```
// DSU on TREE

int sz[maxn];
void getsz(int v, int p){
    sz[v] = 1;  // every vertex has itself in its subtree
    for(auto u : g[v])
        if(u != p){
            getsz(u, v);
            sz[v] += sz[u]; // add size of child u to its parent(v)
}}

// Map Style: n(logn)^2

map<int, int> *cnt[maxn];
void dfs(int v, int p){
    int mx = -1, bigChild = -1;
    for(auto u : g[v])
       if(u != p){
           dfs(u, v);
           if(sz[u] > mx)
               mx = sz[u], bigChild = u;
       }
    if(bigChild != -1)
        cnt[v] = cnt[bigChild];
    else
        cnt[v] = new map<int, int> ();

    (*cnt[v])[ col[v] ] ++;
    for(auto u : g[v])
       if(u != p && u != bigChild){
           for(auto x : *cnt[u])
               (*cnt[v])[x.first] += x.second;
       }
    //now (*cnt[v])[c] is the number of vertices in subtree of vertex v that has color c. You can answer the queries easily.
}

// Vector Style: nlogn

vector<int> *vec[maxn];
int cnt[maxn];
void dfs(int v, int p, bool keep){
    int mx = -1, bigChild = -1;
    for(auto u : g[v])
       if(u != p && sz[u] > mx)
           mx = sz[u], bigChild = u;
    for(auto u : g[v])
       if(u != p && u != bigChild)
           dfs(u, v, 0);
    if(bigChild != -1)
        dfs(bigChild, v, 1), vec[v] = vec[bigChild];
    else
        vec[v] = new vector<int> ();
    vec[v]->push_back(v);
    cnt[ col[v] ]++;
    for(auto u : g[v])
       if(u != p && u != bigChild)
           for(auto x : *vec[u]){
```

```
                cnt[ col[x] ]++;
                vec[v] -> push_back(x);
            }
    //now (*cnt[v])[c] is the number of vertices in subtree of vertex v that has color c. You can answer the queries easily.
    // note that in this step *vec[v] contains all of the subtree of vertex v.
    if(keep == 0)
        for(auto u : *vec[v])
            cnt[ col[u] ]--;
}

// Heavy-Light-Decomposition Style: nlogn

int cnt[maxn];
bool big[maxn];
void add(int v, int p, int x){
    cnt[ col[v] ] += x;
    for(auto u: g[v])
        if(u != p && !big[u])
            add(u, v, x)
}
void dfs(int v, int p, bool keep){
    int mx = -1, bigChild = -1;
    for(auto u : g[v])
       if(u != p && sz[u] > mx)
          mx = sz[u], bigChild = u;
    for(auto u : g[v])
        if(u != p && u != bigChild)
            dfs(u, v, 0);  // run a dfs on small childs and clear them from cnt
    if(bigChild != -1)
        dfs(bigChild, v, 1), big[bigChild] = 1;  // bigChild marked as big and not cleared from cnt
    add(v, p, 1);
    //now cnt[c] is the number of vertices in subtree of vertex v that has color c. You can answer the queries easily.
    if(bigChild != -1)
        big[bigChild] = 0;
    if(keep == 0)
        add(v, p, -1);
}

// KMP EXTRA PART

// p is the pattern where table[] is the pre-made prefix table of pattern
// for any index idx the nxt[idx][j] returns the new index idx where the index
// should point next, this optimizes the kmp in linear time

void getState(string &p, int table[], int nxt[][27]) {
    for(int i = 0; i < p.size(); ++i) {
        for(int j = 0; j < 26; ++j) {
            if(p[i]-'a' == j)
                nxt[i][j] = i+1;
            else
                nxt[i][j] = i == 0 ? 0 : nxt[table[i-1]][j];
}}}

// check function using nxt[idx][j]
// idx is the index from which the string should start matching with the pattern
```

```cpp
// by default idx = 0, also it refers the last index of the pattern to which
// the string matched

int match(string &s, int table[], int nxt[][27], int &idx) {
    int ans = 0;
    for(char c : s) {
        idx = nxt[idx][c-'a'];
        if(idx == p.size())
            ++ans, idx = table[idx-1];
    }
    return ans;
}

// LCA
// Least Common Ancestor with sparse table

void dfs(int u, int p) {
    in[u] = ++cnt;
    revIn[cnt] = u, par[u][0] = p, lvl[u] = lvl[p]+1;

    for(int i = 1; i <= 20; ++i)
        par[u][i] = par[par[u][i-1]][i-1];

    for(auto v : G[u])
        if(v != p)
            dfs(v, u);
    out[u] = cnt;
}

int LCA(int u, int v) {
    if(lvl[u] < lvl[v]) swap(u, v);
    for(int p = 20; p >= 0; --p)
        if(lvl[u] - (1 << p) >= lvl[v])
            u = par[u][p];
    if(u == v) return u;
    for(int p = 20; p >= 0; --p)
        if(par[u][p] != par[v][p])
            u = par[u][p], v = par[v][p];
    return par[u][0];
}

// LCA if the root changes, [first dfs is done with root 1 or any other fixed node]
int LCA(int u, int v, int root) {
    if(isChild(u, root) and isChild(v, root))
        return LCA(u, v);
    if(isChild(u, root) != isChild(v, root))
        return root;
    int x = LCA(u, v), y = LCA(u, root), z = LCA(v, root);
    int a = lvl[root] - lvl[x], b = lvl[root] - lvl[y], c = lvl[root] - lvl[z];
    if(a <= b and a <= c) return x;
    if(b <= a and b <= c) return y;
    return z;
}

// -------------------------- LCA WITH Sparse Table Vector --------------------------
// DFS and LCA INIT is same
void MERGE(vector<int>&u, vector<int>&v) {          // Do what is to be done to merge
    for(auto it : v) u.push_back(it);               // here taking lowest 10 values
```

```
    sort(u.begin(), u.end());
    while((int)u.size() > 10)
        u.pop_back();
}

vector<int> W[MAX][20];              // W[u][0] will contain initial weight/weights at node u
vector<int> LCA(int u, int v) {
    vector<int> T;
    if(level[u] > level[v]) swap(u, v);     // v is deeper
    int p = ceil(log2(level[v]));
    for(int i = p ; i >= 0; --i)            // Pull up v to same level as u
        if(level[v] - (1LL<<i) >= level[u]) {
            MERGE(T, W[v][i]);
            v = sparse[v][i];
        }
    if(u == v) {                                // if u WAS the parent
        MERGE(T, W[u][0]);
        return T;
    }
    for(int i = p; i >= 0; --i)                                      // Pull up u and v together while LCA not found
        if(sparse[v][i] != -1 && sparse[u][i] != sparse[v][i]) {        // -1 check is if 2^i is out of calculated range
            MERGE(T, W[u][i]);
            MERGE(T, W[v][i]);
            u = sparse[u][i], v = sparse[v][i];
        }
    MERGE(T, W[u][0]);              // As W[x][0] denoted the x nodes weight
    MERGE(T, W[v][0]);              // every sparse node must be calculated
    MERGE(T, W[sparse[v][0]][0]);   // we can also calculate summation of distance like this
    return T;
}

// --------------------------- Overlap Path of Tree ---------------------------

// Note: DfsTiming and isChild function required
// a is upper node of path a-b and c is upper node of path c-d
pii overlapPath(int a, int b, int c, int d) {      // returns number of common path of c-d and a-b
    // path a-b and c-d overlaps iff b is a child of c or d or both of c&d
    if(not isChild(b, c)) return {0, 0};
    int u = LCA(b, d);              // u is the lowest point on which c-d and a-b overlaps
    if(level[a]>level[c]) {             // a is below c
        if(isChild(u, a))           // also u is child of a
            return {a, u};
    }
    else {                          // c is above a
        if(isChild(u, c))
            return {c, u};
    }
    return {0, 0};                  // no common path found
}

int EdgeCount(int a, int b, int c, int d) {             // Finds number of edges if we join nodes a, b and
    a = Convert(a), b = Convert(b), c = Convert(c), d = Convert(d); // want to find path from c to d
    int u = LCA(a, b);
```

```
    int v = LCA(c, d);
    int ans = dist(c, d, v);
    pii tt;
    // connected paths are u->a & u->b
    // query paths are v->c & v->d
    // cases:
    // u->a overlaps v->c
    tt = overlapPath(v, c, u, a);
    ans -= tt.fi == 0? 0:dist(tt.fi, tt.se, LCA(tt.fi, tt.se));
    // u->a overlaps v->d
    tt = overlapPath(v, c, u, b);
    ans -= tt.fi == 0? 0:dist(tt.fi, tt.se, LCA(tt.fi, tt.se));
    // u->b overlaps v->c
    tt = overlapPath(v, d, u, a);
    ans -= tt.fi == 0? 0:dist(tt.fi, tt.se, LCA(tt.fi, tt.se));
    // u->b overlaps v->d
    tt = overlapPath(v, d, u, b);
    ans -= tt.fi == 0? 0:dist(tt.fi, tt.se, LCA(tt.fi, tt.se));
    return ans;
}

// ----- return k'th node if we traverse from node u to v of a tree

// NOT TESTED!!
int getKthNode(int u, int v, int k, int lca) {
    int lftChain = lvl[u] - lvl[lca] + 1;
    int rhtChain = lvl[v] - lvl[lca];
    if(k == 1) return u;
    if(lca == v) {
        for(int i = 20; i >= 0; --i)
            if(k - (1 << i) >= 1)
                u = par[u][i], k -= (1 << i);
        return u;
    }
    if(lca == u) {
        k = rhtChain+1-k;
        for(int i = 20; i >= 0; --i)
            if(k - (1 << i) >= 0)
                v = par[v][i], k -= (1 << i);
        return v;
    }
    if(k > lftChain) {
        k -= lftChain;
        k = rhtChain - k;
        for(int i = 20; i >= 0; --i)
            if(k - (1 << i) >= 0)
                v = par[v][i], k -= (1 << i);
        return v;
    }
    for(int i = 20; i >= 0; --i)
        if(k - (1 << i) >= 1)
            u = par[u][i], k -= (1 << i);
    return u;
}

// -------- SUBTREE UPDATE FUNCTIONS --------
// if the root changes

```

```
void subTreeUpdate(int u, int root, int val) {
    // if u is child of root, then subtree of u
    if(u == root)
        DS.update(in[1], out[1], val);
    else if(isChild(u, root))
        DS.update(in[u], out[u], val);
    // if root is child of u
    else if(isChild(root, u)) {
        int x = getChild(root, u);      // get the first child of u
        DS.update(in[1], out[1], val);
        DS.update(in[x], out[x], -val);
    }
    else
        DS.update(in[u], out[u], val);
}

ll getSubTreeSum(int u, int root) {
    // if u is child of root, then subtree of u
    if(u == root)
        return DS.query(in[1], out[1]);
    if(isChild(u, root))
        return DS.query(in[u], out[u]);
    // if root is child of u
    else if(isChild(root, u)) {
        int x = getChild(root, u);      // get the first child of u
        return DS.query(in[1], out[1]) - DS.query(in[x], out[x]);
    }
    else
        return DS.query(in[u], out[u]);
}

// ------- Can Give Total Spanning Tree edges for an particular set of nodes

set<int>nodes;                  // contains nodes according to dfs order
int nodeCost(int u) {           // returns node Query/Insert updated distance
    auto it = nodes.insert(in[u]).first;        // inserted according to dfs in-timing
    auto l = it, r = it;                        // iterator of the inserted index
    if(it == nodes.begin())
        l = --nodes.end();
    else --l;

    if(it == --nodes.end())
        r = nodes.begin();
    else ++r;

    int L = revIn[*l], R = revIn[*r];     // nodes are retrieved from dfs in-timing

    // dst is the spanning distance if the new node is added
    int dst = lvl[u] + lvl[LCA(L, R)] - lvl[LCA(u, L)] - lvl[LCA(u, R)];
    return dst;
}

void removeNode(int u) {
    nodes.erase(in[u]);
}

struct LCATree {
    int tree[MAX*4], lca, n, cost;
```

```
    set<int>nodes;

    void init(int sz) { n = sz, lca = -1, cost = 0; }
    void update(int pos, int l, int r, int idx, int v) {
        if(l == r) {
            tree[pos] += v;
            return;
        }
        int mid = (l+r)>>1;
        if(idx <= mid)  update(pos<<1, l, mid, idx, v);
        else            update(pos<<1|1, mid+1, r, idx, v);
        tree[pos] = tree[pos<<1] + tree[pos<<1|1];
    }
    int query(int pos, int l, int r, int L, int R) {
        if(r < L or R < l)      return 0;
        if(L <= l and r <= R)   return tree[pos];
        int mid = (l+r)>>1;
        return query(pos<<1, l, mid, L, R) + query(pos<<1|1, mid+1, r, L, R);
    }
    int getPar(int u, int p) {
        for(int i = 20; i >= 0; --i)
            if(p & (1 << i))
                u = par[u][i];          // parent sparse table
        return u;
    }
    int LCA() {
        int u = *nodes.begin(), tot = nodes.size(), v, ret = *nodes.begin();
        int lo = 0, hi = lvl[u]-1;
        while(lo <= hi) {
            int mid = (lo+hi)>>1;
            v = getPar(u, mid);
            if(query(1, 1, n, in[v], out[v]) == tot)
                hi = mid-1, ret = v;            // in : dfs in time
            else                                // out : dfs out time
                lo = mid+1;
        }
        return ret;
    }
    int findChainPar(int u, int t) {                // finds parent node of u having
        int lo = 0, hi = lvl[u]-1, ret = u, v, mid; // active child node more than t
        while(lo <= hi) {
            mid = (lo+hi)>>1;
            v = getPar(u, mid);
            if(query(1, 1, n, in[v], out[v]) > t)
                hi = mid-1, ret = v;
            else
                lo = mid+1;
        }
        return ret;
    }
    void addNode(int u) {
        int pstLca = lca;
        nodes.insert(u), update(1, 1, n, in[u], 1);
        if(lca == -1) {
            lca = u;
            return;
        }
        else
```

```
            lca = LCA();
        // if new LCA is same but the new node is on different chain
        if(pstLca == lca and query(1, 1, n, in[u], out[u]) == 1) {
            int v = findChainPar(u, 1);
            cost += lvl[u] - lvl[v];
        }
        // if new LCA changes, newLCA will always be upper from past LCA
        // also the node u is on different chain
        else if(lca != pstLca)
            cost += lvl[u] + lvl[pstLca] - 2*lvl[lca];
    }
    void removeNode(int u) {
        int pstLca = lca;
        nodes.erase(u), update(1, 1, n, in[u], -1);
        if(nodes.empty()) {
            lca = -1, cost = 0;
            return;
        }
        else
            lca = LCA();
        if(pstLca == lca and query(1, 1, n, in[u], out[u]) == 0) {
            int v = findChainPar(u, 0);
            cost -= lvl[u] - lvl[v];
        }
        else if(lca != pstLca)
            cost -= lvl[lca] + lvl[u] - 2*lvl[pstLca];
}};

//Trie
//Complexity : making a trie : O(S), searching : O(S)

bool found;
struct node {
    bool isEnd;
    node *next[11];
    node() {
        isEnd = false;
        for(int i = 0; i < 10; i++)
            next[i] = NULL;
    }};

//trie of a string abc, ax
// [start] --> [a] --> [b] --> [c] --> endMark
//              |
//             [x] --> endMark
//creates trie, returns true if the trie we are creating is a segment of a string
//to only create a trie remove lines which are comment marked

bool create(char str[], int len, node *current) {
    for(int i = 0; i < len; i++) {
        int pos = str[i] - '0';
        if(current->next[pos] == NULL)
            current->next[pos] = new node();
        current = current->next[pos];
        if(current->isEnd)  //
            return true;    //
    }
    current->isEnd = true;  //
```

```
    return false;           //
}

void del(node *current) {
    for(int i = 0; i < 10; i++)
        if(current->next[i] != NULL)
            del(current->next[i]);
    delete current;
}

void check(node *current) {
    for(int i = 0; i < 10; i++) {
        if(current->next[i] != NULL)
            check(current->next[i]);
    }
    if(found) return;
    if(current->isEnd && !found) {
        for(int i = 0; i < 10 && !found; i++)
            if(current->next[i] != NULL) {
                found = 1;
}}}

// NON-Dynamic implementation
// root node is at 0 index of tree
// root node counter contains total number of string insertion
// each inserted char counter is on the child node of the edges

struct Trie {
    struct node {
        int cnt;
        int nxt[60];
    };
    int nodes;
    node tree[MAX];
    void newNode() {
        tree[nodes].cnt = 0;
        memset(tree[nodes].nxt, -1, sizeof tree[nodes].nxt);
        ++nodes;
    }
    void init() {
        nodes = 0;
        newNode();
    }
    int getId(char x) {
        if(x >= 'A' and  x <= 'Z')
            return (x - 'A' + 27);
        return (x - 'a' + 1);
    }
    void insert(string &str, int len = 0, int idx = 0) {
        tree[idx].cnt++;
        if(len == str.size()) return;
        int id = getId(str[len]);
        if(tree[idx].nxt[id] == -1) {
            tree[idx].nxt[id] = nodes;
            newNode();
        }
        insert(str, len+1, tree[idx].nxt[id]);
    }
```

```
    int search(string &str, int len = 0, int idx = 0) {
        if(len == str.size())
            return -2;
        int id = getId(str[len]);
        if(tree[idx].nxt[id] == -1)
            return -1;
        if(tree[idx].cnt == 1)
            return len;
        return search(str, len+1, tree[idx].nxt[id]);
}};
```