

数据结构

- 树状数组
 - 树状数组
 - 树状数组上二分
 - 线段树
 - 线段树
 - 单点修改线段树
 - 可持久化线段树
 - 势能线段树
 - 线段树合并/分裂
 - 字典树
 - 字典树
 - 01-字典树
 - 可持久化01-字典树
 - 并查集
 - 并查集
 - 可撤销并查集
 - 树链剖分
 - 稀疏表
 - Link Cut Tree
 - 珂朵莉树
 - 虚树
-

树状数组

树状数组是最为小巧实用的数据结构之一，能在 $\mathcal{O}(\log n)$ 的时间复杂度内进行单点修改+区间查询。通过维护差分数组也可以实现区间修改+单点查询。

树状数组通过前缀和相减来完成区间操作，所以要求维护的信息具有可减性，否则无法使用树状数组维护。

树状数组

```
template<typename T=int> struct Fenwick {
    int size=0;
    vector<T> tr;

    int lowbit(int x) { return x&(-x); }

    void update(T &aim,const T &val) { aim+=val; }

    void add(int pos,T val) {
        while(pos<=size) update(tr[pos],val),pos+=lowbit(pos);
    }

    T query(int pos) {
```

```

        T res{};
        while(pos) update(res, tr[pos]), pos-=lowbit(pos);
        return res;
    }

    Fenwick(int size):size(size) { tr.resize(size+1); }
};

```

树状数组上二分

类似线段树，我们可以在树上数组上进行二分，从高位向低位枚举即可。

权值树状数组求第k大的例子：

```

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

```

线段树

线段树能够灵活地维护区间信息，区间修改与查询均为 $\mathcal{O}(\log n)$ ，常数较大。

线段树

```

template<class Info, class Tag, int size> struct SegmentTree {
    #define lch ((u)<<1)
    #define rch ((u)<<1|1)

    int rng_l, rng_r;
    constexpr static int node_size=1<<__lg(size)<<2|1;
    array<Tag, node_size> tag;
    array<Info, node_size> info;
    array<bool, node_size> clean;

    void pushup(int u) {
        info[u]=info[lch]+info[rch];
    }

    void update(int u, const Tag &t) {
        info[u]+=t;
        tag[u]+=t;
        clean[u]=0;
    }

    void pushdn(int u) {

```

```

        if(clean[u]) return;
        update(lch, tag[u]);
        update(rch, tag[u]);
        clean[u]=1;
        tag[u].clear();
    }

Info query(int u,int l,int r,int x,int y) {
    if(l>y||r<x) return {};
    if(l>=x&&r<=y) return info[u];
    pushdn(u);
    int mid=(l+r)/2;
    return query(lch,l,mid,x,y)+query(rch,mid+1,r,x,y);
}
Info query(int l,int r) { return query(1,rng_l,rng_r,l,r); }

void modify(int u,int l,int r,int x,int y,const Tag &t) {
    if(l>y||r<x) return;
    if(l>=x&&r<=y) update(u, t);
    else {
        pushdn(u);
        int mid=(l+r)/2;
        if(mid>=x) modify(lch,l,mid,x,y,t);
        if(mid<y) modify(rch,mid+1,r,x,y,t);
        pushup(u);
    }
}
void modify(int l,int r,const Tag &t) { modify(1,rng_l,rng_r,l,r,t); }

template<class F>
int find_first(int u,int l,int r,int x,int y,F check) {
    if(l>y||r<x||l>=x&&r<=y&&!check(info[u])) return -1;
    if(l==r) return l;
    pushdn(u);
    int mid=(l+r)/2;
    int res=find_first(lch,l,mid,x,y,check);
    if(res==-1) res=find_first(rch,mid+1,r,x,y,check);
    return res;
}
template<class F> int find_first(int l,int r,F check) {
    return find_first(1,rng_l,rng_r,l,r,check);
}

template<class F>
int find_last(int u,int l,int r,int x,int y,F check) {
    if(l>y||r<x||l>=x&&r<=y&&!check(info[u])) return -1;
    if(l==r) return l;
    pushdn(u);
    int mid=(l+r)/2;
    int res=find_last(rch,mid+1,r,x,y,check);
    if(res==-1) res=find_last(lch,l,mid,x,y,check);
    return res;
}
template<class F> int find_last(int l,int r,F check) {

```

```

        return find_last(1, rng_l, rng_r, l, r, check);
    }

    void build(int u, int l, int r) {
        clean[u]=1;
        info[u].init(l, r);
        tag[u].clear();
        if(l!=r) {
            int mid=(l+r)/2;
            build(lch, l, mid);
            build(rch, mid+1, r);
            pushup(u);
        }
    }
    void build(int l=1, int r=size) { build(1, rng_l=l, rng_r=r); }

    #undef lch
    #undef rch
};

struct Tag {

    void clear() {

    }

    Tag &operator+=(const Tag &t) {

        return *this;
    }
};

struct Info {

    void init(int l, int r) {
        if(l!=r) return;
    }
    void init(int l) { init(l, l); }

    friend Info operator+(const Info &l, const Info &r) {
        Info res;

        return res;
    }

    Info &operator+=(const Tag &t) {

        return *this;
    }
};

SegmentTree<Info, Tag, N> sgt;

```

单点修改线段树

仅支持单点、修改常数更小更简短的实现。

```
template<class Info,int size> struct SegmentTree {
    #define lch ((u)<<1)
    #define rch ((u)<<1|1)

    int rng_l,rng_r;
    constexpr static int node_size=1<<__lg(size)<<2|1;
    array<Info, node_size> info;
    array<int, size+1> leaf;

    void pushup(int u) {
        info[u]=info[lch]+info[rch];
    }

    Info query(int u,int l,int r,int x,int y) {
        if(l>y||r<x) return {};
        if(l>=x&&r<=y) return info[u];
        int mid=(l+r)/2;
        return query(lch,l,mid,x,y)+query(rch,mid+1,r,x,y);
    }
    Info query(int l,int r) { return query(1,rng_l,rng_r,l,r); }

    void modify(int p,const Info &v) {
        int u=leaf[p];
        info[u]+=v;
        while(u>=1) pushup(u);
    }

    template<class F>
    int find_first(int u,int l,int r,int x,int y,F check) {
        if(l>y||r<x||l>=x&&r<=y&&!check(info[u])) return -1;
        if(l==r) return l;
        int mid=(l+r)/2;
        int res=find_first(lch,l,mid,x,y,check);
        if(res!=-1) res=find_first(rch,mid+1,r,x,y,check);
        return res;
    }
    template<class F> int find_first(int l,int r,F check) {
        return find_first(1,rng_l,rng_r,l,r,check);
    }

    template<class F>
    int find_last(int u,int l,int r,int x,int y,F check) {
        if(l>y||r<x||l>=x&&r<=y&&!check(info[u])) return -1;
        if(l==r) return l;
        int mid=(l+r)/2;
        int res=find_last(rch,mid+1,r,x,y,check);
        if(res!=-1) res=find_last(lch,l,mid,x,y,check);
        return res;
    }
}
```

```

    }
    template<class F> int find_last(int l,int r,F check) {
        return find_last(1,rng_l,rng_r,l,r,check);
    }

    void build(int u,int l,int r) {
        info[u].init(l,r);
        if(l!=r) {
            int mid=(l+r)/2;
            build(lch,l,mid);
            build(rch,mid+1,r);
            pushup(u);
        }
        else leaf[l]=u;
    }
    void build(int l=1,int r=size) { build(1,rng_l=l,rng_r=r); }

    #undef lch
    #undef rch
};

struct Info {

    void init(int l,int r) {
        if(l!=r) return;

    }
    void init(int l) { init(l,l); }

    friend Info operator+(const Info &l,const Info &r) {
        Info res;

        return res;
    }

    Info &operator+=(const Info &v) {

        return *this;
    }
};

SegmentTree<Info, N> sgt;

```

可持久化线段树

通过记录每次修改变化的节点，可以在保存历史信息的同时，大幅地压缩空间复杂度。

```

template<class Info,int node_size>
struct PersistentSegmentTree {
    int idx,rng_l,rng_r;
    vector<int> root;

```

```

array<Info, node_size> info;
array<int, node_size> lch,rch;

int ver() {
    return root.size()-1;
}

int new_node() {
    assert(idx<node_size);
    return ++idx;
}

int new_root() {
    root.emplace_back();
    return ver();
}

void clone(int u,int v) {
    info[u]=info[v];
    lch[u]=lch[v];
    rch[u]=rch[v];
}

void pushup(int u) {
    info[u]=info[lch[u]]+info[rch[u]];
}

Info query(int u,int l,int r,int x,int y) {
    if(l>y||r<x) return {};
    if(l>=x&&r<=y) return info[u];
    int mid=(l+r)/2;
    return query(lch[u],l,mid,x,y)+query(rch[u],mid+1,r,x,y);
}

Info query(int u,int l,int r) {
    return query(root[u],rng_l,rng_r,l,r);
}

Info range_query(int u,int v,int l,int r,int x,int y) {
    if(l>y||r<x) return {};
    if(l>=x&&r<=y) return info[u]-info[v];
    int mid=(l+r)/2;
    return range_query(lch[u],lch[v],l,mid,x,y)+
           range_query(rch[u],rch[v],mid+1,r,x,y);
}

Info range_query(int u,int v,int l,int r) {
    return range_query(root[u],root[v],rng_l,rng_r,l,r);
}

void modify(int &u,int v,int l,int r,int p,const Info &val) {
    u=new_node();
    clone(u, v);
    if(l==r) info[u]+=val;
    else {
        int mid=(l+r)/2;

```

```

        if(p<=mid) modify(lch[u],lch[v],l,mid,p,val);
        else modify(rch[u],rch[v],mid+1,r,p,val);
        pushup(u);
    }
}
void modify(int u,int v,int p,const Info &val) {
    modify(root[u],root[v],rng_l,rng_r,p,val);
}

int update(int p,const Info &val) {
    new_root();
    modify(root[ver()],root[ver()-1],rng_l,rng_r,p,val);
    return ver();
}

template<class F>
int find_first(int u,int l,int r,int x,int y,F check) {
    if(l>y||r<x||l>=x&&r<=y&&!check(info[u])) return -1;
    if(l==r) return l;
    int mid=(l+r)/2;
    int res=find_first(lch[u],l,mid,x,y,check);
    if(res==-1) res=find_first(rch[u],mid+1,r,x,y,check);
    return res;
}
template<class F> int find_first(int u,int l,int r,F check) {
    return find_first(root[u],rng_l,rng_r,l,r,check);
}

template<class F>
int find_last(int u,int l,int r,int x,int y,F check) {
    if(l>y||r<x||l>=x&&r<=y&&!check(info[u])) return -1;
    if(l==r) return l;
    int mid=(l+r)/2;
    int res=find_last(rch[u],mid+1,r,x,y,check);
    if(res==-1) res=find_last(lch[u],l,mid,x,y,check);
    return res;
}
template<class F> int find_last(int u,int l,int r,F check) {
    return find_last(root[u],rng_l,rng_r,l,r,check);
}

void build(int &u,int l,int r) {
    u=new_node();
    info[u].init(l,r);
    if(l!=r) {
        int mid=(l+r)>>1;
        build(lch[u],l,mid);
        build(rch[u],mid+1,r);
        pushup(u);
    }
}
void build(int l,int r) {
    build(root[new_root()],rng_l=l,rng_r=r);
}

```



```
};

struct Info {

    void init(int l,int r) {
        if(l!=r) return;

    }
    void init(int l) { init(l,l); }

    friend Info operator+(const Info &l,const Info &r) {
        Info res;

        return res;
    }

    friend Info operator-(const Info &l,const Info &r) {
        Info res;

        return res;
    }

    Info &operator+=(const Info &v) {

        return *this;
    }
};

PersistentSegmentTree<Info, N*__lg(N)*4> sgt;
```

势能线段树

```
struct Info {
    bool final;

    void init(int l,int r) {
        if(l!=r) return;

    }
    void init(int l) { init(l,l); }

    friend Info operator+(const Info &l,const Info &r) {
        Info res;

        return res;
    }

    void operator--(int) {

    }
}
```

```

    explicit operator bool() const { return final; }
};

template<class Info,int size> struct SegmentTree {
    #define lch (u<<1)
    #define rch (u<<1|1)

    struct Node {
        int l,r;
        Info info;
        void init(int l,int r) {
            this->l=l;
            this->r=r;
            info.init(l, r);
        }
    };

    array<Node, 1<<__lg(size)<<2|1> tr;

    void pushup(int u) {
        tr[u].info=tr[lch].info+tr[rch].info;
    }

    Info query(int u,int l,int r) {
        if(tr[u].l>=l&&tr[u].r<=r) { return tr[u].info; }
        else {
            int mid=(tr[u].l+tr[u].r)/2;
            if(mid>=l&&mid<r) return query(lch,l,r)+query(rch,l,r);
            else if(mid>=l) return query(lch,l,r);
            return query(rch,l,r);
        }
    }
    Info query(int l,int r) { return query(1,l,r); }

    void release(int u,int l,int r) {
        if(tr[u].info) return;
        else if(tr[u].l==tr[u].r) tr[u].info--;
        else {
            int mid=(tr[u].l+tr[u].r)/2;
            if(l<=mid) release(lch,l,r);
            if(r>mid) release(rch,l,r);
            pushup(u);
        }
    }
    void release(int l,int r) { release(1,l,r); }

    void build(int u,int l,int r) {
        tr[u].init(l,r);
        if(l!=r) {
            int mid=(l+r)/2;
            build(lch,l,mid);
            build(rch,mid+1,r);
            pushup(u);
        }
    }
};

```

```

    }
    void build(int l=1,int r=size) { build(1,l,r); }

    #undef lch
    #undef rch
};
SegmentTree<Info, N> sgt;

```

线段树合并/分裂

设区间大小为 n ，分裂次数为 m 。无论以何种顺序合并与分裂，时间复杂度均为 $\mathcal{O}((n+m)\log n)$ ，空间复杂度 $\mathcal{O}((n+m)\log n)$ （常数=1），如果在合并时保留子树结构，则需要多一倍的空间。

```

struct MergeSplitSegmentTree {

    #define lch tr[u].ch[0]
    #define rch tr[u].ch[1]
    constexpr static int MAX_SIZE=1e7+10;

    struct Node {
        int ch[2];
        int cnt;
    } tr[MAX_SIZE];
    int idx;

    int new_node() {
        // assert(idx<MAX_SIZE);
        return ++idx;
    }

    void pushup(int u) {
        if(lch&&rch) ;
        else if(lch) ;
        else if(rch) ;
    }

    // remember to pushdn lazy tag
    void pushdn(int u) {
        if(lch) ;
        if(rch) ;
    }

    void merge(int &u,int v) {
        if(!u&&!v) return;
        if(!u||!v) u=u|v;
        else {
            pushdn(u);pushdn(v);
            merge(lch,tr[v].ch[0]);
            merge(rch,tr[v].ch[1]);
            pushup(u);
        }
    }
};

```

```

    }
}

// k][k+1
void split(int &u,int &v,int l,int r,int k) {
    if(!u||k>=r) return;
    if(k<l) swap(u,v);
    else {
        v=new_node();
        int mid=l+r>>1;
        if(k<=mid) swap(rch,tr[v].ch[1]);
        pushdn(u);
        if(k<mid) split(lch, tr[v].ch[0], l, mid, k);
        else split(rch, tr[v].ch[1], mid+1, r, k);
        pushup(u),pushup(v);
    }
}

int kth(int u,int l,int r,int k) {
    if(tr[u].cnt<k) return -1;
    if(l==r) return l;
    int mid=l+r>>1;
    pushdn(u);
    if(tr[lch].cnt>=k) return kth(lch, l, mid, k);
    return kth(rch, mid+1, r, k-tr[lch].cnt);
}

void build(int &u,int l,int r,int p) {
    u=new_node();
    if(l==r) ;
    else {
        int mid=l+r>>1;
        if(p<=mid) build(lch,l,mid,p);
        else build(rch,mid+1,r,p);
        pushup(u);
    }
}

#undef lch
#undef rch

} sgt;

```

字典树

字典树

```

struct Trie {
    constexpr static int A=26,B='a';
    struct Node {
        int ch[A];
    };
};

```

```

        int cnt;
    };
    vector<Node> tr;

    int new_node() { tr.push_back({}); return tr.size()-1; }

    int extend(int u,int x) {
        if(!tr[u].ch[x-B]) tr[u].ch[x-B]=new_node();
        tr[tr[u].ch[x-B]].cnt++;
        return tr[u].ch[x-B];
    }

    template<typename T> void insert(const T &s) {
        int u=0;
        for(auto x:s) u=extend(u, x);
    }

    void clear() { tr.clear(); new_node(); }
    Trie() { clear(); }
    Trie(int size) { tr.reserve(size); clear(); }
} trie;

```

01-字典树

```

template<typename I> struct BinaryTrie {
    constexpr static int H=sizeof(I)*8-1;
    struct Node {
        int ch[2];
        int cnt;
    };
    vector<Node> tr;

    int new_node() {
        tr.push_back({});
        return tr.size()-1;
    }

    void insert(int v) {
        for(int i=H,u=0;i>=0;i--) {
            bool x=v>>i&1;
            if(!tr[u].ch[x]) tr[u].ch[x]=new_node();
            u=tr[u].ch[x];
            tr[u].cnt++;
        }
    }

    void erase(int v) {
        for(int i=H,u=0;i>=0;i--) {
            bool x=v>>i&1;
            u=tr[u].ch[x];
            tr[u].cnt--;
        }
    }
};

```

```

    }
}

I xor_max(int v) {
    I res{};
    for(int i=H,u=0;i>=0;i--) {
        bool x=v>>i&1^1;
        if(tr[tr[u].ch[x]].cnt) {
            res|=1<<i;
            u=tr[u].ch[x];
        }
        else u=tr[u].ch[x^1];
    }
    return res;
}

void clear() {
    tr.clear();
    new_node();
}

explicit BinaryTrie(int size=0) {
    tr.reserve(size*(H+1));
    clear();
}

};

```

可持久化01-字典树

```

template<typename I> struct PersistentBinaryTrie {
    constexpr static int H=sizeof(I)*8-1;
    struct Node {
        int ch[2];
        int cnt;
    };
    vector<Node> tr;
    vector<int> root;

    int ver() { return root.size()-1; }

    int new_root() {
        root.push_back({});
        return ver();
    }

    int new_node() {
        tr.push_back({});
        return tr.size()-1;
    }

    void insert(int &rt,int v,int val) {

```

```

    int u=rt=new_node();
    tr[u]=tr[v];
    for(int i=H;i>=0;i--) {
        bool x=val>>i&1;
        u=tr[u].ch[x]=new_node();
        v=tr[v].ch[x];
        tr[u]=tr[v];
        tr[u].cnt++;
    }
}

int insert(int val) {
    new_root();
    insert(root[ver()], root[ver()-1], val);
    return ver();
}

I xor_max(int u,int val) {
    u=root[u];
    I res{};
    for(int i=H;i>=0;i--) {
        bool x=val>>i&1^1;
        if(tr[tr[u].ch[x]].cnt) {
            res|=1<<i;
            u=tr[u].ch[x];
        }
        else u=tr[u].ch[x^1];
    }
    return res;
}

I range_xor_max(int u,int v,int val) {
    u=root[u],v=root[v];
    I res{};
    for(int i=H;i>=0;i--) {
        bool x=val>>i&1^1;
        if(tr[tr[u].ch[x]].cnt-tr[tr[v].ch[x]].cnt) {
            res|=1<<i;
            u=tr[u].ch[x];
            v=tr[v].ch[x];
        }
        else u=tr[u].ch[x^1],v=tr[v].ch[x^1];
    }
    return res;
}

void clear() {
    tr.clear();
    new_root();
    new_node();
}

explicit PersistentBinaryTrie(int size=0) {
    tr.reserve(size*(H+1));
    clear();
}

```

```
    }  
};
```

并查集

并查集

并查集能够高效地处理集合信息。

```
struct DisjointUnionSet {  
    vector<int> fa,sz;  
  
    void init(int n) {  
        fa.resize(n+1);  
        sz.assign(n+1,1);  
        iota(fa.begin(), fa.end(), 0);  
    }  
  
    int find(int x) {  
        return x==fa[x]?x:fa[x]=find(fa[x]);  
    }  
  
    bool same(int x,int y) {  
        return find(x)==find(y);  
    }  
  
    bool join(int x,int y) {  
        x=find(x);  
        y=find(y);  
        if(x==y) return false;  
        // if(sz[x]<sz[y]) swap(x,y);  
        sz[x]+=sz[y];  
        fa[y]=x;  
        return true;  
    }  
  
    int size(int x) {  
        return sz[find(x)];  
    }  
  
    DisjointUnionSet() = default;  
    DisjointUnionSet(int n) { init(n); }  
} dsu;
```

可撤销并查集

通过一个额外的栈保存修改历史来实现撤销操作。由启发式合并保证复杂度。


```
struct DisjointUnionSet {
    vector<int> fa,sz;
    vector<pair<int&,int>> fah,szh;

    void init(int n) {
        fah.clear();
        szh.clear();
        fa.resize(n+1);
        sz.assign(n+1,1);
        iota(fa.begin(), fa.end(), 0);
    }

    int find(int x) {
        while(x!=fa[x]) x=fa[x];
        return x;
    }

    bool same(int x,int y) {
        return find(x)==find(y);
    }

    bool join(int x,int y) {
        x=find(x);
        y=find(y);
        if(x==y) {
            fah.emplace_back(fa[0],fa[0]);
            szh.emplace_back(sz[0],sz[0]);
            return false;
        }
        if(sz[x]<sz[y]) swap(x,y);
        fah.emplace_back(fa[y],fa[y]);
        szh.emplace_back(sz[x],sz[x]);
        sz[x]+=sz[y];
        fa[y]=x;
        return true;
    }

    void undo() {
        assert(!fah.empty());
        fah.back().first=fah.back().second;
        szh.back().first=szh.back().second;
        fah.pop_back(),szh.pop_back();
    }

    int size(int x) {
        return sz[find(x)];
    }
}

DisjointUnionSet() = default;
DisjointUnionSet(int n) { init(n); }
} dsu;
```

树链剖分

重链剖分能将树上路径转为 $\mathcal{O}(\log n)$ 级别的连续区间，从而将树上问题转化为区间问题。预处理时间复杂度 $\mathcal{O}(n)$, 单次路径剖分时间复杂度 $\mathcal{O}(\log n)$ 。

关于实现上的易错点：把`id[u]`写成`u`，务必注意。

```
// ! don't confuse dfn id with node id
namespace hpd {
    using PII=pair<int,int>;
    constexpr int N=1e5+10; // ***
    int id[N],w[N],nw[N],cnt;
    int dep[N],sz[N],top[N],p[N],hch[N];
    vector<int> adj[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) {
        id[u]=++cnt,nw[cnt]=w[u],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;
    }

    vector<PII> decompose(int x,int y) {
        vector<PII> res;
        while(top[x]!=top[y]) {
            if(dep[top[x]]<dep[top[y]]) swap(x,y);
            res.emplace_back(id[top[x]],id[x]);
            x=p[top[x]];
        }
        if(dep[x]<dep[y]) swap(x,y);
        res.emplace_back(id[y],id[x]);
        return res;
    }
}
```

```

    }

    PII decompose(int x) {
        return { id[x], id[x]+sz[x]-1 };
    }

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

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

```

稀疏表

倍增维护区间最大值。

```

template<int MAX_SIZE, typename T=int> struct SparseTable {
    constexpr static int M=__lg(MAX_SIZE);
    T arr[MAX_SIZE], st[M][MAX_SIZE];

    void build(int n) {
        for(int i=1; i<=n; i++) st[0][i]=arr[i];
        for(int k=1, t=1<<k; k<M; k++, t<=<=1)
            for(int i=1, j=i+t-1, mid=i+t/2; j<=n; i++, j++, mid++)
                st[k][i]=max(st[k-1][i], st[k-1][mid]);
    }

    T query(int l, int r) {
        if(r<l) return 0;
        int k=__lg(r-l+1);
        return max(st[k][l], st[k][r-(1<<k)+1]);
    }
};

```

Link Cut Tree

*LCT*用来解决动态树问题，加删边/提取树上路径的复杂度均为 $\mathcal{O}(\log n)$ ，常数巨大。

编号从1开始。

```

struct LinkCutTree {

    #define lch tr[u].ch[0]
    #define rch tr[u].ch[1]
    #define wch(u) (tr[tr[u].p].ch[1]==u)

```

```

constexpr static int MAX_SIZE=1e5+10;

struct Node {
    int ch[2],p;
    bool rev;
} tr[MAX_SIZE];
int stk[MAX_SIZE];

bool is_root(int u) {
    return tr[tr[u].p].ch[0]!=u&&tr[tr[u].p].ch[1]!=u;
}

void pushup(int u) {

}

void pushrev(int u) {
    swap(lch,rch);
    tr[u].rev^=1;
}

void pushdn(int u) {
    if(tr[u].rev) pushrev(lch),pushrev(rch),tr[u].rev=0;
}

void rotate(int x) {
    int y=tr[x].p,z=tr[y].p,k=wch(x);
    if(!is_root(y)) tr[z].ch[wch(y)]=x;
    tr[y].ch[k]=tr[x].ch[!k],tr[tr[y].ch[k]].p=y;
    tr[x].ch[!k]=y,tr[y].p=x,tr[x].p=z;
    pushup(y),pushup(x);
}

void splay(int u) {
    int top=0,fa=u;
    stk[++top]=fa;
    while(!is_root(fa)) stk[++top]=fa=tr[fa].p;
    while(top) pushdn(stk[top--]);
    for(;!is_root(u);rotate(u))
        if(!is_root(fa=tr[u].p)) rotate(wch(u)==wch(fa)?fa:u);
}

void access(int u) {
    int t=u;
    for(int v=0;u;v=u,u=tr[u].p)
        splay(u),rch=v,pushup(u);
    splay(t);
}

void make_root(int u) {
    access(u);
    pushrev(u);
}

```

```

int split(int u,int v) {
    make_root(u);
    access(v);
    return v;
}

int find_root(int u) {
    access(u);
    while(lch) pushdn(u),u=lch;
    splay(u);
    return u;
}

void link(int u,int v) {
    make_root(u);
    if(find_root(v)!=u) tr[u].p=v;
}

void cut(int u,int v) {
    make_root(u);
    if(find_root(v)==u&&rch==v&&!tr[v].ch[0])
        rch=tr[v].p=0,pushup(u);
}

void modify(int u,int val) {
    splay(u);

    pushup(u);
}

#undef lch
#undef rch
#undef wch

} lct;

```

珂朵莉树

珂朵莉树通过暴力地合并 set 中信息相同的点来压缩时间复杂度，在保证数据随机的前提下，其时间复杂度为 $\mathcal{O}(n\log^2 n)$ 。

```

struct ChthollyTree {
    struct Node {
        int l,r,v;
        Node(int L,int R,int V) : l(L),r(R),v(V) {}
        bool operator< (const Node &x) const {
            return l<x.l;
        }
    };
    set<Node> st;

```

```

auto split(int pos){
    auto it=st.lower_bound(Node(pos,pos,0));
    if(it!=st.end()&&it->l==pos) return it;
    it=prev(it);
    auto [l,r,v]=*it;
    st.erase(it);
    st.insert(Node(l,pos-1,v));
    return st.insert(Node(pos,r,v)).first;
}

void assign(int l,int r,int v){
    auto end=split(r+1),begin=split(l);
    st.erase(begin,end);
    st.insert(Node(l,r,v));
}
} odt;

```

虚树

能在 $\mathcal{O}(k \log n)$ 时间内提取树上的 k 个关键点建成一棵新树,并且新树的点数不超过 $2k$ 。

```

namespace vt {
    constexpr int N=1e5+10,M=__lg(N); // ***
    vector<int> vt[N],adj[N];
    int stk[N],top,id[N],idx;
    int fa[N][M+1],dep[N];
    bool key[N];

    void lca_init(int u,int p) {
        dep[u]=dep[p]+1;
        for(int v:adj[u]) {
            if(v==p) continue;
            fa[v][0]=u;
            for(int i=1;i<=M;i++)
                fa[v][i]=fa[fa[v][i-1]][i-1];
            lca_init(v,u);
        }
    }

    int lca(int u,int v) {
        if(dep[u]<dep[v]) swap(u,v);
        for(int k=M;~k;k--)
            if(dep[fa[u][k]]>=dep[v])
                u=fa[u][k];
        if(u==v) return u;
        for(int k=M;~k;k--)
            if(fa[u][k]!=fa[v][k])
                u=fa[u][k],v=fa[v][k];
        return fa[u][0];
    }
}

```

```

void relabel(int u,int fa) {
    id[u]=++idx;
    for(int v:adj[u]) if(v!=fa) relabel(v, u);
}

void build(vector<int> &vec) {
    sort(vec.begin(),vec.end(),[](int x,int y) {
        return id[x]<id[y];
    });

    // TODO cleanup dirt memory
    auto clear=[&](int u) {
        vt[u].clear();
        key[u]=0;
    };

    auto add=[&](int u,int v) {
        vt[u].emplace_back(v);
    };

    clear(1);
    stk[top=0]=1;
    for(int u:vec) {
        if(u==1) continue;
        int p=lca(u,stk[top]);
        if(p!=stk[top]) {
            while(id[p]<id[stk[top-1]])
                add(stk[top-1],stk[top]),top--;
            if(id[p]!=id[stk[top-1]])
                clear(p),add(p,stk[top]),stk[top]=p;
            else add(p,stk[top--]);
        }
        clear(u);
        stk[++top]=u;
        key[u]=1;
    }
    for(int i=0;i<top;i++) add(stk[i],stk[i+1]);
}

void init() {
    lca_init(1, 0);
    relabel(1, 0);
}

void clear(int n) {
    idx=0;
    for(int i=0;i<=n;i++) adj[i].clear();
}
}

```