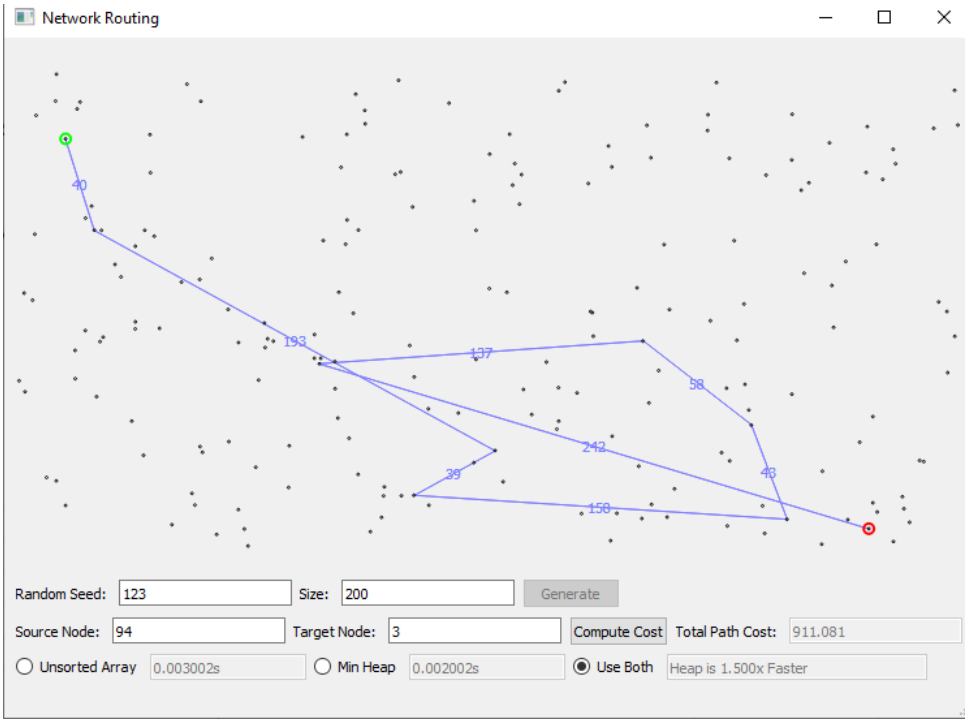
Project 3: Network Routing

1. See Attached Code.
2. I had originally written my data structures completely wrong. My first data structures had the complexity as follows:
    a. Old
        i. List-Based Queue: rq.py - See code for detailed walkthrough
            1. MakeQueue is O(n).
            2. DeleteMin is O(n).
            3. Insert is O(1)
            4. However, my DecreaseKey was made to be O(n) instead of O(1).
        ii. Heap-Based Queue: hq.py - See code for detailed walkthrough
            1. MakeQueue is O(n).
            2. DeleteMin is O(1) because it just picks the value at index 0.
            3. DecreaseKey is, unfortunately, O(n) because it relies on Insert, which I accidentally implemented as O(n).
    b. New
        i. Modified List-Based Queue: rq.py - See code for detailed walkthrough.
            1. MakeQueue is still O(n).
            2. DeleteMin is also O(n).
            3. Insert is O(1) but ultimately is unused for my implementation.
            4. DecreaseKey was rewritten to be O(1). The solution was using the node name as the key to write directly to the “keys” list, rather than searching for the index of the node in the separate node array.
        ii. Rewritten Heap-Based Queue: - hqbetter.py - used https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/?ref=lbp as a resource.
            1. Insert is a simple O(1) append function, but it does require that the tree be shuffled after a new value is added such that it follows the structure of a heap.
            2. DeleteMin is now O(Log(n)) because the new queue is based on a tree structure, whereas my old PQ was based on a list.
            3. DecreaseKey is O(Log(n)), where n is the index of the node passed in.
3. Algorithm Runtimes - See attached code for more details. Constant components are ignored in this breakdown. The code comments are more in-depth.
    a. First Try
        i. The list based algorithm runs in O(n^2) because DecreaseKey, unfortunately, was implemented incorrectly.
        ii. Similarly, the heap algorithm runs in O(n^2) because of my misimplementation.
    b. Second Try (using optimized data structures)
        i. The list based algorithm will run in O(n^2) where n is the number of vertices in the graph. This is because there’s a nested loop which calls

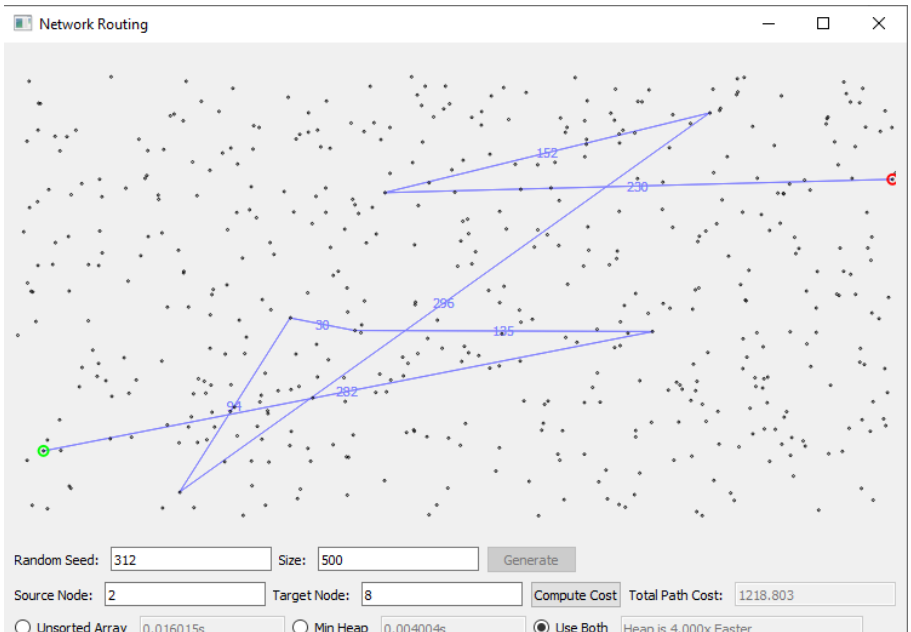
decrease_key, an O(n) operation, at most 3(O(n)) times. The space complexity for this remains pretty constantly x(n) where n is the number of vertices in the graph, and x is the number of arrays that I create in the array.

ii. The heap based algorithm will now run in O(nLog(n)), because, similarly to the list based algorithm, there is a nested call to decrease_key, which is now an O(log(n)) operation. The space complexity for this remains pretty constantly x(n) where n is the number of vertices in the graph, and x is the number of arrays that I create in the array.

4. Screenshots
    a. Unreachable. Crashes program.
    b. See screenshot.

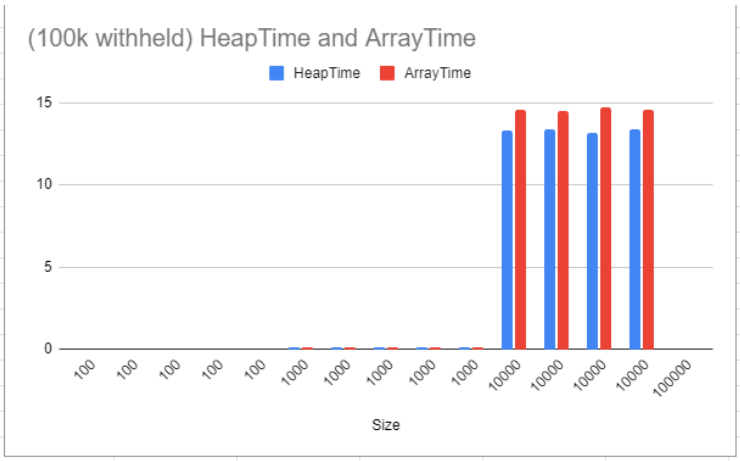


    c. See Screenshot.



5. Average Empirical Runtimes
    a. First attempt:
        i. The average for my heap for 100 is .022 seconds, the same for array. For 1000 points, the average for heap is 1.53 and the average for array is 1.38. For 10000 points, the average for heap is 13.3 and the average for array is 14.59. I was only able to make it run reasonably once other time, for 100k. Heap ran in 2179 seconds, an example of the n^2 complexity I accidentally did.

Size	HeapTime	ArrayTime
100	0.002	0.002
100	0.002	0.002
100	0.002	0.002
100	0.002	0.002
100	0.002	0.002
1000	0.151	0.136
1000	0.152	0.138
1000	0.151	0.14
1000	0.152	0.137
1000	0.151	0.136
10000	13.3	14.59
10000	13.4	14.51
10000	13.2	14.73
10000	13.4	14.6
100000		x



b. Second attempt:
   i. Now, the averages for size 100 are mostly the same. Heap gets an average of .0012, while array averages out to .001. For 1000, heap averages to .009, while array is averaging .0478. For 10000, heap averages to .095, while array averages to 3.899. The heap average for 100000 is 1.97 seconds, while the array average is 1202.778. For 1000000 points, the heap average is 31.18 seconds. Using quadratic regression, we can see that there is a strong correlation between the recorded values and the equation Y=0.460852+-9.492822E^(-4)n+1.297246E^(-7)n^2. Using this curve, we can predict that the time it would take to run on a million points would be approximately 128775 seconds, or about 35 hours. Judging from the deconstructed runtimes seen in part 3, these results seem to make sense, especially with how strong of an r value the correlation between the found quadratic equation and the empirical results of the array based implementation.

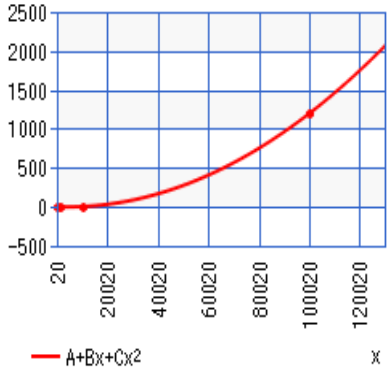
(input by clicking each cell in the table below)

data

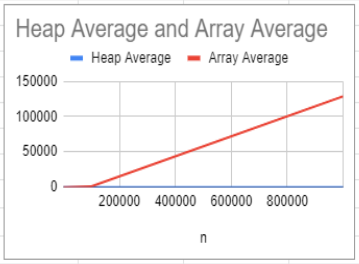
No.	x	y
1	100	.001
2	1000	.0478
3	10000	3.89984
4	100000	1202.779

Execute | Clear | Store/Read | Print | 10digit

function	value
mean of x	27,775
mean of y	301.68191
correlation coefficient r	0.999999861
A	0.460852
B	-9.492822E-4
C	1.297246E-7



Size	HeapTime	ArrayTime	avg heap	avg array
100	0.001001	0.001001		
100	0.001001	0		
100	0.001001	0.001001		
100	0.002	0.002		
100	0.001001	0.001	0.0012008	0.0010004
1000	0.009008	0.05004		
1000	0.009007	0.047043		
1000	0.009008	0.047042		
1000	0.009008	0.049045		
1000	0.009008	0.046042	0.0090078	0.0478424
10000	0.116105	4.842		
10000	0.117106	4.8243		
10000	0.117106	4.90445		
10000	0.116106	4.882437	0.0950862	3.8998458
100000	2.07388	1081.3499		
100000	1.887	1080.094655		
100000	1.9307	1446.8948		
100000	1.9767	x		
100000	1.9727	x	1.968196	1202.779785
1000000	28.945306	x		
1000000	31.3524	x		
1000000	33.149126	x		
1000000	29.255589	x		
1000000	33.194168	x	31.1793178	x



n	Heap Average	Array Average	
100	0.0012008	100	
1000	0.0090078	0.0478424	
10000	0.0950862	3.8998458	
100000	1.968196	1202.779785	
1000000	31.1793178	128775	<---- ESTIMATION