```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main()
{
        //Encodes the following message as "CEO EUWH Ejcrvgt Twngu!"
        //by replacing each letter by the letter 2 places after it
        //with loopback (Z gets replaced by B, etc).
        //The main purpose is to show that characters in strings can
        //be interpreted as integers and used in arithmetic.
        string message = "ACM CSUF Chapter Rules!";
        int offset = 2;

        for (int i=0; i<message.length(); i++)
        {
                if ( isupper(message[i]) )
                {
                        message[i] = (message[i] - 'A' + offset)%26 + 'A';
                }
                else if ( islower(message[i]) )
                {
                        message[i] = (message[i] - 'a' + offset)%26 + 'a';
                }
        }

        cout << message << endl;

        system("pause");
        return 0;
}
```

```
// Authors:  Dustin & Fidel
#include <iostream>
#include "graph.hh"
#include <vector>
#include <memory>
#include <queue>

using namespace std;

// modify this method to do w/e computations are desired on
// the visited node
void visit(int node){ cout << node << endl; };


// Breadth First Search traversal
//   @arg - Graph : a graph
//   @arg - start_node : where to start
//   @return - a pointer to an ordered vector of visited nodes
void BFS(Graph g, int start_node){
        vector<bool> visited(g.n());
        fill(visited.begin(), visited.end(), false);
        queue<int> q;
        vector<int>* result;


        visit(start_node);
        visited[start_node] = true;
        q.push(start_node);


        while(!q.empty()){
                int current = q.front();
                q.pop();

                for( int i =0; i < g.adj[current].size(); i++){
                        int end = g.adj[current][i]->opposite(current);
                        if(visited[end]==false){
                                visit(end);
                                visited[end] = true;
                                q.push(end);
                        }
                }

        }
};



int main(){
    // example from
    // http://en.wikipedia.org/wiki/Kruskal%27s_algorithm#Example
    const int A=0, B=1, C=2, D=3, E=4, F=5, G=6;
    Graph g(7);
    g.add_edge(A, B);  // 0
    g.add_edge(A, D);  // 1
    g.add_edge(B, C);  // 2
    g.add_edge(B, D);  // 3
    g.add_edge(B, E);  // 4
    g.add_edge(C, E);  // 5
    g.add_edge(D, E);  // 6
    g.add_edge(D, F);  // 7
    g.add_edge(E, F);  // 8
    g.add_edge(E, G);  // 9
    g.add_edge(F, G);  // 10


        // Start BFS
        int start_vertex = 0;
        vector<int>* result;
```



```
        BFS(g, start_vertex);


    return 0;
}
```

```
// Author: Dustin Delmer
// Date: 2/2/2013
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main(){
        // initialize vector and fill
        vector<int> values;
        values.push_back(1);
        values.push_back(2);
        values.push_back(3);
        values.push_back(4);
        values.push_back(5);

        // create a bool vector of the same length to mark "active" elements
        vector<bool> active(5);

        // turn on n-k bits in bool vector, off bits will represent active eleme
nts
        int num_to_choose = 3;
        fill(active.begin() + num_to_choose, active.end(), true);

        // loop for generating combinations
        do {
                // if current index is "off" print value at index
                /* alternatively you could push these values into a temp vector
and
                apply your desired computations to the temp vector, clearing the
                vector between each iteration */
                for(int i=0; i<active.size(); i++){
                        if(!active[i]) { cout << values[i] << ' '; }
                }
                cout << endl;
        /* next_permutation generates permutations lexicographically.  Each time

        next_permutation is called it shifts the elements of the argument
        vector to generate the next "largest" order.  In our bool vector 00111 i
s
        the "smallest" permutation, while 11100 is the largest.  This loop will
        end     up generating exactly n choose k combinations */
        }while(next_permutation(active.begin(), active.end()));

        return 0;
}
```

```
#include <iostream>
#include <stdio.h>
#include <set>
#include <queue>
#include "graph.hh"

const int A = 0;
const int B = 1;
const int C = 2;
const int D = 3;
const int E = 4;
const int F = 5;
const int G = 6;
const int H = 7;
const int I = 8;

bool depthFirstSearch(Graph * g, int startingNode, int endingNode)
{
        set <int> visitedSet;
        queue <int> edgeQueue;
        edgeQueue.push(startingNode);
        do
        {
                int currentVertex= edgeQueue.front();
                edgeQueue.pop();
                printf("Current vertex = %d\n", currentVertex);
                //Modify here
                if (currentVertex == endingNode)
                {
                        return true;
                }
                visitedSet.insert(currentVertex);
                for (unsigned int i = 0; i < g->adj[currentVertex].size(); i++)
                {
                        int vert = g->adj[currentVertex][i]->opposite(currentVer
tex);
                        if (visitedSet.find(vert) == visitedSet.end())
                        {
                                edgeQueue.push(vert);
                        }
                }
        }
        while (!edgeQueue.empty());

        return false;
}

int main()
{
        Graph * g = new Graph(9);
        g->add_edge(A,B,false,5);               //0
        g->add_edge(B,C,false,12);              //1
        g->add_edge(C,D,false,6);               //2
        g->add_edge(D,E,false,8);               //3
        g->add_edge(E,F,false,15);              //4
        g->add_edge(F,G,false,6);               //5
        g->add_edge(G,H,false,8);               //6
        g->add_edge(H,I,false,3);               //7
        g->add_edge(A,F,false,15);              //8
        g->add_edge(D,G,false,6);               //9
        g->add_edge(F,H,false,8);               //10
        g->add_edge(C,I,false,3);               //11

        if (depthFirstSearch(g, A, H))
        {
                printf("True\n");
        }
        else
        {
```



```
                printf("False\n");
        }

        if (depthFirstSearch(g, A, 64))
        {
                printf("True\n");
        }
        else
        {
                printf("False\n");
        }
//      for (int j = 0; j < 9; j++)
//      {
//              for (unsigned int i = 0; i < g->adj[j].size(); i++)
//              {
//                      printf("g->adj[%d][%d]->index=%d\n",j, i,g->adj[j][i]->i
ndex);
//              }
//      }
        return 0;
}
```

```
// Dijkstras.cpp : Defines the entry point for the console application.
//

#include "stdio.h"
#include <iostream>
#include <vector>

using namespace std;

void DijkstrasTest();

int main(int argc, char** argv[])
{
	DijkstrasTest();
	return 0;
}


//////////////

class Node;
class Edge;

void Dijkstras();
vector<Node*>* AdjacentRemainingNodes(Node* node);
Node* ExtractSmallest(vector<Node*>& nodes);
int Distance(Node* node1, Node* node2);
bool Contains(vector<Node*>& nodes, Node* node);
void PrintShortestRouteTo(Node* destination);

vector<Node*> nodes;
vector<Edge*> edges;

class Node
{
public:
	Node(char id)
		: id(id), previous(NULL), distanceFromStart(INT_MAX)
	{
		nodes.push_back(this);
	}
public:
	char id;
	Node* previous;
	int distanceFromStart;
};


class Edge
{
public:
	Edge(Node* node1, Node* node2, int distance)
		: node1(node1), node2(node2), distance(distance)
	{
		edges.push_back(this);
	}
	bool Connects(Node* node1, Node* node2)
	{
		return (
			(node1 == this->node1 &&
			node2 == this->node2) ||
			(node1 == this->node2 &&
			node2 == this->node1));
	}
public:
	Node* node1;
	Node* node2;
	int distance;
};
```



```

//////////////////
void DijkstrasTest()
{
	Node* a = new Node('a');
	Node* b = new Node('b');
	Node* c = new Node('c');
	Node* d = new Node('d');
	Node* e = new Node('e');
	Node* f = new Node('f');
	Node* g = new Node('g');

	Edge* e1 = new Edge(a, c, 1);
	Edge* e2 = new Edge(a, d, 2);
	Edge* e3 = new Edge(b, c, 2);
	Edge* e4 = new Edge(c, d, 1);
	Edge* e5 = new Edge(b, f, 3);
	Edge* e6 = new Edge(c, e, 3);
	Edge* e7 = new Edge(e, f, 2);
	Edge* e8 = new Edge(d, g, 1);
	Edge* e9 = new Edge(g, f, 1);

	a->distanceFromStart = 0; // set start node
	Dijkstras();
	PrintShortestRouteTo(f);

	// TODO: Node / Edge memory cleanup not included
}

//////////////////

void Dijkstras()
{
	while (nodes.size() > 0)
	{
		Node* smallest = ExtractSmallest(nodes);
		vector<Node*>* adjacentNodes =
			AdjacentRemainingNodes(smallest);

		const int size = adjacentNodes->size();
		for (int i=0; i<size; ++i)
		{
			Node* adjacent = adjacentNodes->at(i);
			int distance = Distance(smallest, adjacent) +
				smallest->distanceFromStart;

			if (distance < adjacent->distanceFromStart)
			{
				adjacent->distanceFromStart = distance;
				adjacent->previous = smallest;
			}
		}
		delete adjacentNodes;
	}
}

// Find the node with the smallest distance,
// remove it, and return it.
Node* ExtractSmallest(vector<Node*>& nodes)
{
	int size = nodes.size();
	if (size == 0) return NULL;
	int smallestPosition = 0;
	Node* smallest = nodes.at(0);
	for (int i=1; i<size; ++i)
	{
		Node* current = nodes.at(i);
		if (current->distanceFromStart <
			smallest->distanceFromStart)
```

```
                {
                        smallest = current;
                        smallestPosition = i;
                }
        }
        nodes.erase(nodes.begin() + smallestPosition);
        return smallest;
}

// Return all nodes adjacent to 'node' which are still
// in the 'nodes' collection.
vector<Node*>* AdjacentRemainingNodes(Node* node)
{
        vector<Node*>* adjacentNodes = new vector<Node*>();
        const int size = edges.size();
        for(int i=0; i<size; ++i)
        {
                Edge* edge = edges.at(i);
                Node* adjacent = NULL;
                if (edge->node1 == node)
                {
                        adjacent = edge->node2;
                }
                else if (edge->node2 == node)
                {
                        adjacent = edge->node1;
                }
                if (adjacent && Contains(nodes, adjacent))
                {
                        adjacentNodes->push_back(adjacent);
                }
        }
        return adjacentNodes;
}

// Return distance between two connected nodes
int Distance(Node* node1, Node* node2)
{
        const int size = edges.size();
        for(int i=0; i<size; ++i)
        {
                Edge* edge = edges.at(i);
                if (edge->Connects(node1, node2))
                {
                        return edge->distance;
                }
        }
        return -1; // should never happen
}

// Does the 'nodes' vector contain 'node'
bool Contains(vector<Node*>& nodes, Node* node)
{
        const int size = nodes.size();
        for(int i=0; i<size; ++i)
        {
                if (node == nodes.at(i))
                {
                        return true;
                }
        }
        return false;
}

//////////////////

void PrintShortestRouteTo(Node* destination)
{
        Node* previous = destination;
```



```
        cout << "Distance from start: "
                << destination->distanceFromStart << endl;
        while (previous)
        {
                cout << previous->id << " ";
                previous = previous->previous;
        }
        cout << endl;
}



// these two not needed
vector<Edge*>* AdjacentEdges(vector<Edge*>& Edges, Node* node);
void RemoveEdge(vector<Edge*>& Edges, Edge* edge);

vector<Edge*>* AdjacentEdges(vector<Edge*>& edges, Node* node)
{
        vector<Edge*>* adjacentEdges = new vector<Edge*>();

        const int size = edges.size();
        for(int i=0; i<size; ++i)
        {
                Edge* edge = edges.at(i);
                if (edge->node1 == node)
                {
                        cout << "adjacent: " << edge->node2->id << endl;
                        adjacentEdges->push_back(edge);
                }
                else if (edge->node2 == node)
                {
                        cout << "adjacent: " << edge->node1->id << endl;
                        adjacentEdges->push_back(edge);
                }
        }
        return adjacentEdges;
}

void RemoveEdge(vector<Edge*>& edges, Edge* edge)
{
        vector<Edge*>::iterator it;
        for (it=edges.begin(); it<edges.end(); ++it)
        {
                if (*it == edge)
                {
                        edges.erase(it);
                        return;
                }
        }
}
```

```
#ifndef DIJKSTRAS
#define DIJKSTRAS

#include "graph.hh"
#include <vector>
using namespace std;

struct TempNode
{
        bool in_set;
        int number;
        int came_from;
        double path_length;
};

Path Dijkstras(Graph grph, int start, int end, double &path_length)
{
    path_length = 0;
    vector<TempNode> verticies;
        int count = 0, min_dist = 32000000, min_index = 0;
        Path path;

    TempNode temp;
    for (int i=0; i<grph.n(); i++)
     {
                temp.number = i;
                temp.in_set = true;
                if (i == start)
                {
                        temp.path_length = 0;
                }
                else
                {
                        temp.path_length = 32000000;
                }
                verticies.push_back(temp);
     }

        count = verticies.size();
    while (count > 0)
    {
                min_dist = 32000000;
                min_index = 0;
                for (int i=0; i<verticies.size(); i++)
                {
                        if (verticies[i].path_length < min_dist && verticies[i].
in_set == true)
                        {
                                min_dist = verticies[i].path_length;
                                min_index = i;
                        }
                }

                if (min_index == end)
                {
                        path_length = verticies[min_index].path_length;
                        break;
                }
                verticies[min_index].in_set = false;
                count--;

                for (int i=0; i<grph.edges.size(); i++)
                {
                        if (grph.edges[i]->v == min_index)
                        {

                                path_length = verticies[grph.edges[i]->w].path_l
ength;
                                if (verticies[grph.edges[i]->w].path_length > ve
```



```
rticies[min_index].path_length + grph.edges[i]->weight)
                                {
                                        verticies[grph.edges[i]->w].path_length
= verticies[min_index].path_length + grph.edges[i]->weight;
                                        verticies[grph.edges[i]->w].came_from =
min_index;
                                }
                        }
                }
    }

        int a = end;
        int b = 0;
        do
        {
                b = verticies[a].came_from;
                path = path.extend(*grph.connection(a,b));
                a = b;
        }while (verticies[a].path_length > 0);

        return path;
}

#endif
```

```
#include <iostream>
#include "graph.hh"
#include "dijkstras.h"
using namespace std;

enum nodes {A, B, C, D, E, F, G};

int main()
{
        double length;

        Graph graph(7);
        graph.add_edge(A,B,false,7);
        graph.add_edge(A,D,false,5);
        graph.add_edge(B,C,false,8);
        graph.add_edge(B,D,false,9);
        graph.add_edge(B,E,false,7);
        graph.add_edge(C,E,false,5);
        graph.add_edge(D,F,false,6);
        graph.add_edge(E,G,false,9);
        graph.add_edge(E,F,false,8);
        graph.add_edge(F,G,false,11);

        Dijkstras(graph, A, G, length);
        cout << length << endl;

        system("pause");
        return 0;
}
```

```
//
// Disjoint sets data structure, as described in
// http://en.wikipedia.org/wiki/Disjoint-set_data_structure
//
// Maintains a partition of elements into disjoint (non-overlapping)
// subsets. The universe of elements in the n integers
// 0...n-1. Initially each integer is in its own singleton set. The
// data structure supports two operations:
//
// Find(x): return a "representative" for the set containing x. The
// representative is a designated element of that set.
//
// Merge(x, y): combine the set containing x with the set containing y
// so that they are now both in the same set.
//
// (Other sources call this operations "union", but that is a C++
// reserved word, so we use "merge" here.)
//
// For example, after DisjointSets(5) we have 5 singleton sets:
//
// {0} {1} {2} {3} {4}
//
// after Merge(1, 2):
//
// {0} {1, 2} {3} {4}
//
// after Merge(0, 3):
//
// {0, 3} {1, 2} {4}
//
// after Merge(2, 4):
//
// {0, 3} {1, 2, 4}
//

#ifndef _DISJOINT_SETS_HH
#define _DISJOINT_SETS_HH

#include <cassert>
#include <vector>

using namespace std;

struct DisjointSets {
  // Parallel arrays of parent indices, and rank values used for
  // balancing purposes.
  vector<int> parent, rank;

  // Initialize n singleton sets with the values 0...n-1.
  DisjointSets(int n) {
    assert(n > 0);
    for (int x = 0; x < n; ++x) {
      parent.push_back(x);
      rank.push_back(0);
    }
  }

  // Return true if x is a valid element.
  bool is_element(int x) {
    return (x >= 0) && (x < parent.size());
  }

  // Return the representative element for the set containing x.
  int find(int x) {
    assert(is_element(x));
    if (parent[x] != x)
      parent[x] = find(parent[x]);
    return parent[x];
  }
```



```
  // Merge the set containing x with the set containing y.
  void merge(int x, int y) {
    assert(is_element(x));
    assert(is_element(y));

    int x_root = find(x), y_root = find(y);
    if (x_root == y_root)
      return;

    if (rank[x_root] < rank[y_root])
      parent[x_root] = y_root;
    else if (rank[x_root] > rank[y_root])
      parent[y_root] = x_root;
    else {
      parent[y_root] = x_root;
      rank[x_root]++;
    }

    assert(find(x) == find(y));
  }
};

#endif
```

```
#include <iostream>

#include "disjoint_sets.hh"

int main() {
  DisjointSets ds(4);

  assert(ds.find(0) == 0);
  assert(ds.find(1) == 1);
  assert(ds.find(2) == 2);
  assert(ds.find(3) == 3);

  ds.merge(2, 3);
  assert(ds.find(0) == 0);
  assert(ds.find(1) == 1);
  assert(ds.find(2) == 2);
  assert(ds.find(3) == 2);

  ds.merge(1, 0);
  assert(ds.find(0) == 1);
  assert(ds.find(1) == 1);
  assert(ds.find(2) == 2);
  assert(ds.find(3) == 2);

  ds.merge(1, 3);
  assert(ds.find(0) == 1);
  assert(ds.find(1) == 1);
  assert(ds.find(2) == 1);
  assert(ds.find(3) == 1);

  cout << "PASSED" << endl;

  return 0;
}
```

```
// This is NOT COMPLETE :)

// example of string class
// parse input
// compile on command line
// read input until EOF (end of file)
// input/output redirect

#include <iostream>
#include <string>
using namespace std;

// == Compile_on_Command_Line =======================================
//
// $ g++ myfile.cpp -o name
// $ ./name
//
// ==================================================================

// == Read_input_until_(EOF) ========================================
//
// If you are using the command line to enter input, this
// program will wait for input. Once you type something and
// press enter, the program will read each "word" in a loop
// until it reaches the end of the line (where you pressed enter),
// and it will print each "word" out onto the command line
// followed by a new line.
//
// Example: you type "1 2 3", and you see
// 1
// 2
// 3
//
// Then the program waits for you to enter something again.
// If you press "cntrl + c" the program will end. If you press
// "cntrl + d" the program also ends. The difference is that
// "cntrl + d" represents the EOF character which is read by the
// program and tells the while loop to stop looping, while "cntrl + c"
// actually terminates the program no matter what it is doing.
//
// In short, the program is waiting for the EOF character to end,
// even though you did not specify this in the program. This is why
// the program stops when it is reading from an input file (instead
// of reading from the command line).
//
// ==================================================================

// == Input/Output_Redirecting ======================================
//
// The cool thing is that you can give the program a file without
// using file I/O! That's right; no opening and closing files or
// any extra lines of code in your program. How you do this is with
// redirection operators ('<' and '>').
//
// In your command line
//
// ==================================================================

int main(void)
{
        string mystring;

        while(cin >> mystring)
                {
                cout << mystring << endl;
                }

        return 0;
}
```

```
#include <iostream>
#include <cstdlib>
#include <cstddef>
using namespace std;

int GCD(int num1, int num2);
int LCM(int num1, int num2);

void order(int& num1, int& num2);

int main()
{
        int a = 0;
        int b = 0;

        cout<< "Enter two numbers to compute the GCD and LCM: ";
        cin >> a;
        cin >> b;

        order(a,b);

        cout << "GCD of " << a << " " << b << " is "
             << GCD(a,b) << endl;

        cout << "LCM of " << a << " " << b << " is "
             << LCM(a,b)  << endl;

        return 0;
}

////////////////////////////////////
///for the case of GCD for more than 2 numbers use:
///GCD(num1,num2,num3) => GCD(GCD(num1,num2),num3)
////////////////////////////////////
int GCD(int num1, int num2)
{
        //quick check if any of the two is zero
        if(num2 == 0)
        {
                return num1;
        }
        else if (num1 == 0)
        {
                return num2;
        }


        // after having setup num1 to hold the largest value we
        // iterate N times until gcd is found when the remainder is zero
        while(num2 != 0)
        {
                int temp = num2; // temp holds the last value of num2 while the
next remainder is calculated
                num2 = num1 % temp;// num2 gets assigned the remainder of num1%n
um2
                num1 = temp;//num1 holds its predecessor
        }

        return num1;


}



////////////////////////////////////////////////////////////////////////////////
////
///for the case of finding the LCM of more than two numbers use the recursive me
thod:
```



```
///LCM(num1,num2,num3) => LCM(LCM(num1,num2),num3)
////////////////////////////////////////////////////////////////////////////////
////
int LCM(int num1, int num2)
{

        int remainder,largest,smaller;
        int lcm = 0;

        //quick check if any of the two is zero
        if(num2 == 0)
        {
                return num1;
        }
        else if(num1==0)
        {
                return num2;
        }



        //num1 holds the largest value
        largest = num1;
        smaller = num2;
        remainder = largest%smaller;

        while(remainder != 0)
        {
                largest = smaller;
                smaller = remainder;
                remainder = largest % smaller;

        }

        lcm = (num1 * num2)/smaller;

        return lcm;

}


void order(int& num1, int& num2)
{
        //if num2 is greater than num1 values are swapped
        if(num2 > num1)
        {
                swap(num1,num2);
        }
}
```

```
#ifndef _GRAPH_HH
#define _GRAPH_HH

#include <cassert>
#include <vector>

using namespace std;

struct Edge {
  // An edge has two vertex endpoints: v and w. It may or may not be
  // directed and may have a defined numerical weight. When an edge is
  // directed, v is the "from" end and w is the "to" end. In an
  // undirected edge, v and w are interchangeable.
  int index, v, w;
  bool directed;
  double weight;

  Edge(int index_, int v_, int w_, bool directed_, double weight_)
    : index(index_),
      v(v_),
      w(w_),
      directed(directed_),
      weight(weight_) { }

  // Is x one of the ends?
  bool is_incident(int x) const {
    return (x == v) || (x == w);
  }

  // Can this edge flow into vertex x?
  bool enters(int x) const {
    if (directed)
      return x == w;
    else
      return is_incident(x);
  }

  // Can this edge flow out of vertex x?
  bool exits(int x) const {
    if (directed)
      return x == v;
    else
      return is_incident(x);
  }

  // Can this edge flow from "from" to "to"?
  bool connects(int from, int to) const {
    if (from != to)
      return exits(from) && enters(to);
    else
      return (v == from) && (w == from);
  }

  // Assuming x is one of this edge's ends, return the other end.
  int opposite(int x) const {
    assert(is_incident(x));
    return (x != v) ? v : w;
  }
};

// Declaring this constant avoids some pointer casting nonsense later
// on.
Edge const* const EDGE_NULL = 0;

// Adjacency matrix. For AdjMatrix m, m[v][w] is the edge connecting v
// to w, or EDGE_NULL if no such edge exists.
typedef vector<vector<Edge const*> > AdjMatrix;
```



```
// A graph. Stores the edges in a vector, and also an adjacency list
// data structure for fast neighbor lookups of sparse graphs. Each
// individual Edge can be either directed or undirected, so this data
// structure supports "mixed" graphs with both kind of edges, although
// most graph algorithms support only one kind of edge. This data
// structure can also accomodate "multigraphs" where multiple edges
// can be defined between any pair of vertices, although again most
// algorithms don't handle that situation.
struct Graph {
  // Edge objects in the order they were added (unsorted).
  vector<Edge*> edges;

  // Adjacency list: adj[v] is all the edges incident to vertex v. The
  // elements are pointers to the same Edge objects stored in the
  // vector above.
  vector<vector<Edge*> > adj;

  // n is the count of vertices.
  Graph(int n)
  : adj(n) { }

  // destructor
  ~Graph() {
    for (vector<Edge*>::iterator i = edges.begin(); i != edges.end(); ++i)
      delete *i;
  }

  int n() const { return adj.size(); }    // # vertices
  int m() const { return edges.size(); }  // # edges

  bool is_vertex(int x) const { return (x >= 0) && (x < n()); }

  bool empty() const { return n() == 0; }

  int add_edge(int v, int w, bool directed=false, double weight=0.0) {
    assert(is_vertex(v));
    assert(is_vertex(w));
    edges.push_back(new Edge(m(), v, w, directed, weight));
    adj[v].push_back(edges.back());
    adj[w].push_back(edges.back());
  }

  // Find an edge connecting "from" to "to", or EDGE_NULL if no such
  // edge exists.
  Edge const* connection(int from, int to) const {
    for (vector<Edge*>::const_iterator e = adj[from].begin();
         e != adj[from].end(); ++e)
      if ((*e)->connects(from, to))
        return *e;
    return EDGE_NULL;
  }

  // Build adjacency matrix. Caller owns the object.
  AdjMatrix* adj_matrix() const {
    assert(!empty());
    AdjMatrix* m = new AdjMatrix(n(), vector<Edge const*>(n(), EDGE_NULL));
    for (vector<Edge*>::const_iterator i = edges.begin(); i != edges.end(); ++i)
{
      const Edge& e = **i;
      (*m)[e.v][e.w] = &e;
      if (!e.directed)
        (*m)[e.w][e.v] = &e;
    }
    return m;
  }

  // Query the directedness of the graph.

  int count_directed() const {
```

```
    int total = 0;
    for (vector<Edge*>::const_iterator e = edges.begin(); e != edges.end(); ++e)
      if ((*e)->directed)
        ++total;
    return total;
  }

  bool strictly_directed() const { return count_directed() == m(); }
  bool strictly_undirected() const { return count_directed() == 0; }
};

// A path is a sequence of Edges.
struct Path {
  vector<Edge const*> edges;
  double weight; // cache the total weight so it can be computed in
                 // constant time

  Path()
    : weight(0) { }

  // copy constructor
  Path(const Path *p) {
        this->edges = p->edges;
        this->weight = p->weight;
  }

  // Concatenates one path with another
  // @param p - path to add to this
  // @returns - new path
  Path *concat(Path *p) {
        Path *new_path;
        Path *prev_path = new Path(this);

        for(int i = 0; i < p->edges.size(); i++) {
                new_path = prev_path->extend(*(p->edges.at(i)));

                delete prev_path;
                prev_path = new_path;
        }
        return new_path;
  }

  //@postcondition - caller must deallocate path returned
  Path *extend(const Edge& e) const {
        Path *new_path = new Path(this);
        new_path->edges.push_back(&e);
        new_path->weight += e.weight;
        return new_path;
  }
};

#endif
```

```
#include "graph.hh"

#include <iostream>
#include <memory>

int main() {
  Graph empty(0);
  assert(empty.n() == 0);
  assert(empty.m() == 0);
  assert(!empty.is_vertex(-1));
  assert(!empty.is_vertex(0));
  assert(!empty.is_vertex(1));
  assert(empty.empty());
  assert(empty.count_directed() == 0);
  assert(empty.strictly_directed());
  assert(empty.strictly_undirected());

  // example from
  // http://en.wikipedia.org/wiki/Kruskal%27s_algorithm#Example
  const int A=0, B=1, C=2, D=3, E=4, F=5, G=6;
  Graph g(7);

  assert(g.n() == 7);
  assert(g.m() == 0);
  assert(!g.is_vertex(-1));
  assert(g.is_vertex(0));
  assert(g.is_vertex(1));
  assert(g.is_vertex(2));
  assert(g.is_vertex(3));
  assert(g.is_vertex(4));
  assert(g.is_vertex(5));
  assert(g.is_vertex(6));
  assert(!g.is_vertex(7));
  assert(!g.empty());

  g.add_edge(A, B, false, 7);  // 0
  assert(g.m() == 1);
  assert(g.adj[A].size() == 1);
  assert(g.adj[B].size() == 1);
  assert(g.adj[C].empty());
  assert(g.adj[D].empty());
  assert(g.adj[E].empty());
  assert(g.adj[F].empty());
  assert(g.adj[G].empty());
  assert(g.adj[A][0] == g.edges[0]);
  assert(g.adj[B][0] == g.edges[0]);

  g.add_edge(A, D, false, 5);  // 1
  assert(g.m() == 2);
  assert(g.adj[A].size() == 2);
  assert(g.adj[B].size() == 1);
  assert(g.adj[C].empty());
  assert(g.adj[D].size() == 1);
  assert(g.adj[E].empty());
  assert(g.adj[F].empty());
  assert(g.adj[G].empty());
  assert(g.adj[A][1] == g.edges[1]);
  assert(g.adj[D][0] == g.edges[1]);

  g.add_edge(B, C, false, 8);  // 2
  assert(g.m() == 3);
  g.add_edge(B, D, false, 9);  // 3
  assert(g.m() == 4);
  g.add_edge(B, E, false, 7);  // 4
  assert(g.m() == 5);
  g.add_edge(C, E, false, 5);  // 5
  assert(g.m() == 6);
  g.add_edge(D, E, false, 15); // 6
```



```
  assert(g.m() == 7);
  g.add_edge(D, F, false, 6);  // 7
  assert(g.m() == 8);
  g.add_edge(E, F, false, 8);  // 8
  assert(g.m() == 9);
  g.add_edge(E, G, false, 9);  // 9
  assert(g.m() == 10);
  g.add_edge(F, G, false, 11); // 10
  assert(g.m() == 11);

  assert(g.connection(A, A) == EDGE_NULL);
  assert(g.connection(A, B) == g.edges[0]);
  assert(g.connection(A, C) == EDGE_NULL);
  assert(g.connection(A, D) == g.edges[1]);
  assert(g.connection(A, E) == EDGE_NULL);
  assert(g.connection(A, F) == EDGE_NULL);
  assert(g.connection(A, G) == EDGE_NULL);

  assert(g.connection(B, A) == g.edges[0]);
  assert(g.connection(B, B) == EDGE_NULL);
  assert(g.connection(B, C) == g.edges[2]);
  assert(g.connection(B, D) == g.edges[3]);
  assert(g.connection(B, E) == g.edges[4]);
  assert(g.connection(B, F) == EDGE_NULL);
  assert(g.connection(B, G) == EDGE_NULL);

  auto_ptr<AdjMatrix> m(g.adj_matrix());

  assert((*m)[A][A] == EDGE_NULL);
  assert((*m)[A][B] == g.edges[0]);
  assert((*m)[A][C] == EDGE_NULL);
  assert((*m)[A][D] == g.edges[1]);
  assert((*m)[A][E] == EDGE_NULL);
  assert((*m)[A][F] == EDGE_NULL);
  assert((*m)[A][G] == EDGE_NULL);

  assert((*m)[B][A] == g.edges[0]);
  assert((*m)[B][B] == EDGE_NULL);
  assert((*m)[B][C] == g.edges[2]);
  assert((*m)[B][D] == g.edges[3]);
  assert((*m)[B][E] == g.edges[4]);
  assert((*m)[B][F] == EDGE_NULL);
  assert((*m)[B][G] == EDGE_NULL);

  assert((*m)[C][A] == EDGE_NULL);
  assert((*m)[C][B] == g.edges[2]);
  assert((*m)[C][C] == EDGE_NULL);
  assert((*m)[C][D] == EDGE_NULL);
  assert((*m)[C][E] == g.edges[5]);
  assert((*m)[C][F] == EDGE_NULL);
  assert((*m)[C][G] == EDGE_NULL);

  assert((*m)[D][A] == g.edges[1]);
  assert((*m)[D][B] == g.edges[3]);
  assert((*m)[D][C] == EDGE_NULL);
  assert((*m)[D][D] == EDGE_NULL);
  assert((*m)[D][E] == g.edges[6]);
  assert((*m)[D][F] == g.edges[7]);
  assert((*m)[D][G] == EDGE_NULL);

  assert((*m)[E][A] == EDGE_NULL);
  assert((*m)[E][B] == g.edges[4]);
  assert((*m)[E][C] == g.edges[5]);
  assert((*m)[E][D] == g.edges[6]);
  assert((*m)[E][E] == EDGE_NULL);
  assert((*m)[E][F] == g.edges[8]);
  assert((*m)[E][G] == g.edges[9]);

  assert((*m)[F][A] == EDGE_NULL);
```

```
  assert((*m)[F][B] == EDGE_NULL);
  assert((*m)[F][C] == EDGE_NULL);
  assert((*m)[F][D] == g.edges[7]);
  assert((*m)[F][E] == g.edges[8]);
  assert((*m)[F][F] == EDGE_NULL);
  assert((*m)[F][G] == g.edges[10]);

  assert((*m)[G][A] == EDGE_NULL);
  assert((*m)[G][B] == EDGE_NULL);
  assert((*m)[G][C] == EDGE_NULL);
  assert((*m)[G][D] == EDGE_NULL);
  assert((*m)[G][E] == g.edges[9]);
  assert((*m)[G][F] == g.edges[10]);
  assert((*m)[G][G] == EDGE_NULL);

  assert(g.count_directed() == 0);
  assert(!g.strictly_directed());
  assert(g.strictly_undirected());

  cout << "PASSED" << endl;

  return 0;
}
```

```
//
// Kruskal's algorithm for minimum spanning trees.
//

#ifndef _KRUSKAL_MST_HH
#define _KRUSKAL_MST_HH

#include <algorithm>

#include "disjoint_sets.hh"
#include "graph.hh"

// Utility function to compare edge weights, used in sorting.
bool compare_edge_ptr_weight(const Edge* x, const Edge* y) {
  return x->weight < y->weight;
}

// Find the edges in the minimum spanning tree of g. Returns a pointer
// to a vector of Edge pointers. The Edge pointers refer to the Edge
// objects owned by g. The client is responsible for eventually
// deleting the returned vector.
vector<const Edge*>* kruskal_mst_edges(const Graph& g) {
  assert(!g.empty());

  // Sort the edges into nondecreasing order by weight. We use a
  // stable sort to make unit testing easier, but stability is not
  // otherwise necessary.
  vector<Edge*> sorted_edges = g.edges;
  stable_sort(sorted_edges.begin(),
              sorted_edges.end(),
              compare_edge_ptr_weight);

  // Use a disjoint set data structure to keep track of the patchwork
  // forest of spanned subgraphs that we'll create. Initially every
  // vertex is its own component; every time we add an edge, those
  // components get connected. When we're done the whole graph has
  // been connected into one component.
  DisjointSets parts(g.n());

  // Edges in the minimum spanning tree.
  vector<const Edge*>* in_tree(new vector<const Edge*>());

  for (vector<Edge*>::iterator i = sorted_edges.begin();
       i != sorted_edges.end(); ++i) {
    const Edge& e = **i;
    // Does this edge connect distinct components?
    if (parts.find(e.v) != parts.find(e.w)) {
      in_tree->push_back(&e); // Yes, so add it to the tree
      parts.merge(e.v, e.w); // Now the components are one
    }
  }

  return in_tree;
}

#endif
```

```
#include <iostream>
#include <memory>

#include "kruskal_mst.hh"

double total_weight(const vector<const Edge*>& edges) {
  double total = 0;
  for (int i = 0; i < edges.size(); ++i)
    total += edges[i]->weight;
  return total;
}

int main() {
  const int A=0, B=1, C=2, D=3, E=4, F=5, G=6;

  // First test: example from
  // http://en.wikipedia.org/wiki/Kruskal%27s_algorithm#Example
  {
    Graph g(7);
    g.add_edge(A, B, false, 7);  // 0
    g.add_edge(A, D, false, 5);  // 1
    g.add_edge(B, C, false, 8);  // 2
    g.add_edge(B, D, false, 9);  // 3
    g.add_edge(B, E, false, 7);  // 4
    g.add_edge(C, E, false, 5);  // 5
    g.add_edge(D, E, false, 15); // 6
    g.add_edge(D, F, false, 6);  // 7
    g.add_edge(E, F, false, 8);  // 8
    g.add_edge(E, G, false, 9);  // 9
    g.add_edge(F, G, false, 11); // 10
    assert(g.m() == 11);

    auto_ptr<vector<const Edge*> > t(kruskal_mst_edges(g));

    assert(t->size() == 6);
    assert(total_weight(*t) == (5+5+6+7+7+9));

    assert(t->at(0) == g.connection(A, D));
    assert(t->at(1) == g.connection(C, E));
    assert(t->at(2) == g.connection(D, F));
    assert(t->at(3) == g.connection(A, B));
    assert(t->at(4) == g.connection(B, E));
    assert(t->at(5) == g.connection(E, G));
  }

  // Second test:
  // http://en.wikipedia.org/wiki/File:Msp1.jpg
  {
    Graph g(6);
    g.add_edge(A, B, false, 1); // 0
    g.add_edge(A, D, false, 4); // 0
    g.add_edge(A, E, false, 3); // 0
    g.add_edge(B, D, false, 4); // 0
    g.add_edge(B, E, false, 2); // 0
    g.add_edge(C, E, false, 4); // 0
    g.add_edge(C, F, false, 5); // 0
    g.add_edge(D, E, false, 4); // 0
    g.add_edge(E, F, false, 7); // 0

    auto_ptr<vector<const Edge*> > t(kruskal_mst_edges(g));

    assert(t->size() == 5);
    assert(total_weight(*t) == (1+2+4+4+5));

    assert(t->at(0) == g.connection(A, B));
    assert(t->at(1) == g.connection(B, E));
    assert(t->at(2) == g.connection(A, D));
    assert(t->at(3) == g.connection(C, E));
```



```
    assert(t->at(4) == g.connection(C, F));
  }

  cout << "PASSED" << endl;

  return 0;
}
```

```
// Examples of creating 2D, 3D, and 4D STL vectors succinctly.

#include <vector>

using namespace std;

int main() {
  // vector has a constructor
  //   vector<type>(N, default-value)
  // that initializes the vector with N copies of default-value.
  vector<double> arr(30, 3.2); // 30 copies of 3.2

  // 10 rows, 20 columns, all filled with 0.0. We use the constructor
  // described above to create a default-value representing one row,
  // which is copied for each of the 10 requested rows. Note there are
  // 2 dimensions and therefore 2 trailing ">" symbols in the matrix'
  // type and 2 trailing ")" symbols at the end of the line.
  vector<vector<double> > table(10, vector<double>(20, 0.0));

  // you can use usual subscript operators
  table[0][0] = 3;
  table[9][9] = 5;

  // 3D 15x20x25 box, all elements -3.0. 3 dimensions and 3 trailing
  // >'s and )'s.
  vector<vector<vector<double> > > box(15, vector<vector<double> >(20, vector<do
uble>(25, -3.0)));

  // 4D 15x20x25x30 box, all elements 17. 4 dimensions and 4 trailing
  // >'s and )'s.
  vector<vector<vector<vector<double> > > > hyper(15, vector<vector<vector<doubl
e> > >(20, vector<vector<double> >(25, vector<double>(30, 17))));
}
```

```
#include <iostream>
#include <string>
#include <sstream>
#include <vector>
using namespace std;

// == ParseStr ========================================
//
// This function receives a line of input as a string
// and a pointer to a vector of strings which will
// contain the parsed line. If an empty line is
// encountered, it is also pushed into the vector.
//
// ================================================
void ParseStr(string line, vector<string>* result)
{
        stringstream ss;
        string temp;

        if (line.empty())
                {
                result->push_back(line);
                return;
                }
        else
                {
                ss.clear();
                ss.str("");
                ss << line;
                while(ss >> temp)
                        {
                        result->push_back(temp);
                        }
                return;
                }
}

// == main ============================================
//
// This is a sample main that uses ParseStr. It
// displays each element of the resulting vector
// (each string element) on its own line.
//
// ================================================

// Sample Input:
//  Hello! This is a
//  test. :)
//
//  Testing 1 2 3.
//
// Sample Output:
//  Hello!
//  This
//  is
//  a
//  test.
//  :)
//
//  Testing
//  1
//  2
//  3.
//
// ================================================

int main(void)
{
        string myline;
```



```
        vector<string> *myvecptr;
        myvecptr = new vector<string>();

        //get a line and parse it until eof
        while(getline(cin,myline))
                {
                ParseStr(myline, myvecptr);
                }

        //display vector
        for(int i = 0; i < myvecptr->size(); ++i)
                {
                cout << myvecptr->at(i) << endl;
                }

        return 0;
}
```

```
// Author: Dustin Delmer
// Date: 2/2/2013
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
using namespace std;


// function that removes leading and trailing white space
void strip(string& str){
        stringstream ss (str);
        string temp;
        str.clear();
        while(ss>>temp){
                if(!str.empty()){ str += ' '; }
                str += temp;
        }
};


// function that parses a string
//   @arg str - string to be parsed
//   @arg delim - char to be used as delminator
//       @return value - vector of strings
vector<string>* parse_csv(string str, char delim=','){
        vector<string>* result;
        result = new vector<string>();
        string temp;
        for (int i=0; i<str.size(); i++){
                if(str[i]!=delim){ temp += str[i]; }
                else {
                        strip(temp);
                        result->push_back(temp);
                        temp.clear();
                }
        }
        if (!temp.empty()){
                strip(temp);
                result->push_back(temp);
        }
        return result;
};




int main(){
        string str = "this, is a, string, with, some, commas";
        vector<string>* result;
        result = parse_csv(str);

        for (int i=0; i<(result->size()); i++){ cout << result->at(i) << endl; }

        delete result;
        return 0;
}
```

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  vector<char> perm;
  perm.push_back('A');
  perm.push_back('B');
  perm.push_back('C');
  perm.push_back('D');

  cout << "The 4! possible permutations with 4 elements:\n";

  sort (perm.begin(), perm.end());

  do {
       cout << perm[0] << " "
            << perm[1] << " "
            << perm[2] << " "
            << perm[3] << endl;
  } while ( next_permutation (perm.begin(),perm.end()) );

  return 0;
}
```

```
#include <iostream>
using namespace std;

void PrimeFactors(int n)
{
        for (int i=2; i<=n; i++)
        {
                if (n % i == 0)
                {
                        cout << i << " ";
                        n /= i;
                        i--;
                }
        }
}

void UniquePrimeFactors(int n)
{
        int last = 0;

        for (int i=2; i<=n; i++)
        {
                if (n % i == 0 && i != last)
                {
                        cout << i << " ";
                        n /= i;
                        last = i;
                        i--;
                }
        }
}

int main()
{
        int number;
        cout << "Please enter a number: ";
        cin >> number;

        cout << "Prime factorization: ";
        PrimeFactors(number);
        cout << endl;

        cout << "Unique prime factorization: ";
        UniquePrimeFactors(number);
        cout << endl;

        system("pause");
        return 0;
}
```

```
#ifndef PRIMEsieve
#define PRIMEsieve
#include <iostream>
#include <vector>
using namespace std;

/*--------------------------------------------------------------------
Name: Primesieve
Purpose: Utilizes the sieve of Eratosthenes to find lists of primes.
Example Uses:
1)
        Primesieve s;
        s.sieve(2, 100000);             //Finds all primes between 2 and 100000.
                                                        //Subsequent tests are f
ast for n<100000.
        cout << (s.TestPrime(92837)?"prime\n":"not prime\n");
        cout << (s.TestPrime(78391)?"prime\n":"not prime\n");
        cout << (s.TestPrime(78401)?"prime\n":"not prime\n");
        cout << (s.TestPrime(97281)?"prime\n":"not prime\n");
        //=================
2)
        Primesieve s;
        s.TestPrime(92837);             //Since s is empty, finds all primes fro
m
                                                        //2 to sqrt(92837) and t
ests whether any
                                                        //divide 92837 evenly. T
his is faster if
                                                        //only a few tests are p
erformed, but the
                                                        //use #1 is recommended
for multiple
                                                        //tests. Subsequent test
s are slow.
--------------------------------------------------------------------*/

class Primesieve{
private:
        int lastEnd;
        vector< pair<int, int> > primeList;

        bool BinarySearch(int n)
        {
                int L = 0, R = primeList.size()-1, i;
                while (i = int((R-L)/2 + L), L<=R)
                {
                        if (primeList[i].first == n)
                        {
                                return true;
                        }
                        else if (primeList[i].first < n)
                        {
                                L = i+1;
                        }
                        else
                        {
                                R = i-1;
                        }
                }

                return false;
        }

        void Display()
        {
                for (int i=0; i<primeList.size(); i++)
                {
                        cout << primeList[i].first << ' ';
                }
```



```
                cout << endl;
        }
public:
        Primesieve()
        {
                lastEnd = 2;
                pair<int,int> temp(2, 0);
                if (primeList.size() == 0) primeList.push_back(temp);
        }

        void sieve(int start, int end)
        {
                for (int i=lastEnd+1; i<=end; i++)
                {
                        bool prime = true;
                        for (int j=0; j<primeList.size(); j++)
                        {
                                if (primeList[j].first == i)
                                {
                                        prime = false;
                                        break;
                                }

                                primeList[j].second++;
                                if (primeList[j].second == primeList[j].first)
                                {
                                        primeList[j].second = 0;
                                        prime = false;
                                }
                        }

                        if (prime){
                                pair<int,int> temp(i, 0);
                                primeList.push_back(temp);
                        }
                }

                lastEnd = end;
        }

        bool TestPrime(int n)
        {
                bool prime = true;

                if (primeList[primeList.size()-1].first >= n)
                {
                        //The number is already in the sieve; search for it.
                        return BinarySearch(n);
                }
                else if (primeList[primeList.size()-1].first < sqrt(double(n)))
                {
                        //We don't have enough numbers to test n; sieve up to sq
rt(n).
                        sieve(primeList[primeList.size()-1].first+1, sqrt(double
(n)));
                        for (int i=0; i<primeList.size(); i++)
                        {
                                //Test if any of the primes found so far divide
n.
                                //This is theoretically faster than seiving up t
o n, but
                                //it does not save any results for later. If mul
tiple
                                //tests will be made, it is recommended that sie
ve(n) be
                                //called first.
                                if (n % primeList[i].first == 0)
                                {
                                        prime = false;
```

```
                                        break;
                                }
                        }
                }
                //Display();
                return prime;
        }
};
#endif
```

```
#include <iostream>
using namespace std;

#define ARRAY_SIZE 5                                    //change the array size he
re

void PrintArray(int* array, int n);
void QuickSort(int* array, int startIndex, int endIndex);
int SplitArray(int* array, int pivotValue, int startIndex, int endIndex);
void swap(int &a, int &b);

int main(void)
{
        int array[ARRAY_SIZE];
        int i;

        for( i = 0; i < ARRAY_SIZE; i++)            //array elements input
        {
                cout<<"Enter an integer : ";
                cin>>array[i];
        }

        cout<<endl<<"The list you input is : "<<endl;
        PrintArray(array, ARRAY_SIZE);
        QuickSort(array,0,ARRAY_SIZE - 1);              //sort array from first
to last element
        cout<<endl<<"The list has been sorted, now it is : "<<endl;
        PrintArray(array, ARRAY_SIZE);

        cin.get();
        cin.get();
        return 0;
        }

        /* This function swaps two numbers
           Arguments : a, b - the numbers to be swapped
   */

        void swap(int &a, int &b)
        {
            int temp;
            temp = a;
            a = b;
            b = temp;
        }

        /* This function prints an array.
                Arguments : array - the array to be printed
           n - number of elements in the array
        */

void PrintArray(int* array, int n)
{
    int i;
        for( i = 0; i < n; i++) cout<<array[i]<<'\t';
}

/* This function does the quicksort
           Arguments :
                    array - the array to be sorted
                    startIndex - index of the first element of the section
                    endIndex - index of the last element of the section
*/

void QuickSort(int* array, int startIndex, int endIndex)
{

        int pivot = array[startIndex];                  //pivot element is the l
eftmost element
```



```
        int splitPoint;

        if(endIndex > startIndex)                           //if they are equal, i
t means there is
                                                            //only one element and qui
cksort's job
                                                            //here is finished
        {
                splitPoint = SplitArray(array, pivot, startIndex, endIndex);
                                                            //SplitArray() returns the pos
ition where
                                                            //pivot belongs to
        array[splitPoint] = pivot;
        QuickSort(array, startIndex, splitPoint-1);   //Quick sort first half
        QuickSort(array, splitPoint+1, endIndex);    //Quick sort second half
   }
}

/* This function splits the array around the pivot
           Arguments :
            array - the array to be split
            pivot - pivot element whose position will be returned
            startIndex - index of the first element of the section
            endIndex - index of the last element of the section
   Returns :
            the position of the pivot
*/
int SplitArray(int* array, int pivot, int startIndex, int endIndex)
{
    int leftBoundary = startIndex;
    int rightBoundary = endIndex;

        while(leftBoundary < rightBoundary)             //shuttle pivot until th
e boundaries meet
        {
        while( pivot < array[rightBoundary]       //keep moving until a lesser e
lement is found
                                    && rightBoundary > leftBoundary)   //or until th
e leftBoundary is reached
                {
                rightBoundary--;                        //move left
                }

                swap(array[leftBoundary], array[rightBoundary]);
//PrintArray(array, ARRAY_SIZE);            //Uncomment this line for study

                while( pivot >= array[leftBoundary]       //keep moving until a
greater or equal element is found
                        && leftBoundary < rightBoundary)   //or until the rightBo
undary is reached
                {
                leftBoundary++;                          //move right
                }

    swap(array[rightBoundary], array[leftBoundary]);
         //PrintArray(array, ARRAY_SIZE);            //Uncomment this line for s
tudy
        }
        return leftBoundary;                                //leftBoundary is the
split point because
                                                            //the above while loop exi
ts only when
                                                            //leftBoundary and rightBo
undary are equal
}
```

```
#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <iterator>
using namespace std;

struct input {
        string str;
        int     value;
};


int main() {
        string token;
        vector<input> v;

        while (cin >> token) {
                char c;

                stringstream stoken;
                stringstream newtoken;
                stoken << token;

                input in;
                bool first = true;

                while ( stoken >> c ) {
                        if(c != ':') {
                                newtoken << c;
                        }
                        else {
                                newtoken << "*";
                                first = false;
                        }
                }

                //newtoken >> token;
                //cout << token << endl;

                if (first) {
                        newtoken >> in.value;
                }
                else {
                        newtoken >> in.str;
                        //cout << in.value << " " << in.str << endl;
                        v.push_back(in);
                }
        }

        vector<input>::iterator it;
        cout << "output:" << endl;
        for (it = v.begin(); it != v.end(); it++) {
                cout << (*it).str << " " << (*it).value << endl;
        }

        return 0;
}
```

```
// Author: Dustin Delmer
// Date: 2/2/2013
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main(){
        // initialize vector and fill
        vector<int> values;
        values.push_back(1);
        values.push_back(2);
        values.push_back(3);
        values.push_back(4);
        values.push_back(5);

        // create a bool vector of the same length to mark "active" elements
        vector<bool> active(5);

        // outer loop increments set size
        for(int i = 1; i <= active.size(); i++){
                int num_to_choose = i;
                fill(active.begin(), active.end(), false);
                fill(active.begin() + num_to_choose, active.end(), true);

                // loop for generating combinations     of current set size
                do {
                        for(int i=0; i<active.size(); i++){
                                if(!active[i]) { cout << values[i] << ' '; }
                        }
                        cout << endl;
                }while(next_permutation(active.begin(), active.end()));
        }

        return 0;
}
```

```
#include <iostream>
#include <vector>
using namespace std;

int main() {

    vector<char> elements;

    // add elements with duplicates up to 3
    elements.push_back('a');
    elements.push_back('a');
    elements.push_back('a');
    elements.push_back('b');
    elements.push_back('b');
    elements.push_back('b');
    elements.push_back('c');
    elements.push_back('c');
    elements.push_back('c');
    elements.push_back('d');
    elements.push_back('d');
    elements.push_back('d');

    int bits;
    int n = elements.size();

    for (bits = 0; bits < (1 << n); bits++) {
        vector<char> subset;
        int digit;


        for (digit=0; digit<n; digit++) {
            if (((1 << digit) & bits) != 0)
                subset.push_back(elements[digit]);
        }

        // if subset size is less than or equal to 3
        // then do something
        if (subset.size() <= 3) {
            cout << "subset " << bits << ":\t";
            for (int x=0; x<subset.size(); x++) {
                cout << subset[x] << " ";
            }
            cout << endl;
        }
    }

    return 0;
}
```

```
all: graph_test disjoint_sets_test kruskal_mst_test

graph_test: graph.hh graph_test.cpp
	g++ graph_test.cpp -o graph_test
	./graph_test

disjoint_sets_test: disjoint_sets.hh disjoint_sets_test.cpp
	g++ disjoint_sets_test.cpp -o disjoint_sets_test
	./disjoint_sets_test

kruskal_mst_test: graph_test kruskal_mst.hh kruskal_mst_test.cpp
	g++ kruskal_mst_test.cpp -o kruskal_mst_test
	./kruskal_mst_test
```

```
TODO:

    Sieve of eratosthenes
    how to compile on the command line
    getline
    stringstream
    tokenizing
    read until eof
    input/output redirect
    ASCII code trick (modulus)
    sorting
    brute force
    primality testing
    prime generation
    greatest common factor
    basic solutions to NP-C problems?
    vector class
        using algorithm.h with vector
    Full regression testing

IN PROGRESS:

  Floyd/Warshall (Chad)
  Dijkstra's algorithm (Hayden)
  DFS (Brian)
  BFS (Fidel)
  Max flow (Kiet)

DONE:

 Kruskal's minimum spanning tree algorithm
 Disjoint Sets data structure
 Permutation generation example
 Subset generation
 Parsing example
 Matrix initialization example
 CSV parsing
 GCD, LCM
 Some regression testing
```

```
//--------------------inputs-------------------------
// int type : n.
//              - the number of nodes
// array of double type: cMat.
//              - connectivity matrix, 0 means disconnected
//                and distances are all positive.
//              - Is of length (n*n).
//--------------------outputs------------------------
//double** type: dist_mat
//              - the shortest path, (ANSWER)
//int** type: pred_mat
//              - predicate matrix, used for reconstructing
//                shortest routes

void warshall(int n, double *cmat, double **dist_mat, int **pred_mat){
        double *dist;
        int *pred;
        int i, j, k; //counters

        //initialize data structs
        dist = (double *)malloc(sizeof(double) * n * n);//allocate dist[n*n]
        pred = (int *)malloc(sizeof(int) * n * n);//allocate pred[n*n]
        memset(dist, 0, sizeof(double)*n*n);//sets dist[all indices] to 0
        memset(pred, 0, sizeof(int)*n*n);//sets pred[all indices] to 0

        //initialize algorithm
        for ( i = 0; i < n; i++ ){
                for ( j= 0; j < n; j++ ){
                        if (cmat[i*n+j] != 0.0)
                                dist[i*n+j] = cmat[i*n + j];
                        else
                                dist[i*n+j] = HUGE_VAL; //path disconnected

                        if (i == j) //diagonal case
                                dist[i*n+j] = 0;

                        if ((dist[i*n +j] > 0.0) && (dist[i*n+j] < HUGE_VAL))
                                pred[i*n+j] = i;
                }
        }

        //main loop
        for ( k = 0; k < n; k ++ ) {
                for ( i = 0; i < n; i ++ ) {
                        for ( j = 0; j < n; j ++ ) {
                                if ( dist[i*n+j] > (dist[i*n+k] + dist[k*n+j]))
{
                                        dist[i*n+j] = dist[i*n+k] + dist[k*n+j];
                                        pred[i*n+j] = k;
                                }
                        }
                }
        }

        //print out results of all the shortest dist
        //***if we want to output right away***
        /*
        for( i = 0; i < n; i++) {
                for ( j = 0; j < n; j++) {
                        cout << dist[i*n+j] << endl;
                }
        }
        */

        if ( dist_mat )
                *dist_mat = dist;
        else
                free(dist);
```



```
        if ( pred_mat )
                *pred_mat = pred;
        else
                free(pred);
}
```

```
INPUT:
        Weighted, directed graph with no negative weight cycles

OUTPUT:
        Distance matrix. D[i][j] is distance of shortest path from i to j.
        Inf if unreachable

NEED
        numerical limits (infinite)
        test cases

TEST CASES
        Create a graph with known solution.


Pseudo code

matrix Floyd(Graph g) {
Create a 3D matrix A[0...n][0...n−1][0...n−1]
//Base case
for i from 0 to n − 1
     for j from 0 to n − 1
          if i==j
               A[0][i][j] = 0
          else
               A[0][i][j] = min(infinity, D[i][j])
//Main loop
for k from 1 to n
     for i from 0 to n − 1
          for j from 0 to n − 1
               A[k][i][j] = min(A[k − 1][i][k − 1] + A[k − 1][k − 1][j], A[k −
1][i][j])
               //1st arg of min: if layover changes length; 2nd: if it doesn't
               //What happens if you take the existing paths and add a layover
               //at vertex k − 1?
return A[k] //returns a 2D matrix
}
        for(int k = 1; k < attrs.size() + 1; k++) {
                for(int i = 0; i < attrs.size(); i++) {
                        for(int j = 0; j < attrs.size(); j++) {
                                temp[k][i][j] =
                                        std::min(
                                                temp[k − 1][i][k − 1] + temp[k −
 1][k − 1][j],
                                                temp[k − 1][i][j]
                                        );
                        }
                }
        }

        return temp[attrs.size()];
```

```
#ifndef FLOYD
#define FLOYD
#include <vector>
#include <algorithm>
#include <limits>
#include <iostream>
#include "../graph.hh"
typedef std::vector<std::vector<Path *> > DistanceMatrix;

// Computes the shortest path between each pair of edges (Floyd's algorithm)
DistanceMatrix* floyd(Graph *g);

// Intialize a distance matrix
// Used for the "previous" DistanceMatrix in floyd
void floyd_init(DistanceMatrix *d, Graph *g, AdjMatrix *adj);

// Deallocates a distance matrix
void delete_distance_matrix(DistanceMatrix *d);

// Returns the minimum of two paths
// If they're of equal weight, return the one with least vertices.
Path *min_path(Path *p1, Path *p2);

// Computes the shortest path between each pair of edges (Floyd's algorithm)
// @param g - a graph with no negative edge weights
// @postconditions - you must delete the returned pointer
// @return a distance matrix containing the shortest paths between two vertices.
// An entry matrix[i][j] is the shortest path between vertices i and j.
DistanceMatrix* floyd(Graph *g) {

        // Get the adjacency matrix representation of the graph
        AdjMatrix *adj_matrix = g->adj_matrix();

        // Temporary matrix; stores previous distance matrix
        DistanceMatrix *previous = new DistanceMatrix(g->n(),
                                                                        std::vec
tor<Path *>(g->n()));
        // Current shortest paths
        DistanceMatrix *shortest_paths = new DistanceMatrix(g->n(),
                                                                        std::vec
tor<Path *>(g->n()));

        // initialize the "previous" matrix
        floyd_init(previous, g, adj_matrix);

        // calculate all-pairs shortest path
        for(int k = 1; k < g->n() + 1; k++) {
                for(int i = 0; i < g->n(); i++) {
                        for(int j = 0; j < g->n(); j++) {
                                // delete current shortest path value (stale)
                                if(shortest_paths->at(i).at(j) != NULL) {
                                        delete shortest_paths->at(i).at(j);
                                }

                                // calculate shortest path
                                // TODO this breaks if not a complete graph (if
any previous
                                // value is NULL.
                                // TODO concatenation isn't working as expected
                                Path *potential =
                                        previous->at(i).at(k - 1)->concat(previo
us->at(i).at(j));
                                Path *new_path = min_path(potential, previous->a
t(i).at(j));
                                shortest_paths->at(i).at(j) = new Path(new_path)
;
                                //TODO fix memory leak with potential
                        }
                }
```



```
                // Swap which structure is the most recent shortest path.
                // This is an optimation trick to avoid creating a 3D matrix.
                swap(shortest_paths, previous);
        }

        cout << "seg?\n";
        delete_distance_matrix(previous);
        delete adj_matrix;
        return shortest_paths;
}

// Intialize a distance matrix
// Used for the "previous" DistanceMatrix in floyd
// @parm d - distance matrix to initialize
// @param g - graph
// @param adj - adjacency matrix of graph
void floyd_init(DistanceMatrix *d, Graph *g, AdjMatrix *adj) {

        // Initialize distance matrix.
        for(int i = 0; i < g->n(); i++) {
                for(int j = 0; j < g->n(); j++) {
                        if(i == j) {
                                // path from vertex to itself
                                d->at(i).at(j) = new Path(); // empty path objec
t
                        }
                        else {
                                // distance of vertex i to j
                                if(adj->at(i).at(j) != EDGE_NULL) {
                                        // i and j are adjacent
                                        Path p;
                                        // add edge to path
                                        const Edge *e = adj->at(i).at(j);
                                        d->at(i).at(j) = p.extend(*e);
                                }
                                else {
                                        // i and j are not adjacent
                                        d->at(i).at(j) = NULL;
                                }
                        }
                }
        }
}


// Deallocates a distance matrix
// @param distance matrix to delete
void delete_distance_matrix(DistanceMatrix *d) {
        // Distance matrix contains pointers to paths.
        // Non-null pointers must be deleted to avoid memory leaks.
        for(int i = 0; i < d->size(); i++) {
                for(int j = 0; j < d->at(i).size(); j++) {
                        cout << "i,j" << i << "," << j << endl;
                        if(d->at(i).at(j) != NULL) {

                                delete d->at(i).at(j);
                                d->at(i).at(j) = NULL;
                        }
                }
        }
}

// Returns the shortest of two paths.
// If they're of equal weight, return the one with least edges.
// @param p1 - a path
// @param p2 - another path
// @return the shorter of the two paths. If they're of equal weight, return the
// one with the least edges.
Path *min_path(Path *p1, Path *p2) {
```

```
        Path *shortest;
        if(p1->weight < p2->weight) {
                // p1 shorter than p2
                shortest = p1;
        }
        else if(p2->weight < p1->weight) {
                // p2 shorter than p1
                shortest = p2;
        }
        else {
                // p1 and p2 have same weight
                if(p1->edges.size() < p2->edges.size()) {
                        shortest = p1;
                }
                else {
                        shortest = p2;
                }
        }
        return shortest;
}

// Gets the new potential path
Path *get_potential(Path *p1, Path *p2) {

}
#endif
```

```
#ifndef FLOYD
#define FLOYD
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <stdio.h>
#include "attraction.h"

const int INF =  std::numeric_limits<int>::max();

//INPUT: complete graph, with edge weights calculated by distanceTo
//OUTPUT: matrix where [i][j] represents the shortest path weight from i to j
std::vector<std::vector<int> > floyd(std::vector<attraction> &attrs) {
        //initialize apsp matrix
        std::vector<std::vector<std::vector<int> > > temp(attrs.size() + 1,
                                                                                std::vec
tor<std::vector<int> >(attrs.size(),

std::vector<int>(attrs.size(), INF)));

        for(int i = 0; i < attrs.size(); i++) {
                for(int j = 0; j < attrs.size(); j++) {
                        if(i == j)
                                temp[0][i][j] = 0;
                        else //this is the only line that needs to be changed
                                temp[0][i][j] = std::min(INF, attrs[i].distanceT
o(attrs[j]));
                }
        }

        //calculate all-pairs shortest path
        for(int k = 1; k < attrs.size() + 1; k++) {
                for(int i = 0; i < attrs.size(); i++) {
                        for(int j = 0; j < attrs.size(); j++) {
                                temp[k][i][j] =
                                        std::min(
                                                temp[k - 1][i][k - 1] + temp[k -
 1][k - 1][j],
                                                temp[k - 1][i][j]
                                        );
                        }
                }
        }

        return temp[attrs.size()];
}

//print apsp
void printApsp(std::vector<std::vector<int> > &apsp, int numItems) {
        for(int i = 0; i < numItems; i++) {
                for(int j = 0; j < numItems; j++) {
                        printf("%-7d", apsp[i][j]);
                }
                std::cout << std::endl;
        }
}

#endif
```

```
#include "floyd.h"
#include "../graph.hh"
// undirected graph tests
void undirected_tests(void); // runs all undirected graph tests
void undirected_one_vertex(void);
void undirected_two_vertices(void);
void undirected_three_vertices(void);

// unit tests for undirected graphs
void undirected_tests(void) {
        undirected_one_vertex();
        undirected_two_vertices();
        undirected_three_vertices();
}

// Tests floyd for one vertex (trivial case)
void undirected_one_vertex(void) {
        Graph g(1);
        DistanceMatrix *asps = floyd(&g);
        Path *p = asps->at(0).at(0);
        assert(p != NULL);
        assert(p->weight == 0);
        delete_distance_matrix(asps);
}

// Tests floyd for two vertices
void undirected_two_vertices(void) {
        Graph g(2);
        g.add_edge(0, 1, false, 10); // edge between two vertices

        DistanceMatrix *asps = floyd(&g);
        assert(asps->at(0).at(0)->weight == 0);
        assert(asps->at(1).at(1)->weight == 0);
        assert(asps->at(0).at(1)->weight == 10);
        assert(asps->at(1).at(0)->weight == 10);
        delete_distance_matrix(asps);
}

// Tests floyd with three vertices
void undirected_three_vertices(void) {
        Graph g(3);

        // Initialize edges
        g.add_edge(0, 1, false, 10);
        g.add_edge(1, 2, false, 20);
        g.add_edge(0, 2, false, 200);

        // Check results
        DistanceMatrix *asps = floyd(&g);
        assert(asps->at(0).at(1)->weight == 10);
        assert(asps->at(1).at(2)->weight == 20);
        assert(asps->at(0).at(2)->weight == 30);
        delete_distance_matrix(asps);
}

int main() {
        undirected_tests();
        return 0;
}
```

```
//--------------------inputs-------------------------
// int type : n.
//              - the number of nodes
// array of double type: cMat.
//              - connectivity matrix, 0 means disconnected
//                and distances are all positive.
//              - Is of length (n*n).
//--------------------outputs------------------------
//double** type: dist_mat
//              - the shortest path, (ANSWER)
//int** type: pred_mat
//              - predicate matrix, used for reconstructing
//                shortest routes

void warshall(int n, double *cmat, double **dist_mat, int **pred_mat){
        double *dist;
        int *pred;
        int i, j, k; //counters

        //initialize data structs
        dist = (double *)malloc(sizeof(double) * n * n);//allocate dist[n*n]
        pred = (int *)malloc(sizeof(int) * n * n);//allocate pred[n*n]
        memset(dist, 0, sizeof(double)*n*n);//sets dist[all indices] to 0
        memset(pred, 0, sizeof(int)*n*n);//sets pred[all indices] to 0

        //initialize algorithm
        for ( i = 0; i < n; i++ ){
                for ( j= 0; j < n; j++ ){
                        if (cmat[i*n+j] != 0.0)
                                dist[i*n+j] = cmat[i*n + j];
                        else
                                dist[i*n+j] = HUGE_VAL; //path disconnected

                        if (i == j) //diagonal case
                                dist[i*n+j] = 0;

                        if ((dist[i*n +j] > 0.0) && (dist[i*n+j] < HUGE_VAL))
                                pred[i*n+j] = i;
                }
        }

        //main loop
        for ( k = 0; k < n; k ++ ) {
                for ( i = 0; i < n; i ++ ) {
                        for ( j = 0; j < n; j ++ ) {
                                if ( dist[i*n+j] > (dist[i*n+k] + dist[k*n+j]))
{
                                        dist[i*n+j] = dist[i*n+k] + dist[k*n+j];
                                        pred[i*n+j] = k;
                                }
                        }
                }
        }

        //print out results of all the shortest dist
        //***if we want to output right away***
        /*
        for( i = 0; i < n; i++) {
                for ( j = 0; j < n; j++) {
                        cout << dist[i*n+j] << endl;
                }
        }
        */

        if ( dist_mat )
                *dist_mat = dist;
        else
                free(dist);
```



```
        if ( pred_mat )
                *pred_mat = pred;
        else
                free(pred);
}
```