GEMP - UFC Quixadá - ICPC Library

Contents

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;
```

```cpp
    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

```cpp
#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

```cpp
#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

```cpp
#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 WaveletTree{
private:
  int lo, hi;
  WaveletTree *l = 0;
  WaveletTree *r = 0;
  SplayTree b;
public:
  WaveletTree(int min_value, int max_value) {
    lo = min_value;
```

```
  hi = max_value;
  b.insert(0, 0);
}
~WaveletTree() {
  delete l;
  delete r;
}
//0-indexed
void insert(int idx, int x) {
  if(lo >= hi)
    return;
  int mid = (lo + hi - 1) / 2;
  if(x <= mid) {
    l = l ?: new WaveletTree(lo, mid);
    l->insert(b.insert(idx, 1), x);
  }else{
    r = r ?: new WaveletTree(mid+1, hi);
    r->insert(idx - b.insert(idx, 0), x);
  }
}
//0-indexed
void erase(int idx) {
  if(lo == hi)
    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 smallest element in range [i, j[
//0-indexed
int kth(int i, int j, int k) {
  if(i >= j)
    return 0;
  if(lo == hi)
    return lo;
  int x = b.rank(i);
  int y = b.rank(j);
  if(k <= y-x)
    return l->kth(x, y, k);
  else
    return r->kth(i-x, j-y, k-(y-x));
}
//Amount of numbers in the range [i, j[ Less than or equal to k
//0-indexed
int lte(int i, int j, int k) {
  if(i >= j or k < lo) return 0;
  if(hi <= k) return j - i;
  int x = b.rank(i);
  int y = b.rank(j);
  return l->lte(x, y, k) + r->lte(i-x, j-y, k);
}
//Amount of numbers in the range [i, j[ equal to k
//0-indexed
int count(int i, int j, int k) {
  if(i >= j or k < lo or k > hi) return 0;
  if(lo == hi) return j - i;
    int mid = (lo + hi - 1)/2;
    int x = b.rank(i);
    int y = b.rank(j);
    if(k <= mid) return l->count(x, y, k);
    return r->count(i-x, j-y, k);
  }
  //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);
    }
```

```cpp
      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

```cpp
#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 Range Color

```
#include <bits/stdc++.h>
using namespace std;
class RangeColor{
private:
  typedef long long ll;
  struct Node{
    ll l, r;
    int color;
    Node(){}
    Node(ll l, ll r, int color):l(l), r(r), color(color){}
  };
  struct cmp{
    bool operator()(Node a, Node b){
      if(a.r == b.r) return a.l < b.l;
      return a.r < b.r;
    }
  };
  std::set<Node, cmp> st;
  vector<ll> ans;
public:
  RangeColor(ll first, ll last, int maxColor){
    ans.resize(maxColor + 1);
    ans[0] = last - first + 1LL;
    st.insert(Node(first, last, 0));
  }
  //set newColor in [a, b]
  void set(ll a, ll b, int newColor){
    auto p = st.upper_bound(Node(0, a-1LL, -1));
    assert(p != st.end());
    ll l = p->l;
    ll r = p->r;
    int oldColor = p->color;
    ans[oldColor] -= (r - l + 1LL);
    p = st.erase(p);
    if(l < a){
      ans[oldColor] += (a - l);
      st.insert(Node(l, a - 1LL, oldColor));
    }
    if(b < r){
      ans[oldColor] += (r - b);
```

```
      st.insert(Node(b + 1LL, r, oldColor));
    }
    while( (p != st.end()) and (p->l <= b) ){
      l = p->l;
      r = p->r;
      oldColor = p->color;
      ans[oldColor] -= (r - l + 1LL);
      if(b < r){
        ans[oldColor] += (r - b);
        st.insert(Node(b + 1LL, r, oldColor));
        st.erase(p);
        break;
      }else{
        p = st.erase(p);
      }
    }
    ans[newColor] += (b - a + 1LL);
    st.insert(Node(a, b, newColor));
  }
  ll countColor(int x){
    return ans[x];
  }
};
```

1.11 Segment Tree

```
#include <bits/stdc++.h>
using namespace std;
class SegTree{
private:
  typedef long long 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]
  Node 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);
  }
};
```

1.12 Segment Tree Iterative

```
#include <bits/stdc++.h>
using namespace std;
class SegTreeIterative{
private:
   typedef long long 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;
  }
};
```

1.13 Segment Tree Lazy

```
#include <bits/stdc++.h>
using namespace std;
class SegTreeLazy{
private:
  typedef long long Node;
  vector<Node> st;
  vector<long long> lazy;
  vector<int> 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]
  Node 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);
  }
};
```

1.14 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 minX,
      t_wavelet maxX){
    lo = minX, hi = maxX;
    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 [i, j]
  //1-indexed
  int kth(int i, int j, int k){
    if(i > j) return 0;
    if(lo == hi) return lo;
    int inLeft = a[j] - a[i-1];
    int i1 = a[i-1] + 1, j1 = a[j];
    int i2 = b[i-1] + 1, j2 = b[j];
    if(k <= inLeft) return l->kth(i1, j1, k);
    return r->kth(i2, j2, k-inLeft);
  }
  //Amount of numbers in the range [i, j] Less than or equal to k
  //1-indexed
  int lte(int i, int j, int k) {
    if(i > j or k < lo) return 0;
    if(hi <= k) return j - i + 1;
    int i1 = a[i-1] + 1, j1 = a[j];
    int i2 = b[i-1] + 1, j2 = b[j];
    return l->lte(i1, j1, k) + r->lte(i2, j2, k);
  }
  //Amount of numbers in the range [i, j] equal to k
  //1-indexed
  int count(int i, int j, int k) {
    if(i > j or k < lo or k > hi) return 0;
    if(lo == hi) return j - i + 1;
    int mid = (lo+hi-1)/2;
    int i1 = a[i-1]+1, j1 = a[j];
    int i2 = b[i-1]+1, j2 = b[j];
    if(k <= mid) return l->count(i1, j1, k);
    return r->count(i2, j2, k);
  }
  ~WaveletTree(){
    delete l;
    delete r;
  }
};
```

2 Graph Algorithms

2.1 2-SAT

```
#include "strongly_connected_component.h"
using namespace std;
struct SAT{
  typedef pair<int, int> pii;
  vector<pii> edges;
  int n;
  SAT(int size){
    n = 2*size;
  }
  vector<bool> solve2SAT(){
    vector<bool> vAns(n/2, false);
    vector<int> comp = SCC::scc(n, edges);
    for(int i=0; i<n; i+=2){
      if(comp[i] == comp[i + 1])
        return vector<bool>();
      vAns[i / 2] = (comp[i] > comp[i+1]);
    }
    return vAns;
  }
  int v(int x){
    if(x>=0)
```

```
      return (x<<1);
    x = ~x;
    return (x<<1)^1;
  }
  void add(int a, int b){
    edges.push_back(pii(a, b));
  }
  void addOr(int a, int b){
    add(v(~a), v(b)); add(v(~b), v(a));
  }
  void addImp(int a, int b){
    addOr(~a, b);
  }
  void addEqual(int a, int b){
    addOr(a, ~b);
    addOr(~a, b);
  }
  void addDiff(int a, int b){
    addEqual(a, ~b);
  }
};
```

2.2 Minimum Cost Maximum Flow

```
#include <bits/stdc++.h>
using namespace std;
template <class T = int>
class MCMF{
private:
  struct Edge {
    int to;
    T cap, cost;
    Edge(int a, T b, T c) : to(a), cap(b), cost(c) {}
  };
  int n;
  vector<std::vector<int>> edges;
  vector<Edge> list;
  vector<int> from;
  vector<T> dist, pot;
  vector<bool> visit;
  pair<T, T> augment(int src, int sink) {
    pair<T, T> flow = {list[from[sink]].cap, 0};
    for(int v = sink; v != src; v = list[from[v]^1].to) {
      flow.first = std::min(flow.first, list[from[v]].cap);
      flow.second += list[from[v]].cost;
    }
    for(int v = sink; v != src; v = list[from[v]^1].to) {
      list[from[v]].cap -= flow.first;
      list[from[v]^1].cap += flow.first;
    }
    return flow;
  }
  queue<int> q;
  bool SPFA(int src, int sink) {
    T INF = numeric_limits<T>::max();
    dist.assign(n, INF);
    from.assign(n, -1);
    q.push(src);
    dist[src] = 0;
    while(!q.empty()){
      int on = q.front();
      q.pop();
      visit[on] = false;
      for(auto e: edges[on]){
        auto ed = list[e];
        if(ed.cap == 0) continue;
        T toDist = dist[on] + ed.cost + pot[on] - pot[ed.to];
        if(toDist < dist[ed.to]){
          dist[ed.to] = toDist;
          from[ed.to] = e;
          if(!visit[ed.to]){
            visit[ed.to] = true;
            q.push(ed.to);
          }
        }
      }
    }
    return dist[sink] < INF;
  }
  void fixPot() {
    T INF = numeric_limits<T>::max();
    for(int i = 0; i < n; i++) {
      if(dist[i] < INF) pot[i] += dist[i];
    }
  }
public:
  MCMF(int size) {
    n = size;
    edges.resize(n);
    pot.assign(n, 0);
    dist.resize(n);
    visit.assign(n, false);
  }
  pair<T, T> solve(int src, int sink) {
    pair<T, T> ans(0, 0);
    // Can use dijkstra to speed up depending on the graph
    if(!SPFA(src, sink)) return ans;
    fixPot();
    // Can use dijkstra to speed up depending on the graph
    while(SPFA(src, sink)) {
      auto flow = augment(src, sink);
      ans.first += flow.first;
      ans.second += flow.first * flow.second;
      fixPot();
    }
    return ans;
  }
  void addEdge(int from, int to, T cap, T cost) {
    edges[from].push_back(list.size());
    list.push_back(Edge(to, cap, cost));
    edges[to].push_back(list.size());
    list.push_back(Edge(from, 0, -cost));
  }
};
/*bool dij(int src, int sink){
  T INF = numeric_limits<T>::max();
  dist.assign(n, INF);
  from.assign(n, -1);
  visit.assign(n, false);
  dist[src] = 0;
  for(int i = 0; i < n; i++){
```

```
    int best = -1;
    for(int j = 0; j < n; j++){
      if(visit[j]) continue;
      if(best == -1 || dist[best] > dist[j]) best = j;
    }
    if(dist[best] >= INF) break;
    visit[best] = true;
    for(auto e : edges[best]){
      auto ed = list[e];
      if(ed.cap == 0) continue;
      T toDist = dist[best] + ed.cost + pot[best] - pot[ed.to];
      assert(toDist >= dist[best]);
      if(toDist < dist[ed.to]){
        dist[ed.to] = toDist;
        from[ed.to] = e;
      }
    }
  }
  return dist[sink] < INF;
}*/
```

2.3 Strongly Connected Component

```
#include "topological_sort.h"
using namespace std;
namespace SCC{
  typedef pair<int, int> pii;
  vector<vector<int>> revAdj;
  vector<int> component;
  void dfs(int u, int c) {
    component[u] = c;
    for(int to: revAdj[u]){
      if(component[to] == -1)
        dfs(to, c);
    }
  }
  vector<int> scc(int n, vector<pii> &edges){
    revAdj.assign(n, vector<int>());
    for(pii p: edges)
      revAdj[p.second].push_back(p.first);
    vector<int> tp = TopologicalSort::order(n, edges);
    component.assign(n, -1);
    int comp = 0;
    for(int u: tp){
      if(component[u] == -1)
        dfs(u, comp++);
    }
    return component;
  }
}
```

2.4 Topological Sort

```
#include <bits/stdc++.h>
using namespace std;
namespace TopologicalSort{
  typedef pair<int, int> pii;
  vector<vector<int>> adj;
  vector<bool> visited;
  vector<int> vAns;
  void dfs(int u){
    visited[u] = true;
    for(int to : adj[u]){
      if(!visited[to])
        dfs(to);
    }
    vAns.push_back(u);
  }
  vector<int> order(int n, vector<pii> &edges){
    adj.assign(n, vector<int>());
    for(pii p: edges)
      adj[p.first].push_back(p.second);
    visited.assign(n, false);
    vAns.clear();
    for(int i = 0; i < n; i++){
      if(!visited[i])
        dfs(i);
    }
    reverse(vAns.begin(), vAns.end());
    return vAns;
  }
};
```

3 Dynamic Programming

3.1 Divide and Conquer Optimization

Reduces the complexity from $O(n^2k)$ to $O(nk \log n)$ of PD's in the following ways (and other variants):

$$dp[n][k] = \max_{0 \le i < n} (dp[i][k-1] + C[i+1][n]), \ \mathit{base\ case}: \ dp[0][j], dp[i][0] \tag{1}$$

- $C[i][j]$ = the cost only depends on i and j.
- $opt[n][k] = i$ is the optimal value that maximizes $dp[n][k]$.

It is necessary that opt is increasing along each column: $opt[j][k] \le opt[j+1][k]$.

3.2 Divide and Conquer Optimization Implementation

```
#include <bits/stdc++.h>
using namespace std;
int C(int i, int j);
const int MAXN = 100010;
const int MAXK = 110;
const int INF = 0x3f3f3f3f;
int dp[MAXN][MAXK];
void calculateDP(int l, int r, int k, int opt_l, int opt_r) {
  if(l > r) return;
  int mid = (l+r)>>1;
  int ans = -INF, opt;
  for(int i=opt_l; i<=min(opt_r, mid-1); i++) {
    if(ans < dp[i][k-1] + C(i+1, mid)) {
      opt = i;
```

```
      ans = dp[i][k-1] + C(i+1, mid);
    }
  }
  dp[mid][k] = ans;
  calculateDP(l, mid-1, k, opt_l, opt);
  calculateDP(mid+1, r, k, opt, opt_r);
}
int solve(int n, int k){
  for(int i=0; i<=n; i++) dp[i][0] = -INF;
  for(int j=0; j<=k; j++) dp[0][j] = -INF;
  dp[0][0] = 0;
  for(int j=1; j<=k; j++)
    calculateDP(1, n, j, 0, n-1);
  return dp[n][k];
}
```

4 Math

4.1 Basic Math

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
ll fastPow(ll base, ll exp, ll mod){
  base %= mod;
  //exp %= phi(mod) if base and mod are relatively prime
  ll ans = 1LL;
  while(exp > 0){
    if(exp & 1LL)
      ans = (ans *(__int128_t)base)%mod;
    base = (base *(__int128_t)base)%mod;
    exp >>= 1;
  }
  return ans;
}
ll extGcd(ll a, ll b, ll &x, ll &y) {
  if(b == 0){
    x = 1; y = 0; return a;
  }else{
    ll g = extGcd(b, a % b, y, x);
    y -= (a / b) * x;
    return g;
  }
}
ll gcd(ll a, ll b){
  return __gcd(a, b);
}
ll lcm(ll a, ll b){
  return (a/gcd(a, b))*b;
}
void enumeratingAllSubmasks(int mask){
  for(int s=mask; s; s=(s-1)&mask)
    cout << s << endl;
}
bool checkComposite(ll n, ll a, ll d, int s) {
  ll x = fastPow(a, d, n);
  if (x == 1 or x == n-1)
    return false;
  for (int r = 1; r < s; r++) {
    x = (x*(__int128_t)x)%n;
    if (x == n-1LL)
      return false;
  }
  return true;
};
bool millerRabin(ll n) {
  if(n < 2)
    return false;
  int r = 0;
  ll d = n - 1LL;
  while((d & 1LL) == 0){
    d >>= 1;
    r++;
  }
  for(int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
    if(n == a)
      return true;
    if(checkComposite(n, a, d, r))
      return false;
  }
  return true;
}
```

4.2 Binomial Coefficients

```
#include <bits/stdc++.h>
#include "./basic_math.h"
using namespace std;
typedef long long ll;
//O(k)
ll C1(int n, int k) {
  ll res = 1LL;
  for(int i = 1; i <= k; ++i)
    res = (res * (n - k + i)) / i;
  return res;
}
//O(n^2)
vector<vector<ll> > C2(int maxn, int mod){
  vector<vector<ll> > mat(maxn+1, vector<ll>(maxn+1, 0));
  mat[0][0] = 1;
  for(int n = 1; n <= maxn; n++) {
    mat[n][0] = mat[n][n] = 1;
    for(int k = 1; k < n; k++)
      mat[n][k] = (mat[n - 1][k - 1] + mat[n - 1][k])%mod;
  }
  return mat;
}
//O(N)
vector<int> factorial, inv_factorial;
void prevC3(int maxn, int mod){
  factorial.resize(maxn + 1);
  factorial[0] = 1;
  for(int i = 1; i <= maxn; i++)
    factorial[i] = (factorial[i-1]*1LL*i)%mod;
  inv_factorial.resize(maxn + 1);
  inv_factorial[maxn] = fastPow(factorial[maxn], mod-2, mod);
  for(int i = maxn-1; i >= 0; i--)
    inv_factorial[i] = (inv_factorial[i+1]*1LL*(i+1))%mod;
```

```
}
int C3(int n, int k, int mod){
  if(n < k)
    return 0;
  return (((factorial[n]*1LL*inv_factorial[k])%mod)*1LL*inv_factorial[
      n-k])%mod;
}
//O(P*log(P))
//C4(n, k, p) = Comb(n, k)%p
vector<int> changeBase(int n, int p){
  vector<int> v;
  while(n > 0){
    v.push_back(n%p);
    n/=p;
  }
  return v;
}
int C4(int n, int k, int p){
  auto vn = changeBase(n, p);
  auto vk = changeBase(k, p);
  int mx = max(vn.size(), vk.size());
  vn.resize(mx, 0);
  vk.resize(mx, 0);
  prevC3(p-1, p);
  int ans = 1;
  for(int i=0; i<mx; i++)
    ans = (ans * 1LL * C3(vn[i], vk[i], p))%p;
  return ans;
}
```

4.3 Chinese Remainder Theorem

```
#include <bits/stdc++.h>
#include "extended_euclidean.h"
using namespace std;
typedef long long ll;
namespace CRT{
  inline ll normalize(ll x, ll mod){
    x %= mod;
    if(x < 0) x += mod;
    return x;
  }
  ll solve(vector<ll> a, vector<ll>m){
    int n = a.size();
    for(int i=0; i<n; i++)
      normalize(a[i], m[i]);
    ll ans = a[0];
    ll lcm1 = m[0];
    for(int i = 1; i < n; i++){
      ll x, y;
      ll g = extGcd(lcm1, m[i], x, y);
      if((a[i] - ans) % g != 0)
        return -1;
      ans = normalize(ans + ((((a[i]-ans)/g)*x)%(m[i]/g))*lcm1, (lcm1/
          g)*m[i]);
      lcm1 = (lcm1/g)*m[i];//lcm(lcm1, m[i]);
    }
    return ans;
  }
}
```

4.4 Euler's totient

```
#include <bits/stdc++.h>
using namespace std;
int nthPhi(int n){
  int result = n;
  for(int i = 2; i <= n/i; i++){
    if(n%i == 0){
      while(n%i == 0)
        n /= i;
      result -= result/i;
    }
  }
  if(n > 1)
    result -= result/n;
  return result;
}
vector<int> phiFrom1toN(int n){
  vector<int> vPhi(n + 1);
  vPhi[0] = 0;
  vPhi[1] = 1;
  for(int i=2; i <= n; i++)
    vPhi[i] = i;
  for(int i=2; i <= n; i++){
    if(vPhi[i] == i){
      for(int j=i; j <= n; j+=i)
        vPhi[j] -= vPhi[j]/i;
    }
  }
  return vPhi;
}
```

4.5 Extended Euclidean

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
ll extGcd(ll a, ll b, ll &x, ll &y) {
  if(b == 0){
    x = 1; y = 0; return a;
  }else{
    ll g = extGcd(b, a % b, y, x);
    y -= (a / b) * x;
    return g;
  }
}
```

4.6 Gray Code

```
int grayCode(int nth){
  return nth^(nth>>1);
}
int revGrayCode(int g) {
  int nth = 0;
  for(; g>0; g>>=1)
    nth ^= g;
```

```
    return nth;
}
```

5 Geometry

6 String Algorithms

7 Miscellaneous

8 Theorems and Formulas

8.1 Binomial Coefficients

$(a+b)^n = \binom{n}{0}a^n + \binom{n}{1}a^{n-1}b + \binom{n}{2}a^{n-2}b^2 + \cdots + \binom{n}{k}a^{n-k}b^k + \cdots + \binom{n}{n}b^n$

Pascal's Triangle: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

Symmetry rule: $\binom{n}{k} = \binom{n}{n-k}$

Factoring in: $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$

Sum over k: $\sum_{k=0}^{n}\binom{n}{k} = 2^n$

Sum over n: $\sum_{m=0}^{n}\binom{m}{k} = \binom{n+1}{k+1}$

Sum over n and k: $\sum_{k=0}^{m}\binom{n+k}{k} = \binom{n+m+1}{m}$

Sum of the squares: $\binom{n}{0}^2 + \binom{n}{1}^2 + \cdots + \binom{n}{n}^2 = \binom{2n}{n}$

Weighted sum: $1\binom{n}{1} + 2\binom{n}{2} + \cdots + n\binom{n}{n} = n2^{n-1}$

Connection with the Fibonacci numbers: $\binom{n}{0} + \binom{n-1}{1} + \cdots + \binom{n-k}{k} + \cdots + \binom{0}{n} = F_{n+1}$

More formulas: $\sum_{k=0}^{m}(-1)^k \cdot \binom{n}{k} = (-1)^m \cdot \binom{n-1}{m}$

8.2 Catalan Number

Recursive formula: $C_0 = C_1 = 1$

$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}, n \geq 2$

Analytical formula: $C_n = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n}, n \geq 0$

The first few numbers Catalan numbers, C_n (starting from zero): $1, 1, 2, 5, 14, 42, 132, 429, 1430, \ldots$

The Catalan number C_n is the solution for:

- Number of correct bracket sequence consisting of n opening and n closing brackets.
- The number of rooted full binary trees with $n+1$ leaves (vertices are not numbered). A rooted binary tree is full if every vertex has either two children or no children.
- The number of ways to completely parenthesize $n+1$ factors.
- The number of triangulations of a convex polygon with $n+2$ sides (i.e. the number of partitions of polygon into disjoint triangles by using the diagonals).
- The number of ways to connect the $2n$ points on a circle to form n disjoint chords.
- The number of non-isomorphic full binary trees with n internal nodes (i.e. nodes having at least one son).
- The number of monotonic lattice paths from point $(0,0)$ to point (n,n) in a square lattice of size $n \times n$, which do not pass above the main diagonal (i.e. connecting $(0,0)$ to (n,n)).
- Number of permutations of length n that can be stack sorted (i.e. it can be shown that the rearrangement is stack sorted if and only if there is no such index $i < j < k$, such that $a_k < a_i < a_j$).
- The number of non-crossing partitions of a set of n elements.
- The number of ways to cover the ladder $1 \ldots n$ using n rectangles (The ladder consists of n columns, where i^{th} column has a height i).

8.3 Euler's Totient

If p is a prime number: $\phi(p) = p - 1$ and $\phi(p^k) = p^k - p^{k-1}$

If a and b are relatively prime, then: $\phi(ab) = \phi(a) \cdot \phi(b)$

In general: $\phi(ab) = \phi(a) \cdot \phi(b) \cdot \dfrac{gcd(a,b)}{\phi(gcd(a,b))}$

This interesting property was established by Gauss: $\sum_{d|n}\phi(d) = n$, Here the sum is over all positive divisors d of n.

Euler's theorem: $a^{\phi(m)} \equiv 1 \pmod{m}$, if a and m are relatively prime.

Generalization: $a^n \equiv a^{\phi(m)+[n \bmod \phi(m)]} \mod m$, for arbitrary a, m and n $\geq log_2(m)$.

8.4 Primes

If $n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$ então , then: Number of divisors is $d(n) = (e_1+1) \cdot (e_2+1) \cdots (e_k+1)$.

Sum of divisors is $\sigma(n) = \frac{p_1^{e_1+1}-1}{p_1-1} \cdot \frac{p_2^{e_2+1}-1}{p_2-1} \cdots \frac{p_k^{e_k+1}-1}{p_k-1}$