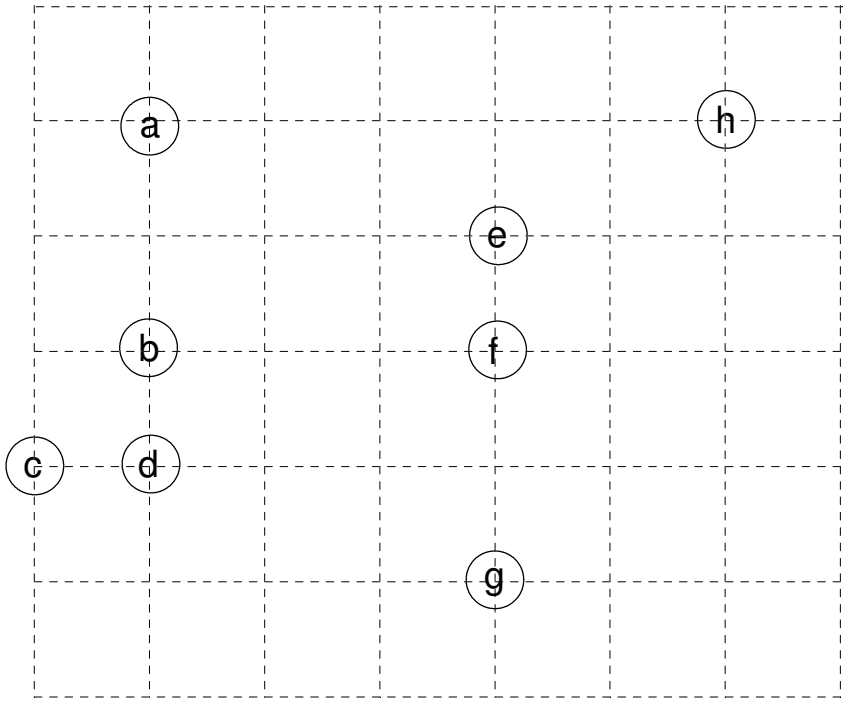
1. Suppose you had a map of settlements on the planet X
   (Assume edges could connecting all vertices with their Euclidean distances as their costs)



We want to build roads that allow us to travel between any pair of cities. Because resources are scarce, we want the total length of all roads build to be minimal. Since all cities will be connected anyway, it does not matter where we start, but assume we start at "a".

a) Assuming we start at city "a" which city would you connect first? Why this city?

b) What city would you connect next to expand your partial road network?

c) What would be some characteristics of the resulting "graph" after all the cities are connected?

d) Does your algorithm come up with the overall best (globally optimal) result?

2. Prim's algorithm for determining the minimum-spanning tree (MST) of a graph is another example of a *greedy algorithm*. Unlike divide-and-conquer and dynamic programming algorithms, greedy algorithms DO NOT divide a problem into smaller subproblems. Instead a greedy algorithm builds a solution by making a sequence of choices that look best ("locally" optimal) at the moment without regard for past or future choices (no backtracking to fix bad choices).

a) What greedy criteria does Prim's algorithm use to select the next vertex and edge to the partial minimum spanning tree?

b) What data structure would be useful it finding the next vertex to add to the partial MST?

The textbook's Prim's Algorithm code (Listing 7.12 p. 346) was incorrect, but **fixed on-line to**:

```
def prim(G,start):
    pq = PriorityQueue()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in G])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
          newCost = currentVert.getWeight(nextVert) + currentVert.getDistance()
          if nextVert in pq and newCost < nextVert.getDistance():
              nextVert.setPred(currentVert)
              nextVert.setDistance(newCost)
              pq.decreaseKey(nextVert,newCost)
```

c) The `PriorityQueue` class used is shown on the next page and is similar to the `BinHeap` class we have been using, except:

- `self.heapArray` is a list of tuples with the first tuple value (tuple index 0) being the "priority" and second tuple value (tuple index 1) being its associated value.
  In Prim's algorithm what is the priority value?

  In Prim's algorithm what is the associated value?

- a `__contains__` method is added to check if a value is in the priority queue.
  In Prim's algorithm where is the `__contains__` method invoked?

- a `decreaseKey` method is added to allow a priority value to be reduced (i.e., increasing its priority).
  In Prim's algorithm when is the `decreaseKey` method used?

d) As written what is the big-oh of each of the methods?

- `__contains__` method?

- `decreaseKey` method?

The `PriorityQueue` class including the `__contains__` and `decreaseKey` methods:.

```
class PriorityQueue:
    def __init__(self):
        self.heapArray = [(0,0)]
        self.currentSize = 0

    def buildHeap(self,alist):
        self.currentSize = len(alist)
        self.heapArray = [(0,0)]
        for i in alist:
            self.heapArray.append(i)
        i = len(alist) // 2
        while (i > 0):
            self.percDown(i)
            i = i - 1

    def percDown(self,i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapArray[i][0] > self.heapArray[mc][0]:
                tmp = self.heapArray[i]
                self.heapArray[i] = self.heapArray[mc]
                self.heapArray[mc] = tmp
            i = mc

    def minChild(self,i):
        if i*2 > self.currentSize:
            return -1
        else:
            if i*2 + 1 > self.currentSize:
                return i*2
            else:
                if self.heapArray[i*2][0] < self.heapArray[i*2+1][0]:
                    return i*2
                else:
                    return i*2+1

    def percUp(self,i):
        while i // 2 > 0:
            if self.heapArray[i][0] < self.heapArray[i//2][0]:
               tmp = self.heapArray[i//2]
               self.heapArray[i//2] = self.heapArray[i]
               self.heapArray[i] = tmp
            i = i//2

    def add(self,k):
        self.heapArray.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)

    def delMin(self):
        retval = self.heapArray[1][1]
        self.heapArray[1] = self.heapArray[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapArray.pop()
        self.percDown(1)
        return retval

    def isEmpty(self):
        if self.currentSize == 0:
            return True
        else:
            return False

    def decreaseKey(self,val,amt):
        # this is a little wierd, but we need to find the
        # heap thing to decrease by looking at its value
        done = False
        i = 1
        myKey = 0
        while not done and i <= self.currentSize:
            if self.heapArray[i][1] == val:
                done = True
                myKey = i
            else:
                i = i + 1
        if myKey > 0:
            self.heapArray[myKey] = (amt,self.heapArray[myKey][1])
            self.percUp(myKey)

    def __contains__(self,vtx):
        for pair in self.heapArray:
            if pair[1] == vtx:
                return True
        return False
```

3. If we want to speed-up the `__contains__` and `decreaseKey` methods, then what type of data structure could we add to aid in:

- checking for the existence of a key value, and
- if a key value exists in the heap, then at what index does it reside?

4. What modifications would need be needed to other methods to keep the data structure in question 3 up-to-date?