Template

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;
template <typename T> using Treap = tree <T, null_type, less<T>,
rb_tree_tag, tree_order_statistics_node_update>;
std::mt19937
rng(chrono::steady_clock::now().time_since_epoch().count());
#define Unique(V) sort(all(V)), V.erase(unique(all(V)), V.end())
#define ran(a, b) rng()%((b) - (a) + 1) + (a)
#define endl "\n"
typedef long long LL;
const int mod = 1e9 + 7;
const double pi = acos(-1.0);
const int inf = 1e9;
const LL INF = 1e18;
const int N = 1e5 + 5;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0); return 0;
}
```

Segment Tree - Lazy Propagation

```
template <typename T> class segment_tree {
public:
  vector <T> tree; vector <T> lazy;
  segment_tree(int sz) {
    tree.resize(sz + 5, 0);  lazy.resize(sz + 5, 0);
  }
  void pushdown(int node, int b, int e) {
    if (lazy[node] == 0) return;
    int l = node << 1, r = l | 1, m = (b + e) >> 1;
    tree[l] += (lazy[node] * (m - b + 1));
    tree[r] += (lazy[node] * (e - m));
    lazy[l] += lazy[node]; lazy[r] += lazy[node]; lazy[node] = 0;
  }
  void update(int node, int b, int e, int i, int j, T val) {
    if( i > e || j < b || b > e ) return;
    if (i <= b && j >= e) {
      tree[node] += (val * (e - b + 1));
      lazy[node] += val; return;
    } pushdown(node, b, e);
    int l = node << 1, r = l | 1, m = (b + e) >> 1;
    update(l, b, m, i, j, val); update(r, m + 1, e, i, j, val);
    tree[node] = tree[l] + tree[r];
  }
  T query(int node, int b, int e, int i, int j) {
    if (i > e || j < b || b > e) return 0;
    if (i <= b && j >= e) return tree[node];
    pushdown( node , b , e );
    int l = node << 1, r = l | 1, m = (b + e) >> 1;
    return query(l, b, m, i, j) + query(r, m + 1, e, i, j);
  }
};
```

Segment Tree - String Hashing

```
struct data {
  int l, fhash, rhash;
  data():l(0), fhash(0), rhash(0) {}
  data(int l, int fhash, int rhash):l(l), fhash(fhash), rhash(rhash){}
};
inline data combine(data a, data b) {
```

```
  data ret = data(0, 0, 0); ret.l = a.l + b.l;
  ret.fhash = add(a.fhash, mul(b.fhash, p[a.l], MOD), MOD);
  ret.rhash = add(b.rhash, mul(a.rhash, p[b.l], MOD), MOD);return ret;
}
inline void gen_power() {
  p[0] = 1; for (int i = 1; i < N; i++) p[i] = mul(p[i-1], base, MOD);
}
inline void build(int node, int b, int e) {
  if (b > e) return;
  if (b == e) {
    tree[node] = data(1, a[b], a[b]); return;
  }  int l = node << 1, r = l | 1, m = (b + e) >> 1;
  build(l, b, m); build(r, m + 1, e);
  tree[node] = combine(tree[l], tree[r]);
}
inline void update(int node, int b, int e, int pos, int val) {
  if (b > e || pos > e || pos < b) return;
  if (b == e && b == pos) {
    tree[node] = data(1, val, val); return;
  } int l = node << 1, r = l | 1, m = (b + e) >> 1;
  update(l, b, m, pos, val); update(r, m + 1, e, pos, val);
  tree[node] = combine( tree[l] , tree[r] );
}
inline data query(int node, int b, int e, int i, int j) {
  if (i > e || j < b || b > e) return data(0, 0, 0);
  if (i <= b && j >= e) return tree[node];
  int l = node << 1, r = l | 1, m = (b + e) >> 1;
  return combine(query(l, b, m, i, j), query(r, m + 1, e, i, j));
}
```

Binary Indexed Tree

```
template <typename T> class fenwick_tree {
public:
  int N; vector <T> bit;
  fenwick_tree(int sz) {N = sz; bit.resize(sz + 5, 0);}
  inline T r_query(int l, int r) {
    return p_query(r) - p_query(l - 1);
  }
  inline void p_update(int i, T v) {
    if (i <= 0) return; for (; i <= N; i += (i & -i)) bit[i] += v;
  }
  inline T p_query(int i) {
    T ret = 0; for (; i > 0; i -= (i & -i)) ret += bit[i]; return ret;
  }
  inline void r_update(int i, int j, T v) {
    for (; i <= N; i += (i & -i)) bit[i] += v;
    j++; for (; j <= N; j += (j & -j)) bit[j] -= v;
  }
};
```

Disjoint Set Union

```
template <typename T> class dsu {
public:
  T compo; vector <T> p; vector <T> sz;
  dsu(T mx) {
    compo = mx; p.resize(mx + 5, 0); sz.resize(mx + 5, 0);
  }
  inline void make() {
    for (T i = 1; i <= compo; i++) p[i] = i, sz[i] = 1;
  }
  inline T root(T x) {
    return p[x] == x ? x : p[x] = root(p[x]);
  }
  inline bool same(T x, T y) {return root(x) == root(y);}
```

```
  inline void unite(T x, T y) {
    T u = root(x), v = root(y);    if (u != v) { compo--;
      if (rand()%2) p[u] = v, sz[v] += sz[u];
      else p[v] = u, sz[u] += sz[v];
    }
  }
};
```

Matrix Exponentiation

```
struct matrix_exponentiation {
  static const int INF = 2e9 + 5, MOD = 1e9 + 7;
  Matrix Initialize() {
    Matrix ret; for (int i = 1; i <= 20; i++) {
      for (int j = 1; j <= 20; j++) {ret.M[i][j] = 0;}
    } return ret;
  }
  Matrix Identity(int r, int c) {
    Matrix ret;
    for (int i = 1; i <= r; i++) for (int j = 1; j <= c; j++)
    (i == j ? ret.M[i][j] = 1 : 0); return ret;
  }
  Matrix MatrixSub(Matrix A, Matrix B, int r, int c) {
    Matrix ret; for( int i = 1; i <= r; i++ ) {
      for (int j = 1; j <= c; j++) {
        ret.M[i][j] = (A.M[i][j] - B.M[i][j]);
      }
    } return ret;
  }
 Matrix MatrixMul(Matrix A, Matrix B, int r) {
    Matrix t = Initialize();
    for (int i = 1; i <= r; i++) {
      for (int j = 1; j <= r; j++) {
        for (int k = 1; k <= r; k++) {
          LL c = (A.M[i][k]%MOD * B.M[k][j]%MOD)%MOD;
          t.M[i][j] = (t.M[i][j]%MOD + c%MOD)%MOD;
        }
      }
    } return t;
  }
  Matrix MatrixExpo(Matrix A, LL P, int r) {
    Matrix ret; if (P == 1) return A;
    ret = MatrixExpo(A, P / 2, r);
    if ((P&1)) return MatrixMul(MatrixMul(ret, ret, r), m, r);
    else return MatrixMul(ret, ret, r)
  }
} mat;
```

Treap Implemented

```
template <typename T> class treap {
public:
  struct node { T val, prior, sz; node *l, *r; };
  typedef node* pnode; pnode root;
  treap() { root = NULL; }
  inline int size(pnode t) {return t? t -> sz : 0;}
  int size() { return size(root); }
  void update_size(pnode t) {
    if (t) t -> sz = size(t -> l) + size(t ->r) + 1;
  }
  pnode initialize(T x) {
    pnode ret = (pnode)malloc(sizeof(node));
    ret -> val = x, ret -> prior = rand(), ret -> sz = 1;
    ret -> l = NULL, ret -> r = NULL; return ret;
  }
  void split(pnode t, pnode &l, pnode &r, T key) {
```

```
  if (!t) l = NULL, r = NULL;
  else if (t -> val <= key) split(t -> r, t -> r, r, key), l = t;
  else split(t -> l, l, t -> l, key), r = t;
  update_size(t);
 }
 void merge(pnode &t, pnode l, pnode r) {
  if (!l || !r) t = l ? l : r;
  else if (l -> prior > r -> prior) merge(l -> r, l -> r, r), t = l;
  else merge(r -> l, l, r -> l), t = r;
  update_size(t);
 }
 void insert(pnode it, pnode &t) {
  if (!t) t = it;
  else if (it -> prior > t -> prior) {
   split(t, it -> l, it -> r, it -> val), t = it;
  }
  else insert(it, t -> val <= it -> val ? t -> r : t -> l);
  update_size(t);
 }
void insert(T val) { insert(initialize(val), root); }
 void erase(T key, pnode &t) {
  if (!t) return;
  if (t -> val == key) {
    pnode temp = t; merge(t, t -> l, t -> r); free(temp);
  }
  else erase(key, t -> val < key ? t -> r : t -> l);
  update_size(t);
 }
 void erase(T key) { erase(key, root); }
 int less(T s, pnode t) {
   if (!t) return 0;
   if (t -> val >= s) return less(s, t -> l);
   return size(t -> l) + 1 + less(s, t -> r);
 }
 int less(T s) { return less(s, root); }
 int less_equal(T s, pnode t) {
   if (!t) return 0;
   if (t -> val > s) return less_equal(s, t -> l);
   return size(t -> l) + 1 + less_equal(s, t -> r);
 }
 int less_equal(T s) { return less_equal(s, root); }
 void print(pnode root) {
   if (root == NULL) return; print(root -> l);
   cerr << root -> val << " "; print(root -> r);
 }
 inline void print() { print(root); cout << "\n"; }
};
```

Sieve Prime

```
vector <int> prime; vector <bool> isprime( N , true );
void sieve() {
 isprime[0] = isprime[1] = false; prime.push_back(2);
 for (int i = 4; i <= N; i += 2) isprime[i] = false;  int sq = sqrt(N);
 for (int i = 3; i <= sq; i += 2) {
  if (isprime[i]) {
   for (int j = i * i; j <= N; j += 2 * i) isprime[j] = false;
  }
 }
 for (int i = 3; i <= N; i += 2) if (isprime[i]) prime.push_back(i);
}
```

Segmented Sieve

```
void segmented_sieve( ) {
 if( a > b ) swap( a , b );
```

```
  if ( b < 2 ) return;
  if ( a < 2 ) a = 2;
  LL cap = sqrt( b ) + 1 , start , cnt = 0;
  memset( isp , true , sizeof(isp) );
  for ( int i = 0; i < prime.size(); i++ ) {
    if( prime[i] >= cap ) break;
    if( prime[i] >= a ) start = prime[i] * 2;
    else start = a + ( ( prime[i] - a % prime[i] ) % prime[i] );
    for ( LL j = start; j <= b; j += prime[i] ) {
      isp[ j - a ] = false;
    }
  }
  start = ( a % 2 ) ? a : a + 1;
  If (a == 2) printf("%lld\n",a) , cnt++;
  for (LL i = start; i <= b; i += 2)  if (isp[ i - a ]) printf("%lld\n",i),cnt++;
  printf("Total Prime : %d\n",cnt);
}
```

BigMod

```
LL bigmod(LL b, LL p, LL mod) {
    LL res = 1%mod, x = b%mod;
    while (p) {
      if (p & 1) res = (res * x)%mod; x = (x * x)%mod; p >>= 1;
    } return res%mod;
 }
```

Modular Inverse

```
return BigMod(val, mod - 2, mod);

void modularInverse() { // Mod Inverse O(N)
  Fact[0] = 1LL;
  for (int i = 1; i < 2*N; i++){
    Fact[i] = (Fact[i-1] % MOD * i % MOD)%MOD;
  }
  modInv[N-1] = ModInv(Fact[N-1], MOD);
  for (int i = N-2; i >= 0; i--) {
    modInv[i] = (modInv[i+1] % MOD * (i+1) % MOD)%MOD;
  }
}
```

Shank’s Baby Step Giant Step Discrete Logarithm

```
struct discrete_logarithm_shanks {
  // Need BigMod & gcd function
  LL bsgs(LL a, LL b, LL p) {
    a %= p , b %= p;
    if( b == 1 ) return 0;
    LL cnt = 0 , t = 1;
    for (LL g = gcd(a, p); g != 1; g = gcd(a, p)) {
      if (b%g) return -1;
      p /= g, b /= g, t = t * a / g % p;  ++cnt;
      if (b == t) return cnt;
    }
    map <LL,LL> Hash;
    LL m = LL( sqrt(1.0 * p) + 1 ), base = b;
    for (LL i = 0; i != m; ++i) {
      Hash[ base ] = i; base = base * a % p;
    }
    base = bigmod(a, m, p); LL now = t;
    for (LL i = 1; i <= m + 1; ++i) {
      now = now * base % p;
      if (Hash.count(now)) return (i*m - Hash[now] + cnt);
    }
    return -1;
  }
```

```
} ds;
```

Euler Totient

```
LL PHI(LL x) {
  vector <LL> val;
  LL temp = x, vag = 1;
  for (int i = 0; i < prime.size(); i++) {
    if (x%prime[i] == 0) {
      vag *= prime[i]; val.push_back( prime[i] );
      x /= prime[i];
      while (x%prime[i] == 0) x /= prime[i];
    }
  }
  if (x > 1) val.push_back(x), vag *= x;
  for (int i = 0; i < val.size(); i++) temp *= (val[i] - 1);
  temp /= vag; return temp;
}
```

Sieve PHI

```
void SievePHI(int SZ) {
  for (int i = 1; i <= SZ; i++) PHI[i] = i;
  for (int i = 1; i <= SZ; i++) {
    for (int j = 2*i; j <= SZ; j += i) PHI[j] -= PHI[i];
  }
}
```

KMP Prefix Function

```
vector <int> prefix_function(string &s) {
  int n = (int)s.size();
  vector <int> pi(n);
  for (int i = 1; i < n; i++) {
    int j = pi[i-1];
    while (j > 0 && s[i] != s[j]) j = pi[j-1];
    if (s[i] == s[j]) j++;
    pi[i] = j;
  }
  return pi;
}
```

KMP Find Occurences

```
vector <int> find_occurences(string &text, string &pattern) {
  string cur = pattern + '#' + text;
  int sz1 = text.size(), sz2 = pattern.size();
  vector <int> v; vector <int> lps = prefix_function(cur);
  for (int i = sz2 + 1; i <= sz1 + sz2; i++ ) {
    if( lps[i] == sz2 ) v.push_back(i - 2 * sz2); /// positions
  } return v;
}
```

KMP Checker

```
int KMP(string &text, string &pattern) {
  int lt = text.size() , lp = pattern.size();
  vector <int> pi = prefix_function(pattern);
  int i = 0, j = 0, ret = 0;
  while (i < lt) {
    if (pattern[j] == text[i]) i++, j++;
    if (j == lp) ret++, j = pi[j-1];
    else if (i < lt && pattern[j] != text[i]) {
      if( j != 0 ) j = pi[j-1];
      else i++;
    }
  } return ret;
}
```

Z Function

```
vector <int> z_function(string &s) {
  int n = (int)s.size();
  vector<int> z(n);
  for (int i = 1, l = 0 , r = 0; i < n; i++) {
    if (i <= r) z[i] = min(r - i + 1, z[i-l]);
    while (i + z[i] < n && s[z[i]] == s[i+z[i]]) z[i]++;
    if (i + z[i] - 1 > r) l = i , r = i + z[i] - 1;
  } return z;
}
```

Minimum Expression O(N)

```
int minimum_expression() {
  int n = len << 1, i = 0, j = 1, k = 0, a, b;
  while (i + k < n && j + k < n) {
    a = (i + k >= len) ? s[i + k - len] : s[i + k];
    b = (j + k >= len) ? s[j + k - len] : s[j + k];
    if (a == b) k++;
    else if (a > b ) {i = i + k + 1; if (i <= j) i = j + 1; k = 0;}
    else {j = j + k + 1; if (j <= i) j = i + 1; k = 0;}
  } return (i < j ? i : j);
}
```

MO's Algorithm

```
struct data { // needed query structure
  int l, r, k, idx;
  data(){}
  data (int l, int r, int k, int idx):l(l), r(r), k(k), idx(idx){}
  bool operator <( const data &q ) const {
    int block_a = l / block, block_b = q.l / block;
    if (block_a == block_b) return (r < q.r);
    return ( block_a < block_b );
  }
} Q[ N ]; // according to query numbers

void MO() {
  sort(Q + 1, Q + q + 1);
  for (int i = 1; i <= q; i++) {
    while ( l < Q[i].l) Remove( l++ );  while ( l > Q[i].l) Add( --l );
    while ( r < Q[i].r) Add( ++r ); while ( r > Q[i].r) Remove( r-- );
    ans[Q[i].idx] = get_answer();
  }
}
```

Trie INT

```
template <typename T> class trie_int {
public:
  int mx_bit = -1;
  trie_int(int sz) {mx_bit = sz; root = new node();}
  void del(node* cur) {
    for (int i = 0; i < 26; i++) if (cur -> nxt[i]) del(cur -> nxt[i]);
    delete(cur);
  }
  void insert(T num) {
    node *cur = root;
    for (int i = mx_bit; i >= 0; i--) {
      int id = check(num, i);
      if (cur -> nxt[id] == NULL) cur -> nxt[id] = new node();
      cur = cur -> nxt[id];
    }  cur -> endmark = true;
  }
  bool find(T num) {
    node *cur = root;
    for (int i = mx_bit; i >= 0; i--) {
```

```
      int id = check(num, i);
      if (cur -> nxt[id] == NULL) return false; cur = cur -> nxt[id];
    } return cur -> endmark;
  }
  T maxxor(T num) {
    node *cur = root; T ans = 0;
    for (int i = mx_bit; i >= 0; i--) {
      int id = check(num, i);
      if (cur -> nxt[id^1] != NULL) {
ans = on(ans, i); cur = cur -> nxt[id^1];
      }
      else if (cur -> nxt[id] != NULL) cur = cur -> nxt[id];
    } return ans;
  }
};
```

NCR Iterative

```
void BinoCoeff( ) {
  for (int i = 0; i < 1005; i++) nCr[i][0] = 1;
  for (int i = 1; i < 1005; i++) {
    for (int j = 1; j <= i; j++) nCr[i][j] = (nCr[i-1][j] + nCr[i-1][j-1]);
  }
}
```

Berlecamp Massey

```
struct linear_recurrence_berlekamp_massey {
  static const int SZ = 2e5 + 5, MOD = 1e9 + 7;
  // mod must be a prime
  LL m, a[SZ], h[SZ], t_[SZ], s[SZ], t[SZ];
  // BigMod needed here
  vector <LL> BM( vector <LL> &x ) {
    LL lf , ld; vector <LL> ls , cur;
    for (int i = 0; i < int(x.size()); ++i) {
      LL t = 0;
      for (int j = 0; j < int(cur.size()); ++j) {
        t = (t + x[i-j-1] * cur[j])%MOD;
      }
      if ((t - x[i])%MOD == 0) continue;
      if (!cur.size()) {
        cur.resize(i + 1);
        lf = i; ld = (t - x[i])%MOD; continue;
      }
      LL k = -(x[i] - t) * bigmod(ld, MOD - 2, MOD)%MOD;
      vector <LL> c(i - lf - 1);
      c.push_back(k);
      for (int j = 0; j < int(ls.size()); ++j) {
        c.push_back(-ls[j] * k%MOD);
      }
      if ((int)c.size() < (int)cur.size()) c.resize(cur.size());
      for (int j = 0; j < int(cur.size()); ++j) c[j] = (c[j] + cur[j])%MOD;
      if (i - lf + (int)ls.size() >= (int)cur.size()) {
        ls = cur, lf = i, ld = (t-x[i])%MOD;
      } cur = c;
    }
    for (int i = 0; i < int(cur.size()); ++i) {
      cur[i] = (cur[i]%MOD + MOD)%MOD;
    } return cur;
  }
  void mull( LL *p , LL *q ) {
    for (int i = 0; i < m + m; ++i) t_[i] = 0;
    for (int i = 0; i < m; ++i) if(p[i])
      for (int j = 0; j < m; ++j) t_[i+j] = (t_[i+j] + p[i] * q[j])%MOD;
    for (int i = m + m - 1; i >= m; --i) if(t_[i])
      for( int j = m - 1; ~j; --j) t_[i-j-1] = (t_[i-j-1] + t_[i] * h[j])%MOD;
```

```
    for (int i = 0; i < m; ++i ) p[i] = t_[i];
  }
  LL calc( LL K ){
    for (int i = m; ~i; --i) s[i] = t[i] = 0;
    s[0] = 1; if (m != 1) t[1] = 1; else t[0] = h[0];
    while (K) {
        if (K&1) mull(s, t);
        mull (t , t); K >>= 1;
    }
    LL su = 0;
    for( int i = 0; i < m; ++i ) su = (su + s[i] * a[i])%MOD;
    return (su%MOD + MOD)%MOD;
  }
  /// already calculated upto k , now calculate upto n.
  vector <LL> process( vector <LL> &x , int n , int k ) {
    auto re = BM(x); x.resize(n + 1);
    for (int i = k + 1; i <= n; i++) {
      for (int j = 0; j < re.size(); j++) {
        x[i] += 1LL * x[i - j - 1]%MOD * re[j] % MOD; x[i] %= MOD;
      }
    } return x;
  }
  LL work(vector <LL> &x, LL n) {
    if (n < int(x.size())) return x[n]%MOD;
    vector <LL> v = BM( x ); m = v.size(); if (!m) return 0;
    for (int i = 0; i < m; ++i) h[i] = v[i], a[i] = x[i];
    return calc( n )%MOD;
  }
} rec;
```

Dynamic Segment Tree

```
struct Node {
   Node *l, *r; LL sum, lazy;
   Node() {
    l = NULL, r = NULL, sum = 0, lazy = 0;
   }
   Node(LL b, LL e) {
    l = NULL, r = NULL, sum = 0, lazy = 0;
   }
   void Merge(int b, int e) {
    sum = 0;
    int mid = (b + e) >> 1;
    if (l) sum += l -> sum; if (r) sum += r -> sum;
   }
} *root;

typedef Node* pNode;

void Propagate(pNode &cur, int b, int e) {
  if (cur -> lazy == 0) return;
  int mid = (b + e) >> 1;
  if (!( cur -> l )) cur -> l = new Node();
  if (!( cur -> r )) cur -> r = new Node();
  cur -> l -> sum += (cur -> lazy * (mid - b + 1));
  cur -> r -> sum += (cur -> lazy * (e - mid));
  cur -> l -> lazy += cur -> lazy;
  cur -> r -> lazy += cur -> lazy; cur -> lazy = 0;
}
void Build(pNode &cur, int b, int e) {
  if (b > e) return;
  if (!cur) cur = new Node();
  if (b == e) {
    cur -> sum = a[b]; return;
  }
```

```
  int m = ( b + e ) >> 1;
  Build(cur -> l, b, m); Build(cur -> r, m + 1, e);
  cur -> Merge(b, e);
}
void Update(pNode &cur, int b, int e, int i, int j, int val) {
  if (i > e || j < b || b > e) return;
  if (!cur) cur = new Node();
  if (i <= b && j >= e) {
    cur -> sum += (1LL * val * ( e - b + 1 ));
    cur -> lazy += val; return;
  }
  Propagate(cur, b, e);
  int mid = (b + e) >> 1;
  Update(cur -> l, b, mid, i, j, val);
  Update(cur -> r, mid + 1, e, i, j, val);
  cur -> Merge(b , e);
}
LL Query(pNode &cur, int b, int e, int i, int j) {
  if (i > e || j < b || b > e) return 0;
  if (!cur) return 0;
  if (i <= b && j >= e) return cur -> sum;
  Propagate(cur, b, e);
  int mid = (b + e) >> 1;
  LL x = Query(cur -> l, b, mid, i, j);
  LL y = Query(cur -> r, mid + 1, e, i, j);
  return (x + y);
}
```

Heavy Light Decomposition

```
struct HEAVYLIGHT_DECOMPOSITION {
  static const int N = 5e4 + 5, LOG = 20;
  typedef pair <int,int> ii;
  int n , Node; int cost[ N ];
  int A[ N ], sub[N], par[N], depth[N];
  int chainNo, chainInd[N], chainHead[N], posInTree[N];
  vector <int> graph[ N ];
  int tree[ 4*MAXN ], lazy[ 4*MAXN ]; for Segment Tree

  void Initialize( int _n ) {
    n = _n;
    memset( tree , 0 , sizeof(tree) );
    memset( lazy , 0 , sizeof(lazy) );
    chainNo = 0 , Node = 1;
    for (int i = 0; i < MAXN; i++) {
      graph[i].clear();
      A[i] = 0, sub[i] = 0, cost[i] = 0;
      chainInd[i] = 0, chainHead[i] = -1, posInTree[i] = 0;
    }
  }
  void AddEdge(int u, int v) {
    graph[ u ].push_back( v ); graph[ v ].push_back( u );
  }
  void dfs( int s , int p ) {
    par[s] = p, sub[s] = 1, depth[s] = depth[p] + 1;
    for (int i = 0; i < graph[s].size(); i++) {
      int v = graph[s][i];
      if (v != p) {
        dfs( v , s ); sub[s] += sub[v];
      }
    }
  }
  void HLD(int s, int p) {
    if (chainHead[ chainNo ] == -1) chainHead[chainNo] = s;
    chainInd[ s ] = chainNo;
```

```
  posInTree[ s ] = Node;
  A[ Node++ ] = cost[ s ];
  int mxChild = -1;
  for (int i = 0; i < graph[s].size(); i++) {
   int v = graph[s][i];
   if ( v != p ) {
    if (mxChild == -1 || sub[mxChild] < sub[v]) {
     mxChild = v;
    }
   }
  }
  if (mxChild != -1) HLD( mxChild, s );
  for (int i = 0; i < graph[s].size(); i++) {
   int v = graph[s][i];
   if (v != p && v != mxChild) {
    chainNo++; HLD( v , s );
   }
  }
 }
 int LCA( int u , int v ) {
  while (1) {
   int pu = chainHead[chainInd[u]], pv = chainHead[chainInd[v]];
   if (pu == pv) return (depth[ u ] < depth[ v ] ? u : v);
   if (depth[pu] < depth[pv] ) v = par[ pv ];
   else u = par[ pu ];
  }
 }
 // Segment Tree as necessary
 int QueryUP(int u, int v) {
  if (u == v) return Query(1, 1, Node, posInTree[u], posInTree[u]);
  int uchain , vchain = chainInd[ v ];
  int ans = 0;
  while (true) {
   uchain = chainInd[ u ];
   if (uchain == vchain) {
    int st = posInTree[ v ] , en = posInTree[ u ];
    int ret = Query( 1 , 1 , Node , st , en );
    ans = ans + ret; break;
   }
   int st = posInTree[chainHead[ uchain ]], en = posInTree[u];
   int ret = Query( 1 , 1 , Node , st , en );
   ans = ans + ret;
   u = chainHead[ uchain ] , u = par[ u ];
  }
  return ans;
 }
 void UpdateUP(int u, int v, int w) {
  if (u == v) {
   Update(1, 1, Node, posInTree[ u ], posInTree[ u ], w);
   return;
  }
  int uchain , vchain = chainInd[ v ];
  while (true) {
   uchain = chainInd[ u ];
   if ( uchain == vchain ) {
    int st = posInTree[ v ] , en = posInTree[ u ];
    Update( 1 , 1 , Node , st , en , w ); break;
   }
   int st = posInTree[chainHead[uchain]], en = posInTree[ u ];
   Update( 1 , 1 , Node , st , en , w );
   u = chainHead[ uchain ] , u = par[ u ];
  }
 }
 void UpdateHLD(int u, int v, int w) {
```

```
    int lca = LCA( u , v ); UpdateUP( u , lca , w );
    UpdateUP( v , lca , w ); UpdateUP( lca , lca , -w );
  }
  int QueryHLD(int u, int v) {
    int lca = LCA( u , v );
    int x = QueryUP( u , lca ), y = QueryUP( v , lca ),
    z = QueryUP( lca , lca ); return ( x + y - z );
  }
} hld;
```

Sparse Table MAX, MIN with Index (leftmost)

```
void buildTable( ) {
  for (int i = 1; i <= n; i++) {
    MN[ i ][ 0 ] = a[ i ], MNID[ i ][ 0 ] = i;
    MX[ i ][ 0 ] = a[ i ], MXID[ i ][ 0 ] = i;
  }
  for (int i = 2; i <= n; i++) lg[i] = lg[i/2] + 1;
  for (int j = 1; ( 1 << j ) <= n; j++) {
    for (int i = 1; i + ( 1 << j ) - 1 <= n; i++) {
      if (MN[ i ][ j-1 ] <= MN[ i + ( 1 << (j-1) ) ][ j-1 ]) {
        MN[ i ][ j ] = MN[ i ][ j-1 ]; MNID[ i ][ j ] = MNID[ i ][ j-1 ];
      }
      else {
        MN[ i ][ j ] = MN[ i + ( 1 << (j-1) ) ][ j-1 ];
        MNID[ i ][ j ] = MNID[ i + ( 1 << (j-1) ) ][ j-1 ];
      }
      if (MX[ i ][ j-1 ] >= MX[ i + ( 1 << (j-1) ) ][ j-1 ]) {
        MX[ i ][ j ] = MX[ i ][ j-1 ]; MXID[ i ][ j ] = MXID[ i ][ j-1 ];
      }
      else {
        MX[ i ][ j ] = MX[ i + ( 1 << (j-1) ) ][ j-1 ];
        MXID[ i ][ j ] = MXID[ i + ( 1 << (j-1) ) ][ j-1 ];
      }
    }
  }
}
pair <int,int> Query( int l , int r ) {
  if( l > r ) swap( l , r );
  int k = lg[r - l + 1];
  int mn = min( MN[ l ][ k ] , MN[ r - ( 1 << k ) + 1 ][ k ] );
  int mx = max( MX[ l ][ k ] , MX[ r - ( 1 << k ) + 1 ][ k ] );
  return {mn, mx };
}
pair <int,int> QueryID(int l, int r) {
  if( l > r ) swap( l , r );
  int k = lg[ r-l+1 ], mnid , mxid;
  if (MN[l][k] <= MN[r - (1 << k) + 1][k]) mnid = MNID[l][k];
  else mnid = MNID[ r - (1 << k) + 1 ][ k ];
  if (MX[l][k] >= MX[r - (1 << k) + 1][k]) mxid = MXID[l][k];
  else mxid = MXID[r - (1 << k) + 1][k]; return {mnid, mxid};
}
```

LCA KTH Node, Distance

```
void dfs(int s, int p) {
  parent[s][0] = p; depth[s] = depth[p] + 1;
  for (int i = 0; i < graph[s].size(); i++) {
    pair <int,int> k = graph[s][i];
    int next = k.first , cost = k.second;
    if( next == p ) continue;
    Dist[next] = Dist[s] + cost; dfs( next , s );
  }
}
void Precompute_LCA() {
  for (int i = 1; i <= level; i++) {
```

```
    for (int node = 1; node <= n; node++) {
      if (parent[node][i-1] != -1) {
        parent[node][i] = parent[parent[node][i-1]][i-1];
      }
    }
  }
}
int LCA(int u, int v) {
  if (depth[ v ] < depth[ u ] ) swap(u, v);
  int dif = depth[ v ] - depth[ u ];
  for (int i = 0; i <= level; i++) {
    if ((dif >> i)&1) v = parent[v][i];
  }
  if( u == v ) return u;
  for (int i = level; i >= 0; i--) {
    if (parent[u][i] != parent[v][i]) u = parent[ u ][ i ], v = parent[ v ][ i ];
  } return parent[ u ][ 0 ];
}
int GetDist(int u, int v) {
  int lca = LCA(u, v);
  int ans = Dist[u] + Dist[v] - 2*Dist[lca]; return ans;
}
int JumpToKTH(int u, int k) {
  for (int i = 0; i <= level; i++) if((k>>i)&1) u = parent[u][i]; return u;
}
int GetKthNode(int u, int v, int k) {
  int lca = LCA(u, v);
  int du = depth[u] - depth[lca] + 1;
  if (k <= du) return JumpToKTH(u, k - 1);
  int dv = depth[v] - depth[lca]; k -= du;
  int x = dv - k; return JumpToKTH(v, x);
}
```

Maximum Matching - Hopcroft Carp: O(sqrt(V) * E)

if DIRECTED , then add i to (j + const) & Increase MAXN accordingly in order to maintain bipartiteness.

```
struct MAX_MATCHING_HOPCROFT_KARP {
  static const int MAXN = 1e5 + 5;
  static const int INF = 2e9 + 9;
  int nodes, dist[ 2*MAXN ], match[ 2*MAXN ];
  vector <int> graph[ 2*MAXN ];
  void Initialize(int n) {
    nodes = n;
    for (int i = 0; i < 2*MAXN; i++) graph[i].clear();
  }
  void AddEdge(int u, int v) {
    graph[u].push_back(v), graph[v].push_back(u);
  }
  bool BFS() {
    queue <int> Q; dist[ 0 ] = INF;
    for (int i = 1; i <= nodes; i++) {
      if (match[i]) dist[i] = INF;
      else Q.push( i ), dist[i] = 0;
    }
    while (!Q.empty()) {
      int u = Q.front(); Q.pop(); if (!u) continue;
      for (int i = 0; i < graph[u].size(); i++) {
        int v = graph[u][i];
        if (dist[ match[v] ] != INF) continue;
        dist[ match[v] ] = dist[u] + 1; Q.push( match[v] );
      }
    }
    return (dist[0] != INF);
```

```
  }
  bool DFS(int u) {
   if(u) {
    for (int i = 0; i < graph[u].size(); i++) {
     int v = graph[u][i];
     if (dist[ match[v] ] == dist[u] + 1) {
      if (DFS(match[v])) {
       match[v] = u; match[u] = v; return true;
      }
     }
    }
    dist[u] = INF; return false;
   } return true;
  }
  int MaximumMatch( ) {
   memset(match, 0, sizeof(match)); int cnt = 0;
   while(BFS( )) {
    for (int i = 1; i <= nodes; i++) if(!match[i] && DFS(i)) cnt++;
   } return cnt;
  }
} HK;
```

Maximum Matching - KUHN O(V * E)

If edges are bidirected add mx to change node value where n , m denotes left , right resoectively.

```
struct bipartite_matchingKUHN {
  static const int N = 1e3 + 5;  int n , m , match[ N ];
  bool vis[ N ]; vector <int> graph[ N ];
  void init(int l, int r) {
   n = l, m = r; for (int i = 0; i < N; i++) graph[i].clear();
  }
  void add_edge(int u, int v) { graph[u].push_back(v); }
  bool find_match(int s) {
   vis[s] = true;
   for (int i = 0; i < graph[s].size(); i++) {
    int node = graph[s][i], next = match[node];
    if (vis[next]) continue;
    if (next == 0 || find_match( next )) {
     match[node] = s; return true;
    }
   } return false;
  }
  int maximum_match() {
   int matching = 0; memset( match , 0 , sizeof(match) );
   for (int i = 1; i <= n; i++) {
    memset(vis, false, sizeof(vis)); if (find_match(i)) matching++;
   } return matching;
  }
} kuhn;
```

Strongly Connected Component - Koraraju

```
struct SCC_KOSARAJU {
  static const int MAXN = 1e5 + 5;
  int n , m , component , ID;
  vector <int> graph[ MAXN ]; vector <int> rev_graph[ MAXN ];
  vector <int> compo[ MAXN ]; vector <int> c_graph[ MAXN ];
  bool visited[ MAXN ]; int compoID[ MAXN ];
  int inDegree[ MAXN ], outDegree[ MAXN ]; stack <int> st;

  void initialize(int _n, int _m) {
   n = _n , m = _m; component = 0; ID = 0;
   memset( inDegree , 0 , sizeof( inDegree ) );
   memset( outDegree , 0 , sizeof( outDegree ) );
```

```
  for (int i = 0; i < MAXN; i++) {
   graph[i].clear(), rev_graph[i].clear();
   compo[i].clear(), c_graph[i].clear();
  }
 }
 void AddEdge(int u, int v) {
  graph[u].push_back(v); rev_graph[v].push_back(u);
 }
 void TopoSort(int s) {
  visited[s] = true;
  for (auto x : graph[s]) if(!visited[x]) TopoSort(x); st.push(s);
 }
 void Kosaraju(int s) {
  visited[s] = true;
  compo[component].push_back(s);
  for (auto x:rev_graph[s]) if(!visited[x]) Kosaraju(x);
 }
 void SCCGraph( ) {
  for (int i = 1; i <= component; i++) {
   for (auto x: compo[i]) {
    compoID[x] = i;
   }
  }
  for (int i = 1; i <= n; i++) {
   for (auto x: graph[i]) {
    if (compoID[x] != compoID[i]) {
     inDegree[compoID[x]]++; outDegree[compoID[i]]++;
     c_graph[compoID[i]].push_back(compoID[x]);
    }
   }
  }
 }
 void SCC() {
  memset(visited, false, sizeof(visited));
  for (int i = 1; i <= n; i++) if (!visited[i]) TopoSort(i);
  memset(visited , false , sizeof(visited));
  while (!st.empty()) {
   int tp = st.top(); st.pop();
   if (!visited[tp]) component++, Kosaraju(tp);
  }
 }
} scc;
```

Euler Path & Cycle - Directed Graph

```
vector <int> EulerPathDirected(int startingNode, int totalEdges) {
 vector <int> res; stack <int> S; S.push(startingNode);
 while (!S.empty()) {
  int st = S.top(); S.pop();
  while (curEdge[st] < graph[st].size()) {
   S.push(st); st = graph[st][curEdge[st]++];
  }
  res.push_back(st);
 }
 if ((int)res.size() != totalEdges + 1) return {};
 reverse(res.begin(), res.end());
 return res;
}
vector <int> EulerPathDirected() {
 int startingNode = -1, totalEdges = 0;
 for (int i = 1; i <= n; i++) {
  for (auto x : graph[i]) inDegree[x]++;
  if ((int)graph[i].size() > 0) startingNode = i;
  totalEdges += (int)(graph[i].size());
 } int deficit = 0;
```

```
  for (int i = 1; i <= n; i++) {
    if ((int)graph[i].size() > inDegree[i]) {
      deficit += ((int)graph[i].size() - inDegree[i]);
      startingNode = i;
    }
  }
  if (deficit > 1 || startingNode == -1) return {};
  return EulerPathDirected(startingNode, totalEdges);
}
vector <int> EulerCycleDirected(int startingNode, int totalEdges) {
  memset( curEdge , 0 , sizeof( curEdge ) );
  vector <int> res; stack <int> S;
  S.push(startingNode);
  while (!S.empty()) {
    int st = S.top(); S.pop();
    while (curEdge[st] < graph[st].size()) {
      S.push(st); st = graph[st][curEdge[st]++];
    }
    res.push_back( st );
  }
  if ((int)res.size() != totalEdges + 1) return {};
  reverse(res.begin(), res.end());
  return res;
}
vector <int> EulerCycleDirected() {
  int startingNode = -1, totalEdges = 0;
  for (int i = 1; i <= n; i++) {
    for (auto x: graph[i]) inDegree[x]++;
    if ((int)graph[i].size() > 0) startingNode = i;
    totalEdges += (int)(graph[i].size());
  }
  for (int i = 1; i <= n; i++) {
    if (inDegree[i] != (int)graph[i].size()) return {};
  }
  if (startingNode == -1) return {};
  return EulerCycleDirected(startingNode, totalEdges);
}
```

Euler Path & Cycle - Undirected

for cycle all degree must be even & graph must be connected.

vector < pair <int,int> > graph[N] denoted val , idx

```
vector <int> EulerPathUndirected(int u, int Edges) {
  vector <int> res; stack <int> S; S.push( u );
  while (!S.empty()) {
    u = S.top(); S.pop();
    while (curEdge[u] < (int)(graph[u].size())) {
      pair <int,int> node = graph[u][curEdge[u]++];
      int v = node.first , id = node.second;
      if (!visited[id]) {
        S.push( u ), u = v, visited[id] = true;
      }
    }
    res.push_back(u);
  }
  if ((int)res.size() != Edges + 1) return {};
  reverse(res.begin(), res.end()); return res;
}
vector <int> EulerPathUndirected() {
  int u = -1 , Edges = 0;
  for (int i = 1; i <= n; i++) {
    if (graph[i].size() > 0 && graph[i].size()%2 == 0 && u == -1) u = i;
    if (graph[i].size()%2 == 1) return {};
    Edges += (int)(graph[i].size());
```

```
  }
  return EulerPathUndirected(u, Edges/2);
}
vector <int> EulerCycleUndirected(int u, int Edges) {
  vector <int> res; stack <int> S; S.push( u );
  while (!S.empty()) {
    u = S.top(); S.pop();
    while (curEdge[u] < (int)(graph[u].size())) {
      pair <int,int> node = graph[u][curEdge[u]++];
      int v = node.first, id = node.second;
      if (!visited[id]) {
        S.push(u), u = v, visited[id] = true;
      }
    } res.push_back( u );
  }
  if ((int)res.size() != Edges + 1) return {};
  reverse(res.begin(), res.end()); return res;
}
vector <int> EulerCycleUndirected() {
  int startingNode = -1 , totalEdges = 0;
  for (int i = 1; i <= n; i++) {
    if ((int)graph[i].size() > 0 && (int)graph[i].size()%2 == 0) {
      startingNode = i;
    }
    if ((int)graph[i].size()%2 == 1) return {};
    totalEdges += (int)graph[i].size();
  }
  if( startingNode == -1 ) return {};
  return EulerCycleUndirected(startingNode, totalEdges/2);
}
```

Shortest Path - Dijkstra

```
struct data {
  int node; LL w;
  data(){}
  data(int node, LL w) : node(node), w(w){}
  bool operator <(const data &other) const {
    return w > other.w;
  }
};

template <typename T> class dijkstra {
public:
  int n; static const int N = 1e5 + 5;
  vector <T> cost; vector <data> g[ N ];
  bool relaxed[ N ]; int par[ N ];
  dijkstra(int node) {
    n = node; cost.resize(n + 1, INF);
    for (int i = 0; i <= n; i++) {
      g[i].clear(); relaxed[i] = false; par[i] = -1;
    }
  }
  void addedge(int u, int v, T w) {
    g[u].push_back(data(v, w));
    g[v].push_back(data(u, w));
  }
  vector <T> shortest_path(int s) {
    priority_queue <data> q; q.push(data(s, 0)); cost[s] = 0;
    while (!q.empty()) {
      data cur = q.top(); q.pop(); int pnode = cur.node;
      T pw = cur.w; if (relaxed[pnode]) continue;
      relaxed[pnode] = true;
      for (auto it: g[pnode]) {
        int cnode = it.node; T cw = it.w;
```

```
        if (cost[cnode] == INF || cost[cnode] > cost[pnode] + cw) {
          cost[cnode] = cost[pnode] + cw; par[cnode] = pnode;
          q.push(data(cnode, cost[cnode]));
        }
      }
    } return cost;
  }
};
```

Topological Sort - BFS

```
graph[v].push_back(u); indegree[u]++; //u depends on v
for (int i = 1; i <= n; i++) if (indegree[i] == 0) Q.push(i);

vector<int> ans;
while (!Q.empty()) {
  int to = Q.top(); Q.pop(); ans.push_back(to);
  for (int i = 0; i < graph[to].size(); i++) {
    indegree[graph[to][i]]--;
    if (indegree[graph[to][i]] == 0) Q.push(graph[to][i]);
  }
}
```

0-1 BFS

```
deque <pair <int,int> > Q;
Q.push_front(make_pair(0, 0));
pair <int,int> f = Q.front(); Q.pop_front();
if (c == 0) Q.push_front(make_pair(nx,ny));
else Q.push_back(make_pair(nx,ny));
```

Extended GCD

```
LL ExtGCD(LL a, LL b, LL &x, LL &y) {
  if (b == 0) { x = 1, y = 0; return a; }
  LL x1, y1, gcd = ExtGCD(b, a % b, x1, y1);
  x = y1; y = x1-(a / b)*y1; return gcd;
}
```

Pollard Rho & Miller Rabin

```
struct PRIME_FACTORIZE_POLLARD_RHO { Needs GCD function.
  const long double range = 1e18;
  LL MulMod(LL a, LL b, LL c) {
    LL x = 0, y = a%c;
    while (b > 0) {
      if (b&1) x = (x + y)%c; y = (y << 1)%c; b = b >> 1;
    } return x;
  }
  LL Modulo(LL a, LL b, LL c) {
    LL x = 1 , y = a % c;
    while (b > 0) {
      if (b&1) x = MulMod(x, y, c); y = MulMod(y, y, c); b = b >> 1;
    } return x;
  }
  bool Miller(LL p, int iter) {
    if (p < 2) return false;
    if(p == 2) return true; if(!(p&1)) return false;
    LL s = p - 1, a, temp, mod;  while (!(s&1)) s = s >> 1;
    for (int i = 0; i < iter; i++) {
      a = rand()%(p - 1) + 1; temp = s;
      mod = Modulo(a, temp, p);
      while (temp != ( p - 1) && mod != 1 && mod != (p - 1)) {
        mod = MulMod(mod, mod, p); temp = temp << 1;
      }
      if (mod != (p - 1) && !(temp&1)) return false;
    } return true;
  }
```

```
 LL Rand() {
   long double pseudo = (long double)rand()/(long
double)RAND_MAX;
   return (long long)(round((long double)range * pseudo )) + 1;
 }
 LL Calc(LL x, LL n, LL c) {return (MulMod(x, x, n) + c )%n;}
 LL Pollard_Rho(LL n) {
   LL d = 1, i = 1, k = 1, x = 2, y = x, c;
   do {c = Rand()%n;} while (( c == 0) || (c + 2)%n == 0);
   while (d != n) {
     if (i == k) k *= 2LL, y = x, i = 0; x = Calc( x , n , c );
     i++; d = GCD( abs( y - x ) , n ); if( d != 1 ) return d;
   }
 }
 vector <LL> GetPrimeFactors(LL n) {
   vector <LL> ret;
   if( n == 1 ) return ret;
   if( Miller( n , 5 ) ) { ret.push_back( n ); return ret; }
   LL d = Pollard_Rho( n ); ret = GetPrimeFactors( d );
   vector <LL> temp = GetPrimeFactors( n/d );
   for( int i = 0; i < temp.size(); i++ ) ret.push_back( temp[i] );
   return ret;
 }
} rho;
```

Factors Using Pollard Rho Prime Factorization

Call genFactors(0 , 1);

V[idx].first = Prime, V[idx].second = Count of Prime.

```
void genFactors(int idx, LL mul) {
 if (idx >= V.size()) {Div.push_back(mul);return;}
 genFactors(idx + 1, mul);
  for (int i = 1; i <= V[idx].second; i++) {
    mul *= V[idx].first; genFactors( idx + 1 , mul );
  }
}
```

FFT Base

```
struct Complex {
 long double real, img; Complex( ) {real = 0.0, img = 0.0;}
 Complex( long double x ) { real = x, img = 0.0;}
 Complex( long double x , long double y ) {real = x, img = y;}
 const void operator += ( Complex &q ) {
   real += q.real , img += q.img;
 }
 const void operator -= ( Complex &q ) {
   real -= q.real , img -= q.img;
 }
 const Complex operator + ( Complex &q ) {
   return Complex( real + q.real , img + q.img );
 }
 const Complex operator - ( Complex &q ) {
   return Complex( real - q.real , img - q.img );
 }
 const Complex operator * ( Complex &q ) {
   long double a = ( real * q.real ) - ( img * q.img );
   long double  b = ( real * q.img ) + ( img * q.real );
   return Complex( a , b );
 }
};

struct FAST_FOURIER_TRANSFORM {
 void FFT(vector <Complex> &V, int n, int invert) {
   int i, j, l, len;
```

```
    for (i = 1, j = 0; i < n; i++) {
      for (l = n >> 1; j >= l; l >>= 1) {j -= l;}
      j += l; if (i < j) swap(V[i], V[j]);
    }
    for (len = 2; len <= n; len <<= 1) {
      long double ang = 2 * PI / len * invert;
      Complex wlen(cos(ang), sin(ang));
      for (i = 0; i < n; i += len ) { Complex w(1);
        for ( j = 0; j < len / 2; j++) {
          Complex u = V[ i + j ], Complex v = V[i+j+len/2]*w;
          V[i + j] = (u + v), V[i + j + len / 2] = (u - v); w = w * wlen;
        }
      }
    }
    if (invert == -1) for (i = 0; i < n; i++) V[i].real /= n;
  }
  vector <Complex> Multiply(const vector <Complex> &x, const
vector <Complex> &y) {
    int n = 1; while (n <= (x.size() + y.size())) n <<= 1;
    vector <Complex> A(n), vector <Complex> B(n);
    for( int i = 0; i < x.size(); i++ ) A[i] = x[i];
    for( int i = 0; i < y.size(); i++ ) B[i] = y[i];
    FFT( A , n , 1 ); FFT( B , n , 1 );
    for (int i = 0; i < n; i++) A[i] = (A[i] * B[i]); FFT( A , n , -1 ); return A;
  }
} fft;
```

Hashing

```
struct hashing {
  const int N = 1000005; int tlen;
  const int mod[2] = {1000000007, 1000000009};
  const int base[2] = {43, 37 };
  const int invb[2] = {395348840, 297297300};
  vector <vector <int> > p; vector <vector <int> > invm;
  vector <vector <int> > fh; vector <vector <int> > bh;

  void gen_pow(int sz) {
    p.resize(sz + 2); invm.resize(sz + 2);
    for (int i = 0; i <= sz; i++) p[i].resize(2), invm[i].resize(2);
    p[0][0] = 1, p[0][1] = 1, invm[0][0] = 1, invm[0][1] = 1;
    for (int i = 1; i <= sz; i++) {
      for (int id = 0; id <= 1; id++) {
        p[i][id] = mul(p[i-1][id], base[id], mod[id]);
        invm[i][id] = mul(invm[i-1][id], invb[id], mod[id]);
      }
    }
  }
  void build_hash(string &txt) {
    tlen = txt.size(); fh.resize(tlen + 2); bh.resize(tlen + 2);
    for (int i = 0; i <= tlen; i++) fh[i].resize(2), bh[i].resize(2);
    for (int i = 0 , j = tlen - 1; i < tlen; i++ , j--) {
      for (int id = 0; id <= 1; id++) {
        fh[i][id] = mul(p[i][id], txt[i], mod[id]);
        if (i) fh[i][id] = add(fh[i][id], fh[i-1][id], mod[id]);
        bh[i][id] = mul(p[j][id], txt[i], mod[id]);
        if (i) bh[i][id] = add(bh[i][id], bh[i-1][id], mod[id]);
      }
    }
  }
  LL combine_hash(LL x, LL y) { return ( x << 31 ) | y; }
  LL fhash(string &s) {
    int l = s.size() , x = 0 , y = 0;
    for (int i = 0; i < l; i++) {
      x = add(x, mul( p[i][0], s[i], mod[0]), mod[0]);
```

```
      y = add(y, mul( p[i][1], s[i], mod[1]), mod[1]);
    } return combine_hash( x , y );
  }
  LL bhash(string &s) {
    int l = s.size() , x = 0 , y = 0;
    for (int i = l - 1 , j = 0; i >= 0; i-- , j++) {
      x = add(x, mul( p[j][0], s[i], mod[0]), mod[0]);
      y = add(y, mul( p[j][1], s[i], mod[1]), mod[1]);
    } return combine_hash( x , y );
  }
  LL get_fhash(int l, int r) {
    if (l == 0) return combine_hash(fh[r][0], fh[r][1]);
    int x = sub(fh[r][0], fh[l-1][0], mod[0]); x = mul(x, invm[l][0], mod[0]);
    int y = sub(fh[r][1], fh[l-1][1], mod[1]); y = mul(y, invm[l][1], mod[1]);
    return combine_hash( x , y );
  }
  LL get_bhash(int l, int r) {
    if (l == 0) {
      int x = bh[r][0]; x = mul(x, invm[tlen-r-1][0], mod[0]);
      int y = bh[r][1]; y = mul(y, invm[tlen-r-1][1], mod[1]);
      return combine_hash(x , y );
    }
    int x = sub(bh[r][0], bh[l-1][0], mod[0]);
    x = mul(x, invm[tlen-r-1][0], mod[0]);
    int y = sub(bh[r][1], bh[l-1][1], mod[1]);
    y = mul(y, invm[tlen-r-1][1], mod[1]); return combine_hash(x, y);
  }
} h;
```

Geometry Base

```
struct point {
  int x, y;
  point( ) {}
  point( int x , int y ) : x( x ), y( y ) {}
  void operator = ( const point &p ) { x = p.x, y = p.y; }
  bool operator < (const point &p) {return x == p.x? y < p.y:x < p.x;}
  point operator + (point p) {return point (x + p.x, y + p.y);}
  int cross( const point &p ) const { return x * p.y - y * p.x; }
  int dot( const point &p ) const { return x * p.x + y * p.y; }
  int dist(point q) {return ((x - q.x)*(x - q.x) + (y - q.y)*(y - q.y));}
};

bool comp(point &p1, point &p2) {
  return p1.x != p2.x ? p1.x < p2.x : p1.y < p2.y;
}
bool cw(point &a, point &b, point &c) {
  return (a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y)) < 0;
}
bool ccw( point &a, point &b, point &c ) {
  return (a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y)) > 0;
}
```

Convex Hull - Graham Scan

```
vector <point> convex_hull(vector<point> & ) {
  if( v.size() == 1 ) return v; sort( v.begin(), v.end() );
  point p1 = v[0], p2 = v.back(); vector <point> up , down;
  up.push_back( p1 ) , down.push_back( p1 );
  for (int i = 1; i < v.size(); i++) {
    if (i == v.size() - 1 || cw(p1, v[i], p2)) {
      while (up.size() >= 2 && !cw(up[up.size()-2], up[up.size()-1], v[i]))
        up.pop_back(); up.push_back( v[i] );
    }
    if (i == v.size() - 1 || ccw( p1, v[i], p2)) {
```

```
    while (down.size() >= 2 && !ccw(down[down.size()-2],
down[down.size()-1], v[i]))
      down.pop_back(); down.push_back( v[i] );
    }
  }
  for (int i = down.size() - 2; i > 0; i--) up.push_back(down[i]);
  return up;
}
```

Polygon Area

```
double PolygonArea(vector <Point> poly) {
  double area = 0.0;
  for (int i = 1; i + 1 < poly.size(); i++)
  area += (poly[i].y - poly[0].y) * (poly[i+1].x - poly[i].x) -
  (poly[i].x - poly[0].x) * (poly[i+1].y - poly[i].y);return fabs(area/2.0);
}

int Orientation(Point st, Point mid, Point ed) {
  LL v = ( ed - st ).cross( mid - st );
  if (!v) return 0; //co-linear return v < 0 ? 1 : -1; // acw , cw
}
bool CheckConvex(vector <Point> V) { //check if a polygon is convex
  bool hasPos = false, hasNeg = false;
  for (int i = 0; i < V.size(); i++) {
    int ori = Orientation(V[i], V[(i+1)%n] , V[(i+2)%n]);
    if( ori > 0 ) hasPos = true; if( ori < 0 ) hasNeg = true;
  } return !( hasPos && hasNeg );
}
```

Point Inside Polygon

```
struct lineSegment {
  Point A , B;
  lineSegment() {}
  lineSegment(Point A, Point B) : A(A), B(B) {}
  lineSegment(LL ax, LL ay, LL bx, LL by) {
    Point A = Point(ax, ay), Point B = Point(bx, by);
  }
} L;
bool coLinear(Point a, Point b, Point c) {
  LL ori = Orientation(a, b, c); return ori == 0;
}
bool onSegment(Point p, lineSegment l) {
  if (coLinear(l.A, l.B, p )) {
    Point r = l.A, s = l.B;
    return (1LL * ( p.x - r.x ) * ( p.x - s.x ) <= 0 &&
        1LL * ( p.y - r.y ) * ( p.y - s.y ) <= 0 );
  } else return false;
}
bool lineIntersect(lineSegment p, lineSegment q) {
  if (Orientation(p.A, p.B, q.A) == -Orientation(p.A, p.B, q.B) &&
  Orientation(q.A, q.B, p.A) == -Orientation(q.A, q.B, p.B)) return true;
  else return false;
}
LL RayShoot(vector <lineSegment> V, Point A, Point B) {
  LL cnt = 0; lineSegment Q = lineSegment(A , B);
  for (int i = 0; i < V.size(); i++ ) { // V contains the line segments
    lineSegment P = V[i]; if (onSegment(A , P )) return 1;
    cnt += lineIntersect(P, Q ) ? 1 : 0;
  } return cnt;
}
Make Line Segments from the given polygon, p is the point to check.
if (RayShoot(V, p, Point(p.x+inf, p.y+inf+1))&1) then true
```

Bits Manipulations

```
int  Set( int N , int pos )   { return N |= ( 1 << pos ); }
int  Reset( int N , int pos )  { return N = N & ~( 1 << pos ); }
bool Check( int N , int pos )  { return (bool)( N & ( 1 << pos ) );}
int  Toggle( int N , int pos ) { return ( N ^= ( 1 << pos ) ); }
#define RIGHTMOST      __builtin_ctzll
#define POPCOUNT       __builtin_popcountll
#define LEFTMOST(x)    ( 63-__builtin_clzll( x ) )
```

Treap Builtin

```
if( treap.find( val ) == treap.end() ) // find
treap.erase(treap.find_by_order(treap.order_of_key(val))); // erase
treap.order_of_key(val) // cnt of elem. smaller than val
find - *treap.find_by_order( idx ); //index wise find, if one bases --idx
```

Ternary Search

```
int TernarySearchMAX(int l, int r) {
 int lo = l, hi = r, ret = -, iter = 300;
 while (iter --) {
  int midL = (2 * lo + hi)/3, midR = (lo + 2 * hi)/3;
  int valL = calc(midL), valR = calc(midR);
  if (valL > valR) hi = midR; else lo = midL;
 }
 for (int i = lo; i <= hi; i++) ret = max(ret, calc(i)); return ret;
}
```

LIS (only length)

```
multiset <int> S;
int LIS() { /// strictly Increasing
 for (int i = 1; i <= n; i++) {
  S.insert( a[i] ); auto it = S.lower_bound(a[i]);
  it++; if (it != S.end()) S.erase(it);
 } return (int)S.size();
}
int LIS( ) { /// non-decreasing
 for (int i = 1; i <= n; i++) {
  S.insert( a[i] ); auto it = S.upper_bound(a[i]);
  if (it != S.end()) S.erase(it);
 } return (int)S.size();
}
```

Digit DP Optimized

```
int solve(int idx, bool lead, bool tight) {
 if (idx == -1) return something.
 if (!tight && dp[idx][lead][tight] != -1) return dp[idx][lead][tight];
 int ret = 0, up = tight ? dig[idx] : 9;
 for (int i = 0; i <= up; i++) {
  ret += go(idx - 1, lead&&(i==0) , tight&&(dig[idx]==i));
 }
 if (!tight) dp[idx][lead][tight] = ret; return ret;
}
int len = 0; while (x) dig[len++] = x%10, x/= 10;
return solve(len - 1, 1, 1); /// len-1, lead, tight
```

Matrix Chain Multiplication

```
int MCM(int l, int r) {
 if (l >= r) return 0; if (dp[l][r] != -1) return dp[l][r];
 int ret = inf;
 for (int i = l; i < r; i++) {
  int temp = getsum(l, i) * getsum(i + 1, r);
  ret = min(ret, MCM( l , i ) + MCM(i + 1, r) + temp);
 } return dp[l][r] = ret;
}
```

MAX Flow Dinitz

```
struct MAXFLOW_DINITZ {
 static const int MAXN = 2*105, INF = 2e9 + 5;
 int nodes, src, snk, dist[ MAXN ], start[ MAXN ];
 struct Edge { int u , v , f , c; };
 vector <Edge> E; vector <int> graph[ MAXN ];
 void initialize(int n) {
  nodes = n; E.clear();
  for (int i = 0; i <= nodes; i++) graph[i].clear();
 }
 void AddEdge(int u, int v, int w) {
  Edge p = { u , v , 0 , w }; Edge q = { v , u , 0 , 0 };
  graph[u].push_back((int)(E.size() ) ), E.push_back(p);
  graph[v].push_back((int)(E.size() ) ), E.push_back(q);
 }
 bool BFS() {
  memset(dist, -1, sizeof(dist));
  queue <int> Q; Q.push(src); dist[src] = 0;
  while (!Q.empty()) {
   int u = Q.front(); Q.pop();
   for (int i = 0; i < graph[u].size(); i++) {
    int id = graph[u][i]; int v = E[id].v;
    if( dist[v] != -1 || E[id].f == E[id].c ) continue;
    dist[v] = dist[u] + 1; Q.push( v );
   }
  } return ( dist[snk] != -1 );
 }
 int DFS(int u, int flow) {
  if (u == snk) return flow;
  for ( ;start[u] < graph[u].size(); start[u]++) {
   int id = graph[u][start[u]]; if (E[id].f == E[id].c) continue;
   int v = E[id].v;
   if (dist[v] == dist[u] + 1) {
    int df = DFS(v, min(flow , E[id].c - E[id].f));
    if (df > 0) {
     E[id].f += df; E[id^1].f -= df; return df;
    }
   }
  } return 0;
 }
 int MaxFlow(int _src, int _snk) {
  src = _src , snk = _snk; int res = 0;
  while ( BFS() ) {
   memset( start , 0 , sizeof(start) );
   while( int f = DFS( src , INF ) ) res += f;
  } return res;
 }
} FLOW; // Careful with Source, Sink, Graph construction
```

Mobius Function

```
bool isprime[N]; int prime[N], mobius[N] = {0, 1};
void mobiusCalc( ) {
 memset(isprime, true, sizeof(isprime)); int primecnt = 0;
 for (int i = 2; i < N; i++) {
  if (isprime[i]) {
   prime[ ++primecnt ] = i; mobius[i] = -1;
  }
  for (int j = 1; i * prime[j] < N; j++) {
   isprime[i * prime[j]] = false;
   if (i%prime[j] == 0) {
    mobius[i * prime[j]] = 0; break;
   }
   else mobius[i * prime[j]] -= mobius[i];
  }
 }
```

```
}
```

Chinese Remainder Theorem

```
struct CHINESE_REMAINDER_THEOREM {
 typedef pair <LL,LL> pll;
 LL Normalize(LL x, LL m) {x = x%m; return (x < 0? x+m: x);}
 // Needs Extended GCD, GCD, LCM here
 pll CRT_1(vector <LL> A, vector <LL> M) { // coprime, lcm fits
  if( A.size() != M.size() ) return {-1LL,-1LL};
  LL mul = 1 , ret = 0 , p , q; int l = A.size();
  for (int i = 0; i < l; i++) mul *= M[i];
  for (int i = 0; i < l; i++) {
   ExtGCD( M[i] , mul / M[i] , p , q );
   ret += ( A[i] * q * ( mul / M[i] ) ), ret %= mul;
  } return { (ret < 0 ? ret + mul : ret ) , mul };
 }
 pll CRT_2(vector <LL> A, vector <LL> M) { // non coprime, lcm fits
  if( A.size() != M.size() ) return {-1LL,-1LL};
  int len = A.size(); LL m = M[0] , r = A[0] , p , q , d;
  for (int i = 1; i < len; i++ ) {
   d = ExtGCD( m , M[i] , p , q );
   if( ( A[i] - r ) % d ) return {-1LL,-1LL};
   p = ( A[i] - r ) / d * p % ( M[i] / d );
   r += ( p * m ); m = m / d * M[i]; r %= m;
  } return { ( r < 0 ? r + m : r ) , m };
 }
 pll CRT_3(vector <LL> A, vector <LL> M ) { //non coprime, lcm !fits
  if( A.size() != M.size() ) return {-1LL,-1LL};
  LL ans = A[0] , lcm = M[0] , p , q;
  for (int i = 0; i < A.size(); i++) {
   LL g = ExtGCD( lcm , M[i] , p , q ); LL x1 = p;
   if (( A[i] - ans )%g != 0 ) return {-1LL,-1LL};
   ans = Normalize(ans+x1*(A[i]-ans)/g%( M[i]/g )*lcm, lcm*M[i]/g);
   lcm = LCM( lcm , M[i] );
  } return { (ans < 0 ? ans + lcm : ans ) , lcm };
 }
} crt;
```

SQRT Decomposition

```
inline void Init() {//problem wise sum, max, min wise
 memset(BLOCK, 0, sizeof(BLOCK));
}
inline int Block_ID(int id) {//id index kon block e jabe.
 int pos = ((id + block_size - 1)/block_size);return pos;
}
inline void SetBlock(int id, int v) {
 int pos = Block_ID(id); BLOCK[pos] += v;
}
pair <int,int> StartEndofBlock(int id) {//start-end idx of id'th BLOCK.
 int st = (id - 1) * block_size + 1;
 int en = min(n, (st + block_size - 1)); return {st,en};
}
inline int GetAnd(int l, int r) {
 int block_a = Block_ID(l), block_b = Block_ID(r);
 if (block_a == block_b) {
  int sum = 0;for (int i = l; i <= r; i++)sum += a[i]; return sum;
 } int ret = 0;
 for (int i = l; ; i++) {
  int pos = Block_ID(i); if (pos != block_a) break; ret += a[i];
 }
 for (int i = block_a + 1; i < block_b; i++) ret += BLOCK[i];
 for (int i = block_size*(block_b-1)+1; i <= r; i++) ret += a[i];
 return ret;
```