DFS:

public class DepthFirstPaths

{

private boolean[] marked;

private int[] edgeTo;

private int s;

public DepthFirstPaths(Graph G, int s)

{

...

dfs(G, s);

}

private void dfs(Graph G, int v)

{

marked[v] = true;

for (int w : G.adj(v))

if (!marked[w])

{

dfs(G, w);

edgeTo[w] = v;

}

}

}

public boolean hasPathTo(int v)

{ return marked[v]; }

public Iterable<Integer> pathTo(int v)

{

if (!hasPathTo(v)) return null;

Stack<Integer> path = new Stack<Integer>();

for (int x = v; x != s; x = edgeTo[x])

path.push(x);

path.push(s);

return path;

}

BFS:

public class BreadthFirstPaths

{

private boolean[] marked;

private int[] edgeTo;

private int[] distTo;

…

private void bfs(Graph G, int s) {

Queue<Integer> q = new Queue<Integer>();

q.enqueue(s);

marked[s] = true;

distTo[s] = 0;

while (!q.isEmpty()) {

int v = q.dequeue();

for (int w : G.adj(v)) {

if (!marked[w]) {

q.enqueue(w);

marked[w] = true;

edgeTo[w] = v;

distTo[w] = distTo[v] + 1;

}

}

}

}

}

CONNECTED COMPONENTS:

public class CC

{

private boolean[] marked;

private int[] id;

private int count;

public CC(Graph G)

{

marked = new boolean[G.V()];

id = new int[G.V()];

for (int v = 0; v < G.V(); v++)

{

if (!marked[v])

{

dfs(G, v);

count++;

}

}

}

public int count()

public int id(int v)

public boolean connected(int v, int w)

private void dfs(Graph G, int v)

}

Frequency counter:

public class FrequencyCounter

{

public static void main(String[] args)

{

int minlen = Integer.parseInt(args[0]);

ST<String, Integer> st = new ST<String, Integer>();

while (!StdIn.isEmpty())

{

String word = StdIn.readString();

if (word.length() < minlen) continue;

if (!st.contains(word)) st.put(word, 1);

else st.put(word, st.get(word) + 1);

}

String max = "";

st.put(max, 0);

for (String word : st.keys())

if (st.get(word) > st.get(max))

max = word;

StdOut.println(max + " " + st.get(max));

}

}

//Symbol tables- equable

public class Equable {

public Equable(Iterable<?> elements) {

this.elements = elements;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Equable equable = (Equable) o;

Iterator<?> thisIterator = elements.iterator();

Iterator<?> thatIterator = equable.elements.iterator();

while (thisIterator.hasNext())

if (thatIterator.hasNext())

if (thisIterator.next().equals(thatIterator.next()))

continue;

else

return false;

else

return false;

return true;

}

@Override

public int hashCode() {

int result = 0;

for (Object element : elements) result = 31 \* result + element.hashCode();

return result;

}

protected final Iterable<?> elements;

}

//----------------------------------------------------------------------------------------------

// frequency counter for string

//----------------------------------------------------------------------------------------------

//Binary search

public Value get(Key key)

{

if (isEmpty()) return null;

int i = rank(key);

if (i < N && keys[i].compareTo(key) == 0) return vals[i];

else return null;

}

private int rank(Key key)

{

int lo = 0, hi = N-1;

while (lo <= hi)

{

int mid = lo + (hi - lo) / 2;

int cmp = key.compareTo(keys[mid]);

if (cmp < 0) hi = mid - 1;

else if (cmp > 0) lo = mid + 1;

else if (cmp == 0) return mid;

}

return lo;

}

//---------------------------------------------------------------------------------------------

//BST search

public Value get(Key key)

{

Node x = root;

while (x != null)

{

int cmp = key.compareTo(x.key);

if (cmp < 0) x = x.left;

else if (cmp > 0) x = x.right;

else if (cmp == 0) return x.val;

}

return null;

}

//---------------------------------------------------------------------------------------------

//BST insert

private Node put(Node x, Key key, Value val)

{

if (x == null) return new Node(key, val);

int cmp = key.compareTo(x.key);

if (cmp < 0)

x.left = put(x.left, key, val);

else if (cmp > 0)

x.right = put(x.right, key, val);

else if (cmp == 0)

x.val = val;

return x;

}

//---------------------------------------------------------------------------------------------

//BST floor

public Key floor(Key key)

{

Node x = floor(root, key);

if (x == null) return null;

return x.key;

}

private Node floor(Node x, Key key)

{

if (x == null) return null;

int cmp = key.compareTo(x.key);

if (cmp == 0) return x;

if (cmp < 0) return floor(x.left, key);

Node t = floor(x.right, key);

if (t != null) return t;

else return x;

}

//---------------------------------------------------------------------------------------------

//BST rank

public int rank(Key key)

{ return rank(key, root); }

private int rank(Key key, Node x)

{

if (x == null) return 0;

int cmp = key.compareTo(x.key);

if (cmp < 0) return rank(key, x.left);

else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);

else if (cmp == 0) return size(x.left);

}

//In order traversal

public Iterable<Key> keys()

{

Queue<Key> q = new Queue<Key>();

inorder(root, q);

return q;

}

private void inorder(Node x, Queue<Key> q)

{

if (x == null) return;

inorder(x.left, q);

q.enqueue(x.key);

inorder(x.right, q);

}

//---------------------------------------------------------------------------------------------

//delete minimum

public void deleteMin()

{ root = deleteMin(root); }

private Node deleteMin(Node x)

{

if (x.left == null) return x.right;

x.left = deleteMin(x.left);

x.count = 1 + size(x.left) + size(x.right);

return x;

}

//---------------------------------------------------------------------------------------------

//Hibbard deletion

public void delete(Key key)

{ root = delete(root, key); }

private Node delete(Node x, Key key) {

if (x == null) return null;

int cmp = key.compareTo(x.key);

if (cmp < 0) x.left = delete(x.left, key);

else if (cmp > 0) x.right = delete(x.right, key);

else {

if (x.right == null) return x.left;

if (x.left == null) return x.right;

Node t = x;

x = min(t.right);

x.right = deleteMin(t.right);

x.left = t.left;

}

x.count = size(x.left) + size(x.right) + 1;

return x;

}

//---------------------------------------------------------------------------------------------

//identical BST functions =for red black

//hash table – separate chaining

public Value get(Key key) {

int i = hash(key);

for (Node x = st[i]; x != null; x = x.next)

if (key.equals(x.key)) return (Value) x.val;

return null;

}

public void put(Key key, Value val) {

int i = hash(key);

for (Node x = st[i]; x != null; x = x.next)

if (key.equals(x.key)) { x.val = val; return; }

st[i] = new Node(key, val, st[i]);

}

//---------------------------------------------------------------------------------------------

//hash table – linear probing

//---------------------------------------------------------------------------------------------

//DFS

public class DepthFirstPaths

{

private boolean[] marked;

private int[] edgeTo;

private int s;

public DepthFirstPaths(Graph G, int s)

{

...

dfs(G, s);

}

private void dfs(Graph G, int v)

{

marked[v] = true;

for (int w : G.adj(v))

if (!marked[w])

{

dfs(G, w);

edgeTo[w] = v;

}

}

}

//DFS : path to

public boolean hasPathTo(int v)

{ return marked[v]; }

public Iterable<Integer> pathTo(int v)

{

if (!hasPathTo(v)) return null;

Stack<Integer> path = new Stack<Integer>();

for (int x = v; x != s; x = edgeTo[x])

path.push(x);

path.push(s);

return path;

}

//-----------------------------------------------------------------

//BFS

public class BreadthFirstPaths

{

private boolean[] marked;

private int[] edgeTo;

private int[] distTo;

…

private void bfs(Graph G, int s) {

Queue<Integer> q = new Queue<Integer>();

q.enqueue(s);

marked[s] = true;

distTo[s] = 0;

while (!q.isEmpty()) {

int v = q.dequeue();

for (int w : G.adj(v)) {

if (!marked[w]) {

q.enqueue(w);

marked[w] = true;

edgeTo[w] = v;

distTo[w] = distTo[v] + 1;

}

}

}

}

}

//----------------------------------------------------------------------

//find connected components in DFS

public class CC

{

private boolean[] marked;

private int[] id;

private int count;

public CC(Graph G)

{

marked = new boolean[G.V()];

id = new int[G.V()];

for (int v = 0; v < G.V(); v++)

{

if (!marked[v])

{

dfs(G, v);

count++;

}

}

}

public int count()

public int id(int v)

public boolean connected(int v, int w)

private void dfs(Graph G, int v)

}

public int count()

{ return count; }

public int id(int v)

{ return id[v]; }

public boolean connected(int v, int w)

{ return id[v] == id[w]; }

private void dfs(Graph G, int v)

{

marked[v] = true;

id[v] = count;

for (int w : G.adj(v))

if (!marked[w])

dfs(G, w);

}

//--------------------------------------------------------------------------------

//topological

public class DepthFirstOrder

{

private boolean[] marked;

private Stack<Integer> reversePostorder;

public DepthFirstOrder(Digraph G)

{

reversePostorder = new Stack<Integer>();

marked = new boolean[G.V()];

for (int v = 0; v < G.V(); v++)

if (!marked[v]) dfs(G, v);

}

private void dfs(Digraph G, int v)

{

marked[v] = true;

for (int w : G.adj(v))

if (!marked[w]) dfs(G, w);

reversePostorder.push(v);

}

public Iterable<Integer> reversePostorder()

{ return reversePostorder; }

}

public Value get(Key key)

{

for (int i = hash(key); keys[i] != null; i = (i+1) % M)

if (key.equals(keys[i]))

return vals[i];

return null;

}

public void put(Key key, Value val)

{

int i;

for (i = hash(key); keys[i] != null; i = (i+1) % M)

if (keys[i].equals(key))

break;

keys[i] = key;

vals[i] = val;

}

public class FrequencyCounter

{

public static void main(String[] args)

{

int minlen = Integer.parseInt(args[0]);

ST<String, Integer> st = new ST<String, Integer>();

while (!StdIn.isEmpty())

{

String word = StdIn.readString();

if (word.length() < minlen) continue;

if (!st.contains(word)) st.put(word, 1);

else st.put(word, st.get(word) + 1);

}

String max = "";

st.put(max, 0);

for (String word : st.keys())

if (st.get(word) > st.get(max))

max = word;

StdOut.println(max + " " + st.get(max));

}

}