

Contents

1	Compile	2	5	Geometry	21
1.1	Compile	2	6	Graphs	21
1.2	Template	2	7	Linear Algebra	21
2	Data Structures	2	8	Math	21
2.1	BIT	2	9	More Topics	21
2.2	Bitset	2	10	Polynomials	21
2.3	Bit Trie	2	11	Scripts	21
2.4	Disjoint Set Union Bipartite	3	12	Strings	21
2.5	Disjoint Set Union	3	13	Trees	21
2.6	Dynamic Conectivity	3			
2.7	Fenwick Tree	5			
2.8	Fenwick Tree 2D	6			
2.9	Merge Sort Tree	6			
2.10	Minimum Cartesian Tree	6			
2.11	Multi Ordered Set	7			
2.12	Ordered Set	7			
2.13	Palindromic Tree	8			
2.14	Persistent Array	8			
2.15	Persistent Segment Tree	9			
2.16	Segment Tree	10			
2.17	Segment Tree 2D	10			
2.18	Segment Tree Dynamic	11			
2.19	Segment Tree Lazy Types	11			
2.20	Segment Tree Lazy	12			
2.21	Segment Tree Lazy Range Set	13			
2.22	Segment Tree Max Subarray Sum	14			
2.23	Segment Tree Range Update	15			
2.24	Segment Tree Struct Types	15			
2.25	Segment Tree Struct	15			
2.26	Segment Tree Walk	16			
2.27	Sparse Table	17			
2.28	Square Root Decomposition	17			
2.29	Treap	18			
2.30	Treap 2	19			
2.31	Treap With Inversion	20			
3	Dynamic Programming	21			
4	Flow	21			

1 Compile

1.1 Compile

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

1.2 Template

```
1 #include <bits/stdc++.h>
2 #pragma GCC optimize("O3,unroll-loops")
3 #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
4 using namespace std;
5 #define pb push_back
6 #define ll long long
7 #define s second
8 #define f first
9 #define MOD 1000000007
10 #define INF 1000000000000000
11
12 void solve(){
13
14 }
15
16 int main() {
17     ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0);
18     int t;cin>>t;for(int T=0;T<t;T++)
19         solve();
20 }
```

2 Data Structures

2.1 BIT

```
1 #define MAXN 10000
2 int bit[MAXN];
3 void update(int x, int val){
4     for(; x < MAXN; x+=x&-x)
5         bit[x] += val;
6 }
7 int get(int x){
8     int ans = 0;
9     for(; x; x-=x&-x)
```

```
10     ans += bit[x];
11     return ans;
12 }
```

2.2 Bitset

```
1 bitset<3001> b[3001];
2
3 //set() Set the bit value at the given index to 1.
4 //count() Count the number of set bits.
5 //any() Checks if any bit is set
6 //all() Check if all bit is set.
7 // count the number of set bits in an integer
8
9 #pragma GCC target("popcnt")
10 (int) __builtin_popcount(x);
11 (int) __builtin_popcountll(x);
12 __builtin_clz(x); // count leading zeros
13
14 // declare bitset
15 bitset<64> b;
16
17 // count set bits in bitset
18 b.count();
```

2.3 Bit Trie

```
1 const int K = 2;
2 struct Vertex {
3     int next[K];
4
5     Vertex() {
6         fill(begin(next), end(next), -1);
7     }
8 };
9
10
11 //insert
12 for(int j=30;j>=0;j--) {
13     int c = 1&(a[j]>>j);
14     if (trie[v].next[c] == -1) {
15         trie[v].next[c] = trie.size();
16         trie.emplace_back();
17         d.pb(-1);
```

```

18     }
19     v = trie[v].next[c];
20 }

```

2.4 Disjoint Set Union Bipartite

```

1 //dsu for checking parity of path length (can be used for checking
  bipartiteness)
2 void make_set(int v) {
3     parent[v] = make_pair(v, 0);
4     rank[v] = 0;
5     bipartite[v] = true;
6 }
7
8 pair<int, int> find_set(int v) {
9     if (v != parent[v].first) {
10         int parity = parent[v].second;
11         parent[v] = find_set(parent[v].first);
12         parent[v].second ^= parity;
13     }
14     return parent[v];
15 }
16
17 void add_edge(int a, int b) {
18     pair<int, int> pa = find_set(a);
19     a = pa.first;
20     int x = pa.second;
21
22     pair<int, int> pb = find_set(b);
23     b = pb.first;
24     int y = pb.second;
25
26     if (a == b) {
27         if (x == y)
28             bipartite[a] = false;
29     } else {
30         if (rank[a] < rank[b])
31             swap(a, b);
32         parent[b] = make_pair(a, x^y^1);
33         bipartite[a] ^= bipartite[b];
34         if (rank[a] == rank[b])
35             ++rank[a];
36     }

```

```

37 }
38
39 bool is_bipartite(int v) {
40     return bipartite[find_set(v).first];
41 }

```

2.5 Disjoint Set Union

```

1 struct DSU {
2     vector<int> e;
3     vector<pair<int, int>> st;
4
5     DSU(int N) : e(N, -1) {}
6
7     int get(int x) { return e[x] < 0 ? x : e[x] = get(e[x]); }
8
9     bool connected(int a, int b) { return get(a) == get(b); }
10
11     int size(int x) { return -e[get(x)]; }
12
13     bool unite(int x, int y) {
14         x = get(x), y = get(y);
15         if (x == y) { return false; }
16         if (e[x] > e[y]) { swap(x, y); }
17         st.push_back({x, e[x]});
18         st.push_back({y, e[y]});
19         e[x] += e[y];
20         e[y] = x;
21         return true;
22     }
23
24     //skip if no rollback
25     int time() {return (int)st.size();}
26
27     void rollback(int t) {
28         for (int i = time(); i --> t;)
29             e[st[i].first] = st[i].second;
30         st.resize(t);
31     }
32 };

```

2.6 Dynamic Conectivity

```

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

```

```
2 using namespace std;
3
4 typedef long long ll;
5
6 struct DSU {
7     vector<int> e;
8     vector<pair<int, int>> st;
9     int cnt;
10
11     DSU(){}
12
13     DSU(int N) : e(N, -1), cnt(N) {}
14
15     int get(int x) { return e[x] < 0 ? x : get(e[x]);}
16
17     bool connected(int a, int b) { return get(a) == get(b); }
18
19     int size(int x) { return -e[get(x)]; }
20
21     bool unite(int x, int y) {
22         x = get(x), y = get(y);
23         if (x == y) { return false; }
24         if (e[x] > e[y]) { swap(x, y); }
25         st.push_back({x, e[x]});
26         st.push_back({y, e[y]});
27         e[x] += e[y];
28         e[y] = x;
29         cnt--;
30         return true;
31     }
32
33     void rollback(){
34         auto [x, y]=st.back();
35         st.pop_back();
36         e[x] = y;
37         auto [a, b]=st.back();
38         st.pop_back();
39         e[a]=b;
40         cnt++;
41     }
42 };
43
44 struct query {
```

```
45     int v, u;
46     bool united;
47     query(int _v, int _u) : v(_v), u(_u) {}
48 };
49
50 struct QueryTree {
51     vector<vector<query>> t;
52     DSU dsu;
53     int T;
54
55     QueryTree(){}
56
57     QueryTree(int _T, int n) : T(_T) {
58         dsu = DSU(n);
59         t.resize(4 * T + 4);
60     }
61
62     void add(int v, int l, int r, int ul, int ur, query& q) {
63         if (ul > ur)
64             return;
65         if (l == ul && r == ur) {
66             t[v].push_back(q);
67             return;
68         }
69         int mid = (l + r) / 2;
70         add(2 * v, l, mid, ul, min(ur, mid), q);
71         add(2 * v + 1, mid + 1, r, max(ul, mid + 1), ur, q);
72     }
73
74     void add_query(query q, int l, int r) {
75         add(1, 0, T - 1, l, r, q);
76     }
77
78     void dfs(int v, int l, int r, vector<int>& ans) {
79         for (query& q : t[v]) {
80             q.united = dsu.unite(q.v, q.u);
81         }
82         if (l == r)
83             ans[l] = dsu.cnt;
84         else {
85             int mid = (l + r) / 2;
86             dfs(2 * v, l, mid, ans);
87             dfs(2 * v + 1, mid + 1, r, ans);
```

```

88     }
89     for (query q : t[v]) {
90         if (q.united)
91             dsu.rollback();
92     }
93 }
94 };
95
96
97 int main(){
98     ios_base::sync_with_stdio(false); cin.tie(NULL);
99     //freopen("connect.in", "r", stdin);
100    //freopen("connect.out", "w", stdout);
101    int n, k; cin >> n >> k;
102    if(k==0) return 0;
103    QueryTree st=QueryTree(k, n);
104    map<pair<int, int>, int> mp;
105    vector<int> ans(k), q;
106    for(int i=0;i<k;i++){
107        char c; cin >> c;
108        if(c=='?'){
109            q.push_back(i);
110            continue;
111        }
112        int u, v; cin >> u >> v;
113        u--; v--;
114        if(u>v) swap(u, v);
115        if(c=='+'){
116            mp[{u, v}]=i;
117        }
118        else{
119            st.add_query(query(u, v), mp[{u, v}], i);
120            mp[{u, v}]=-1;
121        }
122    }
123    for(auto [x, y]:mp){
124        if(y!=-1){
125            st.add_query(query(x.first, x.second), y, k-1);
126        }
127    }
128    st.dfs(1, 0, k-1, ans);
129    for(int x:q){
130        cout << ans[x] << endl;

```

```

131     }
132 }

```

2.7 Fenwick Tree

```

1  template <typename T>
2  struct Fenwick {
3      int n;
4      std::vector<T> a;
5
6      Fenwick(int n_ = 0) {
7          init(n_);
8      }
9
10     void init(int n_) {
11         n = n_;
12         a.assign(n, T{});
13     }
14
15     void add(int x, const T &v) {
16         for (int i = x + 1; i <= n; i += i & -i) {
17             a[i - 1] = a[i - 1] + v;
18         }
19     }
20
21     T sum(int x) {
22         T ans{};
23         for (int i = x; i > 0; i -= i & -i) {
24             ans = ans + a[i - 1];
25         }
26         return ans;
27     }
28
29     T rangeSum(int l, int r) {
30         return sum(r) - sum(l);
31     }
32
33     int select(const T &k) {
34         int x = 0;
35         T cur{};
36         for (int i = 1 << std::__lg(n); i; i /= 2) {
37             if (x + i <= n && cur + a[x + i - 1] <= k) {
38                 x += i;

```

```

39         cur = cur + a[x - 1];
40     }
41 }
42 return x;
43 }
44 };

```

2.8 Fenwick Tree 2D

```

1 struct Fenwick2D{
2     vector<vector<ll>> b;
3     int n;
4
5     Fenwick2D(int _n) : b(_n+5, vector<ll>(_n+5, 0)), n(_n) {}
6
7     void update(int x, int y, int val){
8         for(; x<=n; x+=(x&-x)){
9             for(int j=y; j<=n; j+=(j&-j)){
10                 b[x][j]+=val;
11             }
12         }
13     }
14
15     ll get(int x, int y){
16         ll ans=0;
17         for(; x; x-=x&-x){
18             for(int j=y; j; j-=j&-j){
19                 ans+=b[x][j];
20             }
21         }
22         return ans;
23     }
24
25     ll get1(int x1, int y1, int x2, int y2){
26         return get(x2, y2)-get(x1-1, y2)-get(x2, y1-1)+ get(x1-1, y1-1);
27     }
28 };
29

```

2.9 Merge Sort Tree

```

1 vector<int> t[200005];
2 int a[100005];
3 int n;

```

```

4
5 void build(){
6     for(int i=0;i<n;i++){
7         t[i+n].push_back(a[i]);
8     }
9     for(int i=n-1;i;i--){
10         auto b=t[2*i], c=t[2*i+1];
11         merge(b.begin(), b.end(), c.begin(), c.end(), back_inserter(t[i]));
12     }
13 }
14
15
16 int q(int l, int r, int mid) {
17     int res = 0;
18     for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
19         if (l&1){
20             res+=upper_bound(all(t[l]), mid)-t[l].begin();
21             l++;
22         }
23         if (r&1){
24             r--;
25             res+=upper_bound(all(t[r]), mid)-t[r].begin();
26         }
27     }
28     return res;
29 }

```

2.10 Minimum Cartesian Tree

```

1 struct min_cartesian_tree
2 {
3     vector<int> par;
4     vector<vector<int>> sons;
5     int root;
6     void init(int n, vector<int> &arr)
7     {
8         par.assign(n, -1);
9         sons.assign(n, vector<int>(2, -1));
10        stack<int> st;
11        for (int i = 0; i < n; i++)
12        {
13            int last = -1;
14            while (!st.empty() && arr[st.top()] < arr[i])

```

```

15     {
16         last = st.top();
17         st.pop();
18     }
19     if (!st.empty())
20     {
21         par[i] = st.top();
22         sons[st.top()][1] = i;
23     }
24     if (last != -1)
25     {
26         par[last] = i;
27         sons[i][0] = last;
28     }
29     st.push(i);
30 }
31 for (int i = 0; i < n; i++)
32 {
33     if (par[i] == -1)
34     {
35         root = i;
36     }
37 }
38 }
39 };

```

2.11 Multi Ordered Set

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
4 template <typename T> using oset = __gnu_pbds::tree<
5     T, __gnu_pbds::null_type, less<T>, __gnu_pbds::rb_tree_tag,
6     __gnu_pbds::tree_order_statistics_node_update
7 >;
8
9 //en main
10
11 oset<pair<int,int>> name;
12 map<int,int> cuenta;
13 function<void(int)> meter = [&] (int val) {
14     name.insert({val,++cuenta[val]});
15 };

```

```

16     auto quitar = [&] (int val) {
17         name.erase({val,cuenta[val]--});
18     };
19
20 meter(x);
21 quitar(y);
22 multiset.order_of_key({y+1,-1})-multiset.order_of_key({x,0})

```

2.12 Ordered Set

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
4 template <typename T> using oset = __gnu_pbds::tree<
5     T, __gnu_pbds::null_type, less<T>, __gnu_pbds::rb_tree_tag,
6     __gnu_pbds::tree_order_statistics_node_update
7 >;
8 // order_of_key() primero mayor o igual;
9 // find_by_order() apuntador al elemento k;
10 // oset<pair<int,int>> os;
11 // os.insert({1,2});
12 // os.insert({2,3});
13 // os.insert({5,6});
14 // ll k=os.order_of_key({2,0});
15 // cout<<k<<endl; // 1
16 // pair<int,int> p=os.find_by_order(k);
17 // cout<<p.f<<" "<<p.s<<endl; // 2 3
18 // os.erase(p);
19 // p=os.find_by_order(k);
20 // cout<<p.f<<" "<<p.s<<endl; // 5 6
21
22
23 // check if upperbound or lowerbound does what you want
24 // because they give better time.
25
26 // to allow repetitions
27 #define ordered_set tree<int, null_type,less_equal<int>, rb_tree_tag,
28     tree_order_statistics_node_update>
29
30 // to not allow repetitions
31 #define ordered_set tree<int, null_type,less<int>, rb_tree_tag,
32     tree_order_statistics_node_update>

```

```

32 //order_of_key(x): number of items are strictly smaller than x
33
34 //find_by_order(k) iterator to the kth element

```

2.13 Palindromic Tree

```

1  const int N = 3e5 + 9;
2
3  /*
4  -> cnt contains the number of palindromic suffixes of the node
5  */
6  struct PalindromicTree {
7      struct node {
8          int nxt[26], len, st, en, link, cnt, oc;
9      };
10     string s;
11     vector<node> t;
12     int sz, last;
13     PalindromicTree() {}
14     PalindromicTree(string _s) {
15         s = _s;
16         int n = s.size();
17         t.clear();
18         t.resize(n + 9);
19         sz = 2, last = 2;
20         t[1].len = -1, t[1].link = 1;
21         t[2].len = 0, t[2].link = 1;
22     }
23     int extend(int pos) { // returns 1 if it creates a new palindrome
24         int cur = last, curlen = 0;
25         int ch = s[pos] - 'a';
26         while (1) {
27             curlen = t[cur].len;
28             if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) break;
29             cur = t[cur].link;
30         }
31         if (t[cur].nxt[ch]) {
32             last = t[cur].nxt[ch];
33             t[last].oc++;
34             return 0;
35         }
36         sz++;
37         last = sz;

```

```

38         t[sz].oc = 1;
39         t[sz].len = t[cur].len + 2;
40         t[cur].nxt[ch] = sz;
41         t[sz].en = pos;
42         t[sz].st = pos - t[sz].len + 1;
43         if (t[sz].len == 1) {
44             t[sz].link = 2;
45             t[sz].cnt = 1;
46             return 1;
47         }
48         while (1) {
49             cur = t[cur].link;
50             curlen = t[cur].len;
51             if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) {
52                 t[sz].link = t[cur].nxt[ch];
53                 break;
54             }
55         }
56         t[sz].cnt = 1 + t[t[sz].link].cnt;
57         return 1;
58     }
59     void calc_occurrences() {
60         for (int i = sz; i >= 3; i--) t[t[i].link].oc += t[i].oc;
61     }
62 } t;
63
64 int main() {
65     ios_base::sync_with_stdio(0);
66     cin.tie(0);
67     string s;
68     cin >> s;
69     PalindromicTree t(s);
70     for (int i = 0; i < s.size(); i++) t.extend(i);
71     t.calc_occurrences();
72     long long ans = 0; // number of palindromes
73     for (int i = 3; i <= t.sz; i++) ans += t.t[i].oc;
74     cout << ans << '\n';
75     return 0;
76 }

```

2.14 Persistent Array

```

1  struct Node {

```



```

2   int val;
3   Node *l, *r;

4   Node(ll x) : val(x), l(nullptr), r(nullptr) {}
5   Node(Node *ll, Node *rr) : val(0), l(ll), r(rr) {}
6   };
7
8
9   int n, a[100001];    // The initial array and its size
10  Node *roots[100001]; // The persistent array's roots
11
12  Node *build(int l = 0, int r = n - 1) {
13      if (l == r) return new Node(a[l]);
14      int mid = (l + r) / 2;
15      return new Node(build(l, mid), build(mid + 1, r));
16  }
17
18  Node *update(Node *node, int val, int pos, int l = 0, int r = n - 1) {
19      if (l == r) return new Node(val);
20      int mid = (l + r) / 2;
21      if (pos > mid)
22          return new Node(node->l, update(node->r, val, pos, mid + 1, r));
23      else return new Node(update(node->l, val, pos, l, mid), node->r);
24  }
25
26  int query(Node *node, int pos, int l = 0, int r = n - 1) {
27      if (l == r) return node->val;
28      int mid = (l + r) / 2;
29      if (pos > mid) return query(node->r, pos, mid + 1, r);
30      return query(node->l, pos, l, mid);
31  }
32
33  int get_item(int index, int time) {
34      // Gets the array item at a given index and time
35      return query(roots[time], index);
36  }
37
38  void update_item(int index, int value, int prev_time, int curr_time) {
39      // Updates the array item at a given index and time
40      roots[curr_time] = update(roots[prev_time], index, value);
41  }
42
43  void init_arr(int nn, int *init) {
44      // Initializes the persistent array, given an input array

```

```

45   n = nn;
46   for (int i = 0; i < n; i++) a[i] = init[i];
47   roots[0] = build();
48   }

```

2.15 Persistent Segment Tree

```

1   struct Node {
2       ll val;
3       Node *l, *r;

4       Node(ll x) : val(x), l(nullptr), r(nullptr) {}
5       Node(Node *_l, Node *_r) {
6           l = _l, r = _r;
7           val = 0;
8           if (l) val += l->val;
9           if (r) val += r->val;
10      }
11      Node(Node *cp) : val(cp->val), l(cp->l), r(cp->r) {}
12  };
13
14
15  int n, sz = 1;
16  ll a[200001];
17  Node *t[200001];
18
19  Node *build(int l = 1, int r = n) {
20      if (l == r) return new Node(a[l]);
21      int mid = (l + r) / 2;
22      return new Node(build(l, mid), build(mid + 1, r));
23  }
24
25  Node *update(Node *node, int pos, int val, int l = 1, int r = n) {
26      if (l == r) return new Node(val);
27      int mid = (l + r) / 2;
28      if (pos > mid)
29          return new Node(node->l, update(node->r, pos, val, mid + 1, r));
30      else return new Node(update(node->l, pos, val, l, mid), node->r);
31  }
32
33  ll query(Node *node, int a, int b, int l = 1, int r = n) {
34      if (l > b || r < a) return 0;
35      if (l >= a && r <= b) return node->val;
36      int mid = (l + r) / 2;

```

```

37     return query(node->l, a, b, l, mid) + query(node->r, a, b, mid + 1, r)
38     ;
39 }
40 int main(){
41     ios_base::sync_with_stdio(false); cin.tie(NULL);
42     int q; cin >> n >> q;
43     for(int i=1;i<=n;i++){
44         cin >> a[i];
45     }
46     t[sz++]=build();
47     while(q--){
48         int ty; cin >> ty;
49         if(ty==1){
50             int k, pos, x; cin >> k >> pos >> x;
51             t[k]=update(t[k], pos, x);
52         }
53         else if(ty==2){
54             int k, l, r; cin >> k >> l >> r;
55             cout << query(t[k], l, r) << endl;
56         }
57         else{
58             int k; cin >> k;
59             t[sz++]=new Node(t[k]);
60         }
61     }
62 }

```

2.16 Segment Tree

```

1 struct SegmentTree {
2     vector<ll> a;
3     int n;
4
5     SegmentTree(int _n) : a(2 * _n, 1e18), n(_n) {}
6
7     void update(int pos, ll val) {
8         for (a[pos += n] = val; pos > 1; pos >>= 1) {
9             a[pos / 2] = min(a[pos], a[pos ^ 1]);
10        }
11    }
12
13    ll get(int l, int r) {

```

```

14        ll res = 1e18;
15        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
16            if (l & 1) {
17                res = min(res, a[l++]);
18            }
19            if (r & 1) {
20                res = min(res, a[--r]);
21            }
22        }
23        return res;
24    }
25 };

```

2.17 Segment Tree 2D

```

1 void build_y(int vx, int lx, int rx, int vy, int ly, int ry) {
2     if (ly == ry) {
3         if (lx == rx)
4             t[vx][vy] = a[lx][ly];
5         else
6             t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
7     } else {
8         int my = (ly + ry) / 2;
9         build_y(vx, lx, rx, vy*2, ly, my);
10        build_y(vx, lx, rx, vy*2+1, my+1, ry);
11        t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
12    }
13 }
14
15 void build_x(int vx, int lx, int rx) {
16     if (lx != rx) {
17         int mx = (lx + rx) / 2;
18         build_x(vx*2, lx, mx);
19         build_x(vx*2+1, mx+1, rx);
20     }
21     build_y(vx, lx, rx, 1, 0, m-1);
22 }
23
24 int sum_y(int vx, int vy, int tly, int try_, int ly, int ry) {
25     if (ly > ry)
26         return 0;
27     if (ly == tly && try_ == ry)
28         return t[vx][vy];

```

```

29     int tmy = (tly + try_) / 2;
30     return sum_y(vx, vy*2, tly, tmy, ly, min(ry, tmy))
31         + sum_y(vx, vy*2+1, tmy+1, try_, max(ly, tmy+1), ry);
32 }
33
34 int sum_x(int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {
35     if (lx > rx)
36         return 0;
37     if (lx == tlx && trx == rx)
38         return sum_y(vx, 1, 0, m-1, ly, ry);
39     int tmx = (tlx + trx) / 2;
40     return sum_x(vx*2, tlx, tmx, lx, min(rx, tmx), ly, ry)
41         + sum_x(vx*2+1, tmx+1, trx, max(lx, tmx+1), rx, ly, ry);
42 }
43
44
45 void update_y(int vx, int lx, int rx, int vy, int ly, int ry, int x, int
    y, int new_val) {
46     if (ly == ry) {
47         if (lx == rx)
48             t[vx][vy] = new_val;
49         else
50             t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
51     } else {
52         int my = (ly + ry) / 2;
53         if (y <= my)
54             update_y(vx, lx, rx, vy*2, ly, my, x, y, new_val);
55         else
56             update_y(vx, lx, rx, vy*2+1, my+1, ry, x, y, new_val);
57         t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
58     }
59 }
60
61 void update_x(int vx, int lx, int rx, int x, int y, int new_val) {
62     if (lx != rx) {
63         int mx = (lx + rx) / 2;
64         if (x <= mx)
65             update_x(vx*2, lx, mx, x, y, new_val);
66         else
67             update_x(vx*2+1, mx+1, rx, x, y, new_val);
68     }
69     update_y(vx, lx, rx, 1, 0, m-1, x, y, new_val);
70 }

```

2.18 Segment Tree Dynamic

```

1 struct Vertex {
2     int left, right;
3     int sum = 0;
4     Vertex *left_child = nullptr, *right_child = nullptr;
5
6     Vertex(int lb, int rb) {
7         left = lb;
8         right = rb;
9     }
10
11     void extend() {
12         if (!left_child && left + 1 < right) {
13             int t = (left + right) / 2;
14             left_child = new Vertex(left, t);
15             right_child = new Vertex(t, right);
16         }
17     }
18
19     void add(int k, int x) {
20         extend();
21         sum += x;
22         if (left_child) {
23             if (k < left_child->right)
24                 left_child->add(k, x);
25             else
26                 right_child->add(k, x);
27         }
28     }
29
30     int get_sum(int lq, int rq) {
31         if (lq <= left && right <= rq)
32             return sum;
33         if (max(left, lq) >= min(right, rq))
34             return 0;
35         extend();
36         return left_child->get_sum(lq, rq) + right_child->get_sum(lq, rq);
37     }
38 };

```

2.19 Segment Tree Lazy Types

```

1 struct max_t {
2     long long val;
3     static const long long null_v = -9223372036854775807LL;
4
5     max_t(): val(0) {}
6     max_t(long long v): val(v) {}
7
8     max_t op(max_t& other) {
9         return max_t(max(val, other.val));
10    }
11
12    max_t lazy_op(max_t& v, int size) {
13        return max_t(val + v.val);
14    }
15 };
16
17 struct min_t {
18     long long val;
19     static const long long null_v = 9223372036854775807LL;
20
21     min_t(): val(0) {}
22     min_t(long long v): val(v) {}
23
24     min_t op(min_t& other) {
25         return min_t(min(val, other.val));
26     }
27
28     min_t lazy_op(min_t& v, int size) {
29         return min_t(val + v.val);
30     }
31 }
32 };
33
34 struct sum_t {
35     long long val;
36     static const long long null_v = 0;
37
38     sum_t(): val(0) {}
39     sum_t(long long v): val(v) {}
40
41     sum_t op(sum_t& other) {
42         return sum_t(val + other.val);
43     }

```

```

44    }
45
46    sum_t lazy_op(sum_t& v, int size) {
47        return sum_t(val + v.val * size);
48    }
49 };

```

2.20 Segment Tree Lazy

```

1 template <typename num_t>
2 struct segtree {
3     int n, depth;
4     vector<num_t> tree, lazy;
5
6     void init(int s, long long* arr) {
7         n = s;
8         tree = vector<num_t>(4 * s, 0);
9         lazy = vector<num_t>(4 * s, 0);
10        init(0, 0, n - 1, arr);
11    }
12
13    num_t init(int i, int l, int r, long long* arr) {
14        if (l == r) return tree[i] = arr[l];
15
16        int mid = (l + r) / 2;
17        num_t a = init(2 * i + 1, l, mid, arr),
18              b = init(2 * i + 2, mid + 1, r, arr);
19        return tree[i] = a.op(b);
20    }
21
22    void update(int l, int r, num_t v) {
23        if (l > r) return;
24        update(0, 0, n - 1, l, r, v);
25    }
26
27    num_t update(int i, int tl, int tr, int ql, int qr, num_t v) {
28        eval_lazy(i, tl, tr);
29
30        if (tr < ql || qr < tl) return tree[i];
31        if (ql <= tl && tr <= qr) {
32            lazy[i] = lazy[i].val + v.val;
33            eval_lazy(i, tl, tr);
34            return tree[i];

```

```

35     }
36
37     int mid = (tl + tr) / 2;
38     num_t a = update(2 * i + 1, tl, mid, ql, qr, v),
39               b = update(2 * i + 2, mid + 1, tr, ql, qr, v);
40     return tree[i] = a.op(b);
41 }
42
43 num_t query(int l, int r) {
44     if (l > r) return num_t::null_v;
45     return query(0, 0, n-1, l, r);
46 }
47
48 // int get_first(int v, int tl, int tr, int l, int r, int x) {
49 //     eval_lazy(0, tl, tr);
50 //     if(tl > r || tr < l) return -1;
51 //     if(tree[v].val < x) return -1;
52
53 //     if (tl== tr) return tl;
54
55 //     int tm = tl + (tr-tl)/2;
56 //     int left = get_first(2*v+1, tl, tm, l, r, x);
57 //     if(left != -1) return left;
58 //     return get_first(2*v+2, tm+1, tr, l, r, x);
59 // }
60
61 num_t query(int i, int tl, int tr, int ql, int qr) {
62     eval_lazy(i, tl, tr);
63
64     if (ql <= tl && tr <= qr) return tree[i];
65     if (tr < ql || qr < tl) return num_t::null_v;
66
67     int mid = (tl + tr) / 2;
68     num_t a = query(2 * i + 1, tl, mid, ql, qr),
69             b = query(2 * i + 2, mid + 1, tr, ql, qr);
70     return a.op(b);
71 }
72
73 void eval_lazy(int i, int l, int r) {
74     tree[i] = tree[i].lazy_op(lazy[i], (r - l + 1));
75     if (l != r) {
76         lazy[i * 2 + 1] = lazy[i].val + lazy[i * 2 + 1].val;
77         lazy[i * 2 + 2] = lazy[i].val + lazy[i * 2 + 2].val;

```

```

78     }
79
80     lazy[i] = num_t();
81 }
82 };

```

2.21 Segment Tree Lazy Range Set

```

1
2 int N, Q;
3 int a[maxN];
4
5 struct node {
6     ll val;
7     ll lzAdd;
8     ll lzSet;
9     node(){};
10 } tree[maxN << 2];
11
12 #define lc p << 1
13 #define rc (p << 1) + 1
14
15 inline void pushup(int p) {
16     tree[p].val = tree[lc].val + tree[rc].val;
17     return;
18 }
19
20 void pushdown(int p, int l, int mid, int r) {
21     // lazy: range set
22     if (tree[p].lzSet != 0) {
23         tree[lc].lzSet = tree[rc].lzSet = tree[p].lzSet;
24         tree[lc].val = (mid - l + 1) * tree[p].lzSet;
25         tree[rc].val = (r - mid) * tree[p].lzSet;
26         tree[lc].lzAdd = tree[rc].lzAdd = 0;
27         tree[p].lzSet = 0;
28     } else if (tree[p].lzAdd != 0) { // lazy: range add
29         if (tree[lc].lzSet == 0) tree[lc].lzAdd += tree[p].lzAdd;
30         else {
31             tree[lc].lzSet += tree[p].lzAdd;
32             tree[lc].lzAdd = 0;
33         }
34         if (tree[rc].lzSet == 0) tree[rc].lzAdd += tree[p].lzAdd;
35         else {

```

```

36     tree[rc].lzSet += tree[p].lzAdd;
37     tree[rc].lzAdd = 0;
38 }
39 tree[lc].val += (mid - l + 1) * tree[p].lzAdd;
40 tree[rc].val += (r - mid) * tree[p].lzAdd;
41 tree[p].lzAdd = 0;
42 }
43 return;
44 }
45
46 void build(int p, int l, int r) {
47     tree[p].lzAdd = tree[p].lzSet = 0;
48     if (l == r) {
49         tree[p].val = a[l];
50         return;
51     }
52     int mid = (l + r) >> 1;
53     build(lc, l, mid);
54     build(rc, mid + 1, r);
55     pushup(p);
56     return;
57 }
58
59 void add(int p, int l, int r, int a, int b, ll val) {
60     if (a > r || b < l) return;
61     if (a <= l && r <= b) {
62         tree[p].val += (r - l + 1) * val;
63         if (tree[p].lzSet == 0) tree[p].lzAdd += val;
64         else tree[p].lzSet += val;
65         return;
66     }
67     int mid = (l + r) >> 1;
68     pushdown(p, l, mid, r);
69     add(lc, l, mid, a, b, val);
70     add(rc, mid + 1, r, a, b, val);
71     pushup(p);
72     return;
73 }
74
75 void set(int p, int l, int r, int a, int b, ll val) {
76     if (a > r || b < l) return;
77     if (a <= l && r <= b) {
78         tree[p].val = (r - l + 1) * val;

```

```

79     tree[p].lzAdd = 0;
80     tree[p].lzSet = val;
81     return;
82 }
83 int mid = (l + r) >> 1;
84 pushdown(p, l, mid, r);
85 set(lc, l, mid, a, b, val);
86 set(rc, mid + 1, r, a, b, val);
87 pushup(p);
88 return;
89 }
90
91 ll query(int p, int l, int r, int a, int b) {
92     if (a > r || b < l) return 0;
93     if (a <= l && r <= b) return tree[p].val;
94     int mid = (l + r) >> 1;
95     pushdown(p, l, mid, r);
96     return query(lc, l, mid, a, b) + query(rc, mid + 1, r, a, b);
97 }

```

2.22 Segment Tree Max Subarray Sum

```

1  const ll inf=1e18;
2
3  struct Node {
4      ll maxi, l_max, r_max, sum;
5
6      Node(ll _maxi, ll _l_max, ll _r_max, ll _sum){
7          maxi=_maxi;
8          l_max=_l_max;
9          r_max=_r_max;
10         sum=_sum;
11     }
12
13     Node operator+(Node b) {
14         return {max(maxi, b.maxi), r_max + b.l_max,
15                 max(l_max, sum + b.l_max), max(b.r_max, r_max + b.sum),
16                 sum + b.sum};
17     }
18
19 };
20
21 struct SegmentTreeMaxSubSum{

```

```

22 int n;
23 vector<Node> t;
24
25 SegmentTreeMaxSubSum(int _n) : n(_n), t(2 * _n, Node(-inf, -inf, -inf,
    -inf)) {}
26
27 void update(int pos, ll val) {
28     t[pos += n] = Node(val, val, val, val);
29     for (pos >>= 1; pos ; pos >>= 1) {
30         t[pos] = t[2*pos] + t[2*pos+1];
31     }
32 }
33
34 Node query(int l, int r) {
35     Node node_l = Node(-inf, -inf, -inf, -inf);
36     Node node_r = Node(-inf, -inf, -inf, -inf);
37     for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
38         if (l & 1) {
39             node_l = node_l + t[l++];
40         }
41         if (r & 1) {
42             node_r = t[--r] + node_r;
43         }
44     }
45     return node_l + node_r;
46 }
47 };

```

2.23 Segment Tree Range Update

```

1 struct SegmentTree {
2     vector<ll> a;
3     int n;
4
5     SegmentTree(int _n) : a(2 * _n, 1e18), n(_n) {}
6
7
8     ll get(int pos) {
9         ll res = 1e18;
10        for (pos += n; pos; pos >>= 1) {
11            res = min(res, a[pos]);
12        }
13        return res;

```

```

14    }
15
16    void update(int l, int r, ll val) {
17        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
18            if (l & 1) {
19                a[l] = min(a[l], val);
20                l++;
21            }
22            if (r & 1) {
23                r--;
24                a[r] = min(a[r], val);
25            }
26        }
27    }
28 };

```

2.24 Segment Tree Struct Types

```

1 struct sum_t{
2     ll val;
3     static const long long null_v = 0;
4
5     sum_t(): val(null_v) {}
6     sum_t(long long v): val(v) {}
7
8     sum_t operator + (const sum_t &a) const {
9         sum_t ans;
10        ans.val = val + a.val;
11        return ans;
12    }
13 };
14 // agregar max subarray sum

```

2.25 Segment Tree Struct

```

1 // works as a 0-indexed segtree (not lazy)
2 template <typename num_t>
3 struct segtree
4 {
5     int n, k;
6     vector<num_t> tree;
7
8     void init(int s, vector<ll> arr)
9     {

```

```

10     n = s;
11     k = 0;
12     while ((1 << k) < n)
13         k++;
14     tree = vector<num_t>(2 * (1 << k) + 1);
15     for (int i = 0; i < n; i++)
16     {
17         tree[(1 << k) + i] = arr[i];
18     }
19     for (int i = (1 << k) - 1; i > 0; i--)
20     {
21         tree[i] = tree[i * 2] + tree[i * 2 + 1];
22     }
23 }
24
25 void update(int a, ll b)
26 {
27     a += (1 << k);
28     tree[a] = b;
29     for (a /= 2; a >= 1; a /= 2)
30     {
31         tree[a] = tree[a * 2] + tree[a * 2 + 1];
32     }
33 }
34 num_t find(int a, int b)
35 {
36     a += (1 << k);
37     b += (1 << k);
38     num_t s;
39     while (a <= b)
40     {
41         if (a % 2 == 1)
42             s = s + tree[a++];
43         if (b % 2 == 0)
44             s = s + tree[b--];
45         a /= 2;
46         b /= 2;
47     }
48     return s;
49 }
50 };

```

2.26 Segment Tree Walk

```

1 struct SegmentTreeWalk {
2     vector<ll> a, final_pos;
3     int n;
4
5     SegmentTreeWalk(int _n) : a(4 * _n, 1e18), final_pos(_n), n(_n) {}
6
7     // l = 0, r = n - 1
8     void build(int l, int r, int node, const vector<ll> &vals) {
9         if (l == r){
10             final_pos[l] = node;
11             a[node] = vals[l];
12         }
13         else {
14             int mid = (l + r) / 2;
15             build(l, mid, node * 2, vals);
16             build(mid + 1, r, node * 2 + 1, vals);
17             a[node] = min(a[node * 2], a[node * 2 + 1]);
18         }
19     }
20
21     void update(int pos, ll val){
22         pos = final_pos[pos];
23         a[pos] = val;
24         pos /= 2;
25         while(pos){
26             a[pos] = min(a[2 * pos], a[2 * pos + 1]);
27             pos /= 2;
28         }
29     }
30
31     //inclusive
32     ll get(int l, int r, int L, int R, int node) {
33         if (L > R)
34             return 1e18;
35         if (l == L && r == R) {
36             return a[node];
37         }
38         int mid = (l + r) / 2;
39         return min(get(l, mid, L, min(R, mid), 2 * node), get(mid + 1, r,
40             max(L, mid + 1), R, 2 * node + 1));
41     }
42 }

```



```

41 // l = 0, r = n - 1, L = query start, R = query end
42 // you can just do ll if you only care about value and not index or no
43 // update
44 pair<ll, ll> query(int l, int r, int L, int R, int node, int val){
45     //cout << l << " " << r << endl;
46     if(l > R || r < L) return {-1, 0};
47     if(a[node] < val) return {-1, 0};
48     if(l == r){
49         // depending on what you want to do
50         return {a[node], l};
51     }
52
53     int mid = (l + r) / 2;
54     auto left = query(l, mid, L, R, 2 * node, val);
55     if(left.first != -1) return left;
56     auto right = query(mid + 1, r, L, R, 2 * node + 1, val);
57     return right;
58 }
59 };

```

2.27 Sparse Table

```

1 const int MAXN=100005, K=30;
2 int lg[MAXN+1];
3 int st[K + 1][MAXN];
4
5 int mini(int L, int R){
6     int i = lg[R - L + 1];
7     int minimum = min(st[i][L], st[i][R - (1 << i) + 1]);
8     return minimum;
9 }
10
11 int main(){
12     lg[1]=0;
13     for (int i = 2; i <= MAXN; i++)
14         lg[i] = lg[i/2] + 1;
15     std::copy(a.begin(), a.end(), st[0]);
16
17     for (int i = 1; i <= K; i++)
18         for (int j = 0; j + (1 << i) <= n; j++)
19             st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
20 }

```

2.28 Square Root Decomposition

```

1
2 int n, numBlocks;
3 string s;
4
5 struct Block{
6     int l, r;
7     int sz(){
8         return r-l;
9     }
10 };
11
12 Block blocks[2*MAXI];
13 Block newBlocks[2*MAXI];
14
15 void rebuildDecomp(){
16     string newS=s;
17     int k=0;
18     for(int i=0;i<numBlocks;i++){
19         for(int j=blocks[i].l;j<blocks[i].r;j++){
20             newS[k++]=s[j];
21         }
22     }
23     numBlocks=1;
24     blocks[0]={0, n};
25     s=newS;
26 }
27
28 void cut(int a, int b){
29     int pos=0, curBlock=0;
30     for(int i=0;i<numBlocks;i++){
31         Block B=blocks[i];
32         bool containsA = pos < a && pos + B.sz() > a;
33         bool containsB = pos < b && pos + B.sz() > b;
34         int cutA = B.l + a - pos;
35         int cutB = B.l + b - pos;
36         if(containsA && containsB){
37             newBlocks[curBlock++]={B.l, cutA};
38             newBlocks[curBlock++]={cutA, cutB};
39             newBlocks[curBlock++]={cutB, B.r};
40         }
41         else if(containsA){

```

```

42     newBlocks[curBlock++]={B.l, cutA};
43     newBlocks[curBlock++]={cutA, B.r};
44 }
45 else if(containsB){
46     newBlocks[curBlock++]={B.l, cutB};
47     newBlocks[curBlock++]={cutB, B.r};
48 }
49 else{
50     newBlocks[curBlock++]=B;
51 }
52 pos += B.sz();
53 }
54 pos=0;
55 numBlocks=0;
56 for(int i=0;i<curBlock;i++){
57     if(pos<a || pos>=b){
58         blocks[numBlocks++]=newBlocks[i];
59     }
60     pos+=newBlocks[i].sz();
61 }
62 pos=0;
63 for(int i=0;i<curBlock;i++){
64     if(pos>=a && pos<b){
65         blocks[numBlocks++]=newBlocks[i];
66     }
67     pos+=newBlocks[i].sz();
68 }
69 }
70
71 // while doing operations
72 if(numBlocks>MAXI){
73     rebuildDecomp();
74 }
75
76 // rebuild before final ans
77 rebuildDecomp();
78 cout << ans << endl;

```

2.29 Treap

```

1 struct Node {
2     Node *l = 0, *r = 0;
3     int val, y, c = 1;

```

```

4     Node(int val) : val(val), y(rand()) {}
5     void recalc();
6 };
7
8 int cnt(Node* n) { return n ? n->c : 0; }
9 void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
10
11 template<class F> void each(Node* n, F f) {
12     if (n) { each(n->l, f); f(n->val); each(n->r, f); }
13 }
14
15 pair<Node*, Node*> split(Node* n, int k) {
16     if (!n) return {};
17     if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
18         auto pa = split(n->l, k);
19         n->l = pa.second;
20         n->recalc();
21         return {pa.first, n};
22     } else {
23         auto pa = split(n->r, k - cnt(n->l) - 1); // and just "k"
24         n->r = pa.first;
25         n->recalc();
26         return {n, pa.second};
27     }
28 }
29
30 Node* merge(Node* l, Node* r) {
31     if (!l) return r;
32     if (!r) return l;
33     if (l->y > r->y) {
34         l->r = merge(l->r, r);
35         l->recalc();
36         return l;
37     } else {
38         r->l = merge(l, r->l);
39         r->recalc();
40         return r;
41     }
42 }
43
44 Node* ins(Node* t, Node* n, int pos) {
45     auto pa = split(t, pos);
46     return merge(merge(pa.first, n), pa.second);

```

```

47 }
48
49 // Example application: move the range [l, r) to index k
50 void move(Node*& t, int l, int r, int k) {
51     Node *a, *b, *c;
52     tie(a,b) = split(t, l); tie(b,c) = split(b, r - l);
53     if (k <= l) t = merge(ins(a, b, k), c);
54     else t = merge(a, ins(c, b, k - r));
55 }
56
57 // Usage
58 // create treap
59 // Node* name=nullptr;
60 // insert element
61 // name=ins(name, new Node(val), pos);
62 // Node* x = new Node(val);
63 // name = ins(name, x, pos);
64 // merge two treaps (name before x)
65 // name=merge(name, x);
66 // split treap (this will split treap in two treaps,
67 // first with elements [0, pos) and second with elements [pos, n))
68 // pa will be pair of two treaps
69 // auto pa = split(name, pos);
70 // move range [l, r) to index k
71 // move(name, l, r, k);
72 // iterate over treap
73 // each(name, [&](int val) {
74 //     cout << val << ' ';
75 // });

```

2.30 Treap 2

```

1 typedef struct item * pitem;
2 struct item {
3     int prior, value, cnt;
4     bool rev;
5     pitem l, r;
6 };
7
8 int cnt (pitem it) {
9     return it ? it->cnt : 0;
10 }
11

```

```

12 void upd_cnt (pitem it) {
13     if (it)
14         it->cnt = cnt(it->l) + cnt(it->r) + 1;
15 }
16
17 void push (pitem it) {
18     if (it && it->rev) {
19         it->rev = false;
20         swap (it->l, it->r);
21         if (it->l) it->l->rev ^= true;
22         if (it->r) it->r->rev ^= true;
23     }
24 }
25
26 void merge (pitem & t, pitem l, pitem r) {
27     push (l);
28     push (r);
29     if (!l || !r)
30         t = l ? l : r;
31     else if (l->prior > r->prior)
32         merge (l->r, l->r, r), t = l;
33     else
34         merge (r->l, l, r->l), t = r;
35     upd_cnt (t);
36 }
37
38 void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
39     if (!t)
40         return void( l = r = 0 );
41     push (t);
42     int cur_key = add + cnt(t->l);
43     if (key <= cur_key)
44         split (t->l, l, t->l, key, add), r = t;
45     else
46         split (t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;
47     upd_cnt (t);
48 }
49
50 void reverse (pitem t, int l, int r) {
51     pitem t1, t2, t3;
52     split (t, t1, t2, l);
53     split (t2, t2, t3, r-l+1);
54     t2->rev ^= true;

```

```

55     merge (t, t1, t2);
56     merge (t, t, t3);
57 }
58
59 void output (pitem t) {
60     if (!t) return;
61     push (t);
62     output (t->l);
63     printf ("%d_", t->value);
64     output (t->r);
65 }

```

2.31 Treap With Inversion

```

1 struct Node {
2     Node *l = 0, *r = 0;
3     int val, y, c = 1;
4     bool rev = 0;
5     Node(int val) : val(val), y(rand()) {}
6     void recalc();
7     void push();
8 };
9
10 int cnt(Node* n) { return n ? n->c : 0; }
11 void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
12 void Node::push() {
13     if (rev) {
14         rev = 0;
15         swap(l, r);
16         if (l) l->rev ^= 1;
17         if (r) r->rev ^= 1;
18     }
19 }
20
21 template<class F> void each(Node* n, F f) {
22     if (n) { n->push(); each(n->l, f); f(n->val); each(n->r, f); }
23 }
24
25 pair<Node*, Node*> split(Node* n, int k) {
26     if (!n) return {};
27     n->push();
28     if (cnt(n->l) >= k) {
29         auto pa = split(n->l, k);

```

```

30     n->l = pa.second;
31     n->recalc();
32     return {pa.first, n};
33 } else {
34     auto pa = split(n->r, k - cnt(n->l) - 1);
35     n->r = pa.first;
36     n->recalc();
37     return {n, pa.second};
38 }
39 }
40
41 Node* merge(Node* l, Node* r) {
42     if (!l) return r;
43     if (!r) return l;
44     l->push();
45     r->push();
46     if (l->y > r->y) {
47         l->r = merge(l->r, r);
48         l->recalc();
49         return l;
50     } else {
51         r->l = merge(l, r->l);
52         r->recalc();
53         return r;
54     }
55 }
56
57 Node* ins(Node* t, Node* n, int pos) {
58     auto pa = split(t, pos);
59     return merge(merge(pa.first, n), pa.second);
60 }
61
62 // Example application: reverse the range [l, r]
63 void reverse(Node*& t, int l, int r) {
64     Node *a, *b, *c;
65     tie(a,b) = split(t, l);
66     tie(b,c) = split(b, r - l + 1);
67     b->rev ^= 1;
68     t = merge(merge(a, b), c);
69 }
70
71 void move(Node*& t, int l, int r, int k) {
72     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