

```

1. // DSU on TREE
2.
3. int sz[maxn];
4. void getsz(int v, int p){
5.     sz[v] = 1; // every vertex has itself in its subtree
6.     for(auto u : g[v])
7.         if(u != p){
8.             getsz(u, v);
9.             sz[v] += sz[u]; // add size of child u to its parent(v)
10. }
11.
12. // Map Style: n(logn)^2
13.
14. map<int, int> *cnt[maxn];
15. void dfs(int v, int p){
16.     int mx = -1, bigChild = -1;
17.     for(auto u : g[v])
18.         if(u != p){
19.             dfs(u, v);
20.             if(sz[u] > mx)
21.                 mx = sz[u], bigChild = u;
22.         }
23.     if(bigChild != -1)
24.         cnt[v] = cnt[bigChild];
25.     else
26.         cnt[v] = new map<int, int> ();
27.
28.     (*cnt[v])[ col[v] ] ++;
29.     for(auto u : g[v])
30.         if(u != p && u != bigChild){
31.             for(auto x : *cnt[u])
32.                 (*cnt[v])[x.first] += x.second;
33.         }
34.     //now (*cnt[v])[c] is the number of vertices in subtree of vertex v that has color c.
    You can answer the queries easily.
35. }
36.
37. // Vector Style: nlogn
38.
39. vector<int> *vec[maxn];
40. int cnt[maxn];
41. void dfs(int v, int p, bool keep){
42.     int mx = -1, bigChild = -1;
43.     for(auto u : g[v])
44.         if(u != p && sz[u] > mx)
45.             mx = sz[u], bigChild = u;
46.     for(auto u : g[v])
47.         if(u != p && u != bigChild)
48.             dfs(u, v, 0);
49.     if(bigChild != -1)
50.         dfs(bigChild, v, 1), vec[v] = vec[bigChild];
51.     else
52.         vec[v] = new vector<int> ();
53.     vec[v]->push_back(v);
54.     cnt[ col[v] ]++;
55.     for(auto u : g[v])
56.         if(u != p && u != bigChild)
57.             for(auto x : *vec[u]){

```

```

58.         cnt[ col[x] ]++;
59.         vec[v] -> push_back(x);
60.     }
61.     //now (*cnt[v])[c] is the number of vertices in subtree of vertex v that has color c.
    You can answer the queries easily.
62.     // note that in this step *vec[v] contains all of the subtree of vertex v.
63.     if(keep == 0)
64.         for(auto u : *vec[v])
65.             cnt[ col[u] ]--;
66. }
67.
68. // Heavy-Light-Decomposition Style: nlogn
69.
70. int cnt[maxn];
71. bool big[maxn];
72. void add(int v, int p, int x){
73.     cnt[ col[v] ] += x;
74.     for(auto u: g[v])
75.         if(u != p && !big[u])
76.             add(u, v, x)
77. }
78. void dfs(int v, int p, bool keep){
79.     int mx = -1, bigChild = -1;
80.     for(auto u : g[v])
81.         if(u != p && sz[u] > mx)
82.             mx = sz[u], bigChild = u;
83.     for(auto u : g[v])
84.         if(u != p && u != bigChild)
85.             dfs(u, v, 0); // run a dfs on small childs and clear them from cnt
86.     if(bigChild != -1)
87.         dfs(bigChild, v, 1), big[bigChild] = 1; // bigChild marked as big and not cleared
    from cnt
88.     add(v, p, 1);
89.     //now cnt[c] is the number of vertices in subtree of vertex v that has color c. You can
    answer the queries easily.
90.     if(bigChild != -1)
91.         big[bigChild] = 0;
92.     if(keep == 0)
93.         add(v, p, -1);
94. }
95.
96. // KMP EXTRA PART
97.
98. // p is the pattern where table[] is the pre-made prefix table of pattern
99. // for any index idx the nxt[idx][j] returns the new index idx where the index
100. // should point next, this optimizes the kmp in linear time
101.
102. void getState(string &p, int table[], int nxt[][27]) {
103.     for(int i = 0; i < p.size(); ++i) {
104.         for(int j = 0; j < 26; ++j) {
105.             if(p[i] - 'a' == j)
106.                 nxt[i][j] = i+1;
107.             else
108.                 nxt[i][j] = i == 0 ? 0 : nxt[table[i-1]][j];
109.         }
110.     }
111. // check function using nxt[idx][j]
112. // idx is the index from which the string should start matching with the pattern

```

```

113. // by default idx = 0, also it refers the last index of the pattern to which
114. // the string matched
115.
116. int match(string &s, int table[], int nxt[][27], int &idx) {
117.     int ans = 0;
118.     for(char c : s) {
119.         idx = nxt[idx][c-'a'];
120.         if(idx == p.size())
121.             ++ans, idx = table[idx-1];
122.     }
123.     return ans;
124. }
125.
126. // LCA
127. // Least Common Ancestor with sparse table
128.
129. void dfs(int u, int p) {
130.     in[u] = ++cnt;
131.     revIn[cnt] = u, par[u][0] = p, lvl[u] = lvl[p]+1;
132.
133.     for(int i = 1; i <= 20; ++i)
134.         par[u][i] = par[par[u][i-1]][i-1];
135.
136.     for(auto v : G[u])
137.         if(v != p)
138.             dfs(v, u);
139.     out[u] = cnt;
140. }
141.
142. int LCA(int u, int v) {
143.     if(lvl[u] < lvl[v]) swap(u, v);
144.     for(int p = 20; p >= 0; --p)
145.         if(lvl[u] - (1 << p) >= lvl[v])
146.             u = par[u][p];
147.     if(u == v) return u;
148.     for(int p = 20; p >= 0; --p)
149.         if(par[u][p] != par[v][p])
150.             u = par[u][p], v = par[v][p];
151.     return par[u][0];
152. }
153.
154. // LCA if the root changes, [first dfs is done with root 1 or any other fixed node]
155. int LCA(int u, int v, int root) {
156.     if(isChild(u, root) and isChild(v, root))
157.         return LCA(u, v);
158.     if(isChild(u, root) != isChild(v, root))
159.         return root;
160.     int x = LCA(u, v), y = LCA(u, root), z = LCA(v, root);
161.     int a = lvl[root] - lvl[x], b = lvl[root] - lvl[y], c = lvl[root] - lvl[z];
162.     if(a <= b and a <= c) return x;
163.     if(b <= a and b <= c) return y;
164.     return z;
165. }
166.
167. // ----- LCA WITH Sparse Table Vector -----
168. // DFS and LCA INIT is same
169. void MERGE(vector<int>&u, vector<int>&v) { // Do what is to be done to merge
170.     for(auto it : v) u.push_back(it); // here taking lowest 10 values

```

```

171.     sort(u.begin(), u.end());
172.     while((int)u.size() > 10)
173.         u.pop_back();
174. }
175.
176. vector<int> W[MAX][20];           // W[u][0] will contain initial weight/weights at node u
177. vector<int> LCA(int u, int v) {
178.     vector<int> T;
179.     if(level[u] > level[v]) swap(u, v);    // v is deeper
180.     int p = ceil(log2(level[v]));
181.     for(int i = p ; i >= 0; --i)           // Pull up v to same level as u
182.         if(level[v] - (1LL<<i) >= level[u]) {
183.             MERGE(T, W[v][i]);
184.             v = sparse[v][i];
185.         }
186.     if(u == v) {                          // if u WAS the parent
187.         MERGE(T, W[u][0]);
188.         return T;
189.     }
190.     for(int i = p; i >= 0; --i)           // Pull up u and v
191.         if(sparse[v][i] != -1 && sparse[u][i] != sparse[v][i]) {           // -1 check is if
2^i is out of calculated range
192.             MERGE(T, W[u][i]);
193.             MERGE(T, W[v][i]);
194.             u = sparse[u][i], v = sparse[v][i];
195.         }
196.     MERGE(T, W[u][0]);                   // As W[x][0] denoted the x nodes weight
197.     MERGE(T, W[v][0]);                   // every sparse node must be calculated
198.     MERGE(T, W[sparse[v][0]][0]);       // we can also calculate summation of distance like this
199.     return T;
200. }
201.
202. // ----- Overlap Path of Tree -----
203.
204. // Note: DfsTiming and isChild function required
205. // a is upper node of path a-b and c is upper node of path c-d
206. pii overlapPath(int a, int b, int c, int d) {           // returns number of common path of c-d
207.     // path a-b and c-d overlaps iff b is a child of c or d or both of c&d
208.     if(not isChild(b, c)) return {0, 0};
209.     int u = LCA(b, d);           // u is the lowest point on which c-d and a-b overlaps
210.     if(level[a]>level[c]) {       // a is below c
211.         if(isChild(u, a))         // also u is child of a
212.             return {a, u};
213.     }
214.     else {                       // c is above a
215.         if(isChild(u, c))
216.             return {c, u};
217.     }
218.     return {0, 0};               // no common path found
219. }
220.
221. int EdgeCount(int a, int b, int c, int d) {           // Finds number of edges if we join
222.     // nodes a, b and
223.     a = Convert(a), b = Convert(b), c = Convert(c), d = Convert(d); // want to find path
224.     // from c to d
225.     int u = LCA(a, b);

```

```

224.     int v = LCA(c, d);
225.     int ans = dist(c, d, v);
226.     pii tt;
227.     // connected paths are u->a & u->b
228.     // query paths are v->c & v->d
229.     // cases:
230.     // u->a overlaps v->c
231.     tt = overlapPath(v, c, u, a);
232.     ans -= tt.fi == 0? 0:dist(tt.fi, tt.se, LCA(tt.fi, tt.se));
233.     // u->a overlaps v->d
234.     tt = overlapPath(v, c, u, b);
235.     ans -= tt.fi == 0? 0:dist(tt.fi, tt.se, LCA(tt.fi, tt.se));
236.     // u->b overlaps v->c
237.     tt = overlapPath(v, d, u, a);
238.     ans -= tt.fi == 0? 0:dist(tt.fi, tt.se, LCA(tt.fi, tt.se));
239.     // u->b overlaps v->d
240.     tt = overlapPath(v, d, u, b);
241.     ans -= tt.fi == 0? 0:dist(tt.fi, tt.se, LCA(tt.fi, tt.se));
242.     return ans;
243. }
244.
245. // ----- return k'th node if we traverse from node u to v of a tree
246.
247. // NOT TESTED!!
248. int getKthNode(int u, int v, int k, int lca) {
249.     int lftChain = lvl[u] - lvl[lca] + 1;
250.     int rhtChain = lvl[v] - lvl[lca];
251.     if(k == 1) return u;
252.     if(lca == v) {
253.         for(int i = 20; i >= 0; --i)
254.             if(k - (1 << i) >= 1)
255.                 u = par[u][i], k -= (1 << i);
256.         return u;
257.     }
258.     if(lca == u) {
259.         k = rhtChain+1-k;
260.         for(int i = 20; i >= 0; --i)
261.             if(k - (1 << i) >= 0)
262.                 v = par[v][i], k -= (1 << i);
263.         return v;
264.     }
265.     if(k > lftChain) {
266.         k -= lftChain;
267.         k = rhtChain - k;
268.         for(int i = 20; i >= 0; --i)
269.             if(k - (1 << i) >= 0)
270.                 v = par[v][i], k -= (1 << i);
271.         return v;
272.     }
273.     for(int i = 20; i >= 0; --i)
274.         if(k - (1 << i) >= 1)
275.             u = par[u][i], k -= (1 << i);
276.     return u;
277. }
278.
279. // ----- SUBTREE UPDATE FUNCTIONS -----
280. // if the root changes
281.

```

```

282. void subTreeUpdate(int u, int root, int val) {
283.     // if u is child of root, then subtree of u
284.     if(u == root)
285.         DS.update(in[1], out[1], val);
286.     else if(isChild(u, root))
287.         DS.update(in[u], out[u], val);
288.     // if root is child of u
289.     else if(isChild(root, u)) {
290.         int x = getChild(root, u);    // get the first child of u
291.         DS.update(in[1], out[1], val);
292.         DS.update(in[x], out[x], -val);
293.     }
294.     else
295.         DS.update(in[u], out[u], val);
296. }
297.
298. ll getSubTreeSum(int u, int root) {
299.     // if u is child of root, then subtree of u
300.     if(u == root)
301.         return DS.query(in[1], out[1]);
302.     if(isChild(u, root))
303.         return DS.query(in[u], out[u]);
304.     // if root is child of u
305.     else if(isChild(root, u)) {
306.         int x = getChild(root, u);    // get the first child of u
307.         return DS.query(in[1], out[1]) - DS.query(in[x], out[x]);
308.     }
309.     else
310.         return DS.query(in[u], out[u]);
311. }
312.
313. // ----- Can Give Total Spanning Tree edges for an particular set of nodes
314.
315. set<int>nodes;                // contains nodes according to dfs order
316. int nodeCost(int u) {        // returns node Query/Insert updated distance
317.     auto it = nodes.insert(in[u]).first;    // inserted according to dfs in-timing
318.     auto l = it, r = it;                // iterator of the inserted index
319.     if(it == nodes.begin())
320.         l = --nodes.end();
321.     else --l;
322.
323.     if(it == --nodes.end())
324.         r = nodes.begin();
325.     else ++r;
326.
327.     int L = revIn[*l], R = revIn[*r];    // nodes are retrieved from dfs in-timing
328.
329.     // dst is the spanning distance if the new node is added
330.     int dst = lvl[u] + lvl[LCA(L, R)] - lvl[LCA(u, L)] - lvl[LCA(u, R)];
331.     return dst;
332. }
333.
334. void removeNode(int u) {
335.     nodes.erase(in[u]);
336. }
337.
338. struct LCATree {
339.     int tree[MAX*4], lca, n, cost;

```

```

340. set<int>nodes;
341.
342. void init(int sz) { n = sz, lca = -1, cost = 0; }
343. void update(int pos, int l, int r, int idx, int v) {
344.     if(l == r) {
345.         tree[pos] += v;
346.         return;
347.     }
348.     int mid = (l+r)>>1;
349.     if(idx <= mid) update(pos<<1, l, mid, idx, v);
350.     else update(pos<<1|1, mid+1, r, idx, v);
351.     tree[pos] = tree[pos<<1] + tree[pos<<1|1];
352. }
353. int query(int pos, int l, int r, int L, int R) {
354.     if(r < L or R < l) return 0;
355.     if(L <= l and r <= R) return tree[pos];
356.     int mid = (l+r)>>1;
357.     return query(pos<<1, l, mid, L, R) + query(pos<<1|1, mid+1, r, L, R);
358. }
359. int getPar(int u, int p) {
360.     for(int i = 20; i >= 0; --i)
361.         if(p & (1 << i))
362.             u = par[u][i]; // parent sparse table
363.     return u;
364. }
365. int LCA() {
366.     int u = *nodes.begin(), tot = nodes.size(), v, ret = *nodes.begin();
367.     int lo = 0, hi = lvl[u]-1;
368.     while(lo <= hi) {
369.         int mid = (lo+hi)>>1;
370.         v = getPar(u, mid);
371.         if(query(1, 1, n, in[v], out[v]) == tot)
372.             hi = mid-1, ret = v; // in : dfs in time
373.         else // out : dfs out time
374.             lo = mid+1;
375.     }
376.     return ret;
377. }
378. int findChainPar(int u, int t) { // finds parent node of u having
379.     int lo = 0, hi = lvl[u]-1, ret = u, v, mid; // active child node more than t
380.     while(lo <= hi) {
381.         mid = (lo+hi)>>1;
382.         v = getPar(u, mid);
383.         if(query(1, 1, n, in[v], out[v]) > t)
384.             hi = mid-1, ret = v;
385.         else
386.             lo = mid+1;
387.     }
388.     return ret;
389. }
390. void addNode(int u) {
391.     int pstLca = lca;
392.     nodes.insert(u), update(1, 1, n, in[u], 1);
393.     if(lca == -1) {
394.         lca = u;
395.         return;
396.     }
397.     else

```

```

398.         lca = LCA();
399.         // if new LCA is same but the new node is on different chain
400.         if(pstLca == lca and query(1, 1, n, in[u], out[u]) == 1) {
401.             int v = findChainPar(u, 1);
402.             cost += lvl[u] - lvl[v];
403.         }
404.         // if new LCA changes, newLCA will always be upper from past LCA
405.         // also the node u is on different chain
406.         else if(lca != pstLca)
407.             cost += lvl[u] + lvl[pstLca] - 2*lvl[lca];
408.     }
409.     void removeNode(int u) {
410.         int pstLca = lca;
411.         nodes.erase(u), update(1, 1, n, in[u], -1);
412.         if(nodes.empty()) {
413.             lca = -1, cost = 0;
414.             return;
415.         }
416.         else
417.             lca = LCA();
418.         if(pstLca == lca and query(1, 1, n, in[u], out[u]) == 0) {
419.             int v = findChainPar(u, 0);
420.             cost -= lvl[u] - lvl[v];
421.         }
422.         else if(lca != pstLca)
423.             cost -= lvl[lca] + lvl[u] - 2*lvl[pstLca];
424.     }
425.
426. //Trie
427. //Complexity : making a trie : O(S), searching : O(S)
428.
429. bool found;
430. struct node {
431.     bool isEnd;
432.     node *next[11];
433.     node() {
434.         isEnd = false;
435.         for(int i = 0; i < 10; i++)
436.             next[i] = NULL;
437.     }
438.
439. //trie of a string abc, ax
440. // [start] --> [a] --> [b] --> [c] --> endMark
441. //      |
442. //      [x] --> endMark
443. //creates trie, returns true if the trie we are creating is a segment of a string
444. //to only create a trie remove lines which are comment marked
445.
446. bool create(char str[], int len, node *current) {
447.     for(int i = 0; i < len; i++) {
448.         int pos = str[i] - '0';
449.         if(current->next[pos] == NULL)
450.             current->next[pos] = new node();
451.         current = current->next[pos];
452.         if(current->isEnd) //
453.             return true; //
454.     }
455.     current->isEnd = true; //

```



```

456.     return false;           //
457. }
458.
459. void del(node *current) {
460.     for(int i = 0; i < 10; i++)
461.         if(current->next[i] != NULL)
462.             del(current->next[i]);
463.     delete current;
464. }
465.
466. void check(node *current) {
467.     for(int i = 0; i < 10; i++) {
468.         if(current->next[i] != NULL)
469.             check(current->next[i]);
470.     }
471.     if(found) return;
472.     if(current->isEnd && !found) {
473.         for(int i = 0; i < 10 && !found; i++)
474.             if(current->next[i] != NULL) {
475.                 found = 1;
476.             }
477.     }
478. // NON-Dynamic implementation
479. // root node is at 0 index of tree
480. // root node counter contains total number of string insertion
481. // each inserted char counter is on the child node of the edges
482.
483. struct Trie {
484.     struct node {
485.         int cnt;
486.         int nxt[60];
487.     };
488.     int nodes;
489.     node tree[MAX];
490.     void newNode() {
491.         tree[nodes].cnt = 0;
492.         memset(tree[nodes].nxt, -1, sizeof tree[nodes].nxt);
493.         ++nodes;
494.     }
495.     void init() {
496.         nodes = 0;
497.         newNode();
498.     }
499.     int getId(char x) {
500.         if(x >= 'A' and x <= 'Z')
501.             return (x - 'A' + 27);
502.         return (x - 'a' + 1);
503.     }
504.     void insert(string &str, int len = 0, int idx = 0) {
505.         tree[idx].cnt++;
506.         if(len == str.size()) return;
507.         int id = getId(str[len]);
508.         if(tree[idx].nxt[id] == -1) {
509.             tree[idx].nxt[id] = nodes;
510.             newNode();
511.         }
512.         insert(str, len+1, tree[idx].nxt[id]);
513.     }

```

```
514.     int search(string &str, int len = 0, int idx = 0) {
515.         if(len == str.size())
516.             return -2;
517.         int id = getId(str[len]);
518.         if(tree[idx].nxt[id] == -1)
519.             return -1;
520.         if(tree[idx].cnt == 1)
521.             return len;
522.         return search(str, len+1, tree[idx].nxt[id]);
523.     };
```