NANJING UNIVERSITY

ACM-ICPC Codebook 3

Data Structures

September 27, 2018

Contents

1 Range Operation Structures

1.1 Binary indexed tree

1.1.1 Point update, range query

Usage:

`init(n)`	Initialize the tree with 0.
`add(n, x)`	Add the n-th element by x.
`sum(n)`	Return the sum of the first n elements.

Time complexity: $O(n)$ for initialization; $O(\log n)$ for each update and query.

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

struct bit_purq{ // point update, range query
    int N;
    vector<LL> tr;

    void init(int n){ // fill the array with 0
        tr.resize(N = n + 5);
    }

    LL sum(int n){
        LL ans = 0;
        while (n){
            ans += tr[n];
            n -= lowbit(n);
        }
        return ans;
    }

    void add(int n, LL x){
        while (n < N){
            tr[n] += x;
            n += lowbit(n);
        }
    }
};
```

1.1.2 Range update, point query

Usage:

`init(n)`	Initialize the tree with 0.
`add(n, x)`	Add the first n element by x.
`query(n)`	Return the value of the n-th element.

Time complexity: $O(n)$ for initialization; $O(\log n)$ for each update and query.

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

struct bit_rupq{ // range update, point query
    int N;
    vector<LL> tr;

    void init(int n){ // fill the array with 0
        tr.resize(N = n + 5);
    }

    LL query(int n){
        LL ans = 0;
        while (n < N){
            ans += tr[n];
            n += lowbit(n);
        }
        return ans;
    }

    void add(int n, LL x){
        while (n){
            tr[n] += x;
            n -= lowbit(n);
        }
    }
};
```

1.1.3 Range update, range query

Usage:

`init(n)`	Initialize the tree with 0.
`add(l, r, x)`	Add the elements in $[l, r]$ by x.
`query(l, r)`	Return the sum of the elements in $[l, r]$.

Requirement:
1.1.1 Point update, range query

Time complexity: $O(n)$ for initialization; $O(\log n)$ for each update and query.

```
struct bit_rurq{
```

```
    bit_purq d, di;

    void init(int n){
        d.init(n); di.init(n);
    }

    void add(int l, int r, LL x){
        d.add(l, x); d.add(r+1, -x);
        di.add(l, x*l); di.add(r+1, -x*(r+1));
    }

    LL query(int l, int r){
        return (r+1)*d.sum(r) - di.sum(r) - l*d.sum(l-1) + di.sum(l-1);
    }
};
```

2 Miscellaneous Data Structures

2.1 Sparse table, range extremum query (RMQ)

Usage:

`ext(x, y)`	Return the extremum of x and y. **Modify this function before use!**
`init(n)`	Calculate the sparse table for array a from `a[0]` to `a[n-1]`.
`rmq(l, r)`	Query range extremum from `a[l]` to `a[r]`.

Time complexity: $O(n \log n)$ for initialization; $O(1)$ for each query.

```
const int MAXN = 100007;
int a[MAXN];
int st[MAXN][32 - __builtin_clz(MAXN)];

inline int ext(int x, int y){return x>y?x:y;} // ! max

void init(int n){
    int l = 31 - __builtin_clz(n);
    rep (i, n) st[i][0] = a[i];
    rep (j, l)
        rep (i, 1+n-(1<<j))
            st[i][j+1] = ext(st[i][j], st[i+(1<<j)][j]);
}

int rmq(int l, int r){
    int k = 31 - __builtin_clz(r-l+1);
    return ext(st[l][k], st[r-(1<<k)+1][k]);
```

```
}
```

3 Tree

3.1 Heavy-light decomposition

Usage:

`sz[x]`	Size of subtree rooted at x.
`top[x]`	Top node of the chain that x belongs to.
`fa[x]`	Father of x if exists; otherwise 0.
`son[x]`	Child node of x in its chain if exists; otherwise 0.
`depth[x]`	Depth of x. The depth of root is 1.
`id[x]`	Index of x used in data structure.
`decomp(r)`	Perform heavy-light decomposition on tree rooted at r.
`query(u, v)`	Query the path between u and v.

Time complexity: $O(n)$ for decomposition; $O(f(n) \log n)$ for each query, where $f(n)$ is the time-complexity of data structure.

```
const int MAXN = 100005;
vector<int> adj[MAXN];
int sz[MAXN], top[MAXN], fa[MAXN], son[MAXN], depth[MAXN], id[MAXN];

void dfs1(int x, int dep, int par){
    depth[x] = dep;
    sz[x] = 1;
    fa[x] = par;
    int maxn = 0, s = 0;
    for (int c: adj[x]){
        if (c == par) continue;
        dfs1(c, dep + 1, x);
        sz[x] += sz[c];
        if (sz[c] > maxn){
            maxn = sz[c];
            s = c;
        }
    }
    son[x] = s;
}

int cid = 0;
void dfs2(int x, int t){
    top[x] = t;
    id[x] = ++cid;
```

```
    if (son[x]) dfs2(son[x], t);
    for (int c: adj[x]){
        if (c == fa[x]) continue;
        if (c == son[x]) continue;
        else dfs2(c, c);
    }
}

void decomp(int root){
    dfs1(root, 1, 0);
    dfs2(root, root);
}

void query(int u, int v){
    while (top[u] != top[v]){
        if (depth[top[u]] < depth[top[v]]) swap(u, v);
        // id[top[u]] to id[u]
        u = fa[top[u]];
    }
    if (depth[u] > depth[v]) swap(u, v);
    // id[u] to id[v]
}
```

3.2 Order Statistics and Splay

⚠ Like `std::set`, this structure does not support multiple equivalent elements.

Usage:

See comments in code.

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>
  rkt;
// null_tree_node_update

// SAMPLE USAGE
rkt.insert(x);          // insert element
rkt.erase(x);           // erase element
rkt.order_of_key(x);    // obtain the number of elements less than x
rkt.find_by_order(i);   // iterator to i-th (numbered from 0) smallest element
rkt.lower_bound(x);
rkt.upper_bound(x);
rkt.join(rkt2);         // merge tree (only if their ranges do not intersect)
rkt.split(x, rkt2);     // split all elements greater than x to rkt2
```

3.3 Segment Tree

This implementation follows the "FlashHu" convention: tags does not apply to current node but applies to all child nodes. **Time complexity:** $O(\log n)$ per operation.

```
LL p;
const int MAXN = 4 * 100006;
struct segtree {
  int l[MAXN], m[MAXN], r[MAXN];
  LL val[MAXN], tadd[MAXN], tmul[MAXN];

#define lson (o<<1)
#define rson (o<<1|1)

  void pull(int o) {
    val[o] = (val[lson] + val[rson]) % p;
  }

  void push_add(int o, LL x) {
    val[o] = (val[o] + x * (r[o] - l[o])) % p;
    tadd[o] = (tadd[o] + x) % p;
  }

  void push_mul(int o, LL x) {
    val[o] = val[o] * x % p;
    tadd[o] = tadd[o] * x % p;
    tmul[o] = tmul[o] * x % p;
  }

  void push(int o) {
    if (l[o] == m[o]) return;
    if (tmul[o] != 1) {
      push_mul(lson, tmul[o]);
      push_mul(rson, tmul[o]);
      tmul[o] = 1;
    }
    if (tadd[o]) {
      push_add(lson, tadd[o]);
      push_add(rson, tadd[o]);
      tadd[o] = 0;
    }
  }

  void build(int o, int ll, int rr) {
```

```
    int mm = (ll + rr) / 2;
    l[o] = ll; r[o] = rr; m[o] = mm;
    tmul[o] = 1;
    if (ll == mm) {
      scanf("%lld", val + o);
      val[o] %= p;
    } else {
      build(lson, ll, mm);
      build(rson, mm, rr);
      pull(o);
    }
  }

  void add(int o, int ll, int rr, LL x) {
    if (ll <= l[o] && r[o] <= rr) {
      push_add(o, x);
    } else {
      push(o);
      if (m[o] > ll) add(lson, ll, rr, x);
      if (m[o] < rr) add(rson, ll, rr, x);
      pull(o);
    }
  }

  void mul(int o, int ll, int rr, LL x) {
    if (ll <= l[o] && r[o] <= rr) {
      push_mul(o, x);
    } else {
      push(o);
      if (ll < m[o]) mul(lson, ll, rr, x);
      if (m[o] < rr) mul(rson, ll, rr, x);
      pull(o);
    }
  }

  LL query(int o, int ll, int rr) {
    if (ll <= l[o] && r[o] <= rr) {
      return val[o];
    } else {
      LL ans = 0;
      push(o);
      if (m[o] > ll) ans += query(lson, ll, rr);
      if (m[o] < rr) ans += query(rson, ll, rr);
      return ans % p;
    }
  }
} seg;
```

3.4 Persistent array

Usage:

`init(size, il)`	(Re)initialize an array of size `size` and initial values `il`.
`access(pos)`	Access the position with index `pos`.
`update(pos, val)`	Change the value at `pos` to `val`.

Time complexity: $O(\log n)$ per operation.

```
struct node {
  static int n, pos;

  union {
    int value;
    struct {
      node *left, *right;
    };
  };

  void* operator new(size_t size);

  static node* build(int l, int r, int* il) {
    node* a = new node;
    if (r > l + 1) {
      int mid = (l + r) / 2;
      a->left = build(l, mid, il);
      a->right = build(mid, r, il);
    } else {
      a->value = il[l];
    }
    return a;
  }

  static node* init(int size, int* il) {
    n = size;
    pos = 0;
    return build(0, n, il);
  }

  node *Update(int l, int r, int pos, int val) const {
    node* a = new node(*this);
    if (r > l + 1) {
      int mid = (l + r) / 2;
      if (pos < mid)
        a->left = left->Update(l, mid, pos, val);
      else
```

```
        a->right = right->Update(mid, r, pos, val);
    } else {
      a->value = val;
    }
    return a;
  }

  int Access(int l, int r, int pos) const {
    if (r > l + 1) {
      int mid = (l + r) / 2;
      if (pos < mid) return left->Access(l, mid, pos);
      else return right->Access(mid, r, pos);
    } else {
      return value;
    }
  }

  int access(int index) {
    return Access(0, n, index);
  }

  node *update(int index, int val) {
    return Update(0, n, index, val);
  }
} nodes[30000000];

int node::n, node::pos;
inline void* node::operator new(size_t size) {
  return nodes + (pos++);
}
```

3.5 Persistent union-find set

Persistent union-find set with union-by-rank.

Usage:

`init(size)`	(Re)initialize a ufs of size `size` with indices $[0, \text{size})$.
`find(pos)`	Get the parent of `pos`.
`unite(u, v)`	Unite the two sets containing `u`, `v`.

Time complexity: $O(\log^2 n)$ per operation.

```
// ~0.1s per 100000 operations @ luogu.org
struct node {
  static int n, pos;

```

```
  union {
    struct {
      int value, rank;
    };
    struct {
      node *left, *right;
    };
  };

  void* operator new(size_t size);

  static node* build(int l, int r) {
    node* a = new node;
    if (r > l + 1) {
      int mid = (l + r) / 2;
      a->left = build(l, mid);
      a->right = build(mid, r);
    } else {
      a->value = l;
      a->rank = 0;
    }
    return a;
  }

  static node* init(int size) {
    n = size;
    pos = 0;
    return build(0, n);
  }

  node *Update(int l, int r, int pos, node nd) {
    node* a = new node(*this);
    if (r > l + 1) {
      int mid = (l + r) / 2;
      if (pos < mid)
        a->left = left->Update(l, mid, pos, nd);
      else
        a->right = right->Update(mid, r, pos, nd);
    } else {
      *a = nd;
    }
    return a;
  }

  node *Access(int l, int r, int pos) {
    if (r > l + 1) {
      int mid = (l + r) / 2;
```

```
      if (pos < mid) return left->Access(l, mid, pos);
      else return right->Access(mid, r, pos);
    } else {
      return this;
    }
  }

  int find(int x) {
    int fa;
    while ((fa = Access(0, n, x)->value) != x)
      x = fa;
    return x;
  }

  node* unite(int u, int v) {
    u = find(u); v = find(v);
    if (u == v) return this;
    int ru = Access(0, n, u)->rank, rv = Access(0, n, v)->rank;
    if (ru == rv)
      return Update(0, n, u, {v, ru})->Update(0, n, v, {v, ru+1});
    if (ru > rv) {
      swap(u, v);
      swap(ru, rv);
    }
    return Update(0, n, u, {v, rv});
  }
} nodes[20000000];

int node::n, node::pos;
inline void* node::operator new(size_t size) {
  return nodes + (pos++);
}
```

3.6 Persistent segment tree; Range nth element query

Usage:

`init(size)`	(Re)initialize with indices $[0, \text{size})$.
`inc(pos)`	Increment element with index pos.
`query(l, r, k)`	Find the k-th element between versions l and r.

Time complexity: $O(\log n)$ per operation.

```
struct node {
  static int n, pos;

```

```
int value;
node *left, *right;

void* operator new(size_t size);

static node* Build(int l, int r) {
  node* a = new node;
  if (r > l + 1) {
    int mid = (l + r) / 2;
    a->left = Build(l, mid);
    a->right = Build(mid, r);
  } else {
    a->value = 0;
  }
  return a;
}

static node* init(int size) {
  n = size;
  pos = 0;
  return Build(0, n);
}

static int Query(node* lt, node *rt, int l, int r, int k) {
  if (r == l + 1) return l;
  int mid = (l + r) / 2;
  if (rt->left->value - lt->left->value < k) {
    k -= rt->left->value - lt->left->value;
    return Query(lt->right, rt->right, mid, r, k);
  } else {
    return Query(lt->left, rt->left, l, mid, k);
  }
}

static int query(node* lt, node *rt, int k) {
  return Query(lt, rt, 0, n, k);
}

node *Inc(int l, int r, int pos) const {
  node* a = new node(*this);
  if (r > l + 1) {
    int mid = (l + r) / 2;
    if (pos < mid)
      a->left = left->Inc(l, mid, pos);
    else
      a->right = right->Inc(mid, r, pos);
  }
```

```
    a->value++;
    return a;
  }

  node *inc(int index) {
    return Inc(0, n, index);
  }
} nodes[8000000];

int node::n, node::pos;
inline void* node::operator new(size_t size) {
  return nodes + (pos++);
}
```

3.7 Dynamic tree connectivity; link/cut tree

Maintaining dynamic tree connectivity as well as supporting path aggregation.

Usage:

`Root(u)`	Query the root of `u` in represented tree.
`Link(u, v)`	Add edge between `u` and `v`. The edge must not exist before.
`Cut(u, v)`	Remove edge between `u` and `v`. The edge must exist before.
`Query(u, v)`	Query path aggregation value between `u` and `v`.
`Update(u, x)`	Update node value of `u` to `x`.

Rewrite `pull(x)` to customize aggregation function.

⚠ All indices are numbered from 1.

Time complexity: Amortized $O(\log n)$ per operation.

```
// about 0.13s per 100k ops @Luogu.org

namespace LCT {
  const int MAXN = 300005;
  int fa[MAXN], ch[MAXN][2], val[MAXN], sum[MAXN];
  bool rev[MAXN];

  bool isroot(int x) {
    return ch[fa[x]][0] == x || ch[fa[x]][1] == x;
  }

  void pull(int x) {
    sum[x] = val[x] ^ sum[ch[x][0]] ^ sum[ch[x][1]];
  }

```

```
void reverse(int x) {
  swap(ch[x][0], ch[x][1]);
  rev[x] ^= 1;
}

void push(int x) {
  if (rev[x]) {
    if (ch[x][0]) reverse(ch[x][0]);
    if (ch[x][1]) reverse(ch[x][1]);
    rev[x] = 0;
  }
}

void rotate(int x) {
  int y = fa[x], z = fa[y], k = ch[y][1] == x, w = ch[x][!k];
  if (isroot(y)) ch[z][ch[z][1] == y] = x;
  ch[x][!k] = y; ch[y][k] = w;
  if (w) fa[w] = y;
  fa[y] = x; fa[x] = z;
  pull(y);
}

void pushall(int x) {
  if (isroot(x)) pushall(fa[x]);
  push(x);
}

void splay(int x) {
  int y = x, z = 0;
  pushall(y);
  while (isroot(x)) {
    y = fa[x]; z = fa[y];
    if (isroot(y)) rotate((ch[y][0] == x) ^ (ch[z][0] == y) ? x : y);
    rotate(x);
  }
  pull(x);
}

void access(int x) {
  int z = x;
  for (int y = 0; x; x = fa[y = x]) {
    splay(x);
    ch[x][1] = y;
    pull(x);
  }
  splay(z);
}
```

```

  void chroot(int x) {
    access(x);
    reverse(x);
  }

  void split(int x, int y) {
    chroot(x);
    access(y);
  }

  int Root(int x) {
    access(x);
    while (ch[x][0]) {
      push(x);
      x = ch[x][0];
    }
    splay(x);
    return x;
  }

  void Link(int u, int v) {  // assume unconnected before
    chroot(u);
    fa[u] = v;
  }

  void Cut(int u, int v) {  // assume connected before
    split(u, v);
    fa[u] = ch[v][0] = 0;
    pull(v);
  }

  int Query(int u, int v) {
    split(u, v);
    return sum[v];
  }

  void Update(int u, int x) {
    splay(u);
    val[u] = x;
  }
};
```

4 Block Decomposition

4.1 Range nth element query (Block + bitset)

Usage:

`query(l, r, k)`	Find the k-th element between versions l and r.

Performance: Comparable to persistent segment tree up to 10^5 operations.

```
typedef array<ULL, 64>              block;
typedef array<pair<int, int>, 64>   hdr;

block b[200005];
hdr   h[200005];

int n, m;
pair<int, int> s[200005];
int a[200005], rk[200005];

int query(int l, int r, int k) {
  int delta;
  unsigned bpos, ipos, pos = 0;
  for (bpos = 0; (delta = h[r][bpos].first - h[l][bpos].first) < k;
      bpos++, pos += 4096) k -= delta;
  const auto &bl = b[h[l][bpos].second], &br = b[h[r][bpos].second];
  for (ipos = 0; (delta = __builtin_popcountll(bl[ipos] ^ br[ipos])) < k;
      ipos++, pos += 64) k -= delta;
  ULL mask = br[ipos] ^ bl[ipos], cmask;
  while (k) {
    cmask = mask & -mask;
    mask -= cmask;
    k--;
  }
  return pos + __builtin_ctzll(cmask);
}

int main() {
  scanf("%d%d", &n, &m);
  rep (i, n) scanf("%d", a+i);
  rep (i, n) s[i] = {a[i], i};
  sort(s, s+n);
  rep (i, n) rk[s[i].second] = i;
  rep (i, n) {
    h[i+1] = h[i];
    int crk = rk[i];
    int blk = crk >> 12, bpos = crk & 0xfff;
    int popcnt, bid; tie(popcnt, bid) = h[i][blk];
```

```
    popcnt++;
    b[i+1] = b[bid];
    b[i+1][bpos >> 6] |= 1ull << (bpos & 0x3f);
    h[i+1][blk] = {popcnt, i+1};
  }
  rep (i, m) {
    int l, r, k; scanf("%d%d%d", &l, &r, &k);
    printf("%d\n", s[query(l-1, r, k)].first);
  }
  return 0;
}
```