Project 3

Dijkstra's algorithm:

```
# Time: The complexity changes if we use a MinHeap or an UnSortedArray. For UnSortedArray we get a total complexity of
#       O(V^2+E), for MinHeap, we end up with O((V+E)logV) time complexity, due to the nature of the MinHeap functions
# Space: For both the UnSortedArray and the MinHeap, the space used is O(V) to hold all the nodes and their previous nodes
#       ( as well as distances, but considering we are simply holding two different arrays, O(2V) goes to O(V))
    def computeShortestPaths( self, srcIndex, use_heap=False ):
        self.start = srcIndex
        t1 = time.time()

        distanceToNode = {}
        previousNodeList = {}

        for node in self.network.nodes:
            distanceToNode[node.node_id] = float("inf")
            previousNodeList[node.node_id] = "x"
        distanceToNode[srcIndex] = 0

        if use_heap:
            priorityQueue = MinHeap(srcIndex, self.network.nodes)    #O(V) time
        else:
            priorityQueue = UnSortedArray(srcIndex, self.network.nodes)      #O(V) time

        while priorityQueue.getLength() > 0:
            currentNode = priorityQueue.deleteMin()
            #We hit this part of the code at worst V times as we could loop through each node.
            #We end up with a time complexity for deleteMin() * V loops being either O(VlogV) for MinHeap or O(V^2) for the UnSortedArray

            for edge in self.network.nodes[currentNode].neighbors:      # O(1) time, because each node only has 3 neighbors
                if edge.length + distanceToNode[currentNode] < distanceToNode[edge.dest.node_id]:
                    distanceToNode[edge.dest.node_id] = edge.length + distanceToNode[currentNode]
                    previousNodeList[edge.dest.node_id] = currentNode
                    priorityQueue.decreaseKey(edge.dest.node_id, distanceToNode[edge.dest.node_id])
                    #We only hit this part of the code at worst case E times, resulting in a time complexity of O(ElogV) for the
                    #MinHeap and just O(E) for the UnSortedArray
        self.costArray = distanceToNode
        self.previousNodeList = previousNodeList

        #So we get a combined complexity at the end of O(V^2 + E) for the UnSortedArray, and O((V+E)logV) for the MinHeap
        t2 = time.time()
        return (t2-t1)
```

For each node popped from priority queue, check all edges and see if the distance to that edge from the current node is less than the distance stored, if so, add the new distance, and make that the new distance, and update the queue (deleteMin())

Getting the shortest path:

```
# Time: At worst case we go through all the nodes, and we check each edge at every node, which turns into O(V+E) time
# Space: At worst case, we need to store all the nodes and all the edges, which is a space complexity of O(V+E)
    def getShortestPath( self, destIndex ):
        path = []
        previousNodeList = self.previousNodeList
        costArray = self.costArray
        currentNode = destIndex

        while currentNode != self.start:
            if previousNodeList[currentNode] == "x":
                return {'cost': float("inf"), 'path': []}

            edgeLength = 0
            for edge in self.network.nodes[previousNodeList[currentNode]].neighbors: # O(1) time, because each node only has 3 neighbors
                if edge.dest.node_id == currentNode:
                    edgeLength = edge.length
                    if edge.length == "x":
                        return {'cost': float("inf"), 'path': []}
            path.append((self.network.nodes[currentNode].loc, self.network.nodes[previousNodeList[currentNode]].loc, '{:.0f}'.format(edgeLength)))
            currentNode = previousNodeList[currentNode]
        return {'cost': costArray[destIndex], 'path': path}
```

Start from the destination node, and then add for the path each previous node until we get to the starting node. If the previous node is “x”, or the distance is “x”, then that means there is no possible route from the destination node to the source node. Else, just make the path array, and also return the distance to the destination node.

Unsorted Array Implementation:

```
class UnSortedArray:

# Time: O(V), makeQueue is O(V)
# Space: O(V), same as makeQueue
    def __init__(self, startingNode, allNodes):
        self.makeQueue(startingNode, allNodes)
        return

# Time: O(V), because we just initialize the list of nodes in the network
# Space: O(V), because we store all those values into an array
    def makeQueue(self, startingNode, allNodes):
        self.array = []
        self.length = 0;
        for node in allNodes:   # Runs through for each node V
            self.insert(node.node_id) #O(1) time and space
            self.length += 1
        self.array[startingNode] = 0;


# Time: O(1), its a simple insert
# Space: O(1), inserts a new object into an array of objects.
    def insert(self, node):
        self.array.append(node)
        self.array[node] = float("inf")
        return

# Time: O(1), we don't alter the array
# Space: O(1), we are simply changing a value at an index
    def decreaseKey(self, index, value):
        self.array[index] = value
        return

# Time: O(1)
# Space: O(1)
    def getLength(self):
        return self.length

# Time: Worst case, deleteMin() iterates through each node V, O(V)
# Space: O(1), it just changes the array
    def deleteMin(self):
        index = 0
        min_index = 0
        min = 999999

        for distance in self.array:         # O(V), at most goes through each node
            if distance != "x" and min > self.array[index]:
                min = self.array[index]
                min_index = index
            index += 1

        self.decreaseKey(min_index, "x")
        self.length -= 1
        return min_index
```

MakeQueue():

MakeQueue() runs in O(V) time because it needs to iterate through each node in the graph, and calls insert() for each one. Insert() is O(1), so total we get O(V) for setting up the initial array.

Insert():

Insert() runs in O(1) time, because it only appends a value at the end of an array (O(1)) , and changes it to infinity

DeleteMin():

DeleteMin() runs in O(V) time, because it needs to iterate through each element in the queue to find the minimum index, and return that. Each time it is called, it will go through each element in the array

DecreaseKey():

DecreaseKey() is O(1), because it simply goes to an index in the array, and alters it which is constant time with a given index.

Total Complexity:

The total complexity for Dijkstra's with an unsorted array is O(V^2 + E) as explained in the code comments above my algorithm. Because each call to DeleteMin() is O(V), and Dijkstras at worst case needs to iterate over each node V, we get a complexity of O(V^2), adding on top of that the number of edges of each node with is E, so our final complexity is O(V^2 + E)

Min Heap Implementation:

```python
class MinHeap:

# Time: O(V), due to makeQueue
# Space: O(V), due to makeQueue
    def __init__(self, startingNode, allNodes):
        self.makeQueue(startingNode, allNodes)
        return

# Time: O(V), We loop through every node V and insert it into heap, then we alter distance at startingNode and bubble
#       that up to the root which takes O(logV) time, and we do this once, for a total O(V + logV) which goes to O(V)
# Space: O(V), we keep 3 arrays of size V from this function: O(3V) => O(V)
    def makeQueue(self, startingNode, allNodes):
        self.size = 0
        self.heap = []
        self.heapMap = []
        self.distances = []

        for x in allNodes:
            self.insert(x)
        self.distances[startingNode] = 0
        self.bubbleUp(self.heap[startingNode])
        return

# Time: O(1), simple inserts into the heap and map arrays
# Space: O(1), just inserting values into an array is constant
    def insert(self, node):
        self.heap.append(node.node_id)
        self.distances.append(float("inf"))
        self.heapMap.append(node.node_id)
        self.heapMap[node.node_id] = self.size
        self.size += 1
        return

# Time: O(1)
# Space: O(1)
    def getLength(self):
        return self.size
```

```python
# Time: O(logV), at worst case, we have one node travel all the way up one side of the tree, which results in logV time
# Space: O(1), because we are simply swapping values, but no other array is created
    def bubbleUp(self, index):
        parentGreater = True
        while index != 0 and parentGreater:
            childNode = self.heap[index]
            parentIndex = self.parentIndex(index)
            parentNode = self.heap[parentIndex]
            childDistance = self.distances[self.heap[index]]
            parentDistance = self.distances[self.heap[self.parentIndex(index)]]
            if childDistance < parentDistance:

                self.heap[index] = parentNode
                self.heap[parentIndex] = childNode

                self.heapMap[childNode] = parentIndex
                self.heapMap[parentNode] = index
            else:
                parentGreater = False

            index = self.parentIndex(index)

        return

# Time: O(logV), the only time complexity here comes from the call to bubbleUp(), which is logV time worst case
# Space: O(1), only alteration at a index is done here, even bubbleUp() is O(1) space
    def decreaseKey(self, index, value):
        self.distances[index] = value
        self.bubbleUp(self.heapMap[index])
        return
```

MakeQueue():

MakeQueue() runs in O(V) time, because it populates 3 arrays by iterating through each node in the graph once, and calling Insert(). After populating the array, it calls bubbleUp() once, which runs in O(logV) time, and bubbles the starting node up to the top of the heap to the root. Because we only call bubbleUp() once after the heap has been initialized, we don't add that complexity to the total equation, and we end up only bubbling up once, which ends us with a complexity of O(V)

Insert():

Insert() is only O(1) time, because it only appends and alters the arrays with the new elements, which is a collection of O(1) complexity function calls.

DeleteMin():

DeleteMin() runs in O(logV) time. We first swap the first element with the last before deleting it so that the python delete only run in O(1) time as it deletes the last element of the list. Then we trickle down the last element, with at worst case we trickle it down all the way to the bottom of the tree, which takes O(logV) time, as we only would be trickling it down one side of the tree, the side that the minimum child is on each time.

DecreaseKey():

DecreaseKey() runs in O(logV) time. The bulk of the complexity comes from the call to bubbleUp() after the value in the heap is altered, in order to put it in the new position in the heap

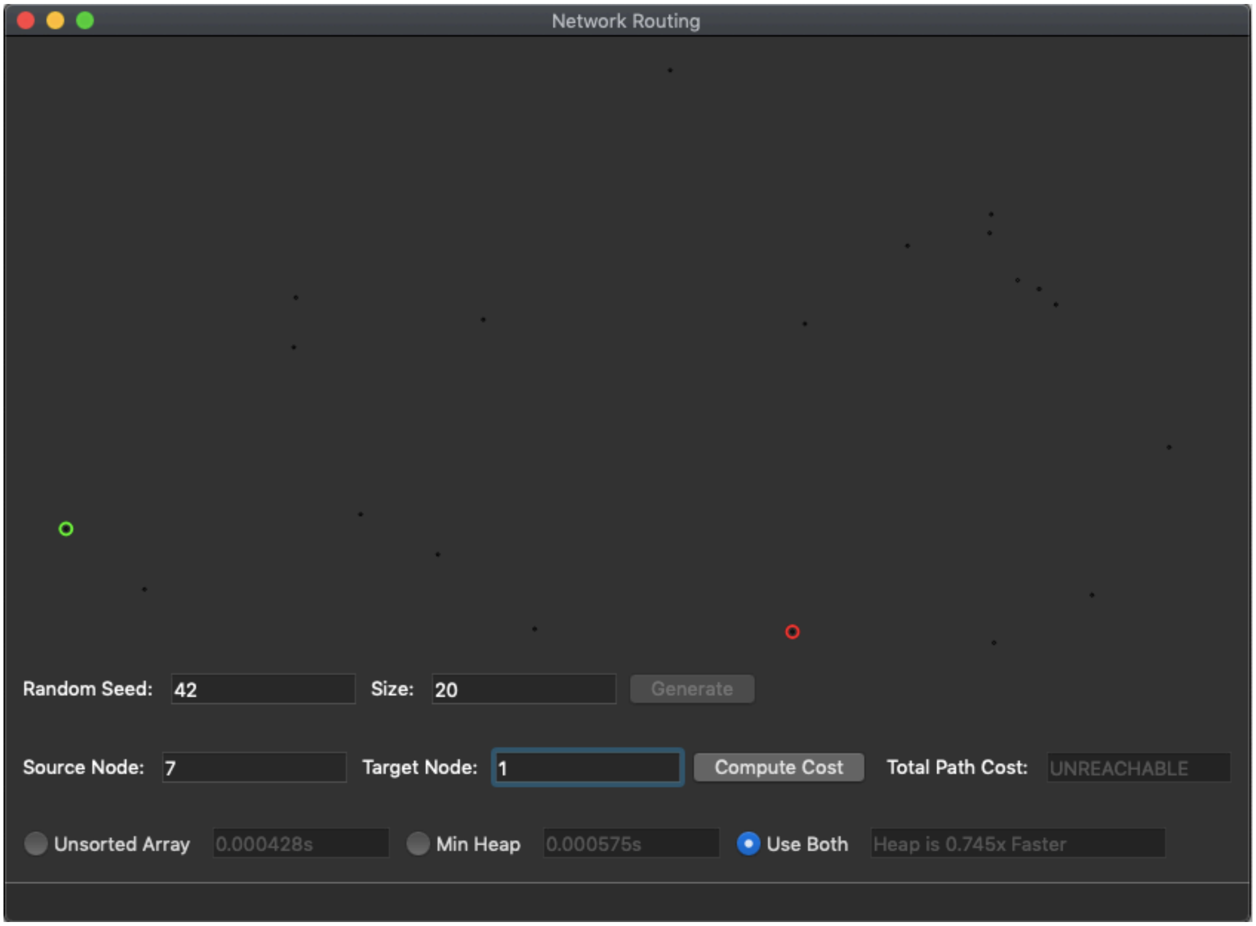
bubbleUp():

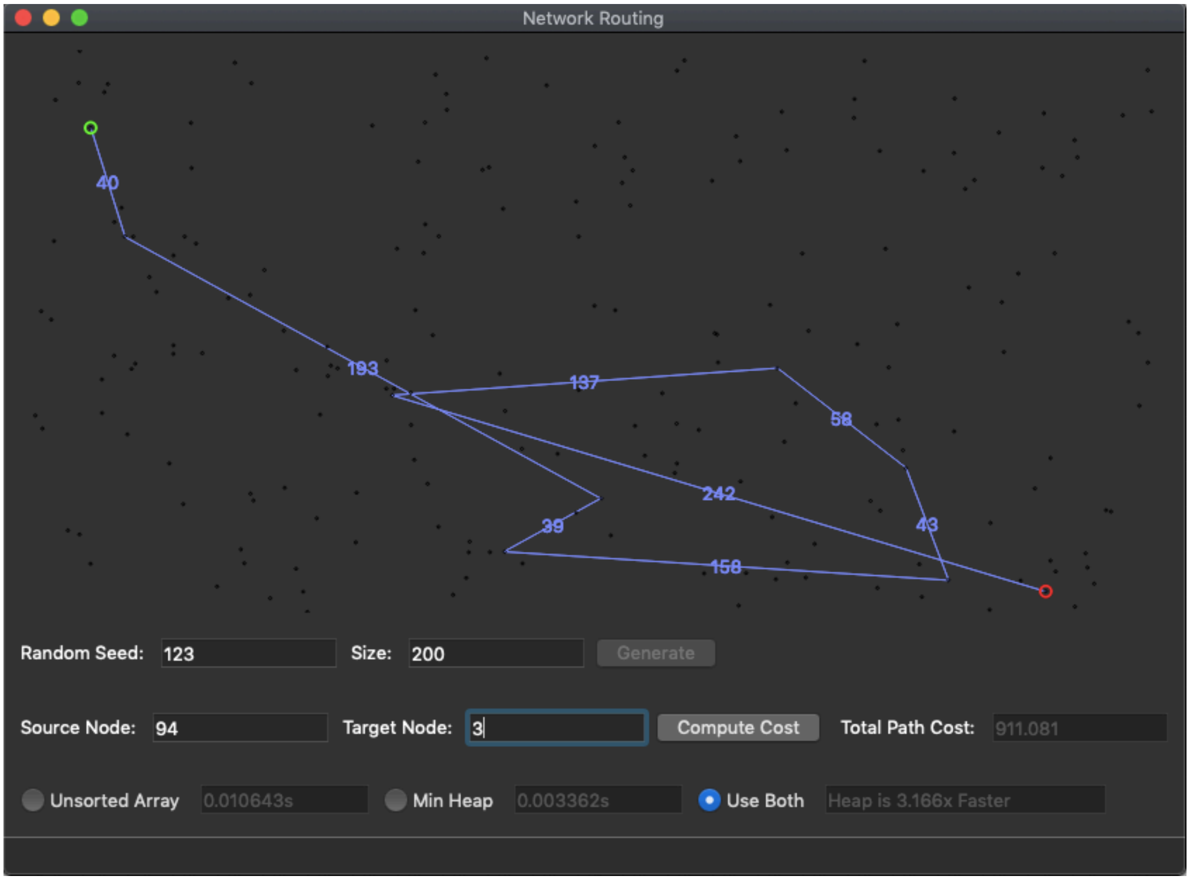
bubbleUp() runs in O(logV) time. The worst case time complexity is logV because at worst case we take a node from the bottom of one side of a tree and swap it all the way to the root of the tree, which takes logV swaps.

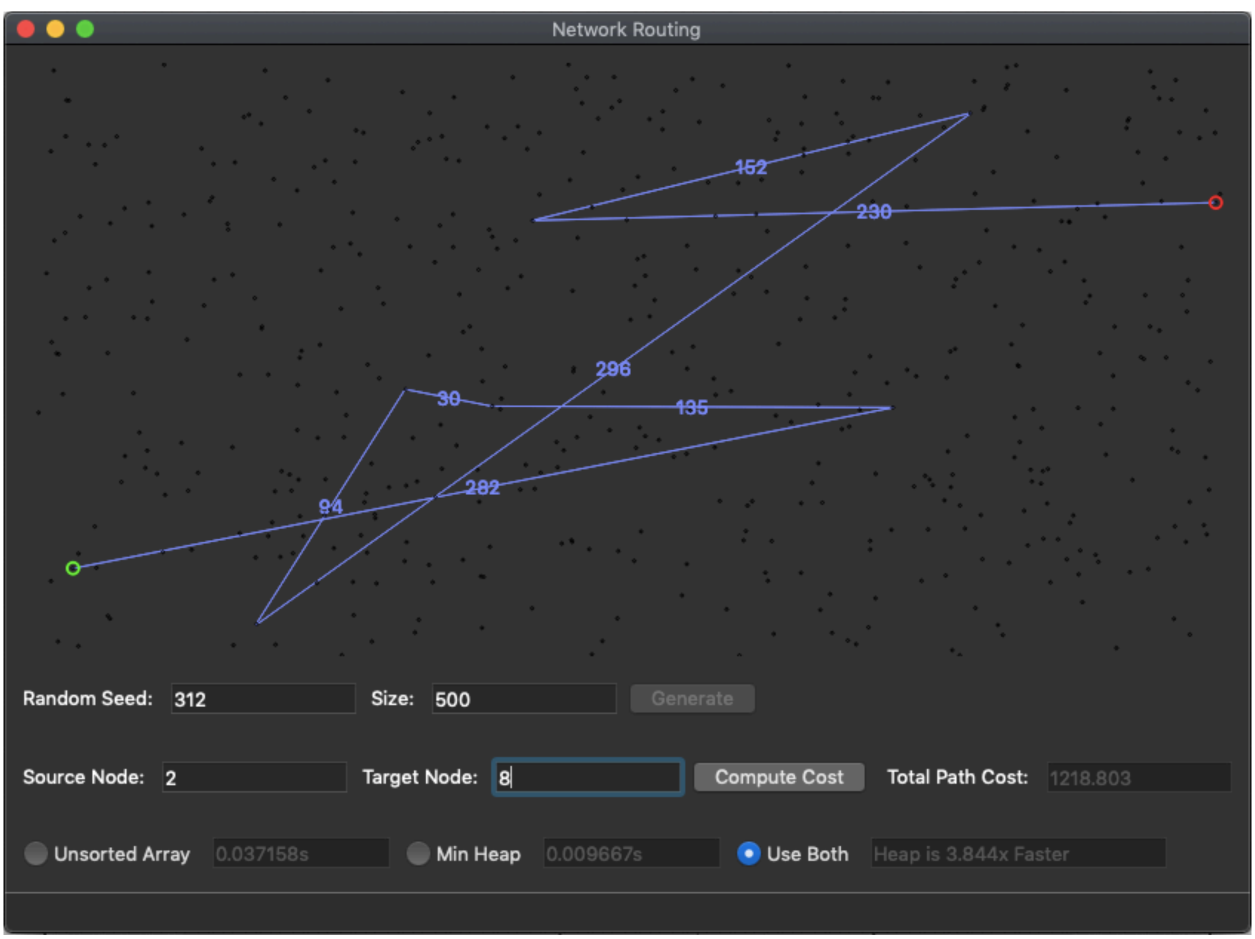
Total Complexity:

The total complexity for Dijkstra's with a MinHeap is O((V+E)logV) as explained in the code comments above my algorithm. Because each call to DeleteMin() is O(logV), and Dijkstras at worst case needs to iterate over each node V, we get a complexity of O(VlogV), adding on top of that the number of edges of each node with is E, and each call to DecreaseKey() is O(logV) complexity, we get a final complexity of O(VlogV + ElogV) or O((V+E)logV)

Expected Output:




Network Routing
40
193
137
58
242
39
43
158
Random Seed: 123
Size: 200
Generate
Source Node: 94
Target Node: 3
Compute Cost
Total Path Cost: 911.081
Unsorted Array 0.010643s
Min Heap 0.003362s
Use Both Heap is 3.166x Faster


Network Routing
152
230
296
30
135
282
94
Random Seed: 312
Size: 500
Generate
Source Node: 2
Target Node: 8
Compute Cost
Total Path Cost: 1218.803
Unsorted Array 0.037158s
Min Heap 0.009667s
Use Both Heap is 3.844x Faster

Raw Data:

n: 100
Times:
0.003717, .002728, 1.363x faster
0.004861, .003324, 1.462x
0.005171, .002391, 2.163x
0.003636, .003289, 1.105x
0.005403, .002391, 2.260x

Average: 0.0045576, 0.0028246, 1.6135x faster

----
n: 1000
Times:
0.141491, .024973, 5.666x
0.155772, .019490, 7.992x
0.138188, .019418, 7.116x
0.132291, .019429, 6.809x
0.132030, .019488, 6.775x

Average: 0.1399544, 0.0205596, 6.807x faster

----
n: 10000
Times:
13.256818, .286239, 46.314x
15.524566, .326156, 47.599x
13.799562, .297933, 46.318x
14.499669, .320354, 45.261x
13.458846, 0. 273931, 49.132x

Average: 14.1078902, .3008984, 49.132x faster

----
n: 100000
Times:
1497.429437, 3.887690, 385.172x
1515.358205, 4.211106, 359.848x
1599.383380, 4.249279, 376.389x
1593.81258, 4.662545, 341.833x
1605.49502, 4.305502, 372.894x

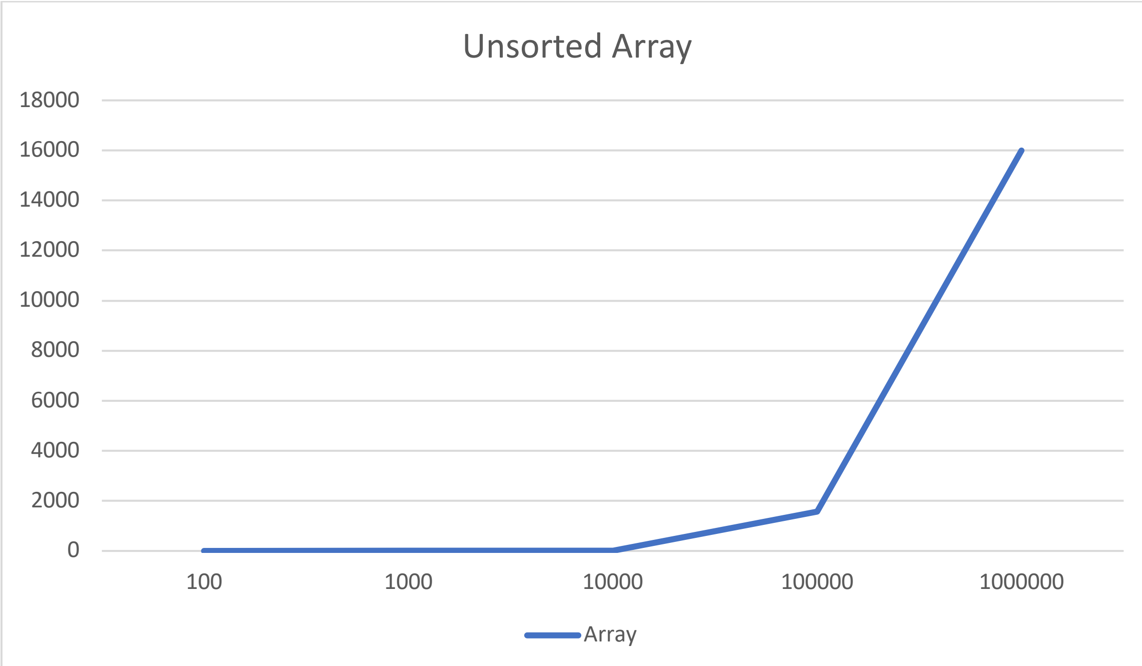
Average: 1562.2957244, 4.263224, 366.459x faster

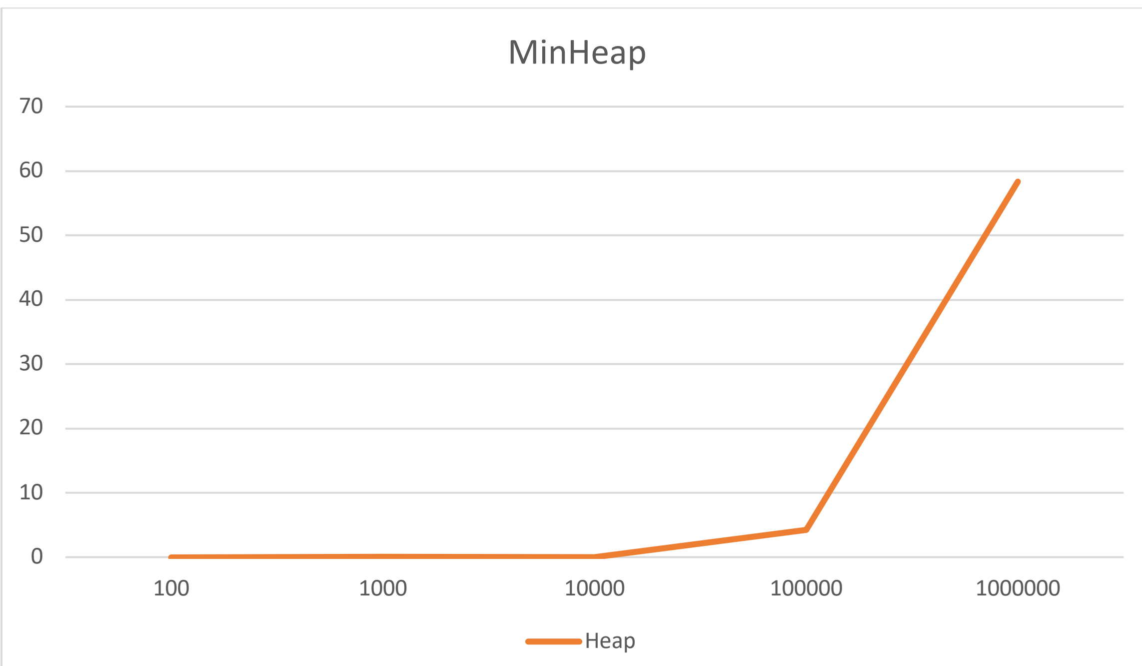
----
n: 1000000
Times:
58.394533
58.955866
57.992041
59.302918
57.059681

Average: 58.3410338

	N = 100		N = 1000		N = 10000		N = 100000		N = 1000	Estimate
	Array	Heap	Array	Heap	Array	Heap	Array	Heap	Heap	
	0.003717	0.002728	0.141491	0.024973	13.256818	0.286239	1497.429437	3.887690	58.394533	
	0.004861	0.003324	0.155772	0.019490	15.524566	0. 326156	1515.358205	4.211106	58.955866	
	0.005171	0.002391	0.138188	0. .019418	13.799562	0. 297933	1599.383380	4.249279	57.992041	
	0.003636	0.003289	0.132291	0.019429	14.499669	0. 320354	1593.81258	4.662545	59.302918	
	0.005403	0.002391	0.132030	0.019488	13.458846	0. 273931	1605.49502	4.305502	57.059681	
Average:	0.0045576	0.0028246	0.1399544	0.0205596	14.1078902	0. 3008984	1562.2957244	4.263224	58.3410338	15996.697

Graph:





After graphing the points and using linear regression formulas, I was able to estimate the array to be around 15996.697 seconds, or 4.4435 hours. This is a huge jump from the lower values of n. We barely see a difference at n=100 and even at n = 1000, but as soon as we get to 10000 points, we really see just how efficient the minimum heap is with its functions. As soon as we reach 100000, the heap is up to 360 times faster than the unsorted array.

Because for the unsorted array, the DeleteMin() function has a complexity of O(V), we can see that this makes a huge difference, especially for a graph that only has three edges for each node. This sparse graph makes it hard for the unsorted array to detect minimum values, and it can't do it as quickly as it could with more edges in each node. The MinHeap is a huge optimization strategy, as we see just how quickly it can outshine the unsorted array with bigger values of points. Although it is harder to implement conceptually, the optimization in the long run as n approaches infinity is a huge increase to efficiency for our algorithm. The simple change in complexity of the whole program from O(V^2 + E) to O((V+E)logV) is a huge change in the efficiency of the program.

Because MinHeap has at worst logV complexity, we see on the graph that logarithmic behavior. At lower values of n, we see that both the UnsortedArray and the MinHeap are fairly similar, but that as n approaches infinity, that the logarithmic nature of MinHeap displays rather accurately, increasing in time very slowly while the UnsortedArray shoots up from 15 seconds to around 25 minutes.