NANJING UNIVERSITY

ACM-ICPC Codebook 3
Data Structures

August 28, 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.

```
1  inline int lowbit(int x){return x&-x;}
2
3  struct bit_purq{ // point update, range query
4      int N;
5      vector<LL> tr;
6
7      void init(int n){ // fill the array with 0
8          tr.resize(N = n + 5);
9      }
10
11     LL sum(int n){
12         LL ans = 0;
13         while (n){
14             ans += tr[n];
15             n -= lowbit(n);
16         }
17         return ans;
18     }
19
20     void add(int n, LL x){
21         while (n < N){
22             tr[n] += x;
23             n += lowbit(n);
24         }
25     }
26 };
```

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.

```
1   inline int lowbit(int x){return x&-x;}
2
3   struct bit_rupq{ // range update, point query
4       int N;
5       vector<LL> tr;
6
7       void init(int n){ // fill the array with 0
8           tr.resize(N = n + 5);
9       }
10
11      LL query(int n){
12          LL ans = 0;
13          while (n < N){
14              ans += tr[n];
15              n += lowbit(n);
16          }
17          return ans;
18      }
19
20      void add(int n, LL x){
21          while (n){
22              tr[n] += x;
23              n -= lowbit(n);
24          }
25      }
26  };
```

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.

```
1   struct bit_rurq{
```

```
2    │    bit_purq d, di;
3    │
4    │    void init(int n){
5    │        d.init(n); di.init(n);
6    │    }
7    │
8    │    void add(int l, int r, LL x){
9    │        d.add(l, x); d.add(r+1, -x);
10   │        di.add(l, x*l); di.add(r+1, -x*(r+1));
11   │    }
12   │
13   │    LL query(int l, int r){
14   │        return (r+1)*d.sum(r) - di.sum(r) - l*d.sum(l-1) + di.sum(l-1);
15   │    }
16   │ };
```

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.

```
1    │ const int MAXN = 100007;
2    │ int a[MAXN];
3    │ int st[MAXN][32 - __builtin_clz(MAXN)];
4    │
5    │ inline int ext(int x, int y){return x>y?x:y;} // ! max
6    │
7    │ void init(int n){
8    │     int l = 31 - __builtin_clz(n);
9    │     rep (i, n) st[i][0] = a[i];
10   │     rep (j, l)
11   │         rep (i, 1+n-(1<<j))
12   │             st[i][j+1] = ext(st[i][j], st[i+(1<<j)][j]);
13   │ }
14   │
15   │ int rmq(int l, int r){
16   │     int k = 31 - __builtin_clz(r-l+1);
17   │     return ext(st[l][k], st[r-(1<<k)+1][k]);
```

```
18  }
```

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.

```
1   const int MAXN = 100005;
2   vector<int> adj[MAXN];
3   int sz[MAXN], top[MAXN], fa[MAXN], son[MAXN], depth[MAXN], id[MAXN];
4
5   void dfs1(int x, int dep, int par){
6       depth[x] = dep;
7       sz[x] = 1;
8       fa[x] = par;
9       int maxn = 0, s = 0;
10      for (int c: adj[x]){
11          if (c == par) continue;
12          dfs1(c, dep + 1, x);
13          sz[x] += sz[c];
14          if (sz[c] > maxn){
15              maxn = sz[c];
16              s = c;
17          }
18      }
19      son[x] = s;
20  }
21
22  int cid = 0;
23  void dfs2(int x, int t){
24      top[x] = t;
25      id[x] = ++cid;
```

```
26      if (son[x]) dfs2(son[x], t);
27      for (int c: adj[x]){
28          if (c == fa[x]) continue;
29          if (c == son[x]) continue;
30          else dfs2(c, c);
31      }
32  }
33
34  void decomp(int root){
35      dfs1(root, 1, 0);
36      dfs2(root, root);
37  }
38
39  void query(int u, int v){
40      while (top[u] != top[v]){
41          if (depth[top[u]] < depth[top[v]]) swap(u, v);
42          // id[top[u]] to id[u]
43          u = fa[top[u]];
44      }
45      if (depth[u] > depth[v]) swap(u, v);
46      // id[u] to id[v]
47  }
```

3.2   Order Statistics and Splay

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

Usage:

See comments in code.

```
1  #include <ext/pb_ds/assoc_container.hpp>
2  using namespace __gnu_pbds;
3
4  tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>
     rkt;
5  // null_tree_node_update
6
7  // SAMPLE USAGE
8  rkt.insert(x);          // insert element
9  rkt.erase(x);           // erase element
10 rkt.order_of_key(x);    // obtain the number of elements less than x
11 rkt.find_by_order(i);   // iterator to i-th (numbered from 0) smallest element
12 rkt.lower_bound(x);
13 rkt.upper_bound(x);
14 rkt.join(rkt2);         // merge tree (only if their ranges do not intersect)
15 rkt.split(x, rkt2);     // split all elements greater than x to rkt2
```

3.3   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.

```
1  struct node {
2    static int n, pos;
3
4    union {
5      int value;
6      struct {
7        node *left, *right;
8      };
9    };
10
11   void* operator new(size_t size);
12
13   static node* build(int l, int r, int* il) {
14     node* a = new node;
15     if (r > l + 1) {
16       int mid = (l + r) / 2;
17       a->left = build(l, mid, il);
18       a->right = build(mid, r, il);
19     } else {
20       a->value = il[l];
21     }
22     return a;
23   }
24
25   static node* init(int size, int* il) {
26     n = size;
27     pos = 0;
28     return build(0, n, il);
29   }
30
31   node *Update(int l, int r, int pos, int val) const {
32     node* a = new node(*this);
33     if (r > l + 1) {
34       int mid = (l + r) / 2;
```

```
35        if (pos < mid)
36          a->left = left->Update(l, mid, pos, val);
37        else
38          a->right = right->Update(mid, r, pos, val);
39      } else {
40        a->value = val;
41      }
42      return a;
43    }
44
45    int Access(int l, int r, int pos) const {
46      if (r > l + 1) {
47        int mid = (l + r) / 2;
48        if (pos < mid) return left->Access(l, mid, pos);
49        else return right->Access(mid, r, pos);
50      } else {
51        return value;
52      }
53    }
54
55    int access(int index) {
56      return Access(0, n, index);
57    }
58
59    node *update(int index, int val) {
60      return Update(0, n, index, val);
61    }
62  } nodes[30000000];
63
64  int node::n, node::pos;
65  inline void* node::operator new(size_t size) {
66    return nodes + (pos++);
67  }
```

3.4   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.

```
1    // ~0.1s per 100000 operations @ luogu.org
2    struct node {
3      static int n, pos;
4
5      union {
6        struct {
7          int value, rank;
8        };
9        struct {
10         node *left, *right;
11       };
12     };
13
14     void* operator new(size_t size);
15
16     static node* build(int l, int r) {
17       node* a = new node;
18       if (r > l + 1) {
19         int mid = (l + r) / 2;
20         a->left = build(l, mid);
21         a->right = build(mid, r);
22       } else {
23         a->value = l;
24         a->rank = 0;
25       }
26       return a;
27     }
28
29     static node* init(int size) {
30       n = size;
31       pos = 0;
32       return build(0, n);
33     }
34
35     node *Update(int l, int r, int pos, node nd) {
36       node* a = new node(*this);
37       if (r > l + 1) {
38         int mid = (l + r) / 2;
39         if (pos < mid)
40           a->left = left->Update(l, mid, pos, nd);
41         else
42           a->right = right->Update(mid, r, pos, nd);
43       } else {
44         *a = nd;
45       }
46       return a;
47     }
```

```
48
49    node *Access(int l, int r, int pos) {
50      if (r > l + 1) {
51        int mid = (l + r) / 2;
52        if (pos < mid) return left->Access(l, mid, pos);
53        else return right->Access(mid, r, pos);
54      } else {
55        return this;
56      }
57    }
58
59    int find(int x) {
60      int fa;
61      while ((fa = Access(0, n, x)->value) != x)
62        x = fa;
63      return x;
64    }
65
66    node* unite(int u, int v) {
67      u = find(u); v = find(v);
68      if (u == v) return this;
69      int ru = Access(0, n, u)->rank, rv = Access(0, n, v)->rank;
70      if (ru == rv)
71        return Update(0, n, u, {v, ru})->Update(0, n, v, {v, ru+1});
72      if (ru > rv) {
73        swap(u, v);
74        swap(ru, rv);
75      }
76      return Update(0, n, u, {v, rv});
77    }
78  } nodes[20000000];
79
80  int node::n, node::pos;
81  inline void* node::operator new(size_t size) {
82    return nodes + (pos++);
83  }
```

3.5   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.

```
1   struct node {
2     static int n, pos;
3
4     int value;
5     node *left, *right;
6
7     void* operator new(size_t size);
8
9     static node* Build(int l, int r) {
10      node* a = new node;
11      if (r > l + 1) {
12        int mid = (l + r) / 2;
13        a->left = Build(l, mid);
14        a->right = Build(mid, r);
15      } else {
16        a->value = 0;
17      }
18      return a;
19    }
20
21    static node* init(int size) {
22      n = size;
23      pos = 0;
24      return Build(0, n);
25    }
26
27    static int Query(node* lt, node *rt, int l, int r, int k) {
28      if (r == l + 1) return l;
29      int mid = (l + r) / 2;
30      if (rt->left->value - lt->left->value < k) {
31        k -= rt->left->value - lt->left->value;
32        return Query(lt->right, rt->right, mid, r, k);
33      } else {
34        return Query(lt->left, rt->left, l, mid, k);
35      }
36    }
37
38    static int query(node* lt, node *rt, int k) {
39      return Query(lt, rt, 0, n, k);
40    }
41
42    node *Inc(int l, int r, int pos) const {
43      node* a = new node(*this);
44      if (r > l + 1) {
45        int mid = (l + r) / 2;
46        if (pos < mid)
47          a->left = left->Inc(l, mid, pos);
```

```
48        else
49           a->right = right->Inc(mid, r, pos);
50      }
51      a->value++;
52      return a;
53    }
54
55    node *inc(int index) {
56       return Inc(0, n, index);
57    }
58  } nodes[8000000];
59
60  int node::n, node::pos;
61  inline void* node::operator new(size_t size) {
62    return nodes + (pos++);
63  }
```

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.

```
1   typedef array<ULL, 64>              block;
2   typedef array<pair<int, int>, 64>   hdr;
3
4   block b[200005];
5   hdr   h[200005];
6
7   int n, m;
8   pair<int, int> s[200005];
9   int a[200005], rk[200005];
10
11  int query(int l, int r, int k) {
12    int delta;
13    unsigned bpos, ipos, pos = 0;
14    for (bpos = 0; (delta = h[r][bpos].first - h[l][bpos].first) < k;
15         bpos++, pos += 4096) k -= delta;
16    const auto &bl = b[h[l][bpos].second], &br = b[h[r][bpos].second];
17    for (ipos = 0; (delta = __builtin_popcountll(bl[ipos] ^ br[ipos])) < k;
18         ipos++, pos += 64) k -= delta;
```

```
19   ULL mask = br[ipos] ^ bl[ipos], cmask;
20   while (k) {
21     cmask = mask & -mask;
22     mask -= cmask;
23     k--;
24   }
25   return pos + __builtin_ctzll(cmask);
26 }
27
28 int main() {
29   scanf("%d%d", &n, &m);
30   rep (i, n) scanf("%d", a+i);
31   rep (i, n) s[i] = {a[i], i};
32   sort(s, s+n);
33   rep (i, n) rk[s[i].second] = i;
34   rep (i, n) {
35     h[i+1] = h[i];
36     int crk = rk[i];
37     int blk = crk >> 12, bpos = crk & 0xfff;
38     int popcnt, bid; tie(popcnt, bid) = h[i][blk];
39     popcnt++;
40     b[i+1] = b[bid];
41     b[i+1][bpos >> 6] |= 1ull << (bpos & 0x3f);
42     h[i+1][blk] = {popcnt, i+1};
43   }
44   rep (i, m) {
45     int l, r, k; scanf("%d%d%d", &l, &r, &k);
46     printf("%d\n", s[query(l-1, r, k)].first);
47   }
48   return 0;
49 }
```