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)

![Graph Diagram]

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?

   \[ \text{It is the closest city to } a. \]

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

   \[ \text{next closest to anything in partial road system (priority queue / min. heap)} \]

   \[ \text{connected graph with no cycles} \Rightarrow \text{spanning} \]

   \[ \text{tree} \]

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

   \[ \text{bestest / min. spanning tree} = \text{MST} \]
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?

```
vertex closest to any vertex in partial MST
```

b) Consider the textbook’s Prim’s Algorithm code (Listing 7.12 p. 346) which is incorrect.

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

c) What is wrong with the code? (Fix the above code.)
(1) BinHeap instead of Priority Queue
(2) newCost calculation wrong
(3) "v" should nextVert in if-statement
(4) pq should a heap of Priority Queue Entry objects
(5) add -- contains -- to BinHeap

3. To avoid “massive” changes to the BinHeap class, it can store PriorityQueueEntry objects:

```
class PriorityQueueEntry:
    def __init__(self, x, y):
        self.key = x
        self.val = y
    def getKey(self):
        return self.key
    def getValue(self):
        return self.val
    def setValue(self, newValue):
        self.val = newValue
```

```
def __lt__(self, other):
    return self.key < other.key
def __gt__(self, other):
    return self.key > other.key
def __eq__(self, other):
    return self.val == other.val
def __hash__(self):
    return hash(self.key)
```

a) Update the above Prim’s algorithm code to use PriorityQueueEntry objects.
b) Why do the _lt_ and _gt_ methods compare key attributes, but _eq_ compare val attributes?
c) When used for Prim's algorithm what type of objects are the `vals` compared by `eq`?

- **Vertex objects**

d) What changes to the Graph and Vertex classes need to be made?

- None?

e) Complete the `__contains__` and `decreaseKey` methods.

```python
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0
    def buildHeap(self,alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            self.percDown(i)
            i = i - 1
    def percDown(self, i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i] > self.heapList[mc]:
                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
                i = mc
    def minChild(self, i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i*2] < self.heapList[i*2+1]:
                return i * 2
            else:
                return i * 2 + 1
    def percUp(self, i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i//2]:
                tmp = self.heapList[i//2]
                self.heapList[i//2] = self.heapList[i]
                self.heapList[i] = tmp
                i = i // 2
    def insert(self, k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)
    def delMin(self):
        retval = self.heapList[1]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval
    def isEmpty(self):
        return self.currentSize == 0
    def size(self):
        return self.currentSize
    def __str__(self):
        return str(self.heapList[1:])
```

```python
def __contains__(self, value):
    for index in range(1, len(self.heapList), 1):
        if value == self.heapList[index]:
            self.foundIndex = index
            return True
    return False

def decreaseKey(self, decreasedValue):
    """Precondition: decreasedValue in heap already""
    if decreasedValue not in self:
        raise ValueError("")
    self.heapList[self.foundIndex] = decreasedValue
    self.percUp(self.foundIndex)
```