Dmitri Stanchevici
Unit 4
Module 4

Ex. 4.1

N	Output
100	Found=true Time taken: 0 Found=true Time taken: 0
1,000	Found=false Time taken: 0 Found=false Time taken: 0
100,000	Found=true Time taken: 1 Found=true Time taken: 0
100,000,000	Found=true Time taken: 11 Found=true Time taken: 0

Ex. 4.4

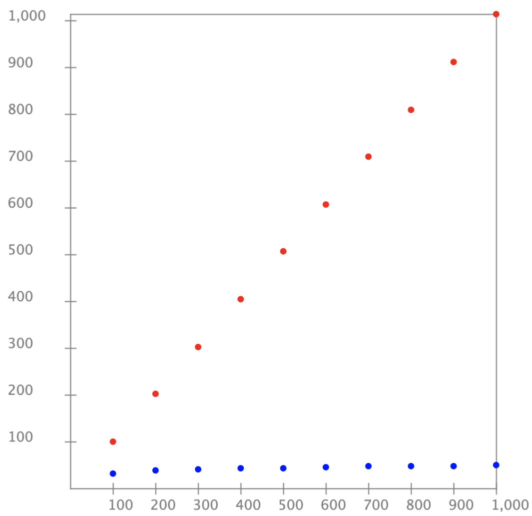
Curves f(x) = x and g(x) = log(x) in the range 0 - 100,000

0	0	
10000	10000	13.28771238
20000	20000	14.28771238
30000	30000	14.87267488
40000	40000	15.28771238
50000	50000	15.60964047
60000	60000	15.87267488
70000	70000	16.0950673
80000	80000	16.28771238
90000	90000	16.45763738

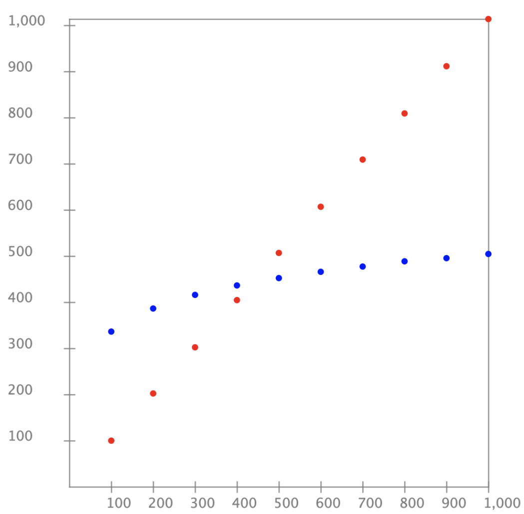
100000	100000	16.60964047

Ex. 4.6

With q = 5



With q = 50



Ex. 4.7

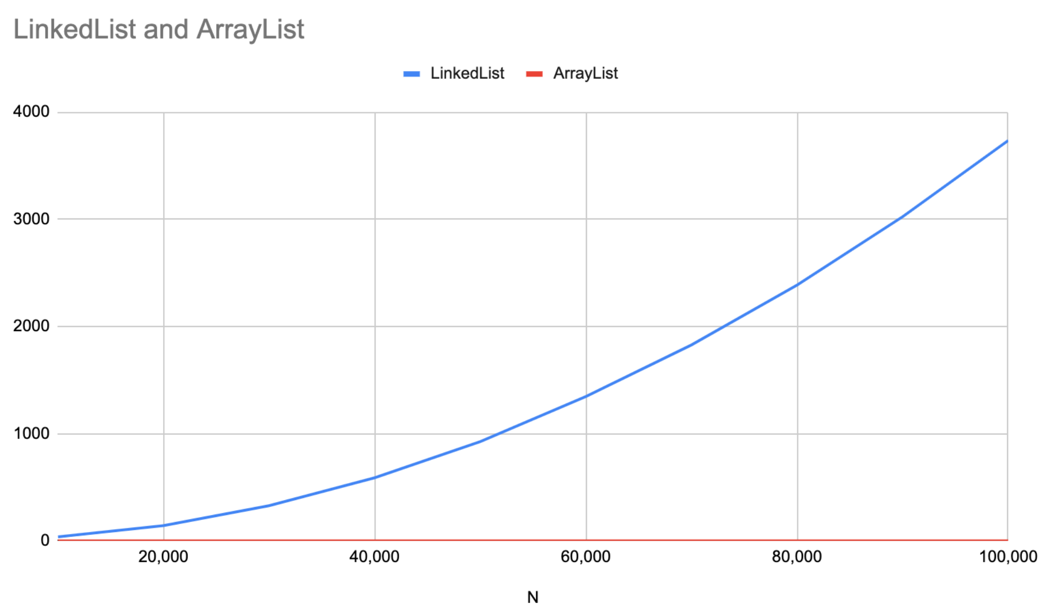
Five executions with N = 10,000

Test	LinkedList	ArrayList
1	40	1
2	40	0
3	42	0
4	43	0
5	40	0
Average	**41**	**0.2**

Ex. 4.8

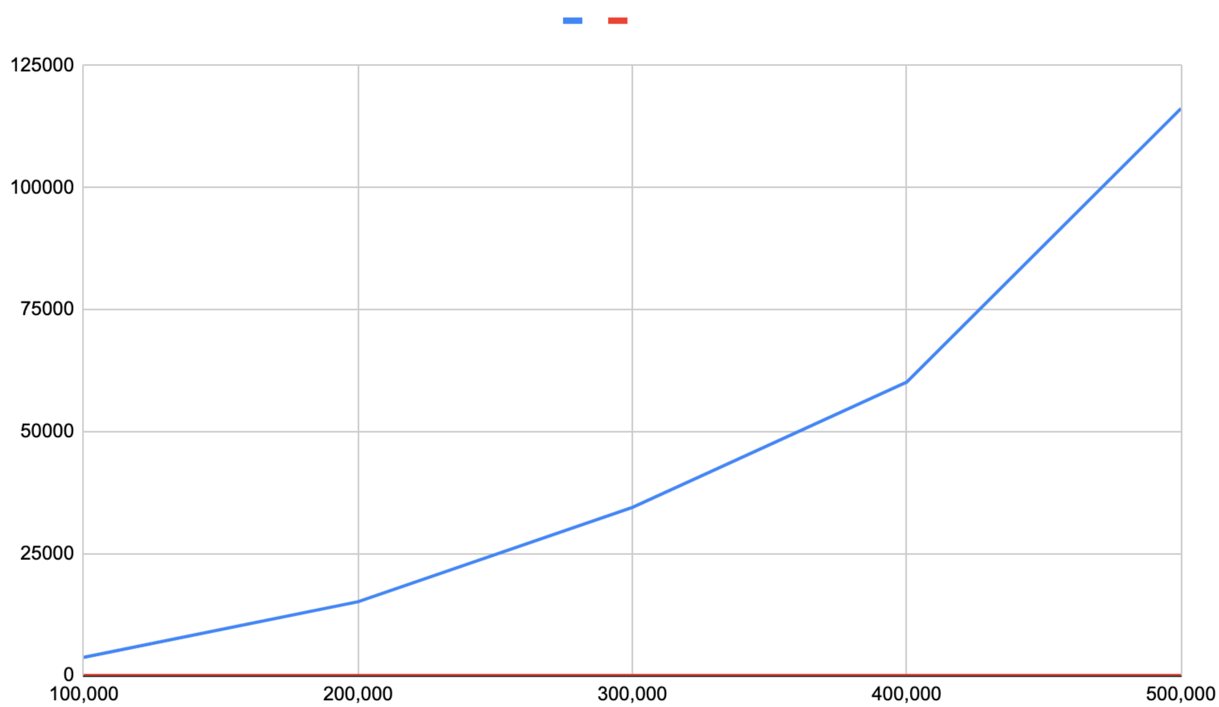
Averages from five executions with N from 10,000 to 100,000

N	LinkedList	ArrayList
10,000	36.8	0.6
20,000	142.2	0.8
30,000	328	0.6
40,000	588.2	0.8
50,000	926.2	1
60,000	1346.4	0.6
70,000	1828	0.4
80,000	2385.8	0.8
90,000	3023.4	0.6
100,000	3736.2	0.6



Averages from five executions with N from 100,000 to 500,000

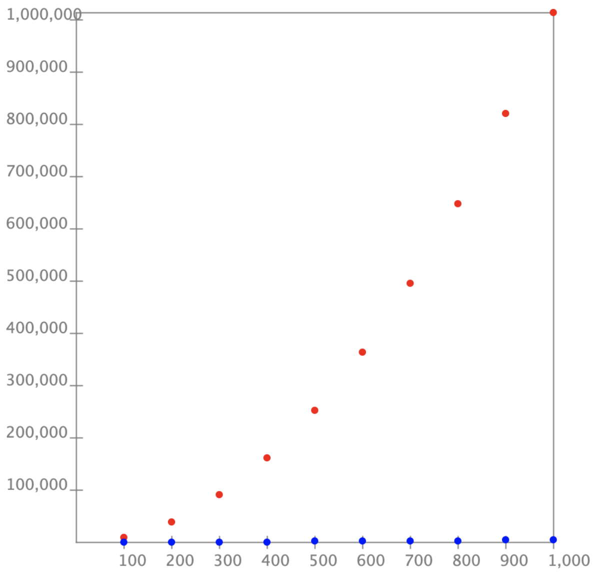
N	LinkedList	ArrayList
100,000	3736.2	0.6
200,000	15,113.20	1.2
300,000	34,467.20	1.6
400,000	60,114.40	1.6
500,000	116,221.80	1.6



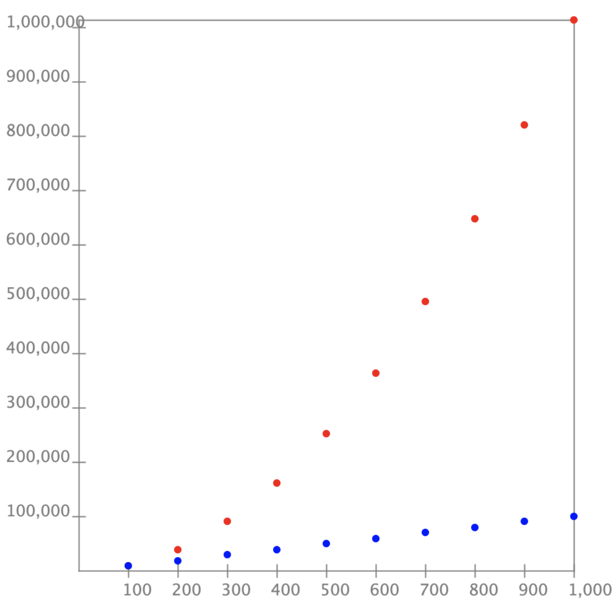
The time taken by the LinkedList grows at a greater than linear rate: When N doubles, the time grows four- to five-fold.

Ex. 4.9

q=5


1,000,000
900,000
800,000
700,000
600,000
500,000
400,000
300,000
200,000
100,000
100
200
300
400
500
600
700
800
900
1,000

q=100


1,000,000
900,000
800,000
700,000
600,000
500,000
400,000
300,000
200,000
100,000
100
200
300
400
500
600
700
800
900
1,000

Ex. 4.11

The curve of Algorithm B will rise above that of Algorithm A at n=13:

n=12 f(n)=7104 g(n)=6912
n=13 f(n)=8736 g(n)=8788
n=14 f(n)=10612 g(n)=10976

Ex. 4.12

Starting with an array of size 1, **10 doublings** will be required to insert 1,024 elements.

In general, the number of doublings for n items equals log to the base of 2 of n: $\mathbf{\log_2 n}$

Ex. 4.14

With the optimization, in which the inner loop starts from the current item of the outer loop (j=i), the exact number of comparisons in terms of n will be

$$n * n/2 = n^2/2$$

In Big-Oh notation, this is equivalent to

$$O(n^2)$$

Ex. 4.15

The total amount of work is

$$n^2/2 - n/2 + n-1$$

or

$$O(n^2)$$

Ex. 4.19

1. The length of the tour that is already in place: **50.22213534438458**

2. The length of the tour that I think is the shortest: **30.33029403737908**

3. Why the for-loop in main() goes up to n-2:

```
for (int i=0; i<n-1; i++) {
   DrawTool.drawLine (x[tour[i]], y[tour[i]], x[tour[i+1]], y[tour[i+1]]);
   tourLength += distance (x[tour[i]], y[tour[i]], x[tour[i+1]], y[tour[i+1]]);
}
```

The for-loop works with i and with i+1. So, if i reached the value of the last index, i+1 would throw an exception LOOP OUT OF BOUND. Therefore, i should not be increased beyond n-2. The distance from the last point to point at [0] is added outside the for-loop.

Ex. 4.20

1. Number of tours evaluated: **40,322**

2. Best route and its length: **0 5 1 6 7 2 3 4; Length=29.492571697547458**

3. What does the number of tours evaluated have to do with n! (n factorial)?

   8! = 40,320. This is the number of evaluations performed inside **recursiveFindTour()**. The remaining two evaluations are called from **findTour()**, one before the call to recursiveFindTour() and one after. Conclusion: This algorithm involves a factorial growth of n.