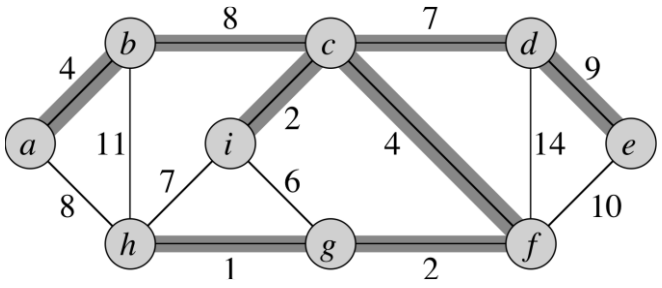
Spring 2016 CSC 325
Assignment 1. Implement Prim's Algorithm for Minimum Spanning Tree
Due: Mon. Feb. 1, 11:59pm

Pseudocode for Prim's Algorithm, p. 634

G is a connected graph
r is the root of the Minimum Spanning Tree
Q is a minimum-priority queue with key attribute
u and v denote a vertex. v.key is the minimum weight of any edge connecting v to some vertex in the tree and v.π is the parent of v in the tree.
w is a weight function on two vertices. w(u, v) is the weight of the edge between vertices u and v.

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```



The vertex coordinates of a connected graph are provided in **standard graphics coordinate system** as a series of integers separated by whitespace: the number N of points, then in sequence x1, y1, x2, y2, . . ., xN, yN.
Example data file:

```
9
50 100
100 150
200 150
300 150
350 100
300 50
200 50
100 50
150 100
```

For each of the following steps, you will turn in a separate C++ program that demonstrates in sensible output the results of the step. In an attempt to reduce file misnaming, dummy empty files for you to edit have been placed into your eccentric folder, named with your MSU ID and each step.

Step A. SETUP. Use **Assn1A_Your_MSU_ID.cpp** for this step.
Read all (x,y) coordinate points from a text file whose name is a parameter to **`main( )`**. The data is a series of integers: the number N of points, then in sequence x1, y1, x2, y2, . . ., xN, yN. The integers are separated by whitespace NOT by commas.
For your own planning: How will you store points?

Step B. ALGORITHM. Use **Assn1B_Your_MSU_ID.cpp** for this step.
Compute lengths of the edge between every pair of points.
For your own planning: How will you loop to identify every pair of points? (Every point acts as a START, every point acts as an END.) How will you store the lengths of the edges and the START/END points that defined that edge?

Step C. ALGORITHM. Use **Assn1C_Your_MSU_ID.cpp** for this step.
Sort edges in increasing length order. The result is essentially a priority queue.
For your own planning: Did the way you STORED the edges make it easy to SORT edges by length?

Step D. ALGORITHM. Use **Assn1D_Your_MSU_ID.cpp** for this step.
Create a "set" mechanism to contain those points which are already in the Minimum Spanning Tree. By default, points not in the set are not yet contained in the MST – but if you prefer, make a second set to explicitly contain the "not yet…" points.
For your own planning: How will you store the set?
How will you query the set storage to determine if an element is a member?

Step E. ALGORITHM. Use **Assn1E_Your_MSU_ID.cpp** for this step.
Create the Minimum Spanning Tree using Prim's Algorithm, p. 634.
For your own planning: In this step, do not display/demonstrate the outcomes of the previous steps.
For your own planning: What printouts will display the MST you found?

Step F. DISPLAY. Use **Assn1F_Your_MSU_ID.cpp** for this step.
Graphically display the edge-by-edge creation of the Minimum Spanning Tree using BearPlot graphics written in Python. BearPlot and a TestPlot.dat demonstration files are provided on eccentric.
In this step, do not display/demonstrate the outcomes of the previous steps.