Contents

1 Compile

1.1 Compile

```
g++-13 nombre.cpp -o nombre (compilar)
./nombre (ejecutar)
```

1.2 Template

```cpp
#include <bits/stdc++.h>
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
using namespace std;
#define pb push_back
#define ll long long
#define s second
#define f first
#define MOD 1000000007
#define INF 100000000000000

void solve(){

}

int main() {
  ios_base::sync_with_stdio(false);  cin.tie(0);  cout.tie(0);
  int t;cin>>t;for(int T=0;T<t;T++)
  solve();
}
```

2 Data Structures

2.1 BIT

```cpp
#define MAXN 10000
int bit[MAXN];
void update(int x, int val){
  for(; x < MAXN; x+=x&-x)
    bit[x] += val;
}
int get(int x){
  int ans = 0;
  for(; x; x-=x&-x)
    ans += bit[x];
  return ans;
}
```

2.2 Bitset

```cpp
bitset<3001> b[3001];

//set() Set the bit value at the given index to 1.
//count() Count the number of set bits.
//any() Checks if any bit is set
//all() Check if all bit is set.
// count the number of set bits in an integer

#pragma GCC target("popcnt")
(int) __builtin_popcount(x);
(int) __builtin_popcountll(x);
__builtin_clz(x); // count leading zeros

// declare bitset
bitset<64> b;

// count set bits in bitser
b.count();
```

2.3 Bit Trie

```cpp
const int K = 2;
struct Vertex {
    int next[K];

    Vertex() {
        fill(begin(next), end(next), -1);
    }
};


//insert
for(int j=30;j>=0;j--) {
    int c = 1&(a[i]>>j);
    if (trie[v].next[c] == -1) {
        trie[v].next[c] = trie.size();
        trie.emplace_back();
        d.pb(-1);
```

```
    }
    v = trie[v].next[c];
}
```

2.4 Disjoint Set Union Bipartite

```
//dsu for checking parity of path length (can be used for checking
    bipartiteness)
void make_set(int v) {
  parent[v] = make_pair(v, 0);
  rank[v] = 0;
  bipartite[v] = true;
}

pair<int, int> find_set(int v) {
  if (v != parent[v].first) {
      int parity = parent[v].second;
      parent[v] = find_set(parent[v].first);
      parent[v].second ^= parity;
  }
  return parent[v];
}

void add_edge(int a, int b) {
  pair<int, int> pa = find_set(a);
  a = pa.first;
  int x = pa.second;

  pair<int, int> pb = find_set(b);
  b = pb.first;
  int y = pb.second;

  if (a == b) {
      if (x == y)
          bipartite[a] = false;
  } else {
      if (rank[a] < rank[b])
          swap (a, b);
      parent[b] = make_pair(a, x^y^1);
      bipartite[a] &= bipartite[b];
      if (rank[a] == rank[b])
          ++rank[a];
  }
}

bool is_bipartite(int v) {
  return bipartite[find_set(v).first];
}
```

2.5 Disjoint Set Union

```
struct DSU {
  vector<int> e;
  vector<pair<int, int>> st;

  DSU(int N) : e(N, -1) {}

  int get(int x) { return e[x] < 0 ? x : e[x] = get(e[x]); }

  bool connected(int a, int b) { return get(a) == get(b); }

  int size(int x) { return -e[get(x)]; }

  bool unite(int x, int y) {
    x = get(x), y = get(y);
    if (x == y) { return false; }
    if (e[x] > e[y]) { swap(x, y); }
    st.push_back({x, e[x]});
    st.push_back({y, e[y]});
    e[x] += e[y];
    e[y] = x;
    return true;
  }

  //skip if no rollback
  int time() {return (int)st.size()};

  void rollback(int t) {
    for (int i = time(); i --> t;)
      e[st[i].first] = st[i].second;
    st.resize(t);
  }
};
```

2.6 Dynamic Conectivity

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;

typedef long long ll;

struct DSU {
  vector<int> e;
  vector<pair<int, int>> st;
  int cnt;

  DSU(){}

  DSU(int N) : e(N, -1), cnt(N) {}

  int get(int x) { return e[x] < 0 ? x : get(e[x]);}

  bool connected(int a, int b) { return get(a) == get(b); }

  int size(int x) { return -e[get(x)]; }

  bool unite(int x, int y) {
    x = get(x), y = get(y);
    if (x == y) { return false; }
    if (e[x] > e[y]) { swap(x, y); }
    st.push_back({x, e[x]});
    st.push_back({y, e[y]});
    e[x] += e[y];
    e[y] = x;
    cnt--;
    return true;
  }

  void rollback(){
    auto [x, y]=st.back();
    st.pop_back();
    e[x] = y;
    auto [a, b]=st.back();
    st.pop_back();
    e[a]=b;
    cnt++;
  }
};

struct query {
  int v, u;
  bool united;
  query(int _v, int _u) : v(_v), u(_u) {}
};

struct QueryTree {
  vector<vector<query>> t;
  DSU dsu;
  int T;

  QueryTree(){}

  QueryTree(int _T, int n) : T(_T) {
    dsu = DSU(n);
    t.resize(4 * T + 4);
  }

  void add(int v, int l, int r, int ul, int ur, query& q) {
    if (ul > ur)
      return;
    if (l == ul && r == ur) {
      t[v].push_back(q);
      return;
    }
    int mid = (l + r) / 2;
    add(2 * v, l, mid, ul, min(ur, mid), q);
    add(2 * v + 1, mid + 1, r, max(ul, mid + 1), ur, q);
  }

  void add_query(query q, int l, int r) {
    add(1, 0, T - 1, l, r, q);
  }

  void dfs(int v, int l, int r, vector<int>& ans) {
    for (query& q : t[v]) {
      q.united = dsu.unite(q.v, q.u);
    }
    if (l == r)
      ans[l] = dsu.cnt;
    else {
      int mid = (l + r) / 2;
      dfs(2 * v, l, mid, ans);
      dfs(2 * v + 1, mid + 1, r, ans);
```

```
    }
    for (query q : t[v]) {
      if (q.united)
        dsu.rollback();
    }
  }
};


int main(){
  ios_base::sync_with_stdio(false); cin.tie(NULL);
  //freopen("connect.in", "r", stdin);
  //freopen("connect.out", "w", stdout);
  int n, k; cin >> n >> k;
  if(k==0) return 0;
  QueryTree st=QueryTree(k, n);
  map<pair<int, int>, int> mp;
  vector<int> ans(k), q;
  for(int i=0;i<k;i++){
    char c; cin >> c;
    if(c=='?'){
      q.push_back(i);
      continue;
    }
    int u, v; cin >> u >> v;
    u--; v--;
    if(u>v) swap(u, v);
    if(c=='+'){
      mp[{u, v}]=i;
    }
    else{
      st.add_query(query(u, v), mp[{u, v}], i);
      mp[{u, v}]=-1;
    }
  }
  for(auto [x, y]:mp){
    if(y!=-1){
      st.add_query(query(x.first, x.second), y, k-1);
    }
  }
  st.dfs(1, 0, k-1, ans);
  for(int x:q){
    cout << ans[x] << endl;
  }
}
```

2.7 Fenwick Tree

```
template <typename T>
struct Fenwick {
    int n;
    std::vector<T> a;

    Fenwick(int n_ = 0) {
        init(n_);
    }

    void init(int n_) {
        n = n_;
        a.assign(n, T{});
    }

    void add(int x, const T &v) {
        for (int i = x + 1; i <= n; i += i & -i) {
            a[i - 1] = a[i - 1] + v;
        }
    }

    T sum(int x) {
        T ans{};
        for (int i = x; i > 0; i -= i & -i) {
            ans = ans + a[i - 1];
        }
        return ans;
    }

    T rangeSum(int l, int r) {
        return sum(r) - sum(l);
    }

    int select(const T &k) {
        int x = 0;
        T cur{};
        for (int i = 1 << std::__lg(n); i; i /= 2) {
            if (x + i <= n && cur + a[x + i - 1] <= k) {
                x += i;
```

```
                cur = cur + a[x - 1];
            }
        }
        return x;
    }
};
```

2.8 Fenwick Tree 2D

```
struct Fenwick2D{
  vector<vector<ll>> b;
  int n;

  Fenwick2D(int _n) : b(_n+5, vector<ll>(_n+5, 0)), n(_n) {}

  void update(int x, int y, int val){
    for(; x<=n; x+=(x&-x)){
      for(int j=y;j<=n;j+=(j&-j)){
        b[x][j]+=val;
      }
    }
  }

  ll get(int x, int y){
    ll ans=0;
    for(; x; x-=x&-x){
      for(int j=y; j ;j-=j&-j){
        ans+=b[x][j];
      }
    }
    return ans;
  }

  ll get1(int x1, int y1, int x2, int y2){
    return get(x2, y2)-get(x1-1, y2)-get(x2, y1-1)+ get(x1-1, y1-1);
  }

};
```

2.9 Merge Sort Tree

```
vector<int> t[200005];
int a[100005];
int n;

void build(){
  for(int i=0;i<n;i++){
    t[i+n].push_back(a[i]);
  }
  for(int i=n-1;i;i--){
    auto b=t[2*i], c=t[2*i+1];
    merge(b.begin(), b.end(), c.begin(), c.end(), back_inserter(t[i]));
  }
}


int q(int l, int r, int mid) {
  int res = 0;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1){
        res+=upper_bound(all(t[l]), mid)-t[l].begin();
        l++;
    }
    if (r&1){
        r--;
        res+=upper_bound(all(t[r]), mid)-t[r].begin();
    }
  }
  return res;
}
```

2.10 Minimum Cartesian Tree

```
struct min_cartesian_tree
{
  vector<int> par;
  vector<vector<int>> sons;
  int root;
  void init(int n, vector<int> &arr)
  {
    par.assign(n, -1);
    sons.assign(n, vector<int>(2, -1));
    stack<int> st;
    for (int i = 0; i < n; i++)
    {
      int last = -1;
      while (!st.empty() && arr[st.top()] < arr[i])
```

```
      {
        last = st.top();
        st.pop();
      }
      if (!st.empty())
      {
        par[i] = st.top();
        sons[st.top()][1] = i;
      }
      if (last != -1)
      {
        par[last] = i;
        sons[i][0] = last;
      }
      st.push(i);
    }
    for (int i = 0; i < n; i++)
    {
      if (par[i] == -1)
      {
        root = i;
      }
    }
  }
};
```

2.11 Multi Ordered Set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T> using oset = __gnu_pbds::tree<
    T, __gnu_pbds::null_type, less<T>, __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update
>;

//en main

oset<pair<int,int>> name;
    map<int,int> cuenta;
    function<void(int)> meter = [&] (int val) {
        name.insert({val,++cuenta[val]});
    };
    auto quitar = [&] (int val) {
        name.erase({val,cuenta[val]--});
    };

meter(x);
quitar(y);
multioset.order_of_key({y+1,-1})-multioset.order_of_key({x,0})
```

2.12 Ordered Set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T> using oset = __gnu_pbds::tree<
    T, __gnu_pbds::null_type, less<T>, __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update
>;
// order_of_key() primero mayor o igual;
// find_by_order() apuntador al elemento k;
// oset<pair<int,int>> os;
// os.insert({1,2});
// os.insert({2,3});
// os.insert({5,6});
// ll k=os.order_of_key({2,0});
// cout<<k<<endl; // 1
// pair<int,int> p=*os.find_by_order(k);
// cout<<p.f<<" "<<p.s<<endl; // 2 3
// os.erase(p);
// p=*os.find_by_order(k);
// cout<<p.f<<" "<<p.s<<endl; // 5 6


// check if upperbound or lowerbound does what you want
// because they give better time.

// to allow repetitions
#define ordered_set tree<int, null_type,less_equal<int>, rb_tree_tag,
    tree_order_statistics_node_update>

// to not allow repetitions
#define ordered_set tree<int, null_type,less<int>, rb_tree_tag,
    tree_order_statistics_node_update>

```

```
//order_of_key(x): number of items are strictly smaller than x

//find_by_order(k) iterator to the kth element
```

2.13 Palindromic Tree

```
const int N = 3e5 + 9;

/*
-> cnt contains the number of palindromic suffixes of the node
*/
struct PalindromicTree {
  struct node {
    int nxt[26], len, st, en, link, cnt, oc;
  };
  string s;
  vector<node> t;
  int sz, last;
  PalindromicTree() {}
  PalindromicTree(string _s) {
    s = _s;
    int n = s.size();
    t.clear();
    t.resize(n + 9);
    sz = 2, last = 2;
    t[1].len = -1, t[1].link = 1;
    t[2].len = 0, t[2].link = 1;
  }
  int extend(int pos) { // returns 1 if it creates a new palindrome
    int cur = last, curlen = 0;
    int ch = s[pos] - 'a';
    while (1) {
      curlen = t[cur].len;
      if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) break;
      cur = t[cur].link;
    }
    if (t[cur].nxt[ch]) {
      last = t[cur].nxt[ch];
      t[last].oc++;
      return 0;
    }
    sz++;
    last = sz;
    t[sz].oc = 1;
    t[sz].len = t[cur].len + 2;
    t[cur].nxt[ch] = sz;
    t[sz].en = pos;
    t[sz].st = pos - t[sz].len + 1;
    if (t[sz].len == 1) {
      t[sz].link = 2;
      t[sz].cnt = 1;
      return 1;
    }
    while (1) {
      cur = t[cur].link;
      curlen = t[cur].len;
      if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) {
        t[sz].link = t[cur].nxt[ch];
        break;
      }
    }
    t[sz].cnt = 1 + t[t[sz].link].cnt;
    return 1;
  }
  void calc_occurrences() {
    for (int i = sz; i >= 3; i--) t[t[i].link].oc += t[i].oc;
  }
} t;

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  string s;
  cin >> s;
  PalindromicTree t(s);
  for (int i = 0; i < s.size(); i++) t.extend(i);
  t.calc_occurrences();
  long long ans = 0; // number of palindromes
  for (int i = 3; i <= t.sz; i++) ans += t.t[i].oc;
  cout << ans << '\n';
  return 0;
}
```

2.14 Persistent Array

```
struct Node {
```

```
  int val;
  Node *l, *r;

  Node(ll x) : val(x), l(nullptr), r(nullptr) {}
  Node(Node *ll, Node *rr) : val(0), l(ll), r(rr) {}
};

int n, a[100001];     // The initial array and its size
Node *roots[100001];  // The persistent array's roots

Node *build(int l = 0, int r = n - 1) {
  if (l == r) return new Node(a[l]);
  int mid = (l + r) / 2;
  return new Node(build(l, mid), build(mid + 1, r));
}

Node *update(Node *node, int val, int pos, int l = 0, int r = n - 1) {
  if (l == r) return new Node(val);
  int mid = (l + r) / 2;
  if (pos > mid)
    return new Node(node->l, update(node->r, val, pos, mid + 1, r));
  else return new Node(update(node->l, val, pos, l, mid), node->r);
}

int query(Node *node, int pos, int l = 0, int r = n - 1) {
  if (l == r) return node->val;
  int mid = (l + r) / 2;
  if (pos > mid) return query(node->r, pos, mid + 1, r);
  return query(node->l, pos, l, mid);
}

int get_item(int index, int time) {
  // Gets the array item at a given index and time
  return query(roots[time], index);
}

void update_item(int index, int value, int prev_time, int curr_time) {
  // Updates the array item at a given index and time
  roots[curr_time] = update(roots[prev_time], index, value);
}

void init_arr(int nn, int *init) {
  // Initializes the persistent array, given an input array
  n = nn;
  for (int i = 0; i < n; i++) a[i] = init[i];
  roots[0] = build();
}
```

2.15 Persistent Segment Tree

```
struct Node {
  ll val;
  Node *l, *r;

  Node(ll x) : val(x), l(nullptr), r(nullptr) {}
  Node(Node *_l, Node *_r) {
    l = _l, r = _r;
    val = 0;
    if (l) val += l->val;
    if (r) val += r->val;
  }
  Node(Node *cp) : val(cp->val), l(cp->l), r(cp->r) {}
};

int n, sz = 1;
ll a[200001];
Node *t[200001];

Node *build(int l = 1, int r = n) {
  if (l == r) return new Node(a[l]);
  int mid = (l + r) / 2;
  return new Node(build(l, mid), build(mid + 1, r));
}

Node *update(Node *node, int pos, int val, int l = 1, int r = n) {
  if (l == r) return new Node(val);
  int mid = (l + r) / 2;
  if (pos > mid)
    return new Node(node->l, update(node->r, pos, val, mid + 1, r));
  else return new Node(update(node->l, pos, val, l, mid), node->r);
}

ll query(Node *node, int a, int b, int l = 1, int r = n) {
  if (l > b || r < a) return 0;
  if (l >= a && r <= b) return node->val;
  int mid = (l + r) / 2;
```

```
  return query(node->l, a, b, l, mid) + query(node->r, a, b, mid + 1, r)
      ;
}

int main(){
  ios_base::sync_with_stdio(false); cin.tie(NULL);
  int q; cin >> n >> q;
  for(int i=1;i<=n;i++){
    cin >> a[i];
  }
  t[sz++]=build();
  while(q--){
    int ty; cin >> ty;
    if(ty==1){
      int k, pos, x; cin >> k >> pos >> x;
      t[k]=update(t[k], pos, x);
    }
    else if(ty==2){
      int k, l, r; cin >> k >> l >> r;
      cout << query(t[k], l, r) << endl;
    }
    else{
      int k; cin >> k;
      t[sz++]=new Node(t[k]);
    }
  }
}
```

2.16 Segment Tree

```
struct SegmentTree {
  vector<ll> a;
  int n;

  SegmentTree(int _n) : a(2 * _n, 1e18), n(_n) {}

  void update(int pos, ll val) {
    for (a[pos += n] = val; pos > 1; pos >>= 1) {
      a[pos / 2] = min(a[pos], a[pos ^ 1]);
    }
  }

  ll get(int l, int r) {
    ll res = 1e18;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
      if (l & 1) {
        res = min(res, a[l++]);
      }
      if (r & 1) {
        res = min(res, a[--r]);
      }
    }
    return res;
  }
};
```

2.17 Segment Tree 2D

```
void build_y(int vx, int lx, int rx, int vy, int ly, int ry) {
    if (ly == ry) {
        if (lx == rx)
            t[vx][vy] = a[lx][ly];
        else
            t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
    } else {
        int my = (ly + ry) / 2;
        build_y(vx, lx, rx, vy*2, ly, my);
        build_y(vx, lx, rx, vy*2+1, my+1, ry);
        t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
    }
}

void build_x(int vx, int lx, int rx) {
    if (lx != rx) {
        int mx = (lx + rx) / 2;
        build_x(vx*2, lx, mx);
        build_x(vx*2+1, mx+1, rx);
    }
    build_y(vx, lx, rx, 1, 0, m-1);
}

int sum_y(int vx, int vy, int tly, int try_, int ly, int ry) {
    if (ly > ry)
        return 0;
    if (ly == tly && try_ == ry)
        return t[vx][vy];
```

```
    int tmy = (tly + try_) / 2;
    return sum_y(vx, vy*2, tly, tmy, ly, min(ry, tmy))
         + sum_y(vx, vy*2+1, tmy+1, try_, max(ly, tmy+1), ry);
}

int sum_x(int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {
    if (lx > rx)
        return 0;
    if (lx == tlx && trx == rx)
        return sum_y(vx, 1, 0, m-1, ly, ry);
    int tmx = (tlx + trx) / 2;
    return sum_x(vx*2, tlx, tmx, lx, min(rx, tmx), ly, ry)
         + sum_x(vx*2+1, tmx+1, trx, max(lx, tmx+1), rx, ly, ry);
}


void update_y(int vx, int lx, int rx, int vy, int ly, int ry, int x, int
     y, int new_val) {
    if (ly == ry) {
        if (lx == rx)
            t[vx][vy] = new_val;
        else
            t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
    } else {
        int my = (ly + ry) / 2;
        if (y <= my)
            update_y(vx, lx, rx, vy*2, ly, my, x, y, new_val);
        else
            update_y(vx, lx, rx, vy*2+1, my+1, ry, x, y, new_val);
        t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
    }
}

void update_x(int vx, int lx, int rx, int x, int y, int new_val) {
    if (lx != rx) {
        int mx = (lx + rx) / 2;
        if (x <= mx)
            update_x(vx*2, lx, mx, x, y, new_val);
        else
            update_x(vx*2+1, mx+1, rx, x, y, new_val);
    }
    update_y(vx, lx, rx, 1, 0, m-1, x, y, new_val);
}
```

2.18 Segment Tree Dynamic

```
struct Vertex {
    int left, right;
    int sum = 0;
    Vertex *left_child = nullptr, *right_child = nullptr;

    Vertex(int lb, int rb) {
        left = lb;
        right = rb;
    }

    void extend() {
        if (!left_child && left + 1 < right) {
            int t = (left + right) / 2;
            left_child = new Vertex(left, t);
            right_child = new Vertex(t, right);
        }
    }

    void add(int k, int x) {
        extend();
        sum += x;
        if (left_child) {
            if (k < left_child->right)
                left_child->add(k, x);
            else
                right_child->add(k, x);
        }
    }

    int get_sum(int lq, int rq) {
        if (lq <= left && right <= rq)
            return sum;
        if (max(left, lq) >= min(right, rq))
            return 0;
        extend();
        return left_child->get_sum(lq, rq) + right_child->get_sum(lq, rq
            );
    }
};
```

2.19 Segment Tree Lazy Types

```
struct max_t {
  long long val;
  static const long long null_v = -9223372036854775807LL;

  max_t(): val(0) {}
  max_t(long long v): val(v) {}

  max_t op(max_t& other) {
    return max_t(max(val, other.val));
  }

  max_t lazy_op(max_t& v, int size) {
    return max_t(val + v.val);
  }
};


struct min_t {
  long long val;
  static const long long null_v = 9223372036854775807LL;

  min_t(): val(0) {}
  min_t(long long v): val(v) {}

  min_t op(min_t& other) {
    return min_t(min(val, other.val));
  }

  min_t lazy_op(min_t& v, int size) {
    return min_t(val + v.val);
  }
};


struct sum_t {
  long long val;
  static const long long null_v = 0;

  sum_t(): val(0) {}
  sum_t(long long v): val(v) {}

  sum_t op(sum_t& other) {
    return sum_t(val + other.val);
  }

  sum_t lazy_op(sum_t& v, int size) {
    return sum_t(val + v.val * size);
  }
};
```

2.20 Segment Tree Lazy

```
template <typename num_t>
struct segtree {
  int n, depth;
  vector<num_t> tree, lazy;

  void init(int s, long long* arr) {
    n = s;
    tree = vector<num_t>(4 * s, 0);
    lazy = vector<num_t>(4 * s, 0);
    init(0, 0, n - 1, arr);
  }

  num_t init(int i, int l, int r, long long* arr) {
    if (l == r) return tree[i] = arr[l];

    int mid = (l + r) / 2;
    num_t a = init(2 * i + 1, l, mid, arr),
          b = init(2 * i + 2, mid + 1, r, arr);
    return tree[i] = a.op(b);
  }

  void update(int l, int r, num_t v) {
  if (l > r) return;
    update(0, 0, n - 1, l, r, v);
  }

  num_t update(int i, int tl, int tr, int ql, int qr, num_t v) {
    eval_lazy(i, tl, tr);

  if (tr < ql || qr < tl) return tree[i];
    if (ql <= tl && tr <= qr) {
      lazy[i] = lazy[i].val + v.val;
      eval_lazy(i, tl, tr);
      return tree[i];
```

```
    }

    int mid = (tl + tr) / 2;
    num_t a = update(2 * i + 1, tl, mid, ql, qr, v),
          b = update(2 * i + 2, mid + 1, tr, ql, qr, v);
    return tree[i] = a.op(b);
  }

  num_t query(int l, int r) {
  if (l > r) return num_t::null_v;
    return query(0, 0, n-1, l, r);
  }

  // int get_first(int v, int tl, int tr, int l, int r, int x) {
  //   eval_lazy(0, tl, tr);
  //   if(tl > r || tr < l) return -1;
  //   if(tree[v].val < x) return -1;

  //   if (tl== tr) return tl;

  //   int tm = tl + (tr-tl)/2;
  //   int left = get_first(2*v+1, tl, tm, l, r, x);
  //   if(left != -1) return left;
  //   return get_first(2*v+2, tm+1, tr, l ,r, x);
  // }

  num_t query(int i, int tl, int tr, int ql, int qr) {
    eval_lazy(i, tl, tr);

    if (ql <= tl && tr <= qr) return tree[i];
    if (tr < ql || qr < tl) return num_t::null_v;

    int mid = (tl + tr) / 2;
    num_t a = query(2 * i + 1, tl, mid, ql, qr),
          b = query(2 * i + 2, mid + 1, tr, ql, qr);
    return a.op(b);
  }

  void eval_lazy(int i, int l, int r) {
    tree[i] = tree[i].lazy_op(lazy[i], (r - l + 1));
    if (l != r) {
      lazy[i * 2 + 1] = lazy[i].val + lazy[i * 2 + 1].val;
      lazy[i * 2 + 2] = lazy[i].val + lazy[i * 2 + 2].val;
    }

    lazy[i] = num_t();
  }
};
```

2.21 Segment Tree Lazy Range Set

```

int N, Q;
int a[maxN];

struct node {
  ll val;
  ll lzAdd;
  ll lzSet;
  node(){};
} tree[maxN << 2];

#define lc p << 1
#define rc (p << 1) + 1

inline void pushup(int p) {
  tree[p].val = tree[lc].val + tree[rc].val;
  return;
}

void pushdown(int p, int l, int mid, int r) {
  // lazy: range set
  if (tree[p].lzSet != 0) {
    tree[lc].lzSet = tree[rc].lzSet = tree[p].lzSet;
    tree[lc].val = (mid - l + 1) * tree[p].lzSet;
    tree[rc].val = (r - mid) * tree[p].lzSet;
    tree[lc].lzAdd = tree[rc].lzAdd = 0;
    tree[p].lzSet = 0;
  } else if (tree[p].lzAdd != 0) {  // lazy: range add
    if (tree[lc].lzSet == 0) tree[lc].lzAdd += tree[p].lzAdd;
    else {
      tree[lc].lzSet += tree[p].lzAdd;
      tree[lc].lzAdd = 0;
    }
    if (tree[rc].lzSet == 0) tree[rc].lzAdd += tree[p].lzAdd;
    else {
```

```
      tree[rc].lzSet += tree[p].lzAdd;
      tree[rc].lzAdd = 0;
    }
    tree[lc].val += (mid - l + 1) * tree[p].lzAdd;
    tree[rc].val += (r - mid) * tree[p].lzAdd;
    tree[p].lzAdd = 0;
  }
  return;
}

void build(int p, int l, int r) {
  tree[p].lzAdd = tree[p].lzSet = 0;
  if (l == r) {
    tree[p].val = a[l];
    return;
  }
  int mid = (l + r) >> 1;
  build(lc, l, mid);
  build(rc, mid + 1, r);
  pushup(p);
  return;
}

void add(int p, int l, int r, int a, int b, ll val) {
  if (a > r || b < l) return;
  if (a <= l && r <= b) {
    tree[p].val += (r - l + 1) * val;
    if (tree[p].lzSet == 0) tree[p].lzAdd += val;
    else tree[p].lzSet += val;
    return;
  }
  int mid = (l + r) >> 1;
  pushdown(p, l, mid, r);
  add(lc, l, mid, a, b, val);
  add(rc, mid + 1, r, a, b, val);
  pushup(p);
  return;
}

void set(int p, int l, int r, int a, int b, ll val) {
  if (a > r || b < l) return;
  if (a <= l && r <= b) {
    tree[p].val = (r - l + 1) * val;
    tree[p].lzAdd = 0;
    tree[p].lzSet = val;
    return;
  }
  int mid = (l + r) >> 1;
  pushdown(p, l, mid, r);
  set(lc, l, mid, a, b, val);
  set(rc, mid + 1, r, a, b, val);
  pushup(p);
  return;
}

ll query(int p, int l, int r, int a, int b) {
  if (a > r || b < l) return 0;
  if (a <= l && r <= b) return tree[p].val;
  int mid = (l + r) >> 1;
  pushdown(p, l, mid, r);
  return query(lc, l, mid, a, b) + query(rc, mid + 1, r, a, b);
}
```

2.22 Segment Tree Max Subarray Sum

```
const ll inf=1e18;

struct Node {
  ll maxi, l_max, r_max, sum;

  Node(ll _maxi, ll _l_max, ll _r_max, ll _sum){
    maxi=_maxi;
    l_max=_l_max;
    r_max=_r_max;
    sum=_sum;
  }

  Node operator+(Node b) {
    return {max(max(maxi, b.maxi), r_max + b.l_max),
      max(l_max, sum + b.l_max), max(b.r_max, r_max + b.sum),
      sum + b.sum};
  }

};

struct SegmentTreeMaxSubSum{
```

```
  int n;
  vector<Node> t;

  SegmentTreeMaxSubSum(int _n) : n(_n), t(2 * _n, Node(-inf, -inf, -inf,
       -inf)) {}

  void update(int pos, ll val) {
    t[pos += n] = Node(val, val, val, val);
    for (pos>>=1; pos ; pos >>= 1) {
      t[pos] = t[2*pos]+t[2*pos+1];
    }
  }

  Node query(int l, int r) {
    Node node_l = Node(-inf, -inf, -inf, -inf);
    Node node_r = Node(-inf, -inf, -inf, -inf);
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
      if (l & 1) {
        node_l=node_l+t[l++];
      }
      if (r & 1) {
        node_r=t[--r]+node_r;
      }
    }
    return node_l+node_r;
  }
};
```

2.23 Segment Tree Range Update

```
struct SegmentTree {
  vector<ll> a;
  int n;

  SegmentTree(int _n) : a(2 * _n, 1e18), n(_n) {}


  ll get(int pos) {
    ll res = 1e18;
    for (pos += n; pos; pos >>= 1) {
      res = min(res, a[pos]);
    }
    return res;
  }

  void update(int l, int r, ll val) {
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
      if (l & 1) {
        a[l] = min(a[l], val);
        l++;
      }
      if (r & 1) {
        r--;
        a[r] = min(a[r], val);
      }
    }
  }
};
```

2.24 Segment Tree Struct Types

```
struct sum_t{
  ll val;
  static const long long null_v = 0;

  sum_t(): val(null_v) {}
  sum_t(long long v): val(v) {}

  sum_t operator + (const sum_t &a) const {
    sum_t ans;
    ans.val = val + a.val;
    return ans;
  }
};
// agregar max subarray sum
```

2.25 Segment Tree Struct

```
// works as a 0-indexed segtree (not lazy)
template <typename num_t>
struct segtree
{
  int n, k;
  vector<num_t> tree;

  void init(int s, vector<ll> arr)
  {
```

```
    n = s;
    k = 0;
    while ((1 << k) < n)
      k++;
    tree = vector<num_t>(2 * (1 << k) + 1);
    for (int i = 0; i < n; i++)
    {
      tree[(1 << k) + i] = arr[i];
    }
    for (int i = (1 << k) - 1; i > 0; i--)
    {
      tree[i] = tree[i * 2] + tree[i * 2 + 1];
    }
  }

  void update(int a, ll b)
  {
    a += (1 << k);
    tree[a] = b;
    for (a /= 2; a >= 1; a /= 2)
    {
      tree[a] = tree[a * 2] + tree[a * 2 + 1];
    }
  }
  num_t find(int a, int b)
  {
    a += (1 << k);
    b += (1 << k);
    num_t s;
    while (a <= b)
    {
      if (a % 2 == 1)
        s = s + tree[a++];
      if (b % 2 == 0)
        s = s + tree[b--];
      a /= 2;
      b /= 2;
    }
    return s;
  }
};
```

2.26 Segment Tree Walk

```
struct SegmentTreeWalk {
  vector<ll> a, final_pos;
  int n;

  SegmentTreeWalk(int _n) : a(4 * _n, 1e18), final_pos(_n), n(_n) {}

  // l = 0, r = n - 1
  void build(int l, int r, int node, const vector<ll> &vals) {
    if (l == r){
      final_pos[l] = node;
      a[node] = vals[l];
    }
    else {
      int mid = (l + r) / 2;
      build(l, mid, node * 2, vals);
      build(mid + 1, r, node * 2 + 1, vals);
      a[node] = min(a[node * 2], a[node * 2 + 1]);
    }
  }

  void update(int pos, ll val){
    pos = final_pos[pos];
    a[pos] = val;
    pos /= 2;
    while(pos){
      a[pos] = min(a[2 * pos], a[2 * pos + 1]);
      pos /= 2;
    }
  }

  //inclusive
  ll get(int l, int r, int L, int R, int node) {
    if (L > R)
      return 1e18;
    if (l == L && r == R) {
      return a[node];
    }
    int mid = (l + r) / 2;
    return min(get(l, mid, L, min(R, mid), 2 * node), get(mid + 1, r,
        max(L, mid + 1), R, 2 * node + 1));
  }
```

```

  // l = 0, r = n - 1, L = query start, R = query end
  // you can just do ll if you only care about value and not index or no
      update
  pair<ll, ll> query(int l, int r, int L, int R, int node, int val){
    //cout << l << " " << r << endl;
    if(l > R || r < L) return {-1, 0};
    if(a[node] < val) return {-1, 0};
    if(l == r){
      // depending on what you want to do
      return {a[node], l};
    }

    int mid = (l + r) / 2;
    auto left = query(l, mid, L, R, 2 * node, val);
    if(left.first != -1) return left;
    auto right = query(mid + 1, r, L, R, 2 * node + 1, val);
    return right;
  }
};
```

2.27 Sparse Table

```
const int MAXN=100005, K=30;
int lg[MAXN+1];
int st[K + 1][MAXN];

int mini(int L, int R){
  int i = lg[R - L + 1];
  int minimum = min(st[i][L], st[i][R - (1 << i) + 1]);
  return minimum;
}

int main(){
  lg[1]=0;
  for (int i = 2; i <= MAXN; i++)
    lg[i] = lg[i/2] + 1;
  std::copy(a.begin(), a.end(), st[0]);

  for (int i = 1; i <= K; i++)
    for (int j = 0; j + (1 << i) <= n; j++)
        st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
}
```

2.28 Square Root Decomposition

```

int n, numBlocks;
string s;

struct Block{
  int l, r;
  int sz(){
    return r-l;
  }
};

Block blocks[2*MAXI];
Block newBlocks[2*MAXI];

void rebuildDecomp(){
  string newS=s;
  int k=0;
  for(int i=0;i<numBlocks;i++){
    for(int j=blocks[i].l;j<blocks[i].r;j++){
      newS[k++]=s[j];
    }
  }
  numBlocks=1;
  blocks[0]={0, n};
  s=newS;
}

void cut(int a, int b){
  int pos=0, curBlock=0;
  for(int i=0;i<numBlocks;i++){
    Block B=blocks[i];
    bool containsA = pos < a && pos + B.sz() > a;
    bool containsB = pos < b && pos + B.sz() > b;
    int cutA = B.l + a - pos;
    int cutB = B.l + b - pos;
    if(containsA && containsB){
      newBlocks[curBlock++]={B.l, cutA};
      newBlocks[curBlock++]={cutA, cutB};
      newBlocks[curBlock++]={cutB, B.r};
    }
    else if(containsA){
```

```
      newBlocks[curBlock++]={B.l, cutA};
      newBlocks[curBlock++]={cutA, B.r};
    }
    else if(containsB){
      newBlocks[curBlock++]={B.l, cutB};
      newBlocks[curBlock++]={cutB, B.r};
    }
    else{
      newBlocks[curBlock++]=B;
    }
    pos += B.sz();
  }
  pos=0;
  numBlocks=0;
  for(int i=0;i<curBlock;i++){
    if(pos<a || pos>=b){
      blocks[numBlocks++]=newBlocks[i];
    }
    pos+=newBlocks[i].sz();
  }
  pos=0;
  for(int i=0;i<curBlock;i++){
    if(pos>=a && pos<b){
      blocks[numBlocks++]=newBlocks[i];
    }
    pos+=newBlocks[i].sz();
  }
}

// while doing operations
if(numBlocks>MAXI){
  rebuildDecomp();
}

// rebuild before final ans
rebuildDecomp();
cout << ans << endl;
```

2.29 Treap

```
struct Node {
  Node *l = 0, *r = 0;
  int val, y, c = 1;
  Node(int val) : val(val), y(rand()) {}
  void recalc();
};

int cnt(Node* n) { return n ? n->c : 0; }
void Node::recalc() { c = cnt(l) + cnt(r) + 1; }

template<class F> void each(Node* n, F f) {
  if (n) { each(n->l, f); f(n->val); each(n->r, f); }
}

pair<Node*, Node*> split(Node* n, int k) {
  if (!n) return {};
  if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
    auto pa = split(n->l, k);
    n->l = pa.second;
    n->recalc();
    return {pa.first, n};
  } else {
    auto pa = split(n->r, k - cnt(n->l) - 1); // and just "k"
    n->r = pa.first;
    n->recalc();
    return {n, pa.second};
  }
}

Node* merge(Node* l, Node* r) {
  if (!l) return r;
  if (!r) return l;
  if (l->y > r->y) {
    l->r = merge(l->r, r);
    l->recalc();
    return l;
  } else {
    r->l = merge(l, r->l);
    r->recalc();
    return r;
  }
}

Node* ins(Node* t, Node* n, int pos) {
  auto pa = split(t, pos);
  return merge(merge(pa.first, n), pa.second);
```

```
}

// Example application: move the range [l, r) to index k
void move(Node*& t, int l, int r, int k) {
  Node *a, *b, *c;
  tie(a,b) = split(t, l); tie(b,c) = split(b, r - l);
  if (k <= l) t = merge(ins(a, b, k), c);
  else t = merge(a, ins(c, b, k - r));
}

// Usage
// create treap
// Node* name=nullptr;
// insert element
// name=ins(name, new Node(val), pos);
// Node* x = new Node(val);
// name = ins(name, x, pos);
// merge two treaps (name before x)
// name=merge(name, x);
// split treap (this will split treap in two treaps,
// first with elements [0, pos) and second with elements [pos, n))
// pa will be pair of two treaps
// auto pa = split(name, pos);
// move range [l, r) to index k
// move(name, l, r, k);
// iterate over treap
// each(name, [&](int val) {
//     cout << val << ' ';
// });
```

2.30 Treap 2

```
typedef struct item * pitem;
struct item {
    int prior, value, cnt;
    bool rev;
    pitem l, r;
};

int cnt (pitem it) {
    return it ? it->cnt : 0;
}

void upd_cnt (pitem it) {
    if (it)
        it->cnt = cnt(it->l) + cnt(it->r) + 1;
}

void push (pitem it) {
    if (it && it->rev) {
        it->rev = false;
        swap (it->l, it->r);
        if (it->l)  it->l->rev ^= true;
        if (it->r)  it->r->rev ^= true;
    }
}

void merge (pitem & t, pitem l, pitem r) {
    push (l);
    push (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;
    upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
    if (!t)
        return void( l = r = 0 );
    push (t);
    int cur_key = add + cnt(t->l);
    if (key <= cur_key)
        split (t->l, l, t->l, key, add),  r = t;
    else
        split (t->r, t->r, r, key, add + 1 + cnt(t->l)),  l = t;
    upd_cnt (t);
}

void reverse (pitem t, int l, int r) {
    pitem t1, t2, t3;
    split (t, t1, t2, l);
    split (t2, t2, t3, r-l+1);
    t2->rev ^= true;
```

```
    merge (t, t1, t2);
    merge (t, t, t3);
}

void output (pitem t) {
    if (!t)  return;
    push (t);
    output (t->l);
    printf ("%d␣", t->value);
    output (t->r);
}
```

2.31 Treap With Inversion

```
struct Node {
  Node *l = 0, *r = 0;
  int val, y, c = 1;
  bool rev = 0;
  Node(int val) : val(val), y(rand()) {}
  void recalc();
  void push();
};

int cnt(Node* n) { return n ? n->c : 0; }
void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
void Node::push() {
  if (rev) {
    rev = 0;
    swap(l, r);
    if (l) l->rev ^= 1;
    if (r) r->rev ^= 1;
  }
}

template<class F> void each(Node* n, F f) {
  if (n) { n->push(); each(n->l, f); f(n->val); each(n->r, f); }
}

pair<Node*, Node*> split(Node* n, int k) {
  if (!n) return {};
  n->push();
  if (cnt(n->l) >= k) {
    auto pa = split(n->l, k);
    n->l = pa.second;
    n->recalc();
    return {pa.first, n};
  } else {
    auto pa = split(n->r, k - cnt(n->l) - 1);
    n->r = pa.first;
    n->recalc();
    return {n, pa.second};
  }
}

Node* merge(Node* l, Node* r) {
  if (!l) return r;
  if (!r) return l;
  l->push();
  r->push();
  if (l->y > r->y) {
    l->r = merge(l->r, r);
    l->recalc();
    return l;
  } else {
    r->l = merge(l, r->l);
    r->recalc();
    return r;
  }
}

Node* ins(Node* t, Node* n, int pos) {
  auto pa = split(t, pos);
  return merge(merge(pa.first, n), pa.second);
}

// Example application: reverse the range [l, r]
void reverse(Node*& t, int l, int r) {
  Node *a, *b, *c;
  tie(a,b) = split(t, l);
  tie(b,c) = split(b, r - l + 1);
  b->rev ^= 1;
  t = merge(merge(a, b), c);
}

void move(Node*& t, int l, int r, int k) {
  Node *a, *b, *c;
```

```
73    tie(a,b) = split(t, l);
74    tie(b,c) = split(b, r - l);
75    if (k <= l) t = merge(ins(a, b, k), c);
76    else t = merge(a, ins(c, b, k - r));
77  }
```

3 Dynamic Programming

4 Flow

5 Geometry

6 Graphs

7 Linear Algebra

8 Math

9 More Topics

10 Polynomials

11 Scripts

12 Strings

13 Trees