by Nathan Pinsker

First, consider any path between two vertices of different colors. There must exist an edge on this path whose endpoints are different colors (otherwise, the entire path would be the same color). If the path comprises two or more edges, then considering just this edge as our path is strictly shorter than the path we started with. It follows that the shortest path must always be a single edge in the graph.

The next observation is to notice that, if vertices i and j are different colors, then ANY path from i to j must have an edge with two differently-colored vertices at some point. How do we know that the edge from i to j is actually the shortest path in the graph? At the very least, we need to know that there isn't any path from i to j comprised only of edges with lower weight -- if there was, then we could take some edge on that path and obtain a better solution. The set of edges in the graph that have this property is equal to the minimum spanning tree of the graph. Since any shortest path must be one of these edges, it is an edge of the minimum spanning tree of the graph.

However, each edge in the minimum spanning tree may or may not be usable at each step, depending on whether its vertices are two different colors. We must still test each edge in the minimum spanning tree at each step to determine whether it is usable, and of the usable edges, which is currently the smallest. In order to facilitate this, we need to track, for each vertex, each of the edges that connect to that vertex. We also need to be able to sort them by their weight, and filter by those that share the same color as the color we're considering.

We do this by keeping several heaps at each vertex -- for each possible color, we maintain a heap containing all adjacent vertices of that color. A little care is required when initializing these heaps, so as not to waste too much memory at each vertex. When we update a vertex's color, we remove it from its neighbors' heaps that correspond to that color, and insert it into its neighbors' heaps corresponding to the new color. In order to query a vertex's minimum-length edge, we take the minimum value over all heaps that don't match the vertex's current color (which can be maintained by another heap).

However, this raises a problem: a vertex may have many neighbors, so updating its color may take a long time. We can solve this by rooting our MST, so that each vertex only needs to compute its minimum-weight edge to one of its children. Thus, changing the color of a vertex will only require updates to the heaps located at that vertex and its parent.

Finally, we keep (yet another) global heap. This heap will contain the minimum value from each vertex-local heap, and we can update the global heap when we update each vertex-local heap. This does not change our overall runtime at all, since the cost of updating this heap is $O(\log n)$ per vertex-local heap update. At each step, we need to update $O(1)$ heaps at a cost of $O(\log n)$ per heap, so the total runtime is $O(n \log n)$.

Here is Lewin Gan's solution:

```
import java.io.OutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.util.Arrays;
import java.io.BufferedWriter;
import java.util.HashMap;
import java.util.InputMismatchException;
import java.io.IOException;
import java.util.TreeSet;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;
import java.io.Writer;
import java.io.OutputStreamWriter;
import java.util.Comparator;
import java.io.InputStream;

/**
 * Built using CHelper plug-in
 * Actual solution is at the top
```

```
 */
public class Main {
    public static void main(String[] args) {
        InputStream inputStream = System.in;
        OutputStream outputStream = System.out;
        InputReader in = new InputReader(inputStream);
        OutputWriter out = new OutputWriter(outputStream);
        RainbowGraph solver = new RainbowGraph();
        solver.solve(1, in, out);
        out.close();
    }

    static class RainbowGraph {
        public int n;
        public int m;
        public int k;
        public int q;
        public List<RainbowGraph.Edge>[] graph;
        public int[] color;
        public HashMap<Integer, TreeSet<Integer>>[] mp;
        public TreeSet<Integer> all;
        public int[] pcost;
        public int[] par;
        public final Comparator<Integer> comp = Comparator.comparingLong(x -> ((1L * pcost[x]) << 32L) | x);

        public void solve(int testNumber, InputReader in, OutputWriter out) {
            n = in.nextInt();
            m = in.nextInt();
            k = in.nextInt() + 1;
            q = in.nextInt();
            RainbowGraph.Edge[] e = new RainbowGraph.Edge[m];
            for (int i = 0; i < m; i++) {
                e[i] = new RainbowGraph.Edge(in.nextInt() - 1, in.nextInt() - 1, in.nextInt());
            }
            Arrays.sort(e, Comparator.comparingInt(x -> x.w));
            int[] dj = DisjointSets.createSets(n);
            graph = Stream.generate(ArrayList::new).limit(n).toArray(List[]::new);
            for (int i = 0; i < m; i++) {
                int a = e[i].a, b = e[i].b, w = e[i].w;
                if (DisjointSets.unite(dj, a, b)) {
                    graph[a].add(new RainbowGraph.Edge(b, w, 0));
                    graph[b].add(new RainbowGraph.Edge(a, w, 0));
                }
            }
            color = in.readIntArray(n);
            pcost = new int[n];
            par = new int[n];
            mp = Stream.generate(HashMap::new).limit(n).toArray(HashMap[]::new);
            all = new TreeSet<>(comp);
            pre_dfs(0, -1);
            while (q-- > 0) {
                int node = in.nextInt() - 1;
                int ncolor = in.nextInt();
                if (color[node] != ncolor) {
                    TreeSet<Integer> tmp;
                    // fix parent edge
                    if (par[node] != -1) {
                        // delete pointers to where node was
                        tmp = mp[par[node]].get(color[node]);
                        all.remove(tmp.first());
                        tmp.remove(node);
                        if (color[node] != color[par[node]] && tmp.size() > 0)
                            all.add(tmp.first());

                        // add node in with its new color
                        tmp = mp[par[node]].get(ncolor);
                        if (tmp == null) {
                            mp[par[node]].put(ncolor, tmp = new TreeSet<>(comp));
```

```
                    }
                    if (tmp.size() > 0)
                        all.remove(tmp.first());
                    tmp.add(node);
                    if (ncolor != color[par[node]])
                        all.add(tmp.first());
                }
                // fix children edges
                tmp = mp[node].get(ncolor);
                if (tmp != null && tmp.size() > 0)
                    all.remove(tmp.first());
                tmp = mp[node].get(color[node]);
                if (tmp != null && tmp.size() > 0)
                    all.add(tmp.first());
            }
            color[node] = ncolor;
            out.println(pcost[all.first()]);
        }
    }

    public void pre_dfs(int node, int pp) {
        par[node] = pp;
        for (RainbowGraph.Edge e : graph[node]) {
            if (e.a == pp) continue;
            pcost[e.a] = e.b;
            pre_dfs(e.a, node);
            TreeSet<Integer> xx = mp[node].get(color[e.a]);
            if (xx == null) {
                mp[node].put(color[e.a], xx = new TreeSet<>(comp));
            }
            xx.add(e.a);
        }
        mp[node].forEach((cc, ts) -> {
            if (cc != color[node]) {
                all.add(ts.first());
            }
        });
    }

    static class Edge {
        public int a;
        public int b;
        public int w;

        public Edge(int a, int b, int w) {
            this.a = a;
            this.b = b;
            this.w = w;
        }

    }

}

static class InputReader {
    private InputStream stream;
    private byte[] buf = new byte[1024];
    private int curChar;
    private int numChars;

    public InputReader(InputStream stream) {
        this.stream = stream;
    }

    public int[] readIntArray(int tokens) {
        int[] ret = new int[tokens];
        for (int i = 0; i < tokens; i++) {
            ret[i] = nextInt();
```

```
        }
        return ret;
    }

    public int read() {
        if (this.numChars == -1) {
            throw new InputMismatchException();
        } else {
            if (this.curChar >= this.numChars) {
                this.curChar = 0;

                try {
                    this.numChars = this.stream.read(this.buf);
                } catch (IOException var2) {
                    throw new InputMismatchException();
                }

                if (this.numChars <= 0) {
                    return -1;
                }
            }

            return this.buf[this.curChar++];
        }
    }

    public int nextInt() {
        int c;
        for (c = this.read(); isSpaceChar(c); c = this.read()) {
            ;
        }

        byte sgn = 1;
        if (c == 45) {
            sgn = -1;
            c = this.read();
        }

        int res = 0;

        while (c >= 48 && c <= 57) {
            res *= 10;
            res += c - 48;
            c = this.read();
            if (isSpaceChar(c)) {
                return res * sgn;
            }
        }

        throw new InputMismatchException();
    }

    public static boolean isSpaceChar(int c) {
        return c == 32 || c == 10 || c == 13 || c == 9 || c == -1;
    }

}

static class OutputWriter {
    private final PrintWriter writer;

    public OutputWriter(OutputStream outputStream) {
        writer = new PrintWriter(new BufferedWriter(new OutputStreamWriter(outputStream)));
    }

    public OutputWriter(Writer writer) {
        this.writer = new PrintWriter(writer);
    }
```

```
        public void close() {
            writer.close();
        }

        public void println(int i) {
            writer.println(i);
        }

    }

    static class DisjointSets {
        public static int[] createSets(int size) {
            int[] p = new int[size];
            for (int i = 0; i < size; i++)
                p[i] = i;
            return p;
        }

        public static int root(int[] p, int x) {
            return x == p[x] ? x : (p[x] = root(p, p[x]));
        }

        public static boolean unite(int[] p, int a, int b) {
            a = root(p, a);
            b = root(p, b);
            if (a != b) {
                p[a] = b;
                return true;
            }
            return false;
        }

    }
}
```