# 1.2 数据结构

## 严格线性 RMQ

- 考虑将序列 $a$ 分为 $\lceil \frac{n}{\lfloor \log_2 n \rfloor} \rceil$ 块，每块大小 $\lfloor \log_2 n \rfloor$，对所有块内最值建 $\mathrm{ST}$ 表，时间/空间复杂度 $\mathcal{O}(\frac{n}{\log_2 n} \log \frac{n}{\log_2 n}) \leq \mathcal{O}(n)$。
- 预处理每个块内前缀/后缀的最值，则当询问的 $l, r$ 在不同块上时，我们只需要在完整块上用 $\mathrm{ST}$ 表查询，不完整块上用预处理值即可实现 $\mathcal{O}(1)$ 回答询问。

- 需要特殊考虑的是 $l, r$ 在同一块内的情况，注意到块大小只有 $\lfloor \log_2 n \rfloor$，对于某一块 $i$ 的某一前缀 $j$，我们可以用一个不超过 $n$ 的二进制数 $pos_{i,j}$ 来状压该前缀的单调栈中元素的分布情况，若 $[l, r]$ 所在块为 $i$，记块 $i$ 的左端点为 $bl_i$，则询问 $[l, r]$ 的答案即为 $pos_{i,r-bl_i}$ 中第 $l - bl_i$ 位以后第一个 1 所表示的元素，可通过位运算以及预处理 $\lfloor \log_2 x \rfloor (1 \le x \le n)$ 快速求得，具体见代码实现。
- 至此，我们得到了预处理时间/空间复杂度 $\mathcal{O}(n)$、单次询问时间复杂度 $\mathcal{O}(1)$ 的严格线性 RMQ 算法。

```
const int N = 1e5 + 5;
const int L = 16;
const int L2 = 13;
const int M = 6255;
int n, m;

struct RMQ
{
    int a[N], bel[N], Log[N];
    int bl[M], br[M], f[L2][M];
    int pre[M][L], suf[M][L], pos[M][L];
    int stk[L + 1], top;

    inline void Init()
    {
        Log[0] = -1;
        for (int i = 1; i <= n; ++i)
        {
            read(a[i]);
            Log[i] = Log[i >> 1] + 1;
        }
        for (int i = 0; i <= bel[n]; ++i)
            bl[i] = br[i] = 0;
        for (int i = 1, t; i <= n; ++i)
        {
            t = bel[i] = (i - 1) / Log[n] + 1;
            if (!bl[t])
                bl[t] = i;
            br[t] = i;
        }
        bl[n + 1] = br[n + 1] = bel[n] + 1;
        for (int i = 1; i <= bel[n]; ++i)
        {
            pre[i][0] = a[bl[i]];
            pos[i][0] = 1;
            stk[top = 1] = bl[i];
            for (int j = bl[i] + 1; j <= br[i]; ++j)
            {
                pre[i][j - bl[i]] = Max(pre[i][j - bl[i] - 1], a[j]);
                int res = pos[i][j - bl[i] - 1];
                while (top && a[stk[top]] <= a[j])
                {
                    res ^= 1 << stk[top] - bl[i];
                    --top;
                }
                stk[++top] = j;
                res |= 1 << j - bl[i];
                pos[i][j - bl[i]] = res;
            }
            suf[i][br[i] - bl[i]] = a[br[i]];
```

```
            for (int j = br[i] - 1; j >= bl[i]; --j)
                suf[i][j - bl[i]] = Max(suf[i][j - bl[i] + 1], a[j]);
        }
        for (int i = 1; i <= bel[n]; ++i)
            f[0][i] = pre[i][br[i] - bl[i]];
        for (int j = 1; j <= Log[bel[n]]; ++j)
            for (int i = 1; i + (1 << j) - 1 <= bel[n]; ++i)
                f[j][i] = Max(f[j - 1][i], f[j - 1][i + (1 << j - 1)]);
    }

    inline int queryMax(int l, int r)
    {
        int tl = bel[l], tr = bel[r];
        if (tl == tr)
        {
            int s = pos[tl][r - bl[tl]] >> l - bl[tl];
            return a[Log[s & -s] + l];
        }
        int res = Max(suf[tl][l - bl[tl]], pre[tr][r - bl[tr]]);
        if (++tl <= --tr)
        {
            int k = Log[tr - tl + 1];
            CkMax(res, Max(f[k][tl], f[k][tr - (1 << k) + 1]));
        }
        return res;
    }
}T;
```

# 线段树常见套路

- 线段树区间 gcd 单次查询 $\mathcal{O}(\log n+\log w)$，建树 $\mathcal{O}(n\log w)$，ST 表预处理 $\mathcal{O}(n(\log n+\log w))$，单次查询 $\mathcal{O}(\log w)$。
- 对于 $n\times n$（$n$ 为 $10^5-10^6$ 左右）二维问题，常见的是将询问离线，每次用数据结构维护一行（或一列）的信息，每次在当前行（或列）上查询相关询问，之后将当前行（或列）的信息转化为下一行（或下一列）的信息，这通常可以转化为数据结构的某些操作，最常见的是线段树。
- 维护位数在 $10^5$ 级别的二进制数，支持在某一位 $+1/-1$。
  - 按位用线段树维护，每个结点记录最长全 $0/1$ 后缀，做 $+1/-1$ 时先确定长度后做区间翻转/区间覆盖。
  - 若需要支持两棵线段树比较操作，在每个结点维护**区间的哈希值**，在线段树上二分找到最高的不同位。
- 对区间中相邻元素满足某种条件的子序列计数。
  - 在线段树结点上记首尾为特定元素的子序列方案数。
- 询问在区间中选取 $k$ 个点（$k$ 为常数），相邻点之间满足一些性质，求选取点的最值。
  - 在线段树结点上记选取 $1\le i\le k$ 个点的最值，暴力分情况讨论。
- 计算区间某一递推式的值，在每个结点维护转移矩阵的乘积，可通过以下方式减少常数：
  - 若有取模，用 `unsigned long long` 暂存计算结果，减少取模次数。
  - 只对结果不为 0 的项进行运算，该方法同时也能节约空间。
  - 循环展开。
- 初始为 0 的序列，若干次区间 $+1$/-1，保证序列元素始终非负，求为 0 位置的个数。
  - 维护区间的最小值以及最小值的个数。
- 区间对一个等差数列取 $\max/\min$，单点查值，即**李超树**。
  - 在区间上记录取 $\max/\min$ 的等差数列，主要难点在于标记的合并。

- 若其中一个等差数列中的所有元素均比另一个大，则可直接合并。
- 否则暴力递归下去（注意是类似**标记永久化**的写法，此时当前区间的标记需保留），显然其中一个子区间一定会存在可直接合并的情况，单次操作时间复杂度 $\mathcal{O}(\log^2 n)$，实际常数很小。
- 以下是支持区间加和线段树合并（取 $\min$）的李超树代码。

```
struct line
{
    ll k, b;

    line() {}
    line(ll K, ll B):
        k(K), b(B) {}

    inline ll ask(int x) const {return k * x + b;}

    inline bool Under(const line &a, int l, int r) const
    {
        return ask(l) <= a.ask(l) && ask(r) <= a.ask(r);
    }
};
int lc[M], rc[M], stk[M];
ll tag[M];
line t[M];

inline void newNode(int &x)
{
    x = top ? stk[top--] : ++T;
    lc[x] = rc[x] = tag[x] = 0;
    t[x] = line(0, Maxn);
}

inline void delNode(int &x)
{
    stk[++top] = x;
    x = 0;
}

inline void addTag(int x, ll v)
{
    if (!x)
        return ;
    tag[x] += v;
    t[x].b += v;
}

inline void pushDown(int x)
{
    if (tag[x] != 0)
    {
        addTag(lc[x], tag[x]);
        addTag(rc[x], tag[x]);
        tag[x] = 0;
    }
}

inline void addLine(int &x, const line &a, int l, int r)
{
```

```
    if (!x)
        newNode(x);
    if (t[x].Under(a, l, r))
        return ;
    if (a.Under(t[x], l, r))
    {
        t[x] = a;
        return ;
    }
    pushDown(x);
    int mid = l + r >> 1;
    addLine(lc[x], a, l, mid);
    addLine(rc[x], a, mid + 1, r);
}

inline void Merge(int &x, int y, int l, int r)
{
    if (!x || !y)
        return (void)(x = x + y);
    addLine(x, t[y], l, r);
    if (l == r)
        return ;
    pushDown(x);
    pushDown(y);
    int mid = l + r >> 1;
    Merge(lc[x], lc[y], l, mid);
    Merge(rc[x], rc[y], mid + 1, r);
}

inline ll Query(int x, int l, int r, int v)
{
    if (!x)
        return Maxn;
    ll res = t[x].ask(v);
    if (l == r)
        return res;
    pushDown(x);
    int mid = l + r >> 1;
    CkMin(res, v <= mid ? Query(lc[x], l, mid, v) : Query(rc[x], mid + 1, r,
v));
    return res;
}
```

- 单点修改，询问 $[l,r]$ 内的前缀最大值个数。
  - 定义函数 $query(s,x)$，表示大于等于 $x$ 且是 $s$ 对应的区间内前缀最大值的个数。
    - 线段树上维护区间最大值，记作 $mx_s$，$s$ 的左右子结点分别为 $sL,sR$。
    - 同时在线段树上维护 $rq_s=query(sR,mx_{sL})$。
    - 若 $mx_s\le x$，则 $query(s,x)=0$。
    - 若 $mx_{sL}<x$，则 $query(s,x)=query(sR,x)$。
    - 否则 $query(s,x)=query(sL,x)+rq_s$。
  - 询问时对查询的 $\mathcal{O}(\log n)$ 个区间调用 $query$ 函数再依次合并即可。
  - 单次操作时间复杂度 $\mathcal{O}(\log^2 n)$。

```
inline int query(int s, const point &x, int l, int r)
{
```

```
    if (mx[s] <= x)
        return 0;
    if (l == r)
        return 1;
    int mid = l + r >> 1;
    if (mx[sL] < x)
        return query(sR, x, mid + 1, r);
    return query(sL, x, l, mid) + rq[s];
}

inline void Update(int s, int l, int r)
{   // l 指 mid+1
    mx[s] = Max(mx[sL], mx[sR]);
    rq[s] = query(sR, mx[sL], l, r);
}
```

- 维护区间括号序列至少要改变多少位才能变成合法的括号序列。
  - 将成对的括号去掉，若剩下的括号为 $x$ 个右括号加上 $y$ 个左括号的形式，不难发现答案就是 $\left\lceil\frac{x}{2}\right\rceil+\left\lceil\frac{y}{2}\right\rceil$，因此将 `(` 记作 1，`)` 记作 -1，记区间 $s$ （保证长度为偶数）的前缀最小值为 $pre_s$（最大为空串，0），后缀最大值为 $suf_s$（最小为空串，0），答案即为 $\left\lceil\frac{|pre_s|}{2}\right\rceil+\left\lceil\frac{suf_s}{2}\right\rceil$，可用线段树维护。
- 矩形加，矩形求和，允许离线。
  - 将其中一维看作时间扫描线，矩形求和转化为历史版本和。

# 区间最值/历史最值/历史版本和

- 区间取 $\min$ / $\max$，区间加，询问区间最值/区间和。
  - 先考虑最简单的情况，修改只有区间取 $\min$，记录区间最大值 $maxv$ 及其个数 $maxc$，以及次大值 $sr$，对修改参数 $v$ 分情况讨论：
    - 若 $v\geq maxv$，无需处理。
    - 若 $sr<v<maxv$，相当于将所有 $maxv$ 变成 $v$，也容易处理。
    - 若 $v\leq sr$，递归左右子树。
  - 注意到每次数的种类数减少 1 时，对应一次从线段树上某结点递归到叶子结点的过程，因而时间复杂度为均摊 $\mathcal{O}(\log n)$ 。
  - 区间取 $\max$ 同理，增加区间加操作后时间复杂度上限为均摊 $\mathcal{O}(\log^2 n)$，实际运行结果近似于 $\mathcal{O}(\log n)$，具体证明见吉如一2016年集训队论文。
  - 将最值与非最值作为多类数分开维护，可与其它操作结合。
  - 核心代码如下：

```
int len[N4], maxt[N4], mint[N4], addt[N4];

struct node
{
    int maxv, minv, sl, sr, maxc, minc;
    ll sum;
    // from the top to the bottom: maxv - sr - sl - minv
    // When there are only two kinds of values, only "maxv" and "minv" are valid.
    // At this time, "sl" is equal to +inf and "sr" is equal to -inf.
    inline void check_seg(int a)
    {
        if (a > minv && a < maxv)
```

```
        {
            CkMin(sl, a);
            CkMax(sr, a);
        }
    }

    friend inline node operator + (const node &a, const node &b)
    {
        node c;
        c.sum = a.sum + b.sum;
        c.maxv = Max(a.maxv, b.maxv);
        c.maxc = a.maxc * (a.maxv == c.maxv) + b.maxc * (b.maxv == c.maxv);
        c.minv = Min(a.minv, b.minv);
        c.minc = a.minc * (a.minv == c.minv) + b.minc * (b.minv == c.minv);

        c.sl = Min(a.sl, b.sl);
        c.sr = Max(a.sr, b.sr);
        if (c.maxv != c.minv)
        {
            c.check_seg(a.maxv);
            if (a.maxv != a.minv)
                c.check_seg(a.minv);
            c.check_seg(b.maxv);
            if (b.maxv != b.minv)
                c.check_seg(b.minv);
        }
        return c;
    }
}tr[N4];

inline void addTag(int s, int v)
{
    addt[s] += v;
    tr[s].maxv += v;
    tr[s].minv += v;
    tr[s].sum += 1ll * v * len[s];
    tr[s].sl != Maxn ? tr[s].sl += v : 0;
    tr[s].sr != Minn ? tr[s].sr += v : 0;
    maxt[s] != Minn ? maxt[s] += v : 0;
    mint[s] != Maxn ? mint[s] += v : 0;
}

inline void maxTag(int s, int v)
{
    if (tr[s].minv >= v || maxt[s] >= v)
        return ;
    tr[s].sum += 1ll * (v - tr[s].minv) * tr[s].minc;
    if (v >= tr[s].maxv)
    {
        if (tr[s].maxv != tr[s].minv)
        {
            tr[s].sum = 1ll * len[s] * v;
            tr[s].maxc = tr[s].minc = len[s];
        }
        tr[s].maxv = v;
    }
    tr[s].minv = v;
    CkMax(mint[s], v);
```

```
    maxt[s] = v;
}

inline void minTag(int s, int v)
{
    if (tr[s].maxv <= v || mint[s] <= v)
        return ;
    tr[s].sum += 1ll * (v - tr[s].maxv) * tr[s].maxc;
    if (v <= tr[s].minv)
    {
        if (tr[s].maxv != tr[s].minv)
        {
            tr[s].sum = 1ll * len[s] * v;
            tr[s].maxc = tr[s].minc = len[s];
        }
        tr[s].minv = v;
    }
    tr[s].maxv = v;
    CkMin(maxt[s], v);
    mint[s] = v;
}

inline void pushDown(int s)
{
    if (addt[s] != 0)
    {
        addTag(sL, addt[s]);
        addTag(sR, addt[s]);
        addt[s] = 0;
    }
    if (maxt[s] != Minn)
    {
        maxTag(sL, maxt[s]);
        maxTag(sR, maxt[s]);
        maxt[s] = Minn;
    }
    if (mint[s] != Maxn)
    {
        minTag(sL, mint[s]);
        minTag(sR, mint[s]);
        mint[s] = Maxn;
    }
}

inline void modifyMax(int s, int l, int r, int x, int y, int v)
{
    if (l == x && r == y && v < tr[s].sl)
        return maxTag(s, v);
    pushDown(s);
    int mid = l + r >> 1;
    if (y <= mid)
        modifyMax(sL, l, mid, x, y, v);
    else if (x > mid)
        modifyMax(sR, mid + 1, r, x, y, v);
    else
    {
        modifyMax(sL, l, mid, x, mid, v);
        modifyMax(sR, mid + 1, r, mid + 1, y, v);
```

```
    }
    Update(s);
}

inline void modifyMin(int s, int l, int r, int x, int y, int v)
{
    if (l == x && r == y && v > tr[s].sr)
        return minTag(s, v);
    pushDown(s);
    int mid = l + r >> 1;
    if (y <= mid)
        modifyMin(sL, l, mid, x, y, v);
    else if (x > mid)
        modifyMin(sR, mid + 1, r, x, y, v);
    else
    {
        modifyMin(sL, l, mid, x, mid, v);
        modifyMin(sR, mid + 1, r, mid + 1, y, v);
    }
    Update(s);
}

inline void Build(int s, int l, int r)
{
    len[s] = r - l + 1;
    maxt[s] = Minn;
    mint[s] = Maxn;
    if (l == r)
    {
        tr[s].maxv = tr[s].minv = tr[s].sum = a[l];
        tr[s].maxc = tr[s].minc = 1;
        tr[s].sl = Maxn; tr[s].sr = Minn;
        return ;
    }
    int mid = l + r >> 1;
    Build(sL, l, mid);
    Build(sR, mid + 1, r);
    Update(s);
}
```

- 区间赋值，区间加，区间最值，**区间历史最值。**
  - **在一个区间被赋值后，后续的区间加都可以看作区间赋值**，维护区间加和区间覆盖标记的历史最值即可。

```
struct node
{
    int max, hmax, cov, hcov, add, hadd;
    bool iscov;

    inline void Init()
    {
        cov = hcov = add = hadd = 0;
        iscov = false;
    }

    inline void Cover(int v, int hv)
```

```
    {
        if (iscov)
            CkMax(hcov, hv);
        else
            iscov = true, hcov = hv;
        cov = max = v;
        CkMax(hmax, hv);
        add = 0;
    }

    inline void Add(int v, int hv)
    {
        if (iscov)
            Cover(cov + v, cov + hv);
        else
        {
            CkMax(hadd, add + hv);
            CkMax(hmax, max + hv);
            add += v, max += v;
        }
    }
}tr[N4];

inline void Update(int s)
{
    tr[s].max = Max(tr[sL].max, tr[sR].max);
    tr[s].hmax = Max(tr[sL].hmax, tr[sR].hmax);
}

inline void pushDown(int s)
{
    int &v = tr[s].add, &hv = tr[s].hadd;
    if (v != 0 || hv != 0)
    {
        tr[sL].Add(v, hv);
        tr[sR].Add(v, hv);
        v = hv = 0;
    }
    if (tr[s].iscov)
    {
        tr[sL].Cover(tr[s].cov, tr[s].hcov);
        tr[sR].Cover(tr[s].cov, tr[s].hcov);
        tr[s].iscov = false;
    }
}
```

- 区间取 $\max$，区间加，区间赋值，**单点询问。**
  - 维护标记组 $(a,b)$，表示 $\max(x+a,b)$，则依次执行 $(a,b),(c,d)$ 可合并为 $(a+c,\max(b+c,d))$，另外区间赋值也可表示为 $(-\infty,0)$，只可用于单点询问，但写起来方便很多。
  - 拓展到单点询问历史最值的方法同上。
- 区间加减，**区间历史最小值和。**
  - 设 $A_i$ 为第 $i$ 个位置当前值，$B_i$ 为第 $i$ 个位置历史最小值，利用辅助数组 $C_i=A_i-B_i$，每次操作变为 $C_i=\max\{C_i+x,0\}$，询问分别对 $A,C$ 数组求和后相加即可。
- 区间加减，**区间历史最大值和。**

    - 设 $B_i$ 为第 $i$ 个位置历史最大值，同样利用 $C_i = B_i - A_i$，每次操作后变为 $C_i = \max\{C_i - x, 0\}$。
- 区间加减，**区间历史版本和。**
    - 设 $B_i$ 为第 $i$ 个位置历史版本和，设 $C_i = B_i - tA_i$，$t$ 为当前版本数，当操作为给 $[l, r]$ 加上 $x$ 时，相当于给 $[l, r]$ 减去 $x \cdot t$，转变为简单的区间加减求区间和问题。

# 势能线段树

- 若序列每个元素被修改的次数有一个上限 $K$，则可在每个结点上记录一个值表示该区间是否每个元素都达到修改上限，区间修改暴力递归到叶子结点，若途中遇到区间内每个元素都达到修改上限则停止递归。
- 时间复杂度 $\mathcal{O}(nK \log n)$。

## 经典应用

- 区间开平方，记录区间内最大值，达到修改上限即最大值小于等于 1。
- 区间取模，记录区间内最大值，达到修改上限即最大值小于模数。
- 区间整除、区间加，记录区间内最大值和最小值，每次整除一个数至少使两者之差减少一半，达到修改上限即最大值与最小值相等。

# 线段树合并/分裂/可持久化

- 参考 changle_cyx xyz32768 的学习笔记。

## 线段树合并

- 当遍历到某个结点，若两棵线段树中这个结点有一棵的对应位置是空的，则没必要遍历下去。
- 因此每次合并的时间就是两棵线段树重合的结点数。
- 设所有线段树的总点数为 $M$，每次合并重合部分后，相当于**删去了其中一棵树的那部分结点**，因此将所有线段树合并成一棵的总时间复杂度为 $\mathcal{O}(M)$。
- 可对被合并的线段树进行空间回收。
- **若需要将每次合并的线段树保存下来，需要每次新开结点（将引用改为返回新结点）**，因为此时 $x$ 可能继承自其它线段树的结点，更新其信息会导致其它线段树维护的信息有误。

```
inline void Merge(int &x, int y, int l, int r)
{
    if (!x || !y)
    {
        x = x + y;
        return ;
    }
    int mid = l + r >> 1;
    Merge(lc(x), lc(y), l, mid);
    Merge(rc(x), rc(y), mid + 1, r);
    sze[x] += sze[y];
    deleteNode(y);
}
```

## 线段树分裂

- 将前 $k$ 个存在的叶子结点分裂出去，形成两棵线段树。
- 在线段树上二分即可。

```
inline void Split(int x, int l, int r, int &a, int &b, int k)
{
    if (l == r)
    {
        a = x;
        b = 0;
        return ;
    }
    int mid = l + r >> 1;
    if (k <= cnt[lc[x]])
    {
        a = newNode();
        b = x;
        Split(lc[x], l, mid, lc[a], lc[b], k);
    }
    else
    {
        a = x;
        b = newNode();
        Split(rc[x], mid + 1, r, rc[a], rc[b], k - cnt[lc[x]]);
    }
    Update(a);
    Update(b);
}
```

## 典例1

### 题目大意

- 长度为 $n$ 的序列，$q$ 次询问，每次询问为以下两种操作之一：
  - 对 $[l,r]$ 按照升序/降序排序。
  - 询问 $[l,r]$ 相关信息。

### 解法

- 我们将排序后满足升序/降序的区间成为一个连续段，通过线段树合并/分裂将每一个连续段对应一棵权值线段树，此时元素在序列中的排列顺序与权值线段树中的排列顺序相同，便于在线段树上维护，同时我们用 `set` 或一般的线段树维护连续段的信息。
- 询问时同样可以通过线段树分裂，将其转化为若干个完整连续段的信息并，一般可以将每个连续段的信息记录在其左端点，外层用一个线段树/树状数组维护。
- 每次分裂只会产生 $\mathcal{O}(\log n)$ 个结点，总时间复杂度仍然正确。
- 用 `set` 维护连续段的部分有一定细节，为避免不必要的调试，这里提供一个模板。

```
struct seg
{
    int l, r;
    bool rev; //是否为倒序

    seg() {}
    seg(int L, int R, bool Rev):
```

```
        l(L), r(R), rev(Rev) {}

    inline bool operator < (const seg &a) const
    {
        return l < a.l;
    }
};
set<seg> s;
typedef set<seg>::iterator it;

inline void Cut(int x) //将 x 和 x 右侧的连续段切割开
{
    it p = s.lower_bound(seg(x + 1, 0, false));
    seg t = *--p;
    if (t.r <= x)
        return ;
    s.erase(p);
    /* remove information in t.l */
    int a, b;
    if (!t.rev)
    {
        Split(rt[t.l], 1, n, a, b, x - t.l + 1);
        rt[t.l] = a;
        rt[x + 1] = b;
        /* add information in t.l */
        /* add information in (x + 1) */
    }
    else
    {
        Split(rt[t.l], 1, n, a, b, t.r - x);
        rt[t.l] = b;
        rt[x + 1] = a;
        /* add information in t.l */
        /* add information in (x + 1) */
    }
    s.insert(seg(t.l, x, t.rev));
    s.insert(seg(x + 1, t.r, t.rev));
}

inline void Reverse(int l, int r)
{
    if (l > 1)
        Cut(l - 1);
    Cut(r);

    vector<seg> cur;
    it p = s.lower_bound(seg(l, 0, false));
    cur.emplace_back(*p);
    /* remove information in p->l */
    for (++p; p != s.end() && p->r <= r; ++p)
    {
        /* remove information in p->l */
        Merge(rt[l], rt[p->l], 1, n);
        cur.emplace_back(*p);
    }
    for (seg x : cur)
        s.erase(x);
    s.insert(seg(l, r, rev));
```

```
    /* add information in l */
}
```

## 典例2 HDU7313

### 题目大意

- 给一棵树（点数 $n \leq 10^6$），点有点权 $a_i$，边有边权 $k_i$。
- 设 $f(x,T)$ 表示树 $T$ 中点权等于 $x$ 的点数，$g(y,T)=\max\{x|f(x,T)\geq y\}$
- 将每条边去除后，若得到的两个子树为 $T_1,T_2$，求 $\max\{g(k_i,T_1),g(k_i,T_2)\}$ 。

### 解法

- 子树内的 $g$ 值通过线段树合并和线段树二分很容易求解。
- 对于子树外的 $g$ 值，同样可以用线段树维护，设 $cnt[v]$ 表示整棵树 $v$ 的出现次数，$cnt_0[v]$ 表示子树内 $v$ 的出现次数，线段树根结点维护的是 $\max\{cnt[a_i]-cnt_0[a_i]\}$。
- 考虑归纳地证明维护方式的正确性，额外建出一棵维护整棵树（即 $\max\{cnt[a_i]\}$ 的以 $z$ 为根的线段树，假设以 $x$ 为根的线段树和以 $y$ 为根的线段树已经维护好了，现需合并 $x$ 和 $y$，将 $z$ 作为参数一同传入，并将合并后的线段树存入 $x$ 中：
  - 若 $x$ 和 $y$ 其中之一为空，返回非空的树即可。
  - 若 $x$ 和 $y$ 均递归到叶子，很容易进行修改。
  - 若合并后 $x$ 的左右子树均存在，直接取 $\max$ 维护即可。
  - 若合并后 $x$ 的左右子树不存在，取 $z$ 的左右子树信息用于更新 $x$ 的信息。
- 插入和询问的写法与合并类似，也需要将 $z$ 作为参数传入，具体见代码。

```
inline void pushdownOut(int x, int y)
{
    int _lc = lc(x) ? lc(x) : lc(y),
        _rc = rc(x) ? rc(x) : rc(y);
    mx(x) = Max(mx(_lc), mx(_rc));
}

inline void insertOut(int &x, int y, int l, int r, int u)
{
    if (!x)
        x = newNode();
    if (l == r)
        return (void)(mx(x) = cnt[l] - 1);
    int mid = l + r >> 1;
    u <= mid ? insertOut(lc(x), lc(y), l, mid, u) : insertOut(rc(x), rc(y),
mid + 1, r, u);
    pushdownOut(x, y);
}

inline void mergeOut(int &x, int y, int z, int l, int r)
{
    if (!x || !y)
        return (void)(x += y);
    if (l == r)
        return (void)(mx(x) += mx(y) - cnt[l], deleteNode(y));
    int mid = l + r >> 1;
    mergeOut(lc(x), lc(y), lc(z), l, mid);
    mergeOut(rc(x), rc(y), rc(z), mid + 1, r);
    pushdownOut(x, z);
    deleteNode(y);
```

```
}

inline int queryOut(int x, int y, int l, int r, int v)
{
    if (!x && !y)
        return 0;
    if (l == r)
        return mx(x ? x : y) >= v ? l : 0;
    int mid = l + r >> 1;
    return mx(rc(x) ? rc(x) : rc(y)) >= v ?
        queryOut(rc(x), rc(y), mid + 1, r, v) : queryOut(lc(x), lc(y), l,
mid, v);
}
```

## 树状数组套权值线段树

- 以单点修改、区间求第 $k$ 小为例。
  - 每次修改利用树状数组，在对应 $\mathcal{O}(\log n)$ 棵权值线段树上修改。
  - 每次询问依然是在权值线段树上二分，利用树状数组将 $\mathcal{O}(\log n)$ 棵权值线段树上询问的结果相加。

```
struct node
{
    int lc, rc, cnt;
    #define lc(x) tr[x].lc
    #define rc(x) tr[x].rc
    #define cnt(x) tr[x].cnt
}tr[M];

inline void newNode(int &x)
{
    x = top ? stk[top--] : ++T;
    lc(x) = rc(x) = cnt(x) = 0;
}

inline void delNode(int &x)
{
    stk[++top] = x;
    x = 0;
}

inline void Init(int l, int r)
{
    ql = qr = 0;
    for (int i = l - 1; i; i -= i & -i)
        lx[++ql] = rt[i];
    for (int i = r; i; i -= i & -i)
        rx[++qr] = rt[i];
}

inline void turnL()
{
    for (int i = 1; i <= ql; ++i)
        lx[i] = lc(lx[i]);
    for (int i = 1; i <= qr; ++i)
        rx[i] = lc(rx[i]);
```

```
}

inline void turnR()
{
    for (int i = 1; i <= ql; ++i)
        lx[i] = rc(lx[i]);
    for (int i = 1; i <= qr; ++i)
        rx[i] = rc(rx[i]);
}

inline void Insert(int &x, int l, int r, int v)
{
    if (!x)
        newNode(x);
    ++cnt(x);
    if (l == r) return ;
    int mid = l + r >> 1;
    if (v <= mid) Insert(lc(x), l, mid, v);
        else Insert(rc(x), mid + 1, r, v);
}

inline void Delete(int &x, int l, int r, int v)
{
    --cnt(x);
    if (l == r)
    {
        if (!cnt(x))
            delNode(x);
        return ;
    }
    int mid = l + r >> 1;
    if (v <= mid) Delete(lc(x), l, mid, v);
        else Delete(rc(x), mid + 1, r, v);
    if (!cnt(x))
        delNode(x);
}

inline int queryKth(int l, int r, int v)
{
    if (l == r)
    {
        int res = 0;
        if (flag)
        {
            for (int i = 1; i <= ql; ++i)
                res -= cnt(lx[i]);
            for (int i = 1; i <= qr; ++i)
                res += cnt(rx[i]);
        }
        return res;
    }

    int mid = l + r >> 1;
    if (v <= mid)
        return turnL(), queryKth(l, mid, v);
    else
    {
        int res = 0;
```

```
        for (int i = 1; i <= ql; ++i)
            res -= cnt(lc(lx[i]));
        for (int i = 1; i <= qr; ++i)
            res += cnt(lc(rx[i]));
        return turnR(), res + queryKth(mid + 1, r, v);
    }
}

inline int findKth(int l, int r, int k)
{
    if (l == r) return l;

    int res = 0;
    for (int i = 1; i <= ql; ++i)
        res -= cnt(lc(lx[i]));
    for (int i = 1; i <= qr; ++i)
        res += cnt(lc(rx[i]));

    int mid = l + r >> 1;
    if (k <= res)
        return turnL(), findKth(l, mid, k);
    else
        return turnR(), findKth(mid + 1, r, k - res);
}

inline void bitDelete(int l)
{
    for (int i = l; i <= n; i += i & -i)
        Delete(rt[i], 0, L, a[l]);
}

inline void bitInsert(int l)
{
    for (int i = l; i <= n; i += i & -i)
        Insert(rt[i], 0, L, a[l]);
}
```

# 分块

- 实现时需注意 $[l,r]$ 在同一块内的情况以及分块的边界问题。
- 区间众数。
  - 将序列平均分成 $\sqrt{n}$ 块，预处理 $cnt[i][j]$ 表示元素 $i$ 在前 $j$ 块中出现的次数，$ans[i][j]$ 表示第 $i$ 块到第 $j$ 块的众数。
  - $cnt[i][j]$ 即先枚举 $j$ 后枚举 $i$ 统计，$ans[i][j]$ 即先枚举 $i$，将第 $i$ 块及其之后的数依次加入，用桶维护众数。
  - 询问时设完整块为第 $L$ 块到第 $R$ 块，则答案要么为 $ans[L][R]$，要么为非完整块中的数，暴力枚举即可。
  - 时间复杂度 $\mathcal{O}(n\sqrt{n})$。
- 插入删除元素，询问元素的最大值/最小值。
  - 离散化后按值域分成 $\sqrt{n}$ 块，记录每块中插入元素的数目。
  - 询问时先暴力找到最值所在块，再在块内暴力找到最值。
  - 即可做到 $\mathcal{O}(1)$ 修改，$\mathcal{O}(\sqrt{n})$ 回答询问，可与莫队结合。
- $\mathcal{O}(1)$ 区间加，$\mathcal{O}(\sqrt{n})$ 区间求和。

- 对于序列 $a$，设差分 $\Delta a_i = a_i - a_{i-1}$，维护 $\sum\limits_{i=1}^{n} \Delta a_i$ 和 $\sum\limits_{i=1}^{n} (i-1)\Delta a_i$，则：

$$\sum_{i=1}^{n} a_i = n\sum_{i=1}^{n} \Delta a_i - \sum_{i=1}^{n} (i-1)\Delta a_i$$

  - 维护单点的差分标记和整块的差分标记即可。
- $\mathcal{O}(1)$ 单点插入删除，$\mathcal{O}(\sqrt{n})$ 询问第 $k$ 小数。
  - 值域分块后，询问逐块查找即可。
- $\mathcal{O}(\sqrt{n})$ 单点插入删除，$\mathcal{O}(1)$ 询问第 $k$ 小数。
  - 按排名分块，块大小固定，每块按排名维护一个 `deque`，修改时暴力找到对应的 `deque`，之后对前后的 `deque` 的首尾进行调整（`push/pop_front()` 和 `push/pop_back()`），询问直接通过下标访问对应的 `deque` 即可（队首从 0 开始编号）。

# 莫队

- 允许离线，无修改，询问区间。
  - 将序列平均分成 $\sqrt{n}$ 块，给每个位置按顺序标上所在块的编号，将询问 $[l,r]$ 按**左端点所在块**为第一关键字，**右端点所在位置的编号**为第二关键字排序，易分析出总时间复杂度为 $\mathcal{O}(n\sqrt{n})$。
  - 单次移动可与其它传统数据结构结合，设单次移动时间复杂度为 $\mathcal{O}(k)$，总时间复杂度 $\mathcal{O}(kn\sqrt{n})$。
  - 若单次移动时间复杂度为 $\mathcal{O}(1)$，单次询问结合分块实现，时间复杂度为 $\mathcal{O}(\sqrt{n})$，总时间复杂度依然为 $\mathcal{O}(n\sqrt{n})$。
- 允许离线，带修改，询问区间。
  - 将序列所有点分块，块的大小为 $n^{\frac{2}{3}}$，共有 $n^{\frac{1}{3}}$ 个块，将询问按**左端点所在块**为第一关键字，**右端点所在块**为第二关键字，**询问的时间**为第三关键字进行排序，易分析出总时间复杂度为 $\mathcal{O}(n^{\frac{5}{3}})$。
  - 时间指针的移动即修改操作正向和逆向的进行。

```
    for (int i = 1, l, r; i <= m; ++i)
    {
        char ch;
        while (ch = getchar(), ch != 'R' && ch != 'Q');
        if (ch == 'Q')
        {
            ++qm;
            q[qm].scan(pm, qm);
        }
        else
        {
            read(l); read(r);
            p[++pm] = modify(l, _a[l], r);
            _a[l] = r;
        }
    }
    std::sort(q + 1, q + qm + 1);

    int tl = 1, tr = 0, tt = 0;
    for (int i = 1; i <= qm; ++i)
    {
        int l = q[i].l, r = q[i].r;
        while (tt < q[i].t)
        {
```

```
            modify b = p[++tt];
            if (b.x >= tl && b.x <= tr)
            {
                if (!--cnt[b.pre])
                    --ans;
                if (!cnt[b.suf]++)
                    ++ans;
            }
            a[b.x] = b.suf;
        }
        while (tt > q[i].t)
        {
            modify b = p[tt--];
            if (b.x >= tl && b.x <= tr)
            {
                if (!--cnt[b.suf])
                    --ans;
                if (!cnt[b.pre]++)
                    ++ans;
            }
            a[b.x] = b.pre;
        }
        while (tl < l)
            if (!--cnt[a[tl++]])
                --ans;
        while (tl > l)
            if (!cnt[a[--tl]]++)
                ++ans;
        while (tr > r)
            if (!--cnt[a[tr--]])
                --ans;
        while (tr < r)
            if (!cnt[a[++tr]]++)
                ++ans;
        fans[q[i].id] = ans;
    }
```

- 高维莫队的情况有时可用差分拆成几个询问降成低维。
- 一般地，对于 $k$ 维莫队，对前 $k-1$ 维分块，关键字为所在块编号，第 $k$ 维关键字为第 $k$ 维坐标，设单维的最大长度为 $n$，操作次数为 $m$，取块大小为 $\frac{n}{m^{\frac{1}{k}}}$ 时最优，时间复杂度 $\mathcal{O}(nm^{\frac{k-1}{k}})$。

# 树上莫队

- 允许离线，询问树上路径。
- 考虑将树上路径转化为序列上的区间。
- 欧拉序：DFS 遍历整棵树，访问到 $x$ 时，加入序列，访问完 $x$ 的子树，再加入序列，记 $x$ 两次加入序列的编号分别为 $st[x], ed[x]$。
- 考虑树上路径 $x \to y$，不妨设 $st[x] < st[y]$。
  - 若 $\mathrm{LCA}(x,y) = x$，则统计 $[st[x], st[y]]$ 只出现一次的点的贡献。
  - 若 $\mathrm{LCA}(x,y) \neq x$，则统计 $[ed[x], st[y]]$ 只出现一次的点的贡献，另外需要补上 $\mathrm{LCA}(x,y)$ 的贡献。
- 实现时可以另外记录 $used[x] = 0/1$ 表示指针经过了点 $x$ 偶数次/奇数次，来判断是将该点的贡献插入还是删除。

- 时间复杂度分析、带修改的处理同一般莫队相同。

# 回滚莫队

- 允许离线，无修改，询问区间，单次移动插入远比删除容易。
- 设法调整指针移动顺序，避免删除操作。
- 将序列平均分成 $\sqrt{n}$ 块，左右端点所在块相同的询问暴力处理，将其余询问按**左端点所在块**分组，每组按照**右端点**从小到大排序。
- 因为每组内右端点单调，从所在块右端点开始移动即可，每组总移动次数为 $\mathcal{O}(n)$ 。
- 对于每次询问，将左端点从所在块右端点开始移动，每次移动次数为 $\mathcal{O}(\sqrt{n})$ 。
- 用栈记录移动左端点发生改变的变量的地址和原来的值，处理完每个询问后还原回去。
- 总时间复杂度 $\mathcal{O}(n\sqrt{n})$。

## 典例1 洛谷P6072

### 题目大意

- 给定一棵 $n(n \leq 3 \times 10^4)$ 个点的无根树，边有边权。
- 选择两条简单路径，满足没有重合的点，且边权异或和之和最大。

### 算法一

- 设 $in[x], out[x]$ 分别表示在以 $x$ 为根的子树内/外选一条路径的异或和最大值，答案即 $\max\limits_{1\leq x\leq n}\{in[x]+out[x]\}$。
- 设 $x$ 在 DFS序 中编号为 $dfn[x]$，子树大小为 $size[x]$，可将 $in[x], out[x]$ 的求解表示成下面两种询问：
  - 求在 $[dfn[x], dfn[x]+size[x]-1]$ 任取两个数异或的最大值。
  - 求在 $[1, dfn[x]-1]\cup[dfn[x]+size[x], n]$ 任取两个数异或的最大值。
- 可将序列复制一遍，使第二种询问也变为连续的区间。
- 套用回滚莫队模板即可，时间复杂度 $\mathcal{O}(n\sqrt{n}\log w)$。

### 算法二

- 先求出最大异或和路径的两个端点 $dx, dy$，令树根为 $dx$ 。
- 不在该路径上的点被划分成若干个互不相干的子树，若最终的答案包含路径 $(dx, dy)$，暴力求这些子树内路径的最大异或和即可。
- 若最终的答案不包含 $(dx, dy)$，只需要求这条路径上所有点的 $in[x], out[x]$ 即可，类似算法一中的转换，由于此时询问的区间均为包含关系，每个数插入 Trie 的次数为 $\mathcal{O}(1)$，同样可以暴力求解。
- 时间复杂度 $\mathcal{O}(n\log w)$。

## 典例2 洛谷P5386

### 题目大意

- 给定一个长度为 $n$ 的一个 1~$n$ 的排列 $A_{1-n}$。
- 给定 $q$ 个询问四元组 $(l, r, x, y)$ :
- 表示询问 有多少个二元组 $(u, v)$ 满足 :
  - $[u, v] \neq \varnothing$
  - $[u, v] \subset [l, r]$
  - $\min\limits_{i\in[u,v]}\{A_i\} \geq x$
  - $\max\limits_{i\in[u,v]}\{A_i\} \leq y$

### 题解

- 令 $B_{A_i}=i$，对于每个询问 $(l,r,x,y)$，令 $C_{B_i}=1(x\le i\le y)$，则问题转化对于数组 $C$ 的 $[l,r]$，设每段连续 1 的长度为 $len_i$，求 $\sum\frac{len_i(len_i+1)}{2}$。
- 容易想到外层套一个莫队，内层用线段树维护答案，时间复杂度 $\mathcal{O}(n\sqrt{n}\log n)$，常数过大难以通过。
- 将内层改为分块，并用并查集维护连续的 1，因为不方便删除将外层改为回滚莫队，时间复杂度不变，但常数小了很多。
- 注意到每个位置的插入操作至多只有一次，并且我们在合并连续 1 的过程中只关心每段连续 1 的起点和终点，我们只需要在每段连续 1 的起点处记录终点、在终点处记录起点即可 $\mathcal{O}(1)$ 完成合并。
- 时间复杂度 $\mathcal{O}(n\sqrt{n})$。

## 二次离线莫队

- 设 $f(x,l,r)$ 表示已经插入了区间 $[l,r]$、现在插入单点 $x$ 对答案产生的贡献。
- 一般莫队共有四种指针的移动，设当前区间为 $[tl,tr]$，目标区间为 $[l,r]$，这里以 $tr<r$ 为例，其它三种移动同理。
- 每次移动即

$$[tl,x-1]\to[tl,x]\ (tr<x\le r)$$

- 记 $F(x,r)=f(x,1,r)$，对答案产生的贡献为

$$f(x,tl,x-1)=F(x,x-1)-F(x,tl-1)$$

- 其中 $F(x,x-1)$ 很容易预处理，$F(x,tl-1)$ 则可以通过扫描线将询问区间 $[tr+1,r]$ 挂在 $tl-1$ 处暴力询问得到。
- 如上所述，二次离线莫队即是把莫队移动的操作预处理以降低总的时间复杂度。
- 常见于询问区间内符合某种性质的点对数，莫队的单次移动可看一次询问（查询符合条件的点数）和一次修改（加入该点），设单次询问的复杂度为 $\mathcal{O}(f(n))$，单次修改的复杂度为 $\mathcal{O}(g(n))$，则上述做法使总时间复杂度由 $\mathcal{O}(n\sqrt{n}(f(n)+g(n)))$ 降至 $\mathcal{O}(ng(n)+n\sqrt{n}f(n))$，且空间复杂度依然为 $\mathcal{O}(n)$。

```cpp
// 假定 F(x, x) = F(x, x - 1)，sum 数组为 F(x, x - 1) 的前缀和
// 以下为扫描线预处理部分，注意 ans 数组存储的只是最终答案的差分形式
for (int i = 1; i <= m; ++i)
{
    int l = q[i].l, r = q[i].r;
    if (tr < r)
    {
        v[tl - 1].push_back(segQuery(tr + 1, r, -1, i));
        ans[i] += sum[r] - sum[tr];
        tr = r;
    }
    if (tr > r)
    {
        v[tl - 1].push_back(segQuery(r + 1, tr, 1, i));
        ans[i] -= sum[tr] - sum[r];
        tr = r;
    }
    if (tl < l)
    {
        v[tr].push_back(segQuery(tl, l - 1, -1, i));
        ans[i] += sum[l - 1] - sum[tl - 1];
        tl = l;
    }
```

```
        if (tl > l)
        {
            v[tr].push_back(segQuery(l, tl - 1, 1, i));
            ans[i] -= sum[tl - 1] - sum[l - 1];
            tl = l;
        }
    }
```

# 树上启发式合并

- 即 dsu on tree，解决的问题类型如下，常见的有子树数内外数颜色和求众数等：
  - 允许离线。
  - 询问关于以某点 $x$ 为根的子树内的信息。
  - 询问的信息在遍历子树时容易维护。
- 先将所有询问挂在对应的子树根结点 $x$ 上，考虑先遍历子结点 $y$ 的子树处理其询问，除了最后一个子树外，每遍历完一棵子树就要清除它的影响，而最后一棵子树的信息则可以继承给 $x$。
- 我们令最后一棵子树为以重儿子为根的子树，由重链剖分的性质，从根到某结点的路径上的轻边个数为 $\mathcal{O}(\log n)$，因此每个点被清除的次数为 $\mathcal{O}(\log n)$。
- 设计算单点贡献的时间复杂度为 $\mathcal{O}(k)$，总时间复杂度 $\mathcal{O}(kn\log n)$。
- 预处理同重链剖分，具体算法流程如下：
  1. 遍历所有轻儿子，处理完某一个轻儿子后就清除它的影响。
  2. 遍历重儿子。
  3. 加上子树根结点 $x$ 的贡献。
  4. 回答询问。

```
inline void addSubtree(int x)
{
    for (int i = pos[x], im = pos[x] + sze[x] - 1; i <= im; ++i)
        addCol(col[idx[i]]);
}

inline void decSubtree(int x)
{
    for (int i = pos[x], im = pos[x] + sze[x] - 1; i <= im; ++i)
        decCol(col[idx[i]]);
}

inline void dfsTraverse(int x)
{
    for (arc *e = adj[x]; e; e = e->nxt)
    {
        int y = e->to;
        if (y == fa[x] || y == son[x])
            continue ;
        dfsTraverse(y);
        decSubtree(y);
    }
    if (son[x])
        dfsTraverse(son[x]);
    addCol(col[x]);
    for (arc *e = adj[x]; e; e = e->nxt)
    {
        int y = e->to;
```

```
        if (y == fa[x] || y == son[x])
            continue ;
        addSubtree(y);
    }
    for (int i = 0, im = ask[x].size(); i < im; ++i)
    {
        pair<int, int> y = ask[x][i];
        ans[y.second] = sum[y.first];
    }
}
```

- 常见的子树数颜色问题实际上还有另一种比较套路的做法：
  - 将同种颜色的所有点按照 DFS 序排序，在各自的位置 $+1$，在相邻两点的 LCA 处 $-1$。
  - 询问子树内颜色种数即子树求和。

# 整体二分

- 主要思想即将多个询问一起二分答案，根据判定的结果分治。
- 使用整体二分的题目应满足如下性质：
  - 询问的答案具有可二分性，且允许离线。
  - 修改对判定答案的贡献相互独立，相互之间不影响效果。
  - 修改若对判定答案有贡献，贡献为一确定的与判定标准无关的值。
  - 贡献满足交换律，结合律，具有可加性。

## 典例 [ZJOI2013]K大数查询

### 题目大意

- 初始给定 $n$ 个可重整数集，初始为空。
- $m$ 个操作为以下两者之一：
  - 将数 $c$ 加到编号为 $[l,r]$ 内的集合中。
  - 查询编号为 $[l,r]$ 内集合并集的第 $c$ 大数。

### 解法

- 为方便起见，对修改的数取反后转化为求第 $k$ 小数。
- 二分答案后将小于等于答案的修改转化成区间加一，具体见代码。

```
inline void solve(int tl, int tr, int l, int r)
{
    if (tl > tr)
        return ;
    if (l == r)
    {
        for (int i = tl; i <= tr; ++i)
            if (k[cur[i]] == 2)
                ans[cur[i]] = l;
        return ;
    }

    int dl = 0, dr = 0, mid = l + r >> 1, tm;
    ++tis; // 对 BIT 做时间戳标记，免去清空
    for (int i = tl; i <= tr; ++i)
        if (k[cur[i]] == 1)
```

```
        {
            if (c[cur[i]] <= mid)
            {
                h1[++dl] = cur[i];
                secModify(a[cur[i]], b[cur[i]]);
            }
            else
                h2[++dr] = cur[i];
        }
        else
        {
            ll tmp = segQuery(a[cur[i]], b[cur[i]]);
            if (c[cur[i]] > tmp)
            {
                h2[++dr] = cur[i];
                c[cur[i]] -= tmp;
            }
            else
                h1[++dl] = cur[i];
        }
    tm = tl + dl;
    for (int i = tl; i < tm; ++i)
        cur[i] = h1[i - tl + 1];
    for (int i = tm; i <= tr; ++i)
        cur[i] = h2[i - tm + 1];

    solve(tl, tm - 1, l, mid);
    solve(tm, tr, mid + 1, r);
}
```

# Splay

- 注意维护和标记下传的细节。
- 注意时间复杂度是均摊 $\mathcal{O}(n \log n)$，每种操作过后均需执行 `Splay` **至少一次**使树的形态改变，防止被特殊构造的数据针对。
- **启发式合并** 多次将较小的 Splay 中所有结点按中序遍历依次取出插入较大的 Splay，可以证明总的时间复杂度为均摊 $\mathcal{O}(n \log n)$。

```
const int N = 2e6 + 5;
int n, bm, m, T_splay, rt;
int a[N], b[N], c[N];
int rev[N], val[N], fa[N], lc[N], rc[N], sze[N], cnt[N];

inline void addRev(int x)
{
    if (!x) return ;
    rev[x] ^= 1;
    std::swap(lc[x], rc[x]);
}

inline void pushDown(int x)
{
    if (rev[x])
    {
        addRev(lc[x]);
        addRev(rc[x]);
```

```
        rev[x] = 0;
    }
}

inline void Update(int x)
{
    sze[x] = sze[lc[x]] + sze[rc[x]] + cnt[x];
}

inline void Rotate(int x)
{
    int y = fa[x], z = fa[y];
    pushDown(y);
    pushDown(x);
    bool flag = lc[y] == x;
    int b = flag ? rc[x] : lc[x];
    fa[x] = z, fa[y] = x;
    b ? fa[b] = y : 0;
    z ? (lc[z] == y ? lc[z] : rc[z]) = x : 0;
    flag ? (rc[x] = y, lc[y] = b) : (lc[x] = y, rc[y] = b);
    Update(y);
}

inline bool whichSide(int x)
{
    return rc[fa[x]] == x;
}

inline void Splay(int x, int tar)
{
    while (fa[x] != tar)
    {
        if (fa[fa[x]] != tar)
            Rotate(whichSide(fa[x]) == whichSide(x) ? fa[x] : x);
        Rotate(x);
    }
    Update(x);
    !tar ? rt = x : 0;
}

inline void Insert(int v)
{
    int x = rt, y = 0, dir;
    while (x)
    {
        pushDown(x);
        ++sze[y = x];
        if (val[x] == v)
        {
            ++cnt[x];
            Splay(x, 0);
            return ;
        }
        if (v < val[x])
            dir = 0, x = lc[x];
        else
            dir = 1, x = rc[x];
    }
```

```
    fa[x = ++T_splay] = y;
    val[x] = v;
    sze[x] = cnt[x] = 1;
    y ? (dir ? rc[y] : lc[y]) = x : 0;
    Splay(x, 0);
}

inline int Find(int v)
{
    int x = rt;
    while (x)
    {
        pushDown(x);
        if (val[x] == v)
            return Splay(x, 0), x;
        x = v < val[x] ? lc[x] : rc[x];
    }
    return 0;
}

inline int getKth(int k)
{
    int x = rt;
    if (sze[x] < k)
        return 0;
    while (x)
    {
        pushDown(x);
        if (k <= sze[lc[x]])
            x = lc[x];
        else
        {
            k -= sze[lc[x]] + cnt[x];
            if (k <= 0)
                return Splay(x, 0), x;
            x = rc[x];
        }
    }
    return 0;
}

inline int getRank(int v)
{
    int x = rt, y, k = 1;
    while (x)
    {
        pushDown(x);
        y = x;
        if (val[x] == v)
        {
            k += sze[lc[x]];
            Splay(y, 0);
            return k;
        }
        if (val[x] < v)
            k += sze[lc[x]] + cnt[x], x = rc[x];
        else
            x = lc[x];
```

```
    }
    return Splay(y, 0), k;
}

inline int findPre(int v)
{
    int x = rt, res = 0;
    while (x)
    {
        pushDown(x);
        if (val[x] < v)
            res = x, x = rc[x];
        else
            x = lc[x];
    }
    Splay(res, 0);
    return val[res];
}

inline int findSuf(int v)
{
    int x = rt, res = 0;
    while (x)
    {
        pushDown(x);
        if (val[x] > v)
            res = x, x = lc[x];
        else
            x = rc[x];
    }
    Splay(res, 0);
    return val[res];
}

inline void Join(int x, int y)
{
    int k = y;
    pushDown(k);
    while (lc[k])
    {
        k = lc[k];
        pushDown(k);
    }
    lc[k] = x;
    fa[x] = k;
    fa[rt = y] = 0;
    Splay(k, 0);
}

inline void Delete(int v)
{
    int x = Find(v);
    if (cnt[x] > 1)
    {
        --cnt[x];
        --sze[x];
        return ;
    }
```

```
    if (!lc[x] || !rc[x])
        fa[rt = lc[x] + rc[x]] = 0;
    else
        Join(lc[x], rc[x]);
}

inline int Build(int _fa, int l, int r)
{ // 如是需要对原序列 a 按权值大小建树，数组 b 为排序去重后的结果，数组 c 为对应权值的种数
    if (l > r)
        return 0;
    int mid = l + r >> 1, x = ++T_splay;
    fa[x] = _fa;
    val[x] = b[mid];
    cnt[x] = c[mid];
    lc[x] = Build(x, l, mid - 1);
    rc[x] = Build(x, mid + 1, r);
    return Update(x), x;
}

inline void Reverse(int l, int r)
{
    int x = getKth(l),
        y = getKth(r + 2);
    Splay(x, 0);
    Splay(y, x);
    addRev(lc[y]);
}

inline void Print(int x)
{
    if (!x)
        return ;
    pushDown(x);
    Print(lc[x]);
    if (val[x] != 0)
        put(val[x]), putchar(' ');
    Print(rc[x]);
}
```

# Link-Cut Tree

- 对原树做实虚链剖分，对每条实链用 Splay 以原树深度为权值维护，在每棵 Splay 的根结点处记录该条实链深度最小的点在原树上的父结点，实现时一并记录在 $fa$ 数组中即可。
- 注意 LCT 的 `Splay` 操作与一般的 Splay 有所不同，一般的 Splay 在找到特定结点前都会经过根结点到该结点的路径，从而完成标记下传，但 LCT 一般都是直接指定某个结点进行操作，`Splay` 前需在对应的 Splay 中先完成从根结点到该节点的标记下传。
- 常见操作的实现见代码。

```
const int N = 3e5 + 5;
int qr, que[N], val[N], rev[N], lc[N], rc[N], fa[N], sze[N];

inline bool whichSide(int x)
{
    return lc[fa[x]] == x;
}
```

```
8
9  inline bool isRoot(int x)
10 {
11     return lc[fa[x]] != x && rc[fa[x]] != x;
12 }
13
14 inline void Update(int x)
15 {
16     sze[x] = sze[lc[x]] + sze[rc[x]] + 1;
17 }
18
19 inline void addRev(int x)
20 {
21     if (!x)
22         return ;
23     rev[x] ^= 1;
24     std::swap(lc[x], rc[x]);
25 }
26
27 inline void pushDown(int x)
28 {
29     if (rev[x])
30     {
31         addRev(lc[x]);
32         addRev(rc[x]);
33         rev[x] = 0;
34     }
35 }
36
37 inline void Rotate(int x)
38 {
39     int y = fa[x], z = fa[y];
40     bool flag = lc[y] == x;
41     int b = flag ? rc[x] : lc[x];
42     !isRoot(y) ? (lc[z] == y ? lc[z] : rc[z]) = x : 0;
43     fa[x] = z, fa[y] = x;
44     b ? fa[b] = y : 0;
45     flag ? (rc[x] = y, lc[y] = b) : (lc[x] = y, rc[y] = b);
46     Update(y);
47 }
48
49 inline void Splay(int x)
50 {
51     que[qr = 1] = x;
52     for (int y = x; !isRoot(y); y = fa[y])
53         que[++qr] = fa[y];
54     for (int i = qr; i >= 1; --i)
55         pushDown(que[i]);
56     while (!isRoot(x))
57     {
58         if (!isRoot(fa[x]))
59             Rotate(whichSide(fa[x]) == whichSide(x) ? fa[x] : x);
60         Rotate(x);
61     }
62     Update(x);
63 }
64
65 inline void Access(int x)
```

```
{
    for (int y = 0; x; y = x, x = fa[x])
    {
        Splay(x);
        rc[x] = y;
        Update(x);
    }
}

inline void makeRoot(int x)
{
    Access(x);
    Splay(x);
    addRev(x);
}

inline int findRoot(int x)
{
    Access(x);
    Splay(x);
    while (pushDown(x), lc[x])
        x = lc[x];
    return x;
}

inline void Link(int x, int y)
{
    makeRoot(x);
    fa[x] = y;
}

inline void Cut(int x, int y)
{
    makeRoot(x);
    Access(y);
    Splay(y);
    lc[y] = fa[x] = 0;
    Update(y);
}

inline int Select(int x, int y)
{
    makeRoot(x);
    Access(y);
    Splay(y);
    return y;
}
```

## 常见应用

### 动态加边维护边双连通分量

- 每次加边时若两端点已连通，取出这条路径的 Splay，暴力遍历用并查集缩为其根结点，只需在 LCT 中修改以下两处就能保证每次 `Rotate` 操作询问 $fa$ 数组的正确性。

```
inline bool isRoot(int x)
{
```

```
    fa[x] = ufs_find(fa[x]);
    return lc[fa[x]] != x && rc[fa[x]] != x;
}

inline void Access(int x)
{
    for (int y = 0; x; y = x, x = ufs_find(fa[x]))
    {
        Splay(x);
        rc[x] = y;
        Update(x);
    }
}
```

### 动态加边维护最小生成树

- 将每条边视作一个有点权的点，每次加边时若两端点已连通，就尝试删去两点路径上权值最大的边，类似的方法可以推广至多种生成树问题。

### 询问以任意点为根的子树大小/权值和

- 在每个点用 $vsze$ 数组维护虚子树大小，若询问以 $rt$ 为根时 $x$ 的子树大小只需要执行完 `makeRoot(rt); Access(x);` 后输出 $vsze[x]$ 即可，具体实现区别如下。

```
inline void Update(int x)
{
    sze[x] = sze[lc[x]] + sze[rc[x]] + vsze[x] + 1;
}

inline void Access(int x)
{
    for (int y = 0; x; y = x, x = fa[x])
    {
        Splay(x);
        vsze[x] += sze[rc[x]];
        rc[x] = y;
        vsze[x] -= sze[rc[x]];
        Update(x);
    }
}

inline void Link(int x, int y)
{
    makeRoot(x);
    Access(y);
    Splay(y);
    fa[x] = y;
    vsze[y] += sze[x];
    Update(y);
}
```

## K-D Tree

- 用于维护 $K$ 维空间中点集的数据结构，在每个结点维护包含以该结点为根的子树内所有点的最小 $K$ 维矩体，支持查询被某一 $K$ 维矩体包含的点集的信息，单次查询的最坏时间复杂度 $\mathcal{O}(N^{1-\frac{1}{K}})$ 。

- 使用 K-D Tree 有以下几点需要注意：
  - 每个结点维护的最小 $K$ 维矩体也可用于查找某些函数最值的剪枝（如查询距离某点最近的点），但查询复杂度不明，只能用于骗分。
  - $\mathcal{O}(N^{1-\frac{1}{K}})$ 是静态建树后每次查询的复杂度，若需支持动态插入，需要采用类似替罪羊树的写法，但理论时间复杂度会增加，尽量不要使用。
  - K-D Tree 并不支持单点查询，因为 `nth_element` 可能会使左右子树中均有与当前结点在当前维度坐标相同的结点，因此重复结点需要分开存储，且需要采用类似查询 $K$ 维矩体的写法，一次查询会找到所有重复点。
  - 删除一般采用懒惰删除，同样需要注意上一项中的问题。
- 更多细节可参考 KDT小记。

```
// 含类似替罪羊树实现的动态插入
const int K = 2;
const double alpha = 0.6;
int top, op, T, rt, m; ll sum[N], val[N];
int stk[N], erav[N], sze[N], cur[N], lc[N], rc[N];
bool del[N];

struct point
{
    int a[K];

    point() {}
    point(int v)
    {
        for (int i = 0; i < K; ++i)
            a[i] = v;
    }

    inline void scan()
    {
        for (int i = 0; i < K; ++i)
            read(a[i]);
    }

    inline bool operator == (const point &x) const
    {
        for (int i = 0; i < K; ++i)
            if (a[i] != x.a[i])
                return false;
        return true;
    }
}p[N], erap[N];

const point minPoint = point(Minn);
const point maxPoint = point(Maxn);

struct rect
{
    point tl, tr;

    rect() {}
    rect(point Tl, point Tr):
        tl(Tl), tr(Tr) {}

    inline void updatePoint(const point &x)
```

```
    {
        for (int i = 0; i < K; ++i)
            CkMin(tl.a[i], x.a[i]);
        for (int i = 0; i < K; ++i)
            CkMax(tr.a[i], x.a[i]);
    }

    inline void updateRect(const rect &x)
    {
        for (int i = 0; i < K; ++i)
            CkMin(tl.a[i], x.tl.a[i]);
        for (int i = 0; i < K; ++i)
            CkMax(tr.a[i], x.tr.a[i]);
    }

    inline bool inPoint(const point &x) const
    {   // whether point x is in the rectangle
        for (int i = 0; i < K; ++i)
            if (x.a[i] < tl.a[i] || x.a[i] > tr.a[i])
                return false;
        return true;
    }

    inline bool intersectRect(const rect &x) const
    {   // whether rectangle x intersects with the rectangle
        for (int i = 0; i < K; ++i)
            if (x.tr.a[i] < tl.a[i] || x.tl.a[i] > tr.a[i])
                return false;
        return true;
    }

    inline bool inRect(const rect &x) const
    {   // whether rectangle x is in the rectangle
        for (int i = 0; i < K; ++i)
            if (x.tl.a[i] < tl.a[i] || x.tr.a[i] > tr.a[i])
                return false;
        return true;
    }
}tr[N];

const rect nullRect = rect(maxPoint, minPoint);

inline bool cmp(const int &x, const int &y)
{
    return erap[x].a[op] < erap[y].a[op];
}

inline void Update(int x)
{
    sze[x] = 1;
    if (del[x])
    {
        tr[x] = nullRect;
        sum[x] = 0;
    }
    else
    {
        tr[x] = rect(p[x], p[x]);
```

```
        sum[x] = val[x];
    }
    if (lc[x])
    {
        sze[x] += sze[lc[x]];
        sum[x] += sum[lc[x]];
        tr[x].updateRect(tr[lc[x]]);
    }
    if (rc[x])
    {
        sze[x] += sze[rc[x]];
        sum[x] += sum[rc[x]];
        tr[x].updateRect(tr[rc[x]]);
    }
}

inline ll querySum(int x, const rect &u)
{
    if (!x || !tr[x].intersectRect(u))
        return 0;
    if (u.inRect(tr[x]))
        return sum[x];
    ll res = 0;
    if (u.inPoint(p[x]))
        res += val[x];
    res += querySum(lc[x], u);
    res += querySum(rc[x], u);
    return res;
}

inline int newNode(ll v, const point &u)
{
    int x = top ? stk[top--] : ++T;
    lc[x] = rc[x] = 0;
    tr[x] = rect(u, u);
    p[x] = u;
    sze[x] = 1;
    val[x] = sum[x] = v;
    del[x] = false;
    return x;
}

inline void Build(int &x, int l, int r, int _op)
{   // 若只需静态建树，请预先将结点信息存入 erap 和 erav 中
    if (l > r)
        return (void)(x = 0);
    op = _op;
    int mid = l + r >> 1;
    std::nth_element(cur + l, cur + mid, cur + r + 1, cmp);

    x = newNode(erav[cur[mid]], erap[cur[mid]]);
    int nxt_op = _op + 1 == K ? 0 : _op + 1;
    Build(lc[x], l, mid - 1, nxt_op);
    Build(rc[x], mid + 1, r, nxt_op);
    Update(x);
}

inline bool inBalanced(int x)
```

```
{
    return sze[x] * alpha < Max(sze[lc[x]], sze[rc[x]]);
}

inline void delNode(int x)
{
    lc[x] = rc[x] = sze[x] = sum[x] = val[x] = 0;
    tr[x] = nullRect;
    stk[++top] = x;
    return ;
}

inline void Erase(int x)
{
    if (!x)
        return ;
    Erase(lc[x]);
    Erase(rc[x]);

    ++m;
    erap[m] = p[x];
    erav[m] = val[x];
    delNode(x);
}

inline void reBuild(int &x, int _op)
{
    m = 0;
    Erase(x);
    for (int i = 1; i <= m; ++i)
        cur[i] = i;
    Build(x, 1, m, _op);
}

inline void Insert(int &x, const point &u, ll v, int _op)
{
    if (!x)
        return (void)(x = newNode(v, u));
    int nxt_op = _op + 1 == K ? 0 : _op + 1;
    u.a[_op] < p[x].a[_op] ? Insert(lc[x], u, v, nxt_op) : Insert(rc[x], u, v, nxt_op);
    Update(x);
    int _sze = sze[x];
    if (inBalanced(x))
        reBuild(x, _op);
    assert(_sze == sze[x]);
}

inline void Delete(int x, const rect &u)
{   // 将以结点 x 为根的子树中所有被 K 维矩体 u 包含的结点删除
    if (!x || !tr[x].intersectRect(u))
        return ;
    if (u.inPoint(p[x]))
        del[x] = true;
    Delete(lc[x], u);
    Delete(rc[x], u);
    Update(x);
}
```

# 块状链表

- 大致思想是用链表维护块，块内是数组，块大小大致为 $\sqrt{n}$。
- 可以支持插入/删除一段区间以及在块内维护信息，为维护块大小大致在 $\sqrt{n}$ 附近，插入元素后若单块的大小超过了 $2\sqrt{n}$，将该块分裂，删除元素后若相邻两块大小之和小于 $\sqrt{n}$，则将两块合并。
- 绝大部分情况下，平衡树是其很好的替代品，实现复杂程度接近甚至更优，且时间复杂度优秀，下面仅给出单点插入/区间删除/单点询问的实现。

```
const int S = 1e3;
const int S2 = 2e3;
int tot_len;

struct node
{
    node *nxt;
    int sze, tag;
    int b[S2 + 5];

    node()
    {
        sze = 0;
        nxt = NULL;
    }

    inline void Push(int c)
    {
        b[sze++] = c;
    }
}*head = NULL;

inline void checkLarge(node *p)
{
    if (p->sze >= S2)
    {
        node *q = new node;
        for (int i = S; i < p->sze; ++i)
            q->Push(p->b[i]);
        p->sze = S;
        q->nxt = p->nxt;
        p->nxt = q;
    }
}

inline void checkSmall(node *p)
{
    if (p->nxt && p->sze + p->nxt->sze <= S)
    {
        node *q = p->nxt;
        for (int i = 0; i < q->sze; ++i)
            p->Push(q->b[i]);
        p->nxt = q->nxt;
        return ;
    }
}

```

```
48  inline void Insert(int c, int pos)
49  {
50      node *p = head;
51      int cur, cnt;
52      if (pos >= ++tot_len)
53      {
54          while (p->nxt != NULL)
55              p = p->nxt;
56          p->Push(c);
57          checkLarge(p);
58          return ;
59      }
60      for (cur = head->sze; p != NULL && cur < pos; p = p->nxt, cur += p-
    >sze);
61      cur -= p->sze;
62      cnt = pos - cur - 1;
63      for (int i = p->sze - 1; i >= cnt; --i)
64          p->b[i + 1] = p->b[i];
65      p->b[cnt] = c;
66      ++p->sze;
67      checkLarge(p);
68  }
69
70  inline void find(node* &p, int &cnt, int pos)
71  {
72      p = head;
73      int cur;
74      for (cur = head->sze; p != NULL && cur < pos; p = p->nxt, cur += p-
    >sze);
75      cur -= p->sze;
76      cnt = pos - cur - 1;
77  }
78
79  inline void find2(node* &p, node* &pre, int &cnt, int pos)
80  {
81      pre = NULL, p = head;
82      int cur;
83      for (cur = head->sze; p != NULL && cur < pos; pre = p, p = p->nxt, cur
    += p->sze);
84      cur -= p->sze;
85      cnt = pos - cur - 1;
86  }
87
88  inline void Delete(int l, int r)
89  {
90      node *p, *q, *pre;
91      int nl, nr;
92      find2(p, pre, nl, l);
93      find(q, nr, r);
94      if (p == q)
95      {
96          int len = nr - nl + 1;
97          for (int i = nr + 1; i < p->sze; ++i)
98              p->b[i - len] = p->b[i];
99          p->sze -= len;
100         if (!p->sze)
101             pre != NULL ? pre->nxt = p->nxt : head = p->nxt;
102         else
```

```
            checkSmall(p);
        return ;
    }

    p->sze = nl;
    int len = nr + 1;
    for (int i = nr + 1; i < q->sze; ++i)
        q->b[i - len] = q->b[i];
    q->sze -= len;

    if (q->sze > 0)
        p->nxt = q;
    else
        p->nxt = q->nxt;
    if (p->sze > 0)
        checkSmall(p);
    else
        pre != NULL ? pre->nxt = p->nxt : head = p->nxt;
}

inline int Query(int pos)
{
    node *p = head;
    int cur;
    for (cur = head->sze; p != NULL && cur < pos; p = p->nxt, cur += p->sze);
    cur -= p->sze;
    return p->b[pos - cur - 1];
}
```