```
#include <set>
#include <list>
#include <vector>
#include <fstream>
#include <iostream>
#include <string>

// Written by Eric Wasiolek Copyright 2008

using namespace std;

class neuron{
	private:
		string name;
		string lineage;
		string maintype;
		string subtype1;
		string subtype2;
		set<string> genes;

	public:
		neuron()
		{}
		void setName(string x)
		{
			name = x;
		}
```

```
void setLineage(string x)
{
	lineage = x;
}
void setMainType(string x)
{
	maintype = x;
}
void setSubType1(string x)
{
	subtype1 = x;
}
void setSubType2(string x)
{
	subtype2 = x;
}
void addGene(string gene)
{
	genes.insert(gene);
}
string getName()
{
	return name;
}
string getLineage()
{
	return lineage;
}
string getMainType()
```

```
		{
			return maintype;
		}
		string getSubType1()
		{
			return subtype1;
		}
		string getSubType2()
		{
			return subtype2;
		}
		set<string> getGenes()
		{
			return genes;
		}
	};


// SET FUNCTION IMPLEMENTATIONS

template <typename T>
bool operator== (const set<T>& lhs, const set<T>& rhs)
{
	typename set<T>::const_iterator myself = lhs.begin(), other = rhs.begin();

	// return false if the sets do not have the same size
	if (lhs.size() == rhs.size())
	{
		// compare until encounter end of the sets or
```

```
                // find two elements that are not equal
                while (myself != lhs.end() && *myself++ == *other++);

                // if we left the loop before reaching the end
                // of the sets, they are not equal
                if (myself != lhs.end())
                        return false;
                else
                        return true;
        }
        else
                return false;
}

template <typename T>
set<T> operator+ (const set<T>& lhs, const set<T>& rhs)
{
        // constuct union
        set<T> setUnion;
        // iterators that traverse the sets
        typename set<T>::const_iterator lhsIter = lhs.begin(), rhsIter = rhs.begin();

        // move forward as long as we have not reached the end of
        // either set
        while (lhsIter != lhs.end() && rhsIter != rhs.end())
                if (*lhsIter < *rhsIter)
                        // *lhsIter belongs to the union. insert and
                        // move iterator forward
                        setUnion.insert(*lhsIter++);
```

```
            else if (*rhsIter < *lhsIter)
                // *rhsIter belongs to the union. insert and
                // move iterator forward
                setUnion.insert(*rhsIter++);
            else
            {
                // the two values are equal. insert just one and
                // move both itertors forward
                setUnion.insert(*lhsIter++);
                rhsIter++;
            }

            // flush any remaining items
            if (lhsIter != lhs.end())
                while (lhsIter != lhs.end())
                    setUnion.insert(*lhsIter++);
            else if (rhsIter != rhs.end())
                while (rhsIter != rhs.end())
                    setUnion.insert(*rhsIter++);

        return setUnion;
}

template <typename T>
set<T> operator* (const set<T>& lhs, const set<T>& rhs)
{
        // constuct intersection
        set<T> setIntersection;
        // iterators that traverse the sets
```

```
	typename set<T>::const_iterator lhsIter = lhs.begin(), rhsIter = rhs.begin();

	// move forward as long as we have not reached the end of
	// either set
	while (lhsIter != lhs.end() && rhsIter != rhs.end())
		if (*lhsIter < *rhsIter)
			// *lhsIter is in lhs and not in rhs. move iterator
			// forward
			lhsIter++;
		else if (*rhsIter < *lhsIter)
			// *rhsIter is in rhs and not in lhs. move iterator
			// forward
			rhsIter++;
		else
		{
			// the same value is in both sets. insert one value
			// and move the iterators forward
			setIntersection.insert(*lhsIter);
			lhsIter++;
			rhsIter++;
		}

		return setIntersection;
}

template <typename T>
set<T> operator- (const set<T>& lhs, const set<T>& rhs)
{
	// construct difference
```

```
set<T> setDifference;
// iterators that traverse the sets
typename set<T>::const_iterator lhsIter = lhs.begin(), rhsIter = rhs.begin();

// move forward as long as we have not reached the end of
// either set
while (lhsIter != lhs.end() && rhsIter != rhs.end())
        if (*lhsIter < *rhsIter)
                // *lhsIter belongs to lhs but not to rhs. put it in
                // the difference
                setDifference.insert(*lhsIter++);
        else if (*rhsIter < *lhsIter)
                // *rhsIter is in the rhs but not in the lhs. pass
                // over it
                rhsIter++;
        else
        {
                // the same value is in both sets. move the
                // iterators forward
                lhsIter++;
                rhsIter++;
        }

        // flush any remaining items from lhs
        if (lhsIter != lhs.end())
                while (lhsIter != lhs.end())
                        setDifference.insert(*lhsIter++);

return setDifference;
```

```
}

set<string> findGeneSet(string neuronname, neuron neuron[], int numneurons)
{
	for(int i=0; i < numneurons; ++i)
		if(neuronname == neuron[i].getName())
			return neuron[i].getGenes();
}

bool find(string genename, set<string>& geneset)
{
	set<string>::iterator geneiter, endgenes;
	geneiter = geneset.begin();
	endgenes = geneset.end();
	int counter = 0;
	if(geneset.empty())
		return false;
	while(geneiter != geneset.end())
	{
		++counter;
		if(genename == *geneiter)
			return true;
		else if(counter == geneset.size())
			return false;
		++geneiter;
	}
}

template<typename Iterator>
```

```
void writeContainer(Iterator first, Iterator last, const string& seperator = " ")
{
	Iterator iter = first;
	while(iter != last)
	{
		cout << *iter << seperator;
		iter++;
	}
}

void writeSet(set<string> s)
{
	set<string>::iterator siter = s.begin(), esiter = s.end();

	while(siter != esiter)
	{
		cout << *siter << " ";
		siter++;
	}
	cout << endl;
}

void writeBVector(vector<bool>& bvect)
{
	int i = 0;
	for(i = 0; i < bvect.size(); ++i)
		cout << bvect[i];
}
```

```
int main(int argc, char *argv[])
{
	// cout << "Entered main" << endl;
	ifstream neurons;
	string inputneuron, outputneuron;



	if(argc != 2)
	{
		cout << "Usage: bitgenes graphname " << endl;
		return 0;
	}
	else
	{
		neurons.open(argv[1]);
		if(!neurons)
		{
			cout << "Could not open graph file!" << endl;
			return 0;
		}
		else
			cout << "Opened graph file " << argv[1] << " successfully." << endl;
	}



	int numneurons = 0;
	string gene;
```

```
vector<neuron> vneurons(numneurons);
neurons >> numneurons;
string temp;
neuron n[numneurons];
int i = 0;
for(i = 0; i < numneurons; ++i)
{
	neurons >> temp;
	n[i].setName(temp);
	neurons >> temp;
	n[i].setLineage(temp);
	neurons >> temp;
	n[i].setMainType(temp);
	neurons >> temp;
	n[i].setSubType1(temp);
	neurons >> temp;
	n[i].setSubType2(temp);
	while(1)
	{
		neurons >> gene;
			if(gene == "end")
				break;
		n[i].addGene(gene);
	}
}


vector<set<string> > vs;
```

```
list<string>::iterator liter;
set<string>::iterator fsiter;
set<string>::iterator esiter;

// Create a vector of all of the gene sets
for(i = 0; i < numneurons; ++i)
{
        vs.push_back(n[i].getGenes());
}

// Create a union set with all of the unique genes in it
set<string> us;

us = vs[0];

for(i=0; i < vs.size(); ++i)
        us = us + vs[i];

cout << "There are " << us.size() << " neuron specific genes in C. Elegans" << endl;
cout << "And here are the genes: " << endl;
fsiter = us.begin();
esiter = us.end();
writeSet(us);

// Create bit strings showing which neurons have which genes
vector< vector<bool> > bvect(numneurons);

for(i = 0; i < numneurons; ++i)
{
```

```
		// cout << "Diag:  Bitstring for neuron " << i << endl;
		while(fsiter != esiter)
		{
			// The essence of this algorithm:
			if(find(*fsiter, vs[i]))  // if you find the gene in the geneset iterated by fsiter in the set of genes per neuron
								// vector of sets vs, set it to 1 else 0
				bvect[i].push_back(true);
			else
				bvect[i].push_back(false);
			++fsiter;
		}
		fsiter = us.begin();
	}

	// Print out bit vectors
	for(i = 0; i < numneurons; ++i)
	{
		cout << "Neuron: " << n[i].getName() << " Bitstring: ";
		writeBVector(bvect[i]);
		cout << endl;
	}



	return 0;
}
```