Modular Arithmetic:

```
ll FM[N];
int INIT = 0;
ll factMod(ll n, ll x){
    if(!INIT){
        FM[0] = 1 % x;
        for (int i = 1; i < N; i++)
            FM[i] = (FM[i-1]*i)%x;
        INIT = 1;
    }
    return FM[n];
}
ll powerMod(ll x, ll y, ll p){
    ll res = 1 % p;
    x = x % p;
    while(y > 0){
        if (y & 1)
            res = (res * x) % p;
        y = y >> 1;
        x = (x * x) % p;
    }
    return res;
}
ll invMod(ll a, ll x){
    return powerMod(a, x - 2, x);
}
ll nCrMod(ll n, ll r, ll x){
    if (r == 0) return 1;
    if (r > n) return 0;
    ll res = factMod(n, x);
    ll fr = factMod(r, x);
    ll zr = factMod(n - r, x);
    res = (res*invMod((fr*zr)%x, x))%x;

    return res;
}
```

String Hash:

```
// invb = inverse mod of base (mod md)
// init() for forward hash
// init_reverse() for reverse hash
// get_hash(l, r) - l <= r (forward hash) | r > l
(reverse hash)
// 1-based indexing
// (1000000007, 31, 129032259)
// (1000000009, 29, 517241384)
struct hash_st {
    string s;
    vector<ll> h, inv, hrev;
    ll md, base, invb;
    hash_st(string s, int md, int base, int invb){
        this->md = md;
        this->s = s;
        this->base = base;
        this->invb = invb;
    }
    void init(){
        h.pb(0);
        inv.pb(1);
        ll pw = 1;
        for(int i = 0; i < s.size(); i++){
            ll z = (h.back() + pw*(s[i]-'a'+1))%md;
            h.pb(z);
            pw = (pw*base)%md;
            inv.pb((inv.back()*invb)%md);
        }
    }
    void init_reverse(){
        string srev = s;
        reverse(srev.begin(), srev.end());
        hrev.pb(0);
        ll pw = 1;
        for(int i = 0; i<srev.size(); i++){
            ll z = (hrev.back() +
pw*(srev[i]-'a'+1))%md;
            hrev.pb(z);
            pw = (pw*base)%md;
        }
    }

    int get_hash(int l, int r){
        if(l<=r){
            return
(((h[r]-h[l-1]+md)%md)*inv[l-1])%md;
        }
        else {
            l = s.size()-l+1;
            r = s.size()-r+1;
            return
(((hrev[r]-hrev[l-1]+md)%md)*inv[l-1])%md;
        }
    }
};
```

Big Integer:

```
string add(string x, string y){
    string res = "";
    int p1 = x.size()-1, p2 = y.size()-1;
    int carry = 0;
    while(p1 >= 0 || p2 >= 0){
        int d1 = (p1 >= 0) ? x[p1]-'0' : 0;
        int d2 = (p2 >= 0) ? y[p2]-'0' : 0;
        int d = (d1+d2+carry);
        carry = d/10;
        d %= 10;
        res += ('0'+d);
        p1--, p2--;
    }
    if(carry) res += ('0'+carry);
    while(res.size() > 1 && res.back() == '0')
res.pop_back();
    reverse(res.begin(), res.end());
    return res;
}
string multiply(string x, string y){
    string res = "0";
    for(int i = x.size()-1; i >= 0; i--){
        string r = "";
        for(int k = x.size()-1; k > i; k--)
            r += '0';
        int d1 = x[i]-'0';
        int carry = 0;
        for(int j = y.size()-1; j>=0; j--){
            int d2 = y[j]-'0';
            int d = d1*d2+carry;
            carry = d/10;
            d %= 10;
            r += ('0'+d);
        }
        if(carry) r += ('0'+carry);
        reverse(r.begin(), r.end());
        res = add(res, r);
    }
    return res;
}
string multiply_string_num(string s, ll n){
    // s is a reversed string
    // returns a reversed string
    string res = "";
    ll carry = 0;
    for(int i = 0; i < s.size(); i++){
        carry += n*(s[i]-'0');
        res += '0'+carry%10;
        carry /= 10;
    }
    while(carry){
        res += '0'+carry%10;
        carry /= 10;
    }
    return res;
}
string num_to_string(ll n){
    string s = "";
    while(n){
        s += '0'+n%10, n /= 10;
    }
    reverse(s.begin(), s.end());
    if(s=="") s = "0";
    return s;
}
string div_by_2(string x){
    int p = 0;
    string res = "";
    int rem = 0;
    while(p < x.size()){
        int num = rem*10+(x[p]-'0');
        if(num < 2){
            if(p+1 < x.size()) num =
num*10+(x[p+1]-'0');
            else {
                res += '0';
                break;
            }
            if(res.size()) res += '0';
            p++;
        }
        res += ('0'+num/2);
        rem = num%2;
        p++;
    }
    return res;
}
```

LCA:

```
vector<int> g[N];
int table[N][25];
int depth[N];

void dfs(int u, int p){
    table[u][0] = p;
    depth[u] = depth[p]+1;
    for(int i = 1; i < 24; i++)
        table[u][i] = table[table[u][i-1]][i-1];

    for(int v: g[u]){
        if(v==p) continue;
        dfs(v, u);
    }
}
int find_lca(int u, int v){
    if(depth[u] > depth[v]) swap(u, v);
    int diff = depth[v]-depth[u];
    for(int i = 0; i < 24; i++){
        if(diff==0) break;
        if(diff & (1<<i)){
            v = table[v][i];
            diff ^= (1<<i);
        }
    }
    if(u==v) return u;
    assert(depth[u]==depth[v]);
    for(int i = 23; i >= 0; i--){
        if(table[u][i]==table[v][i]) continue;
        u = table[u][i];
        v = table[v][i];
    }
    return table[u][0];
}
```

Persistent Segment Tree:

```
int a[N];
int nxt_free = 0;
int versions[N];
int val[N], Left[N], Right[N];
void build(int l, int r, int nd = 0){
    if(l==r){
        val[nd] = a[l];
        return;
    }
    Left[nd] = ++nxt_free;
    Right[nd] = ++nxt_free;
    build(l, (l+r)>>1, Left[nd]);
    build(((l+r)>>1)+1, r, Right[nd]);
    val[nd] = val[Left[nd]] + val[Right[nd]];
}
int update(int l, int r, int indx, int v, int nd){
    if(l>indx || r<indx) return nd;
    int new_nd = ++nxt_free;
    if(l==r){
        val[new_nd] = val[nd] + v;
        return new_nd;
    }
    Left[new_nd] = update(l, (l+r)>>1, indx, v,
Left[nd]);
    Right[new_nd] = update(((l+r)>>1)+1, r, indx, v,
Right[nd]);
    val[new_nd] = val[Left[new_nd]] +
val[Right[new_nd]];
    return new_nd;
}
int query(int l, int r, int i, int j, int nd){
    if(l>j || r<i) return 0;
    if(l>=i && r<=j) return val[nd];
    return query(l, (l+r)>>1, i, j, Left[nd]) +
query(((l+r)>>1)+1, r, i, j, Right[nd]);
}
```

Persistent Segment Tree with Lazy:

```
ll a[N/10];
ll lazy[N];
int flag[N];
int nxt_free = 0;
int versions[N];
ll val[N];
int Left[N], Right[N];
void build(int l, int r, int nd = 0){
    if(l==r){
        val[nd] = a[l];
        return;
    }
    Left[nd] = ++nxt_free;
    Right[nd] = ++nxt_free;
    build(l, (l+r)/2, Left[nd]);
    build((l+r)/2+1, r, Right[nd]);
    val[nd] = val[Left[nd]] + val[Right[nd]];
}
int newLazyKid(int nd, ll delta, int l, int r){
    int new_nd = ++nxt_free;
    Left[new_nd] = Left[nd];
    Right[new_nd] = Right[nd];
    lazy[new_nd] = lazy[nd];
    flag[new_nd] = 1;
    lazy[new_nd] += delta;
    val[new_nd] = val[nd] + (r-l+1)*delta;
    return new_nd;
}
void propagate(int nd, int l, int r){
    if(flag[nd]){
        if(l!=r){
            Left[nd] = newLazyKid(Left[nd], lazy[nd],
l, (l+r)/2);
```

```
           Right[nd] = newLazyKid(Right[nd],
lazy[nd], (l+r)/2+1, r);
       }
       flag[nd] = 0;
       lazy[nd] = 0;
   }
}
int updateRange(int l, int r, int i, int j, ll
delta, int nd){
   if(r < i || l > j) return nd;
   if(i <= l && j >= r) return newLazyKid(nd, delta,
l, r);
   propagate(nd, l, r);
   int new_nd = ++nxt_free;
   Left[new_nd] = updateRange(l, (l+r)/2, i, j,
delta, Left[nd]);
   Right[new_nd] = updateRange((l+r)/2+1, r, i, j,
delta, Right[nd]);
   val[new_nd] = val[Left[new_nd]] +
val[Right[new_nd]];
   return new_nd;
}
ll query(int l, int r, int i, int j, int nd){
   if(r < i || l > j) return 0;
   if(i <= l && j >= r) return val[nd];

   propagate(nd, l, r);

   return query(l, (l+r)/2, i, j, Left[nd]) +
query((l+r)/2+1, r, i, j, Right[nd]);
}
```

Line Segment Intersection:

```
pair<int, pair<double, double>>
pointOfIntersectionOfTwoLines(ll x1, ll y1, ll x2,
ll y2, ll x3, ll y3, ll x4, ll y4){
   // given two points (x1, y1) & (x2, y2) of a line
and two points (x3, y3) & (x4, y4) of another line.
   // if return.first is 1, the lines intersect.
return.second is the point of intersection of two
lines.
   if((y1-y2)*(x3-x4)==(x1-x2)*(y3-y4)) return {0,
{0, 0}};
   double dy1 = y1-y2;
   double dy2 = y3-y4;
   double dx1 = x1-x2;
   double dx2 = x3-x4;
   double z1 = x1*dy1 - y1*dx1;
   double z2 = x3*dy2 - y3*dx2;
   double y = (y1==y2) ? y1 : ((y3==y4) ? y3 :
(z1*dy2 - z2*dy1)/(dx2*dy1 - dx1*dy2));
   double x = (x1==x2) ? x1 : ((x3==x4) ? x3 :
((y1==y2) ? (z2 + y*dx2)/dy2 : (z1 + y*dx1)/dy1));
   return {1, {x, y}};
}
pair<int, pair<double, double>>
pointOfIntersectionOfTwoLineSegments(ll x1, ll y1,
ll x2, ll y2, ll x3, ll y3, ll x4, ll y4){
   // given two end points (x1, y1) & (x2, y2) of a
segment and two end points (x3, y3) & (x4, y4) of
another segment.
   // if return.first is 1, the segments intersect.
return.second is the point of intersection of two
segments.
   if((y1-y2)*(x3-x4)==(x1-x2)*(y3-y4)) return {0,
{0, 0}};
   pair<int, pair<double, double>> p =
pointOfIntersectionOfTwoLines(x1, y1, x2, y2, x3,
y3, x4, y4);
   if(!p.ff) return {0, {0, 0}};
   double x = p.ss.ff;
   double y = p.ss.ss;
   if(((x<=x1 && x>=x2) || (x<=x2 && x>=x1)) &&
((y<=y1 && y>=y2) || (y<=y2 && y>=y1))){
       if(((x<=x3 && x>=x4) || (x<=x4 && x>=x3)) &&
((y<=y3 && y>=y4) || (y<=y4 && y>=y3))){
           return p;
       }
   }
   return {0, {0, 0}};
}
pair<int, pll>
integerPointOfIntersectionOfTwoLineSegments(ll x1,
ll y1, ll x2, ll y2, ll x3, ll y3, ll x4, ll y4){
   // given two end points (x1, y1) & (x2, y2) of a
segment and two end points (x3, y3) & (x4, y4) of
another segment.
   // if return.first is 1, the segments intersect
on a integer cordinate. return.second is the point
of intersection of two segments.
   if((y1-y2)*(x3-x4)==(x1-x2)*(y3-y4)) return {0,
{0, 0}};

   pair<int, pair<double, double>> r =
pointOfIntersectionOfTwoLines(x1, y1, x2, y2, x3,
y3, x4, y4);

   if(!r.ff) return {0, {0, 0}};
   ll x = round(r.ss.ff);
   ll y = round(r.ss.ss);
   if(((x-x1)*(y1-y2)==(y-y1)*(x1-x2)) &&
((x-x3)*(y3-y4)==(y-y3)*(x3-x4))){
       if(((x<=x1 && x>=x2) || (x<=x2 && x>=x1)) &&
((y<=y1 && y>=y2) || (y<=y2 && y>=y1))){
           if(((x<=x3 && x>=x4) || (x<=x4 && x>=x3))
&& ((y<=y3 && y>=y4) || (y<=y4 && y>=y3))){
               return {1, {x, y}};
           }
       }
   }
   return {0, {0, 0}};
}
```

MO's Algorithm:

```
struct query {
   int l, r, ind;
};
vector<query> v;
bool cmp(query a, query b){
   int b1 = a.l/k;
   int b2 = b.l/k;
   if(b1!=b2) return b1 < b2;
   if(b1 & 1) return a.r < b.r;
   return a.r > b.r;
}
void add(int i){}
void remove(int i){}
// main function
for(int i = 0; i < q; i++){
   query z;
   cin >> z.l >> z.r;
   z.l--, z.r--;
   z.ind = i;
   v.pb(z);
}
sort(v.begin(), v.end(), cmp);
int pl = 0, pr = -1;
for(int i = 0; i < q; i++){
   while(pr+1<=v[i].r) add(++pr);
   while(pr>v[i].r) remove(pr--);
   while(pl-1>=v[i].l) add(--pl);
   while(pl<v[i].l) remove(pl++);
   ans[v[i].ind] = res;
}
```

Topological Sort:

```
vector<int> adj[N];
int indegree[N];
vector<int> ans;
void topological_sort(){
  queue<int> q;
  for(int i = 1; i <= n; i++)
      if(indegree[i]==0) q.push(i);
   while(!q.empty()){
      int u = q.front();
      q.pop();
      ans.push_back(u);
      for(int v: adj[u]){
          indegree[v]--;
          if(indegree[v]==0) q.push(v);
      }
  }
}

int n; // number of vertices
vector<vector<int>> adj; // adjacency list of graph
vector<int> visited, ans;
void dfs(int v) {
  visited[v] = true;
  for (int u : adj[v]) {
      if (!visited[u])
          dfs(u);
  }
  ans.push_back(v);
}
void topological_sort() {
  for (int i = 0; i < n; ++i) {
      if (!visited[i])
          dfs(i);
  }
  reverse(ans.begin(), ans.end());
}
```