dfs2(hch[u],t);
            for(int v:adj[u])
                if(v!=p[u]&&v!=hch[u]) dfs2(v,v);
        }

        int lca(int x,int y) {
            while(top[x]!=top[y]) {
                if(dep[top[x]]<dep[top[y]]) swap(x,y);
                x=p[top[x]];
            }
            if(dep[x]<dep[y]) swap(x,y);
            return y;
        }

        void init() {
            dfs1(1,-1,1); dfs2(1,1);
        }

        void clear(int n) {
            fill(hch, hch+n+1, 0);
        }
    }

    void dfs(int u,int fa) {
        uid[++idx]=u;
        first[u]=idx;
        for(int v:adj[u]) if(v!=fa) dfs(v,u);
        uid[++idx]=u;
        last[u]=idx;
    }

    void add_query(int u,int v,int id) {
```

```
        if(first[u]>first[v]) swap(u,v);
        int p=hpd::lca(u,v);
        int l,r,pidx;
        if(u==p) l=first[u],r=first[v],pidx=0;
        else l=last[u],r=first[v],pidx=first[p];
        query.emplace_back(l,r,pidx,id);
    }

    void solve() {
        auto getid=[](int x) {
            return x/block;
        };

        sort(query.begin(),query.end(),[&](const Query &x,const Query &y) {
            const auto &[l,r,_,__]=x;
            const auto &[L,R,___,____]=y;
            if(getid(l)!=getid(L)) return getid(l)<getid(L);
            return getid(l)&1?r<R:r>R;
        });

        auto _add=[&](int u) {

        };

        auto _del=[&](int u) {

        };

        auto extend=[&](int idx) {
            int u=uid[idx];
            if(odd[u]^=1) _add(u);
            else _del(u);
        };

        int l=1,r=0;
        for(const auto &[L,R,pidx,id]:query) {
            while(l>L) extend(--l);
            while(r<R) extend(++r);
            while(l<L) extend(l++);
            while(r>R) extend(r--);
            if(pidx) extend(pidx);
            // TODO ans[id]=res;
            if(pidx) extend(pidx);
        }
    }

    void init() {
        hpd::init();
        dfs(1,-1);
    }

    void clear(int n) {
        idx=0;
        query.clear();
```

```
        fill(odd, odd+n+1, 0);
        for(int i=0;i<=n;i++) adj[i].clear();
        hpd::clear(n);
    }
}
```

CDQ 分治

CDQ 分治用来解决三维偏序问题，在应用 CDQ 分治解题时，通常是将问题转化为三维偏序来解决。

最典型的 CDQ 分治会结合树状数组使用，复杂度为 $\mathcal{O}(n\log^2 n)$。

需要注意的是，三维偏序若存在完全相同的点，则需要特殊处理。

```
namespace cdq {
    constexpr int N=1e5+10; // ***
    struct Point {

        bool operator<(const Point &p) const {

        }
    } p[N],tmp[N],bak[N];

    void solve(const int L,const int R) {
        if(L==R) return;
        int mid=L+R>>1;
        solve(L,mid),solve(mid+1,R);

        int i=L,j=mid+1,idx=L;
        while(j<=R) {
            while(i<=mid&&true) {
                // TODO 双指针更新i
                tmp[idx++]=p[i++];
            }
            // TODO 更新答案
            tmp[idx++]=p[j++];
        }
        for(int k=L;k<i;k++) ; // TODO reset状态
        while(i<=mid) tmp[idx++]=p[i++];
        for(int i=L;i<=R;i++) p[i]=tmp[i];
    }

} using cdq::p,cdq::bak;
```

应用&技巧

合并询问点与数据点

例如二维数点，每次查询相对一个点的某个范围有多少个满足要求的点。我们可以将询问点和数据点合并，将每个点额外加上一维 z 指示点的类型，然后问题就转化为：

- 横坐标 $x_i < x_j$
- 纵坐标 $y_i < y_j$
- 类型 $z_i < z_j$

由于 $z \in 0, 1$，讨论即可，复杂度保持不变。

时间线偏序

对于带修改的问题，仿照带修莫队的思路，加上一个时间维来解决。

例如二维数点附加修改操作，每次增删点然后查询。按照前面修改的次数作为时间戳的分配标准。

- 横坐标 $x_i < x_j$
- 纵坐标 $y_i < y_j$
- 时间 $t_i \le t_j$
- 类型 $z_i < z_j$

绝对值拆分与序列反转

对于带询问的问题，且询问点不同方向的统计标准不一致，我们可以用 CDQ 分治解决其中一个方向，然后想办法复用代码来解决其他方向。

最典型的带绝对值的问题，例如给定一个二维平面，带增加点的修改，每次询问距离一个点最近的点的曼哈顿距离。

将绝对值拆开后变成四个子问题，最简单的是：

- $x_i < x_j$
- $y_i < y_j$
- $t_i < t_j$
- $z_i < z_j$

直接将 $x_i + y_i$ 存进树状数组即可。

而对于其他三种情形，我们可以仿照前面的套路，将点按照对应的顺序反转后再跑cdq。

- $x_i < x_j$ 且 $y_i > y_j$ 上下反转
- $x_i > x_j$ 且 $y_i < y_j$ 左右反转
- $x_i > x_j$ 且 $y_i > y_j$ 上下反转+左右反转

```
auto rev=[&](auto get) {
    int maxx=0,minn=1e9;
    for(int i=1;i<=n+m;i++) maxx=max(maxx,get(i)),minn=min(minn,get(i));
    for(int i=1;i<=n+m;i++) get(i)=maxx-get(i)+minn;
};

sort(p+1,p+1+n+m);
cdq::solve(1,n+m);
```

```
for(int i=1;i<=n+m;i++) p[i]=bak[i];
rev([](int idx) -> int& { return p[idx].x; });
sort(p+1,p+1+n+m);
cdq::solve(1,n+m);

for(int i=1;i<=n+m;i++) p[i]=bak[i];
rev([](int idx) -> int& { return p[idx].y; });
sort(p+1,p+1+n+m);
cdq::solve(1,n+m);

for(int i=1;i<=n+m;i++) p[i]=bak[i];
rev([](int idx) -> int& { return p[idx].x; });
rev([](int idx) -> int& { return p[idx].y; });
sort(p+1,p+1+n+m);
cdq::solve(1,n+m);
```

类似还有带修逆序对，都是差不多的套路。

CDQ 分治优化 1D/1D 动态规划的转移

考虑一个二维版本的最长上升子序列，序列中每个点有两个属性。

可以直接列出dp转移方程：

$$dp_j = 1 + \max_{i=1}^{j-1} dp_i \text{ if } h_i < h_j \text{ and } v_i < v_j$$

直接转移显然是$O(n^2)$的，观察到转移过程其实也是点对之间的关系，使用CDQ分治优化转移过程。

设当前区间为$[L,R]$

- 若$L=R$，那么$[1,L)$已经处理完成，我们直接令$dp_L := dp_L + 1$ 即可
- 递归处理$[L,mid]$
- 像一般的CDQ分治一样处理$[L,mid]$ 与$[mid+1,R]$ 的信息合并
- 递归处理$[mid+1,R]$

和一般的 CDQ 分治最大的区别是，我们不直接递归左右两边，而 是变成了左->中->右的顺序，这主要是为了保证dp转移顺序的正确性。

如果某些问题带修，且一次修改依赖前面的修改（修改之间不独立），那么也可以用类似的思路解决。

修改递归顺序后，不方便再使用双指针进行排序，这个时候直接暴力`std::sort`即可。

线段树分治

线段树分治可以将 “增+删” 转化为 “增+撤销/持久化”，代价是多一个log的复杂度，并且要求问题可离线。

```
namespace sd {
    #define lch (u<<1)
    #define rch (u<<1|1)
```

```
    using T=int;
    vector<vector<T>> seg;
    int L,R;

    void add(int u,int x,int y,int l,int r,T val) {
        if(x>r||y<l) return;
        if(x<=l&&y>=r) seg[u].emplace_back(val);
        else {
            int mid=(l+r)/2;
            add(lch,x,y,l,mid,val);
            add(rch,x,y,mid+1,r,val);
        }
    }
    void add(int x,int y,T val) {
        add(1,x,y,L,R,val);
    }

    void solve(int u,int l,int r) {
        // apply
        for(auto x:seg[u]) {

        }

        // update ans
        if(l==r) ;
        else {
            int mid=(l+r)/2;
            solve(lch,l,mid);
            solve(rch,mid+1,r);
        }

        // undo
    }
    void solve() {
        solve(1,L,R);
    }

    void init(int l,int r) {
        L=l,R=r;
        seg.clear();
        seg.resize(4<<__lg(r-l+1)|1);
    }

    #undef lch
    #undef rch
}
```

哈希

质数表

10^9

```
{
    int(1e9)+0007,int(1e9)+0009,int(1e9)+0021,int(1e9)+0033,int(1e9)+0087,
    int(1e9)+0093,int(1e9)+0097,int(1e9)+0103,int(1e9)+0123,int(1e9)+0181,
    int(1e9)+0207,int(1e9)+0223,int(1e9)+0241,int(1e9)+0271,int(1e9)+0289,
    int(1e9)+0297,int(1e9)+0321,int(1e9)+0349,int(1e9)+0363,int(1e9)+0403,
    int(1e9)+0409,int(1e9)+0411,int(1e9)+0427,int(1e9)+0433,int(1e9)+0439,
    int(1e9)+0447,int(1e9)+0453,int(1e9)+0459,int(1e9)+0483,int(1e9)+0513,
    int(1e9)+0531,int(1e9)+0579,int(1e9)+0607,int(1e9)+0613,int(1e9)+0637,
    int(1e9)+0663,int(1e9)+0711,int(1e9)+0753,int(1e9)+0787,int(1e9)+0801,
    int(1e9)+0829,int(1e9)+0861,int(1e9)+0871,int(1e9)+0891,int(1e9)+0901,
    int(1e9)+0919,int(1e9)+0931,int(1e9)+0933,int(1e9)+0993,int(1e9)+1011
};
```

10^{18}

```
{
    LL(1e18)+0003,LL(1e18)+0009,LL(1e18)+0031,LL(1e18)+0079,LL(1e18)+0177,
    LL(1e18)+0183,LL(1e18)+0201,LL(1e18)+0283,LL(1e18)+0381,LL(1e18)+0387,
    LL(1e18)+0507,LL(1e18)+0523,LL(1e18)+0583,LL(1e18)+0603,LL(1e18)+0619,
    LL(1e18)+0621,LL(1e18)+0799,LL(1e18)+0841,LL(1e18)+0861,LL(1e18)+0877,
    LL(1e18)+0913,LL(1e18)+0931,LL(1e18)+0997,LL(1e18)+1093,LL(1e18)+1191,
    LL(1e18)+1267,LL(1e18)+1323,LL(1e18)+1347,LL(1e18)+1359,LL(1e18)+1453,
    LL(1e18)+1459,LL(1e18)+1537,LL(1e18)+1563,LL(1e18)+1593,LL(1e18)+1659,
    LL(1e18)+1683,LL(1e18)+1729,LL(1e18)+1743,LL(1e18)+1771,LL(1e18)+1827,
    LL(1e18)+1879,LL(1e18)+1953,LL(1e18)+2049,LL(1e18)+2097,LL(1e18)+2137,
    LL(1e18)+2217,LL(1e18)+2271,LL(1e18)+2319,LL(1e18)+2481,LL(1e18)+2493
}
```

Hashint

比较useless，可以直接用`Modint`+ 10^{18} 质数做平替。

```
constexpr int HASHCNT=2;
array<int,HASHCNT> mod;
template<int size,typename I=int,typename L=long long,const array<I,size>
&p=mod>
struct Hashint {

array<I,size> v;
I _pow(int i,L b) const {
    L res=1,a=v[i];
    while(b) { if(b&1) res=res*a%p[i]; b>>=1; a=a*a%p[i]; }
    return res;
}
I _inv(int i) const { return _pow(i,p[i]-2); }
Hashint pow(L b) {
```

```
    Hashint res;
    for(int i=0;i<size;i++) res[i]=_pow(i,b);
    return res;
}

Hashint &operator+=(const Hashint &x)
{ for(int i=0;i<size;i++) v[i]+=x[i],v[i]-=v[i]>=p[i]?p[i]:0; return *this;
}

Hashint &operator-=(const Hashint &x)
{ for(int i=0;i<size;i++) v[i]-=x[i],v[i]+=v[i]<0?p[i]:0; return *this; }

Hashint &operator*=(const Hashint &x)
{ for(int i=0;i<size;i++) v[i]=L(1)*v[i]*x[i]%p[i]; return *this; }

Hashint &operator/=(const Hashint &x)
{ for(int i=0;i<size;i++) v[i]=L(1)*v[i]*x._inv(i)%p[i]; return *this; }

friend Hashint operator+(Hashint l,const Hashint &r) { return l+=r; }
friend Hashint operator-(Hashint l,const Hashint &r) { return l-=r; }
friend Hashint operator*(Hashint l,const Hashint &r) { return l*=r; }
friend Hashint operator/(Hashint l,const Hashint &r) { return l/=r; }

Hashint operator++(int) { auto res=*this; *this+=1; return res; }
Hashint operator--(int) { auto res=*this; *this-=1; return res; }
Hashint operator-  () { return *this*-1; }
Hashint &operator++() { return *this+=1; }
Hashint &operator--() { return *this-=1; }

bool operator< (const Hashint &x) const { return v< x.v; }
bool operator> (const Hashint &x) const { return v> x.v; }
bool operator<=(const Hashint &x) const { return v<=x.v; }
bool operator>=(const Hashint &x) const { return v>=x.v; }
bool operator==(const Hashint &x) const { return v==x.v; }
bool operator!=(const Hashint &x) const { return v!=x.v; }

auto &operator[](int i) { return v[i]; }
auto &operator[](int i) const { return v[i]; }

Hashint(L x=0) { for(int i=0;i<size;i++) v[i]=(x%p[i]+p[i])%p[i]; }

}; using Hint=Hashint<HASHCNT>;
```

集合哈希

为每个元素分配一个随机数，集合哈希就等于每个元素的xor。如果要哈希多重集，将xor改成+即可。

DP 模板

换根dp

```
namespace chrt {
    // 初始化节点u
    void init(int u) {

    }

    // 将子树u加入到子树p下
    void link(int p,int u) {

    }

    // 将子树u从子树p中移除
    void cut(int p,int u) {

    }

    void dfs1(int u,int fa) {
        init(u);
        for(int v:adj[u]) {
            if(v!=fa) {
                dfs1(v,u);
                link(u,v);
            }
        }
    }

    void dfs2(int u,int fa) {
        // TODO 更新答案
        for(int v:adj[u]) {
            if(v!=fa) {
                cut(u,v),link(v,u);
                dfs2(v,u);
                cut(v,u),link(u,v);
            }
        }
    }

    void solve(int rt) {
        dfs1(rt, -1);
        dfs2(rt, -1);
    }
}
```

斜率优化dp

```
LL dp(int n) {
    LL res=0,pre=0;
    vector<pair<LL,LL>> q(1);
    for(int i=1,idx=0;i<=n;i++) {
```

```
        // assert(idx<q.size());
        auto [x,y]=q[idx];
        LL k=0.0;
        LL b=y-k*x;
        while(idx+1<q.size()) {
            auto [x,y]=q[idx+1];
            if(y-k*x<=b) {
                b=y-k*x;
                idx++;
            }
            else break;
        }

        res=0.0;
        x=0.0;
        y=0.0;

        while(q.size()>=2) {
            auto [xl,yl]=q[q.size()-2];
            auto [xr,yr]=q[q.size()-1];
            if((y-yr)*(x-xl)<=(y-yl)*(x-xr)) q.pop_back();
            else break;
        }
        q.emplace_back(x,y);
    }
    return res;
}
```

整数三分

整数域下的三分，要求函数必须为单峰/单谷函数，允许存在多个最值点，但最值处的左/右侧必须严格单调。

```
template<typename I,class F>
I ternary_search_min(I l,I r,F f) {
    while(l<r) {
        I lmid=l+(r-l)/2;
        I rmid=lmid+1;
        if(f(lmid)<f(rmid)) r=rmid-1;
        else l=lmid+1;
    }
    return l;
};

template<typename I,class F>
I ternary_search_max(I l,I r,F f) {
    while(l<r) {
        I lmid=l+(r-l)/2;
        I rmid=lmid+1;
        if(f(lmid)<f(rmid)) l=lmid+1;
        else r=rmid-1;
```

```
    }
    return l;
};
```

大整数乘法

利用 `long double` 代替 `__int128` 实现模意义下的大整数乘法。

```
LL binmul(LL a, LL b, LL m) {
    LL c = (LL)a * b - (LL)((long double) a / m * b + 0.5L) * m;
    return c < 0 ? c + m : c;
}
```

表达式求值

```
namespace eval {
    // 调度场算法 O(n) 输入中缀表达式, 输出后缀表达式
    // 默认每个词(数、运算符、函数)仅为1char长
    // 注意, -可以同时理解为减法和取反时会导致歧义
    string shunting_yard(string &s) {
        // 是否为基础值
        auto is_num=[](char x) {
            return isdigit(x);
        };

        // 是否为运算符
        auto is_oper=[](char x) {
            const static string s="+-*/";
            return find(s.begin(),s.end(),x)!=s.end();
        };

        // 是否为函数
        auto is_func=[](char x) {
            const static string s="";
            return find(s.begin(),s.end(),x)!=s.end();
        };

        // 是否左结合
        auto left_assoc=[](char x) {
            const static string s="+-*/";
            return find(s.begin(),s.end(),x)!=s.end();
        };

        // 运算优先级 优先级越高, 值越小
        auto prio=[](char x) {
            if(x=='*'||x=='/') return 1;
```

```
            if(x=='+'||x=='-') return 2;
            return 3;
        };

        string res,stk;
        for(auto x:s) {
            if(is_num(x)) res.push_back(x);
            else if(is_func(x)) stk.push_back(x);
            else if(is_oper(x)) {
                while(stk.size()&&is_oper(stk.back())) {
                    char y=stk.back();
                    if (
                        prio(y)<prio(x)||
                        left_assoc(x)&&prio(x)==prio(y)
                    ) stk.pop_back(),res.push_back(y);
                    else break;
                }
                stk.push_back(x);
            }
            else if(x==',') {
                while(stk.back()!='(')
                    res.push_back(stk.back()),stk.pop_back();
            }
            else if(x=='(') stk.push_back(x);
            else if(x==')') {
                while(stk.size()&&stk.back()!='(')
                    res.push_back(stk.back()),stk.pop_back();
                assert(stk.size());
                stk.pop_back();
                if(stk.size()&&is_func(stk.back()))
                    res.push_back(stk.back()),stk.pop_back();
            }
            else assert(0);
        }

        while(stk.size()) {
            assert(stk.back()!='(');
            res.push_back(stk.back());
            stk.pop_back();
        }
        return res;
    }

    // 计算中缀表达式
    LL cal(string s) {
        s=shunting_yard(s);
        vector<LL> stk;
        for(char x:s) {
            if(isdigit(x)) stk.emplace_back(x-'0');
            else {
                LL b=stk.end()[-1];
                LL a=stk.end()[-2];
                stk.pop_back();
                if(x=='*') stk.back()=a*b;
```

```
                else if(x=='/') stk.back()=a/b;
                else if(x=='+') stk.back()=a+b;
                else if(x=='-') stk.back()=a-b;
                else assert(0);
            }
        }
        return stk.back();
    }
}
```

约瑟夫环

从 n 个人的循环队列中每次隔 k 个抽出一人出队，求最后出队人编号。

下标从0开始，复杂度 $\mathcal{O}(n)$。

```
int josephus(int n, int k) {
    int s = 0;
    for (int i = 2; i <= n; i++)
        s = (s + k) % i;
    return s;
}
```

如果要求完整的出队编号序列，可以利用树状数组上二分可以在 $\mathcal{O}(n\log n)$ 时间内求出序列。

```
int kth(int k) {
    int pos=0;
    for(int i=__lg(tr.size()-1);~i;i--)
        if(pos+(1<<i)<tr.size()&&tr[pos+(1<<i)]<k)
            pos+=1<<i,k-=tr[pos];
    return pos;
}

auto get=[&](int k) {
    Fenwick<> tr(n);
    for(int i=0;i<n;i++) tr.modify(i, 1);

    vector<int> p(n);
    iota(p.begin(), p.end(), 0);
    int idx=0,tot=n;
    for(int i=0;i<n;i++) {
        int step=(k-1)%tot+1;
        int pre=tr.query(idx-1);
        int suf=tot-pre;
        if(suf>=step) idx=tr.kth(pre+step);
        else idx=tr.kth(step-suf);

        p[i]=idx;
```

```
        tot--;
        tr.modify(idx, -1);
    }
    return p;
};
```