WHAT THE LAB IS ABOUT AND UNDERSTANDING

Purpose of the program:
Help Denver Long reach his destination with several options for pathing

Intro:

1) Vertices -> V = {v0,....vn}
2) Edges -> E = {<vi,vj>,...}: connected the vertices

Edge connects 1 vertex to another

The question is: How to represent visiting each places 1 time and coming back at the starting location?

Use a graph!

	0	1	2	3	4	5	⋯	25
0	0	10	0	0	0	0	0	0
1	0	0	2	5	0	0	0	0
2	0	0	0	0	0	3	0	5
3	0	0	0	0	21	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
⋮	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0

Each edge will be represented as a triple $\langle i, j, k \rangle$. The set of edges in the adjacency matrix above is

$$E = \{\langle 0,1,10\rangle, \langle 1,2,2\rangle, \langle 1,3,5\rangle, \langle 2,5,3\rangle, \langle 2,25,5\rangle, \langle 3,4,21\rangle\}.$$

With k being the weight

If the above adjacency matrix were made to be *undirected*, it would be reflected along the diagonal.

	0	1	2	3	4	5	⋯	25
0	0	10	0	0	0	0	0	0
1	10	0	2	5	0	0	0	0
2	0	2	0	0	0	3	0	5
3	0	5	0	0	21	0	0	0
4	0	0	0	21	0	0	0	0
5	0	0	3	0	0	0	0	0
⋮	0	0	0	0	0	0	0	0
25	0	0	5	0	0	0	0	0

So something like 0,1,10 would be 1,0,10 instead

We must implement **ADTs** for the graph. An **ADT** is an *abstract data type*. With any **ADT** comes an interface composed of *constructor, destructor, accessor, and manipulator functions*.

Pseudo code for graph.c:

```
graph_vertices(Graph *G)
        Return vertices

graph_add_edge(Graph *G, i, j, k)
(this is a manipulator, used for altering field of data types)
        Matrix[i][j] = k
        If i  > j or j > n(number of vertices)
                Matrix[j][i] = k
                Return false
        Return true

graph_has_edge( Graph *G, i, j)
        If i <= j or j <= n(number of vertices)
                Return true
        Return false

graph_edge_weight(Graph *G, i, j)
        If graph_has_edge(G, i, j) == false
                Return 0
        Return matrix[i][j]
```

Depth-First Search:

- Need a way to search thru the graph
- Once we have examined a vertex, we do not want to do so again
- DFS first marks the vertex v as having been visited, then iterates through all of the edges <v,w>, recursively calling itself starting at w if w has not already been visited
- Hamiltonian path then reduces to
    1. Using DFS to find paths that pass through all vertices
    2. There is an edge from the last vertex to the first. The solutions to the traveling salesman problem and then the shortest Hamiltonian path

Stacks:
Mostly storing items/values here
If we observe the struct, we have top, capacity, and items. This is all to keep track of the items in the array and to manipulate the growing stack

Command-line options:

-h prints out a help message describing the purpose of the graph and the command-line options it accepts, exiting the program afterwards

-v enables verbose printing. Prints out all Hamiltonian paths found as well as the total number of recursive calls to dfs()

-u specified the graph to be undirected

-i infile: specify the input file path containing the cities and edges of a graph

-o outfile: specify the output file path to print to. If not, the default output should be set as stdout

Expected input/output

For the input, this is a little different compared to before since we are dealing with multiple lines of input

First, we take in number of vertices, or cities, in the graph
Second, the next n lines of the file are the names of the cities
Third, scanned in as triple <i,j,k> and interpreted as an edge from vertex i to vertex j with weight k

Example of input:

```
$ cat mythical.graph
4
Asgard
Elysium
Olympus
Shangri-La
0 3 5
3 2 4
2 1 10
1 0 2
```

Example of output:

```
$ ./tsp < mythical.graph
Path length: 21
Path: Asgard -> Shangri-La -> Olympus -> Elysium -> Asgard
Total recursive calls: 4
```

If the verbose command-line option was enabled, print out *all* the Hamiltonian paths that were found as well. It is recommended that you print out the paths as you find them.

```
$ ./tsp -v < ucsc.graph
Path length: 7
Path: Cowell -> Stevenson -> Merrill -> Cowell
Path length: 6
Path: Cowell -> Merrill -> Stevenson -> Cowell
Path length: 6
Path: Cowell -> Merrill -> Stevenson -> Cowell
Total recursive calls: 5
```

THE OUTPUT OF THE PROGRAM MUST MATCH THE OUTPUT OF THE REFERENCE PROGRAM WHEN GIVEN A PROPERLY FORMATTED GRAPH TO RECEIVE FULL CREDIT

Files in this lab:

Graph.h
Graph.c
Path.h
Path.c
Stack.h
Stack.c
Tsp.c
Vertices.h

Cannot modify the header files