GEMP - UFC Quixadá - ICPC Library

Contents

1 Data Structures 1
- 1.1 BIT . . . 1
- 1.2 BIT 2D . . . 1
- 1.3 BIT In Range . . . 2
- 1.4 Dynamic Median . . . 2
- 1.5 Dynamic Wavelet Tree . . . 2
- 1.6 Implicit Treap . . . 4
- 1.7 LiChao Tree . . . 5
- 1.8 Policy Based Tree . . . 5
- 1.9 Queue Query . . . 5
- 1.10 Segment Tree . . . 6
- 1.11 Segment Tree Iterative . . . 6
- 1.12 Segment Tree Lazy . . . 7
- 1.13 Set and Clear . . . 7
- 1.14 Sparse Table . . . 8
- 1.15 SQRT Decomposition . . . 9
- 1.16 SQRT Tree . . . 9
- 1.17 Stack Query . . . 10
- 1.18 Treap . . . 10
- 1.19 Union Find . . . 11
- 1.20 Wavelet Tree . . . 12

2 Graph Algorithms 12

3 Dynamic Programming 12

4 Math 12

5 Geometry 12

6 String Algorithms 12

7 Miscellaneous 12

8 Theorems and Formulas 12

1 Data Structures

1.1 BIT

```cpp
#include <bits/stdc++.h>
using namespace std;
class Bit{
private:
  typedef long long t_bit;
  int nBit;
  int nLog;
  vector<t_bit> bit;
public:
  Bit(int n){
    nBit = n;
    nLog = 20;
    bit.resize(nBit+1, 0);
  }
  //1-indexed
  t_bit get(int i){
    t_bit s = 0;
    for(; i > 0; i -= (i & -i))
      s += bit[i];
    return s;
  }
  //1-indexed [l, r]
  t_bit get(int l, int r){
    return get(r) - get(l-1);
  }
  //1-indexed
  void add(int i, t_bit value){
    for(; i <= nBit; i += (i & -i))
      bit[i] += value;
  }
  t_bit position(t_bit value){
    t_bit sum = 0;
    int pos = 0;
    for(int i=nLog; i>=0; i--){
      if( (pos + (1 << i) <= nBit) and (sum + bit[pos + (1 << i)] <
          value) ){
        sum += bit[pos + (1 << i)];
        pos += (1 << i);
      }
    }
    return pos + 1;
  }
};
```

1.2 BIT 2D

```cpp
#include <bits/stdc++.h>
using namespace std;
class Bit2d{
private:
  typedef long long t_bit;
  vector<vector<t_bit> > bit;
  int nBit, mBit;
public:
  Bit2d(int n, int m){
    nBit = n;
    mBit = m;
    bit.resize(nBit+1, vector<t_bit>(mBit+1, 0));
  }
  //1-indexed
  t_bit get(int i, int j){
    t_bit sum = 0;
    for(int a=i; a > 0; a-=(a & -a))
      for(int b=j; b > 0; b-=(b & -b))
        sum += bit[a][b];
    return sum;
  }
  //1-indexed
  t_bit get(int a1, int b1, int a2, int b2){
    return get(a2, b2) - get(a2, b1-1) - get(a1-1, b2) + get(a1-1, b1
        -1);
  }
  //1-indexed [i, j]
  void add(int i, int j, t_bit value){
    for(int a=i; a <= nBit; a+=(a & -a))
      for(int b=j; b <= mBit; b+=(b & -b))
```

```
        bit[a][b] += value;
    }
};
```

1.3 BIT In Range

```
#include <bits/stdc++.h>
using namespace std;
class BitRange{
private:
  typedef long long t_bit;
  vector<t_bit> bit1, bit2;
  t_bit get(vector<t_bit> &bit, int i) {
    t_bit sum = 0;
    for(; i > 0; i -= (i & -i))
      sum += bit[i];
    return sum;
  }
  void add(vector<t_bit> &bit, int i, t_bit value) {
    for(; i < (int)bit.size(); i += (i & -i))
      bit[i] += value;
  }
public:
  BitRange(int n) {
    bit1.assign(n + 1, 0);
    bit2.assign(n + 1, 0);
  }
  //1-indexed [i, j]
  void add(int i, int j, t_bit v) {
    add(bit1, i, v);
    add(bit1, j+1, -v);
    add(bit2, i, v*(i-1));
    add(bit2, j+1, -v*j);
  }
  //1-indexed
  t_bit get(int i) {
    return get(bit1, i)*i - get(bit2, i);
  }
  //1-indexed [i,j]
  t_bit get(int i, int j) {
    return get(j) - get(i-1);
  }
};
```

1.4 Dynamic Median

```
#include <bits/stdc++.h>
using namespace std;
class DinamicMedian{
  typedef int t_median;
private:
  priority_queue<t_median> mn;
  priority_queue<t_median, vector<t_median>, greater<t_median> > mx;
public:
  double median(){
    if(mn.size() > mx.size())
      return mn.top();
    else
      return (mn.top() + mx.top())/2.0;
  }
  void push(t_median x){
    if(mn.size() <= mx.size())
      mn.push(x);
    else
      mx.push(x);
    if((!mx.empty()) and (!mn.empty())){
      while(mn.top() > mx.top()){
        t_median a = mx.top(); mx.pop();
        t_median b = mn.top(); mn.pop();
        mx.push(b);
        mn.push(a);
      }
    }
  }
};
```

1.5 Dynamic Wavelet Tree

```
#include <bits/stdc++.h>
using namespace std;
struct SplayTree{
  struct Node{
    int x, y, s;
    Node *p = 0;
    Node *l = 0;
    Node *r = 0;
    Node(int v){
      x = v;
      y = v;
      s = 1;
    }
    void upd(){
      s = 1;
      y = x;
      if(l) {
        y += l->y;
        s += l->s;
      }
      if(r) {
        y += r->y;
        s += r->s;
      }
    }
    int left_size(){
      return l ? l->s : 0;
    }
  };
  Node *root = 0;
  void rot(Node *c) {
    auto p = c->p;
    auto g = p->p;
    if(g)(g->l == p ? g->l : g->r) = c;
    if(p->l == c) {
      p->l = c->r;
      c->r = p;
      if(p->l) p->l->p = p;
    }else{
      p->r = c->l;
    c->l = p;
    if(p->r) p->r->p = p;
  }
  p->p = c;
  c->p = g;
  p->upd();
  c->upd();
}
void splay(Node *c) {
  while(c->p) {
    auto p = c->p;
    auto g = p->p;
    if(g) rot((g->l == p) == (p->l == c) ? p : c);
    rot(c);
  }
  c->upd();
  root = c;
}
Node* join(Node *l, Node *r) {
  if(not l) return r;
  if(not r) return l;
  while(l->r) l = l->r;
  splay(l);
  r->p = l;
  l->r = r;
  l->upd();
  return l;
}
pair<Node*, Node*> split(Node *p, int idx) {
  if(not p)
    return make_pair(nullptr, nullptr);
  if(idx < 0)
    return make_pair(nullptr, p);
  if(idx >= p->s)
    return make_pair(p, nullptr);
  for(int lf = p->left_size(); idx != lf; lf = p->left_size()) {
    if(idx < lf)
      p = p->l;
    else
      p = p->r, idx -= lf+1;
  }
  splay(p);
  Node *l = p;
  Node *r = p->r;
  if(r) {
    l->r = r->p = 0;
    l->upd();
  }
  return make_pair(l, r);
}
Node* get(int idx) {
  auto p = root;
  for(int lf = p->left_size(); idx != lf; lf = p->left_size()) {
    if(idx < lf)
      p = p->l;
    else
      p = p->r, idx -= lf+1;
  }
  splay(p);
  return p;
}
```

```
  int insert(int idx, int x) {
    Node *l, *r;
    tie(l, r) = split(root, idx-1);
    int v = l ? l->y : 0;
    root = join(l, join(new Node(x), r));
    return v;
  }
  void erase(int idx) {
    Node *l, *r;
    tie(l, r) = split(root, idx);
    root = join(l->l, r);
    delete l;
  }
  int rank(int idx) {
    Node *l, *r;
    tie(l, r) = split(root, idx);
    int x = (l && l->l ? l->l->y : 0);
    root = join(l, r);
    return x;
  }
  int operator[](int idx) {
    return rank(idx);
  }
  ~SplayTree() {
    if(!root)
      return;
    vector<Node*> nodes {root};
    while(nodes.size()) {
      auto u = nodes.back();
      nodes.pop_back();
      if(u->l) nodes.emplace_back(u->l);
      if(u->r) nodes.emplace_back(u->r);
      delete u;
    }
  }
};
class Wavelet{
private:
  int A, Z;
  Wavelet *l = 0;
  Wavelet *r = 0;
  SplayTree b;
public:
  Wavelet(int min_value, int max_value) {
    A = min_value;
    Z = max_value;
    b.insert(0, 0);
  }
  ~Wavelet() {
    delete l;
    delete r;
  }
  //0-indexed
  void insert(int idx, int x) {
    if(A >= Z)
      return;
    int M = (A + Z - 1) / 2;
    if(x <= M) {
      l = l ?: new Wavelet(A, M);
      l->insert(b.insert(idx, 1), x);
    }else{
      r = r ?: new Wavelet(M+1, Z);
      r->insert(idx - b.insert(idx, 0), x);
    }
  }
  //0-indexed
  void erase(int idx) {
    if(A == Z)
      return;
    auto p = b.get(idx);
    int lf = p->l ? p->l->y : 0;
    int x = p->x;
    b.erase(idx);
    if(x == 1)
      l->erase(lf);
    else
      r->erase(idx-lf);
  }
  //kth [l, r[
  int kth(int L, int R, int k) {
    if(A == Z)
      return A;
    int x = b.rank(L);
    int y = b.rank(R);
    if(k <= y-x)
      return l->kth(x, y, k);
    else
      return r->kth(L-x, R-y, k-(y-x));
  }
  //0-indexed
  int get(int idx){
    return kth(idx, idx+1, 1);
  }
};
```

1.6 Implicit Treap

```
#include <bits/stdc++.h>
using namespace std;
class ImplicitTreap {
private:
  typedef int t_treap;
  const t_treap neutral = 0;
  inline t_treap join(t_treap a, t_treap b, t_treap c){
    return a + b + c;
  }
  struct Node{
    int y, size;
    t_treap v, op_value;
    bool rev;
    Node *l, *r;
    Node(t_treap _v){
      v = op_value = _v;
      y = rand();
      size = 1;
      l = r = NULL;
      rev = false;
    }
  };
  Node* root;
  int size(Node* t) { return t ? t->size : 0; }
  t_treap op_value(Node* t) { return t ? t->op_value : neutral; }
  Node* refresh(Node* t) {
    if (t == NULL) return t;
    t->size = 1 + size(t->l) + size(t->r);
    t->op_value = join(t->v, op_value(t->l), op_value(t->r));
    if (t->l != NULL) t->l->rev ^= t->rev;
    if (t->r != NULL) t->r->rev ^= t->rev;
    if (t->rev) {
      swap(t->l, t->r);
      t->rev = false;
    }
    return t;
  }
  void split(Node* &t, int k, Node* &a, Node* &b) {
    refresh(t);
    Node * aux;
    if (!t) a = b = NULL;
    else if (size(t->l) < k) {
      split(t->r, k-size(t->l)-1, aux, b);
      t->r = aux;
      a = refresh(t);
    }
    else {
      split(t->l, k, a, aux);
      t->l = aux;
      b = refresh(t);
    }
  }
  Node* merge(Node* a, Node* b) {
    refresh(a); refresh(b);
    if (!a || !b) return a ? a : b;
    if (a->y < b->y) {
      a->r = merge(a->r, b);
      return refresh(a);
    }
    else {
      b->l = merge(a, b->l);
      return refresh(b);
    }
  }
  Node* at(Node* t, int n) {
    if (!t) return t;
    refresh(t);
    if (n < size(t->l)) return at(t->l, n);
    else if (n == size(t->l)) return t;
    else return at(t->r, n-size(t->l)-1);
  }
  void del(Node* &t) {
    if (!t) return;
    if (t->l) del(t->l);
    if (t->r) del(t->r);
    delete t;
    t = NULL;
  }
public:
  ImplicitTreap() : root(NULL) {
    srand(time(NULL));
  }
  ~ImplicitTreap() { clear(); }
  void clear() { del(root); }
  int size() { return size(root); }
  //0-indexed
  bool insert(int n, int v) {
    Node *a, *b;
    split(root, n, a, b);
    root = merge(merge(a, new Node(v)), b);
    return true;
  }
  //0-indexed
  bool erase(int n) {
    Node *a, *b, *c, *d;
    split(root, n, a, b);
    split(b, 1, c, d);
    root = merge(a, d);
    if (c == NULL) return false;
    delete c;
    return true;
  }
  //0-indexed
  t_treap at(int n) {
    Node* ans = at(root, n);
    return ans ? ans->v : -1;
  }
  //0-indexed [l, r]
  t_treap query(int l, int r) {
    if (l > r) swap(l, r);
    Node *a, *b, *c, *d;
    split(root, l, a, d);
    split(d, r-l+1, b, c);
    t_treap ans = op_value(b);
    root = merge(a, merge(b, c));
    return ans;
  }
  //0-indexed [l, r]
  void reverse(int l, int r) {
    if (l>r) swap(l, r);
    Node *a, *b, *c, *d;
    split(root, l, a, d);
    split(d, r-l+1, b, c);
    if (b != NULL) b->rev ^= 1;
    root = merge(a, merge(b, c));
  }
};
```

1.7 LiChao Tree

```
#include <bits/stdc++.h>
using namespace std;
const int INF = 0x3f3f3f3f;
class LiChaoTree{
private:
  typedef int t_line;
  struct Line{
    t_line k, b;
    Line() {}
    Line(t_line k, t_line b): k(k), b(b) {}
  };
  int n_tree, min_x, max_x;
  vector<Line> li_tree;
  t_line f(Line l, int x){
    return l.k*x + l.b;
  }
  void add(Line nw, int v, int l, int r) {
    int m = (l + r) / 2;
    bool lef = f(nw, l) > f(li_tree[v], l);
    bool mid = f(nw, m) > f(li_tree[v], m);
    if(mid)
      swap(li_tree[v], nw);
    if(r - l == 1)
      return;
    else if(lef != mid)
      add(nw, 2 * v, l, m);
    else
      add(nw, 2 * v + 1, m, r);
  }
  int get(int x, int v, int l, int r) {
    int m = (l + r) / 2;
    if(r - l == 1)
      return f(li_tree[v], x);
    else if(x < m)
      return max(f(li_tree[v], x), get(x, 2 * v, l, m));
    else
      return max(f(li_tree[v], x), get(x, 2 * v + 1, m, r));
  }
public:
  LiChaoTree(int mn_x, int mx_x){
    min_x = mn_x;
    max_x = mx_x;
    n_tree = max_x-min_x+5;
    li_tree.resize(4*n_tree, Line(0, -INF));
  }
  void add(t_line k, t_line b){
    add(Line(k, b), 1, min_x, max_x);
  }
  t_line get(int x){
    return get(x, 1, min_x, max_x);
  }
};
```

1.8 Policy Based Tree

```
#include <bits/stdc++.h>
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
using namespace std;
typedef tree<int, null_type, less<int>, rb_tree_tag,
  tree_order_statistics_node_update> OrderedSet;
typedef tree<int, int, less<int>, rb_tree_tag,
  tree_order_statistics_node_update> OrderedMap;
//order_of_key (k) : Number of items strictly smaller than k .
//find_by_order(k) : K-th element in a set (counting from zero).
```

1.9 Queue Query

```
#include <bits/stdc++.h>
using namespace std;
class QueueQuery{
private:
    typedef int t_queue;
    stack<pair<t_queue, t_queue> > s1, s2;
     t_queue cmp(t_queue a, t_queue b){
       return min(a, b);
     }
     void move(){
      if (s2.empty()) {
        while (!s1.empty()) {
          t_queue element = s1.top().first;
          s1.pop();
          t_queue result = s2.empty() ? element : cmp(element, s2.top().
              second);
          s2.push({element, result});
        }
      }
     }
  public:
    void push(t_queue x){
      t_queue result = s1.empty() ? x : cmp(x, s1.top().second);
      s1.push({x, result});
    }
    void pop(){
      move();
      s2.pop();
    }
    t_queue front(){
      move();
      return s2.top().first;
    }
    t_queue query(){
      if (s1.empty() || s2.empty())
        return s1.empty() ? s2.top().second : s1.top().second;
      else
        return cmp(s1.top().second, s2.top().second);
    }
    t_queue size(){
      return s1.size() + s2.size();
    }
  };
```

1.10 Segment Tree

```
  #include <bits/stdc++.h>
  using namespace std;
  class SegTree{
  private:
    typedef int Node;
    Node neutral = 0;
    vector<Node> st;
    vector<int> v;
    int n;
    Node join(Node a, Node b){
      return (a + b);
    }
    void build(int node, int i, int j){
      if(i == j){
        st[node] = v[i];
        return;
      }
      int m = (i+j)/2;
      int l = (node<<1);
      int r = l + 1;
      build(l, i, m);
      build(r, m+1, j);
      st[node] = join(st[l], st[r]);
    }
    Node query(int node, int i, int j, int a, int b){
      if( (i>b) or (j<a) )
        return neutral;
      if( (a<=i) and (j<=b) )
        return st[node];
      int m = (i+j)/2;
      int l = (node<<1);
      int r = l + 1;
      return join( query(l, i, m, a, b), query(r, m+1, j, a, b) );
    }
    void update(int node, int i, int j, int idx, Node value){
      if(i == j){
        st[node] = value;
        return;
      }
      int m = (i+j)/2;
      int l = (node<<1);
      int r = l + 1;
      if(idx <= m)
        update(l, i, m, idx, value);
      else
        update(r, m+1, j, idx, value);
      st[node] = join(st[l], st[r]);
    }
  public:
    template <class MyIterator>
    SegTree(MyIterator begin, MyIterator end){
      n = end-begin;
      v = vector<int>(begin, end);
      st.resize(4*n + 5);
      build(1, 0, n-1);
    }
    //0-indexed [a, b]
    int query(int a, int b){
      return query(1, 0, n-1, a, b);
    }
    //0-indexed
    void update(int idx, int value){
      update(1, 0, n-1, idx, value);
    }
  };
```

1.11 Segment Tree Iterative

```
  #include <bits/stdc++.h>
  using namespace std;
  class SegTreeIterative{
  private:
     typedef int Node;
     Node neutral = 0;
    vector<Node> st;
    int n;
    inline Node join(Node a, Node b){
      return a + b;
    }
  public:
    template <class MyIterator>
    SegTreeIterative(MyIterator begin, MyIterator end) {
      int sz = end-begin;
      for (n = 1; n < sz; n <<= 1);
      st.assign(n << 1, neutral);
      for(int i=0; i<sz; i++, begin++) st[i+n] = (*begin);
      for(int i=n+sz-1; i>1; i--)
        st[i>>1] = join(st[i>>1], st[i]);
    }
    //0-indexed
    void update(int i, Node x) {
      st[i += n] = x;
      for (i >>= 1; i; i >>= 1)
        st[i] = join(st[i<<1], st[1+(i<<1)]);
    }
    //0-indexed [l, r]
    Node query(int l, int r) {
      Node ans = neutral;
      for (l+=n, r+=n+1; l<r; l>>=1, r>>=1) {
        if (l & 1) ans = join(ans, st[l++]);
        if (r & 1) ans = join(ans, st[--r]);
      }
      return ans;
    }
  };
```

1.12 Segment Tree Lazy

```
  #include <bits/stdc++.h>
  using namespace std;
  class SegTreeLazy{
  private:
    typedef int Node;
    vector<Node> st;
    vector<int> lazy, v;
    int n;
    Node neutral = 0;
    inline Node join(Node a, Node b){
      return a+b;
    }
    inline void upLazy(int &node, int &i, int &j){
      if(lazy[node] != 0){
        st[node] += lazy[node]*(j-i+1);
        //tree[node] += lazy[node];
        if(i != j){
          lazy[(node<<1)] += lazy[node];
          lazy[(node<<1)+1] += lazy[node];
        }
        lazy[node] = 0;
      }
    }
    void build(int node, int i, int j){
      if(i == j){
        st[node] = v[i];
        return;
      }
      int m = (i+j)/2;
      int l = (node<<1);
      int r = l + 1;
      build(l, i, m);
      build(r, m+1, j);
      st[node] = join(st[l], st[r]);
    }
    Node query(int node, int i, int j, int a, int b){
      upLazy(node, i, j);
      if( (i>b) or (j<a) )
        return neutral;
      if( (a<=i) and (j<=b) ){
        return st[node];
      }
      int m = (i+j)/2;
      int l = (node<<1);
      int r = l + 1;
      return join( query(l, i, m, a, b), query(r, m+1, j, a, b) );
    }
    void update(int node, int i, int j, int a, int b, int value){
      upLazy(node, i, j);
      if( (i>j) or (i>b) or (j<a) )
        return;
      if( (a<=i) and (j<=b) ){
        lazy[node] = value;
        upLazy(node, i, j);
      }else{
        int m = (i+j)/2;
        int l = (node<<1);
        int r = l + 1;
        update(l, i, m, a, b, value);
        update(r, m+1, j, a, b, value);
        st[node] = join(st[l], st[r]);
      }
    }
  public:
    template <class MyIterator>
    SegTreeLazy(MyIterator begin, MyIterator end){
      n = end-begin;
      v = vector<int>(begin, end);
      st.resize(4*n + 5);
      lazy.assign(4*n + 5, 0);
      build(1, 0, n-1);
    }
    //0-indexed [a, b]
    int query(int a, int b){
      return query(1, 0, n-1, a, b);
    }
    //0-indexed [a, b]
    void update(int a, int b, int value){
      update(1, 0, n-1, a, b, value);
    }
  };
```

1.13 Set and Clear

```
  #include <bits/stdc++.h>
  using namespace std;
  class SetAndClear{
  private:
    typedef long long ll;
    typedef pair<ll, ll> pii;
    struct cmp{
      bool operator()(pii a, pii b){
        return a.second < b.second;
      }
    };
    std::set<pii, cmp> st;
    ll ans;
  public:
    SetAndClear(ll first, ll last){
      ans = last - first + 1LL;
      clear(first, last);
    }
    //set [a, b]
    void set(ll a, ll b){
      auto p = st.upper_bound(pii(0, a-1LL));
      if( (p == st.end()) or (p->first > b) ){
        return;
      }else{
        ll x = p->first;
        ll y = p->second;
        ans += (y - x + 1LL);
        p = st.erase(p);
        if(x<=(a - 1LL)){
          ans -= (a - x);
          st.insert(pii(x, a - 1LL));
        }
        if( (b + 1LL) <= y){
          ans -= (y - b);
          st.insert(pii(b + 1LL, y));
        }
        while( (p != st.end()) and (p->first <= b) ){
          x = p->first;
          y = p->second;
          ans += (y - x + 1LL);
          if(y > b){
            ans -= (y - b);
            st.erase(p);
            st.insert(pii(b + 1LL, y));
            break;
          }else{
            p = st.erase(p);
          }
        }
      }
    }
    //clear [a, b]
    void clear(ll a, ll b){
      auto p = st.upper_bound(pii(0, a-1LL));
      if( (p == st.end()) or (p->first > b) ){
        ans -= (b - a + 1LL);
        st.insert(pii(a, b));
      }else{
        ll x = p->first;
        ll y = p->second;
        ans += (y - x + 1LL);
        p = st.erase(p);
        a = min(x, a);
        b = max(y, b);
        while( (p != st.end()) and (p->first <= b) ){
          x = p->first;
          y = p->second;
          ans += (y - x + 1LL);
          a = min(x, a);
          b = max(y, b);
          p = st.erase(p);
        }
        ans -= (b - a + 1LL);
        st.insert(pii(a, b));
      }
    }
    ll count(){
      return ans;
    }
  };
```

1.14 Sparse Table

```
  #include <bits/stdc++.h>
  using namespace std;
  class SparseTable{
  private:
    typedef int t_st;
    vector<vector<t_st> > st;
    vector<int> log2;
    t_st neutral = 0x3f3f3f3f;
    int nLog;
    t_st join(t_st a, t_st b){
      return min(a, b);
    }
  public:
    template <class MyIterator>
    SparseTable(MyIterator begin, MyIterator end){
      int n = end-begin;
      nLog = 20;
      log2.resize(n+1);
      log2[1] = 0;
      for (int i = 2; i <=n; i++)
        log2[i] = log2[i/2] + 1;
      st.resize(n, vector<t_st>(nLog, neutral));
      for(int i=0; i<n; i++, begin++)
        st[i][0] = (*begin);
      for(int j=1; j<nLog; j++)
        for(int i=0; (i+(1<<(j-1))) < n; i++)
          st[i][j] = join(st[i][j-1], st[i+(1<<(j-1))][j-1]);
    }
    //0-indexed [a, b]
    t_st query(int a, int b){
      int d = b - a + 1;
      t_st ans = neutral;
      for(int j=nLog-1; j>=0; j--){
        if(d & (1<<j)){
          ans = join(ans, st[a][j]);
          a = a + (1<<(j));
        }
      }
      return ans;
    }
    //0-indexed [a, b]
    t_st queryRMQ(int a, int b){
      int j = log2[b - a + 1];
      return join(st[a][j], st[b - (1 << j) + 1][j]);
  }
};
```

1.15 SQRT Decomposition

```
#include <bits/stdc++.h>
using namespace std;
struct SqrtDecomposition{
  typedef long long t_sqrt;
  int sqrtLen;
  vector<t_sqrt> block;
  vector<t_sqrt> v;
  template <class MyIterator>
  SqrtDecomposition(MyIterator begin, MyIterator end){
    int n = end-begin;
    sqrtLen = (int) sqrt (n + .0) + 1;
    v.resize(n);
    block.resize(sqrtLen + 5);
    for(int i=0; i<n; i++, begin++){
      v[i] = (*begin);
      block[i / sqrtLen] += v[i];
    }
  }
  //0-indexed
  void update(int idx, t_sqrt new_value){
    t_sqrt d = new_value - v[idx];
    v[idx] += d;
    block[idx/sqrtLen] += d;
  }
  //0-indexed [l, r]
  t_sqrt query(int l, int r){
    t_sqrt sum = 0;
    int c_l = l/sqrtLen, c_r = r/sqrtLen;
    if (c_l == c_r){
      for (int i=l; i<=r; i++)
        sum += v[i];
    }else{
      for (int i=l, end=(c_l+1)*sqrtLen-1; i<=end; i++)
        sum += v[i];
      for (int i=c_l+1; i<=c_r-1; i++)
        sum += block[i];
      for (int i=c_r*sqrtLen; i<=r; i++)
        sum += v[i];
    }
    return sum;
  }
};
```

1.16 SQRT Tree

```
#include <bits/stdc++.h>
using namespace std;
class SqrtTree{
private:
  typedef long long t_sqrt;
  t_sqrt op(const t_sqrt &a, const t_sqrt &b){
    return a | b;
  }
inline int log2Up(int n) {
  int res = 0;
  while ((1 << res) < n)
    res++;
  return res;
}
int n, lg, indexSz;
vector<t_sqrt> v;
vector<int> clz, layers, onLayer;
vector< vector<t_sqrt> > pref, suf, between;
inline void buildBlock(int layer, int l, int r) {
  pref[layer][l] = v[l];
  for (int i = l+1; i < r; i++)
    pref[layer][i] = op(pref[layer][i-1], v[i]);
  suf[layer][r-1] = v[r-1];
  for (int i = r-2; i >= l; i--)
    suf[layer][i] = op(v[i], suf[layer][i+1]);
}
inline void buildBetween(int layer, int lBound, int rBound, int
    betweenOffs) {
  int bSzLog = (layers[layer]+1) >> 1;
  int bCntLog = layers[layer] >> 1;
  int bSz = 1 << bSzLog;
  int bCnt = (rBound - lBound + bSz - 1) >> bSzLog;
  for (int i = 0; i < bCnt; i++) {
    t_sqrt ans;
    for (int j = i; j < bCnt; j++) {
      t_sqrt add = suf[layer][lBound + (j << bSzLog)];
      ans = (i == j) ? add : op(ans, add);
      between[layer-1][betweenOffs + lBound + (i << bCntLog) + j] =
          ans;
    }
  }
}
inline void buildBetweenZero() {
  int bSzLog = (lg+1) >> 1;
  for (int i = 0; i < indexSz; i++) {
    v[n+i] = suf[0][i << bSzLog];
  }
  build(1, n, n + indexSz, (1 << lg) - n);
}
inline void updateBetweenZero(int bid) {
  int bSzLog = (lg+1) >> 1;
  v[n+bid] = suf[0][bid << bSzLog];
  update(1, n, n + indexSz, (1 << lg) - n, n+bid);
}
void build(int layer, int lBound, int rBound, int betweenOffs) {
  if (layer >= (int)layers.size())
    return;
  int bSz = 1 << ((layers[layer]+1) >> 1);
  for (int l = lBound; l < rBound; l += bSz) {
    int r = min(l + bSz, rBound);
    buildBlock(layer, l, r);
    build(layer+1, l, r, betweenOffs);
  }
  if (layer == 0)
    buildBetweenZero();
  else
    buildBetween(layer, lBound, rBound, betweenOffs);
}
void update(int layer, int lBound, int rBound, int betweenOffs, int
      x) {
    if (layer >= (int)layers.size())
      return;
    int bSzLog = (layers[layer]+1) >> 1;
    int bSz = 1 << bSzLog;
    int blockIdx = (x - lBound) >> bSzLog;
    int l = lBound + (blockIdx << bSzLog);
    int r = min(l + bSz, rBound);
    buildBlock(layer, l, r);
    if (layer == 0)
      updateBetweenZero(blockIdx);
    else
      buildBetween(layer, lBound, rBound, betweenOffs);
    update(layer+1, l, r, betweenOffs, x);
  }
  inline t_sqrt query(int l, int r, int betweenOffs, int base) {
    if (l == r)
      return v[l];
    if (l + 1 == r)
      return op(v[l], v[r]);
    int layer = onLayer[clz[(l - base) ^ (r - base)]];
    int bSzLog = (layers[layer]+1) >> 1;
    int bCntLog = layers[layer] >> 1;
    int lBound = (((l - base) >> layers[layer]) << layers[layer]) +
        base;
    int lBlock = ((l - lBound) >> bSzLog) + 1;
    int rBlock = ((r - lBound) >> bSzLog) - 1;
    t_sqrt ans = suf[layer][l];
    if (lBlock <= rBlock) {
      t_sqrt add;
      if(layer == 0)
        add = query(n + lBlock, n + rBlock, (1 << lg) - n, n);
      else
        add = between[layer-1][betweenOffs + lBound + (lBlock <<
            bCntLog) + rBlock];
      ans = op(ans, add);
    }
    ans = op(ans, pref[layer][r]);
    return ans;
  }
public:
  template <class MyIterator>
  SqrtTree(MyIterator begin, MyIterator end){
    n = end-begin;
    v.resize(n);
    for(int i=0; i<n; i++, begin++)
      v[i] = (*begin);
    lg = log2Up(n);
    clz.resize(1<<lg);
    onLayer.resize(lg + 1);
    clz[0] = 0;
    for (int i = 1; i < (int)clz.size(); i++)
      clz[i] = clz[i >> 1] + 1;
    int tlg = lg;
    while (tlg > 1) {
      onLayer[tlg] = (int)layers.size();
      layers.push_back(tlg);
      tlg = (tlg+1) >> 1;
    }
    for (int i = lg-1; i >= 0; i--)
      onLayer[i] = max(onLayer[i], onLayer[i+1]);
    int betweenLayers = max(0, (int)layers.size() - 1);
    int bSzLog = (lg+1) >> 1;
    int bSz = 1 << bSzLog;
    indexSz = (n + bSz - 1) >> bSzLog;
    v.resize(n + indexSz);
    pref.assign(layers.size(), vector<t_sqrt>(n + indexSz));
    suf.assign(layers.size(), vector<t_sqrt>(n + indexSz));
    between.assign(betweenLayers, vector<t_sqrt>((1 << lg) + bSz));
    build(0, 0, n, 0);
  }
  //0-indexed
  inline void update(int x, const t_sqrt &item) {
    v[x] = item;
    update(0, 0, n, 0, x);
  }
  //0-indexed [l, r]
  inline t_sqrt query(int l, int r) {
    return query(l, r, 0, 0);
  }
};
```

1.17 Stack Query

```
#include <bits/stdc++.h>
using namespace std;
struct StackQuery{
  typedef int t_stack;
  stack<pair<t_stack, t_stack> > st;
  t_stack cmp(t_stack a, t_stack b){
    return min(a, b);
  }
  void push(t_stack x){
    t_stack new_value = st.empty() ? x : cmp(x, st.top().second);
    st.push({x, new_value});
  }
  void pop(){
    st.pop();
  }
  t_stack top(){
    return st.top().first;
  }
  t_stack query(){
    return st.top().second;
  }
  t_stack size(){
    return st.size();
  }
};
```

1.18 Treap

```
#include <bits/stdc++.h>
using namespace std;
class Treap {
private:
  typedef int t_treap;
  struct Node {
    t_treap x, y, size;
    Node *l, *r;
    Node(t_treap _x) : x(_x), y(rand()), size(1), l(NULL), r(NULL){}
  };
  Node* root;
  int size(Node* t) { return t ? t->size : 0; }
  Node* refresh(Node* t) {
    if (!t) return t;
    t->size = 1 + size(t->l) + size(t->r);
    return t;
  }
  void split(Node* &t, t_treap k, Node* &a, Node* &b) {
    Node* aux;
    if(!t){
      a = b = NULL;
    }else if(t->x < k) {
      split(t->r, k, aux, b);
      t->r = aux;
      a = refresh(t);
    }else{
      split(t->l, k, a, aux);
      t->l = aux;
      b = refresh(t);
    }
  }
  Node* merge(Node* a, Node* b) {
    if (!a || !b) return a ? a : b;
    if (a->y < b->y) {
      a->r = merge(a->r, b);
      return refresh(a);
    }
    else {
      b->l = merge(a, b->l);
      return refresh(b);
    }
  }
  Node* count(Node* t, t_treap k) {
    if (!t) return NULL;
    else if (k < t->x) return count(t->l, k);
    else if (k == t->x) return t;
    else return count(t->r, k);
  }
  Node* nth(Node* t, int n) {
    if (!t) return NULL;
    if (n <= size(t->l)) return nth(t->l, n);
    else if (n == size(t->l) + 1) return t;
    else return nth(t->r, n-size(t->l)-1);
  }
  void del(Node* &t) {
    if (!t) return;
    if (t->l) del(t->l);
    if (t->r) del(t->r);
    delete t;
    t = NULL;
  }
public:
  Treap() : root(NULL) {}
  ~Treap() { clear(); }
  void clear() { del(root); }
  int size() { return size(root); }
  bool count(t_treap k) { return count(root, k) != NULL; }
  bool insert(t_treap k) {
    if (count(k)) return false;
    Node *a, *b;
    split(root, k, a, b);
    root = merge(merge(a, new Node(k)), b);
    return true;
  }
  bool erase(t_treap k) {
    Node * f = count(root, k);
    if (!f) return false;
    Node *a, *b, *c, *d;
    split(root, k, a, b);
    split(b, k+1, c, d);
    root = merge(a, d);
    delete f;
    return true;
  }
  //1-indexed
  t_treap nth(int n) {
    Node* ans = nth(root, n);
    return ans ? ans->x : -1;
  }
};
```

1.19 Union Find

```
#include <bits/stdc++.h>
using namespace std;
class UnionFind{
private:
  vector<int> p, w, sz;
public:
  UnionFind(int n){
    w.resize(n+1, 1);
    sz.resize(n+1, 1);
    p.resize(n+1);
    for(int i=0; i<=n; i++)
      p[i] = i;
  }
  int find(int x){
    if(p[x] == x)
      return x;
    return p[x] = find(p[x]);
  }
  void join(int x, int y){
    x = find(x);
    y = find(y);
    if(x == y)
      return;
    if(w[x] > w[y])
      swap(x, y);
    p[x] = y;
    sz[y] += sz[x];
    if(w[x] == w[y])
      w[y]++;
  }
  bool isSame(int x, int y){
    return find(x) == find(y);
  }
  int size(int x){
    return sz[find(x)];
    }
  };
```

1.20 Wavelet Tree

```
  #include <bits/stdc++.h>
  using namespace std;
  struct WaveletTree{
  private:
    typedef int t_wavelet;
    t_wavelet lo, hi;
    WaveletTree *l, *r;
    vector<int> a, b;
  public:
    template <class MyIterator>
    WaveletTree(MyIterator begin, MyIterator end, t_wavelet x, t_wavelet
         y){
      lo = x, hi = y;
      if(lo == hi or begin >= end) return;
      t_wavelet mid = (lo+hi-1)/2;
      auto f = [mid](int x){
        return x <= mid;
      };
      a.reserve(end-begin+1);
      b.reserve(end-begin+1);
      a.push_back(0);
      b.push_back(0);
      for(auto it = begin; it != end; it++){
        a.push_back(a.back() + f(*it));
        b.push_back(b.back() + !f(*it));
      }
      auto pivot = stable_partition(begin, end, f);
      l = new WaveletTree(begin, pivot, lo, mid);
      r = new WaveletTree(pivot, end, mid+1, hi);
    }
    //kth smallest element in range [l, r]
    //1-indexed
    int kth(int l, int r, int k){
      if(l > r) return 0;
      if(lo == hi) return lo;
      int inLeft = a[r] - a[l-1];
      int lb1 = a[l-1] + 1, rb1 = a[r];
      int lb2 = b[l-1] + 1, rb2 = b[r];
      if(k <= inLeft) return this->l->kth(lb1, rb1, k);
      return this->r->kth(lb2, rb2, k-inLeft);
    }
    //Amount of numbers in the range [l, r] Less than or equal to k
    //1-indexed
    int lte(int l, int r, int k) {
      if(l > r or k < lo) return 0;
      if(hi <= k) return r - l + 1;
      int lb1 = a[l-1] + 1, rb1 = a[r];
      int lb2 = b[l-1] + 1, rb2 = b[r];
      return this->l->lte(lb1, rb1, k) + this->r->lte(lb2, rb2, k);
    }
    //Amount of numbers in the range [l, r] equal to k
    //1-indexed
    int count(int l, int r, int k) {
      if(l > r or k < lo or k > hi) return 0;
      if(lo == hi) return r - l + 1;
      int mid = (lo+hi-1)/2;
      int lb1 = a[l-1]+1, rb1 = a[r];
      int lb2 = b[l-1]+1, rb2 = b[r];
      if(k <= mid) return this->l->count(lb1, rb1, k);
      return this->r->count(lb2, rb2, k);
    }
    ~WaveletTree(){
      delete l;
      delete r;
    }
  };
```

2 Graph Algorithms

3 Dynamic Programming

4 Math

5 Geometry

6 String Algorithms

7 Miscellaneous

8 Theorems and Formulas