```
// another fine solution by misof
#include <algorithm>
#include <numeric>

#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <queue>
#include <set>
#include <map>
#include <stack>

#include <cstdio>
#include <cstdlib>
#include <cctype>
#include <cassert>

#include <cmath>
#include <complex>
using namespace std;

#define SIZE(t) ((int)((t).size()))

// interval tree class
class interval_tree_vertex {
  public:
  int color; // 0 empty, 1-7 rainbow
  int summary[8];
  void set_color(int c) { color=c; for (int i=0; i<8; ++i) summary[i]=0; }
  interval_tree_vertex() { set_color(0); }
};

class interval_tree {
  int L;
  void insert(int lo, int hi, int color, int kde, int left, int length);
  int count(int lo, int hi, int color, int kde, int left, int length);
  public:
  vector<interval_tree_vertex> data;
  interval_tree(int N);
  void insert(int lo, int hi, int color);
  int count(int lo, int hi, int color);
};

interval_tree::interval_tree(int N) {
  for (int i=1; ; i*=2) if (i>N+2) { L=i; break; }
  data.resize(2*L);
}

void interval_tree::insert(int lo, int hi, int color) { insert(lo,hi,color,1,0,L); }
int interval_tree::count(int lo, int hi, int color) { return count(lo,hi,color,1,0,L); }

void interval_tree::insert(int lo, int hi, int color, int kde, int left, int length) {
  if (hi <= left || lo >= left+length) return; // mimo
  if (lo <= left && left+length <= hi) { // cely dnu
    data[kde].set_color(color);
    data[kde].summary[color] = length;
    return;
  }
  if (data[kde].color != 0) {
```

```
    int cc = data[kde].color;
    data[2*kde].set_color(cc);           data[2*kde+1].set_color(cc);
    data[2*kde].summary[cc] = length/2;  data[2*kde+1].summary[cc] = length/2;
  }
  data[kde].set_color(0);
  insert(lo,hi,color,2*kde,left,length/2);
  insert(lo,hi,color,2*kde+1,left+length/2,length/2);
  for (int i=0; i<8; ++i) data[kde].summary[i] += data[2*kde].summary[i];
  for (int i=0; i<8; ++i) data[kde].summary[i] += data[2*kde+1].summary[i];
}

int interval_tree::count(int lo, int hi, int color, int kde, int left, int length) {
  if (hi <= left || lo >= left+length) return 0; // mimo
  if (lo <= left && left+length <= hi) return data[kde].summary[color]; // cely dnu
  if (data[kde].color != 0) {
    int cc = data[kde].color;
    data[2*kde].set_color(cc);           data[2*kde+1].set_color(cc);
    data[2*kde].summary[cc] = length/2;  data[2*kde+1].summary[cc] = length/2;
  }
  return count(lo,hi,color,2*kde,left,length/2) + count(lo,hi,color,2*kde+1,left+length/2,length/2);
}

// the original tree
int N;
vector<vector<int> > G;

// rooted tree as parent/child edges
vector<vector<int> > children;
vector<int> parent;

// vertex processing times for the DFS
vector<int> time_in, time_out;

// heavy-light decomposition of the tree into paths
vector< vector<int> > paths;
vector<int> path_id, path_offset;

// an interval tree for each path
vector<interval_tree> trees;

void load() {
  cin >> N;
  G.clear(); G.resize(N);
  for (int i=0; i<N-1; ++i) {
    int x,y;
    cin >> x >> y;
    G[x].push_back(y);
    G[y].push_back(x);
  }
}

void dfs() {
  parent.clear(); parent.resize(N);
  children.clear(); children.resize(N);
  time_in.clear(); time_in.resize(N);
  time_out.clear(); time_out.resize(N);
  paths.clear();
  vector<bool> visited(N,false);
  vector<int> walk;
  vector<int> subtree_size(N,0);
```

```
  int time = 0;

  // run the DFS to compute lots of information
  stack<int> vertex, edge;
  visited[0]=true; time_in[0]=time; parent[0]=0;
  vertex.push(0); edge.push(0);
  while (!vertex.empty()) {
    ++time;
    int kde = vertex.top(); vertex.pop();
    int e = edge.top(); edge.pop();
    if (e == SIZE(G[kde])) {
      walk.push_back(kde);
      time_out[kde] = time;
      subtree_size[kde] = 1;
      for (int i=0; i<SIZE(children[kde]); ++i) subtree_size[kde] += subtree_size[children[kde][i]];
    } else {
      vertex.push(kde); edge.push(e+1);
      int kam = G[kde][e];
      if (!visited[kam]) {
        visited[kam]=true; time_in[kam]=time; parent[kam]=kde; children[kde].push_back(kam);
        vertex.push(kam); edge.push(0);
      }
    }
  }

  // compute the heavy-light decomposition
  vector<bool> parent_edge_processed(N,false);
  parent_edge_processed[0] = true;
  for (int i=0; i<SIZE(walk); ++i) {
    int w = walk[i];
    if (parent_edge_processed[w]) continue;
    vector<int> this_path;
    this_path.push_back(w);
    while (1) {
      bool is_parent_edge_heavy = (2*subtree_size[w] >= subtree_size[parent[w]]);
      parent_edge_processed[w] = true;
      w = parent[w];
      this_path.push_back(w);
      if (!is_parent_edge_heavy) break;
      if (parent_edge_processed[w]) break;
    }
    paths.push_back(this_path);
  }

  path_id.clear(); path_id.resize(N); path_id[0]=-1;
  path_offset.clear(); path_offset.resize(N);

  for (int i=0; i<SIZE(paths); ++i)
    for (int j=0; j<SIZE(paths[i])-1; ++j) {
      path_id[ paths[i][j] ] = i;
      path_offset[ paths[i][j] ] = j;
    }

  trees.clear();
  for (int i=0; i<SIZE(paths); ++i) trees.push_back( interval_tree( SIZE(paths[i])-1 ) );
}

// return whether x is an ancestor of y
inline bool is_ancestor(int x, int y) {
  return (time_in[y] >= time_in[x] && time_out[y] <= time_out[x]);
```

```
}

// return the number of edges on the x-y path that do NOT have color c
// afterwards, color all edges on the x-y path using the color c
int query(int x, int y, int c) {
  if (x==y) return 0;
  if (is_ancestor(x,y)) return query(y,x,c);
  int p = path_id[x];
  int lo = path_offset[x], hi = SIZE(paths[p])-1;
  if (is_ancestor(paths[p][hi], y)) {
    while (hi-lo > 1) {
      int med = (hi+lo)/2;
      if (is_ancestor(paths[p][med], y)) hi=med; else lo=med;
    }
    lo = path_offset[x]; // keep hi at found value, restore lo
  }
  int result = hi-lo - trees[p].count(lo,hi,c);
  trees[p].insert(lo,hi,c);
  return result + query(paths[p][hi],y,c);
}

string color[7] = {"red","orange","yellow","green","blue","indigo","violet"};
map<string,int> C;

int main() {
  for (int i=0; i<7; ++i) C[color[i]]=i+1;
  int TC; cin >> TC;
  while (TC--) {
    load();
    dfs();
    int Q; cin >> Q;
    vector<long long> totals(8,0);
    while (Q--) {
      int x, y; string c; cin >> x >> y >> c;
      totals[C[c]] += query(x,y,C[c]);
    }
    for (int i=1; i<8; ++i) cout << color[i-1] << " " << totals[i] << endl;
  }
  return 0;
}
// vim: fdm=marker:commentstring=\ \"\ %s:nowrap:autoread
```