Exercise 1

The get() method is faster for an ArrayList because an ArrayList uses direct indices. When calling the get() method on an ArrayList, we can specify the numeric index that we want to access an element at, directly access the index, and return that element.

For the LinkedList, on the other hand, the index of a given element cannot be accessed directly. LinkedLists do not use indices, and so we will have to traverse through the entire list and update a counter variable to track our index. This will use a while() loop, which slows the program down by increasing the number of operations needed to find the element.

Exercise 2

The curve for g(n) does eventually rise above f(n), and it does so at n=13.

Exercise 3

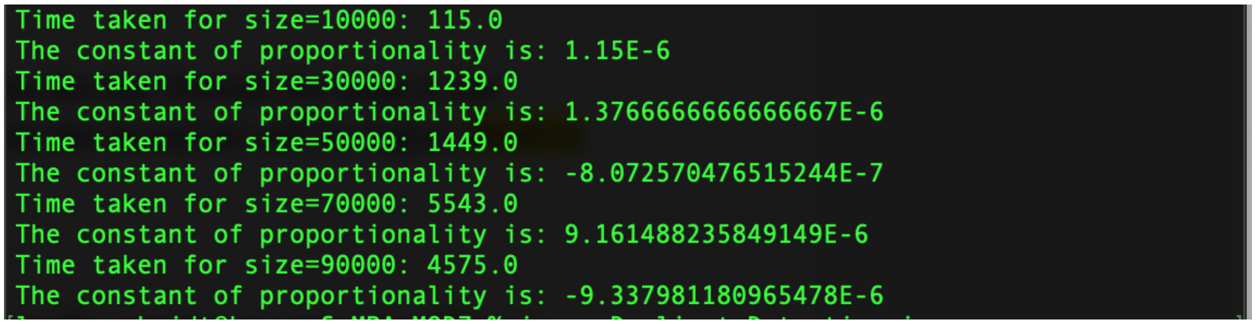
10 doublings will be needed to insert 1024 items into the list. Overall, for large *n*, the number of doublings needed to insert *n* elements into the ArrayList will be log(n).

Exercise 4

When the size of the list is *n* elements, and we observe the worst case scenario of linear search (traversing the whole list), the search method takes O(n) time.
When the size of the list is *n* elements, an array-list will always take O(1) time to use the get method. This is because arrays are indexed, and we can directly retrieve the value at the given index. If the index does not exist, a null value will be returned. Either way, only one operation takes place during the get method, making its time complexity O(1).

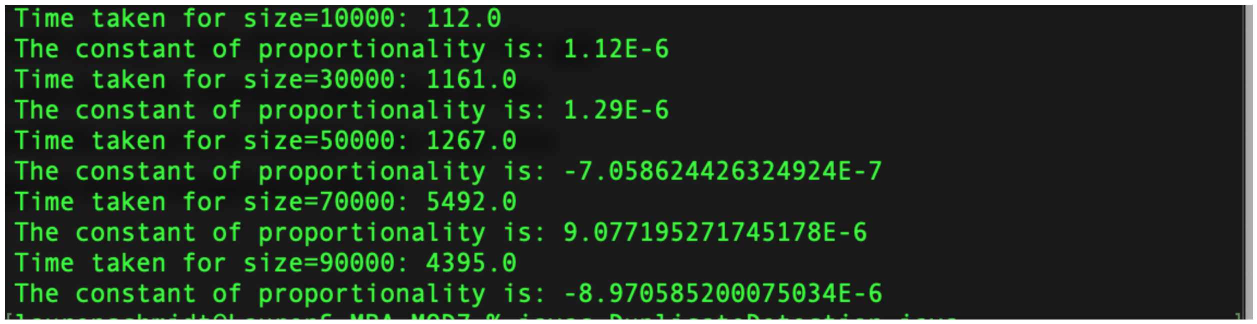
Exercise 6

Constant(s) of proportionality before adjustments:



```
Time taken for size=10000: 115.0
The constant of proportionality is: 1.15E-6
Time taken for size=30000: 1239.0
The constant of proportionality is: 1.3766666666666667E-6
Time taken for size=50000: 1449.0
The constant of proportionality is: -8.072570476515244E-7
Time taken for size=70000: 5543.0
The constant of proportionality is: 9.161488235849149E-6
Time taken for size=90000: 4575.0
The constant of proportionality is: -9.337981180965478E-6
```

After adjustments:



```
Time taken for size=10000: 112.0
The constant of proportionality is: 1.12E-6
Time taken for size=30000: 1161.0
The constant of proportionality is: 1.29E-6
Time taken for size=50000: 1267.0
The constant of proportionality is: -7.058624426324924E-7
Time taken for size=70000: 5492.0
The constant of proportionality is: 9.077195271745178E-6
Time taken for size=90000: 4395.0
The constant of proportionality is: -8.970585200075034E-6
```

The exact number of comparisons is O(n-1), because starting the index of the second loop at 1 instead of 0 removes one of the elements.

Exercise 7

<u>Insert:</u>

The number of elements in the sorted LinkedList can be defined as *n*. The "worst case" scenario here will be that we have to place the element at the end of the list (or that our element is larger than any of the elements that already exist within the list). So, because in the worst case we will have to traverse the entire LinkedList, the time needed is O(n).

<u>Search:</u>

The number of elements in the LinkedList can be defined as *n*. In the worst case scenario, where the element is either at the end of the LinkedList, or does not exist in the LinkedList at all, the list will still have to be traversed. So, when searching a LinkedList, the time needed is also O(n).

In an unsorted LinkedList, the time needed (O(n)) will be the same, because in the worst case, we will still have to traverse the entire LinkedList until we find our element.

Exercise 9

<u>Insert:</u>

The number of elements in our ArrayList is defined as *n*. In the worst case scenario, we will have to shift *n* elements in order to create space for the new element to be inserted. Because of this, despite an ArrayList's characteristic direct indexing, it will still take O(n) time to shift over *n* elements in the list.

<u>Search:</u>

Search is much faster on sorted ArrayLists because we can directly access the index of a pivot point and perform a binary search from there. If our element is greater or less than our midpoint, we can eliminate half of the sorted data at a time. In a dataset of *n* elements, with our worst case scenario (the target is not in the search list), it will take O(log(n)) time to carry out the search operation.

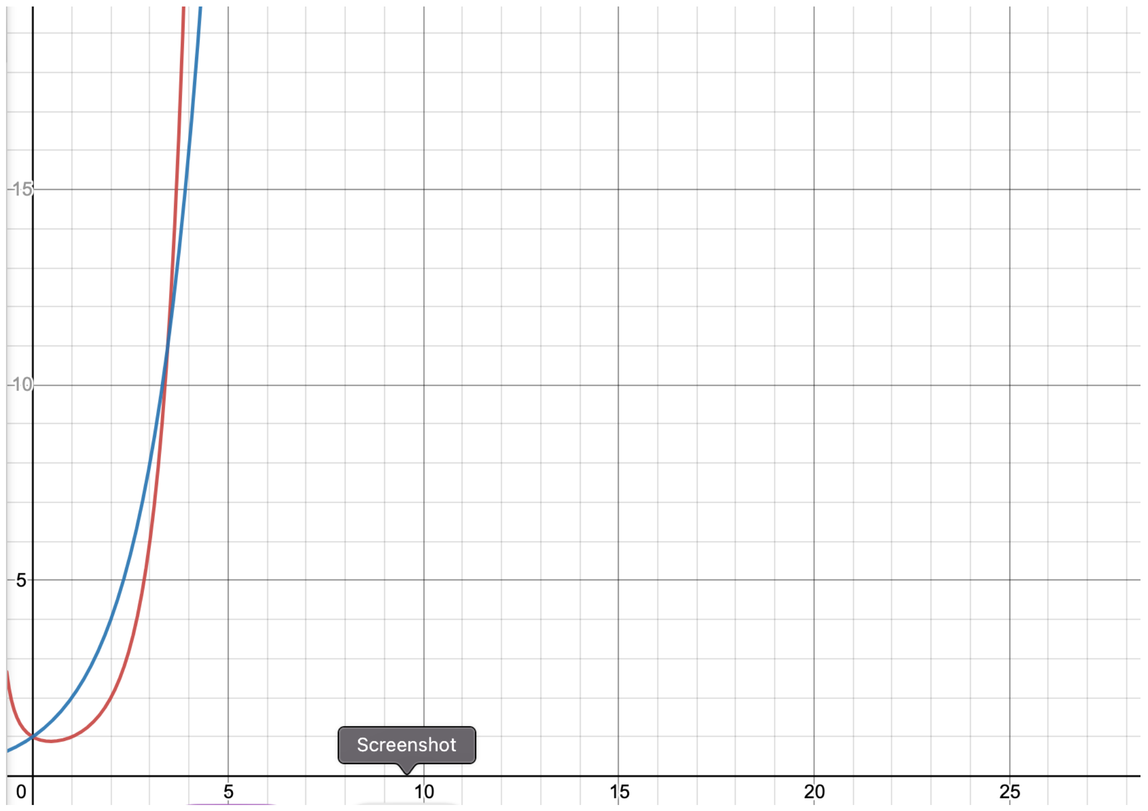
Exercise 10
This algorithm is related to binary search because the time complexity of binary search is O(log(n)), and both algorithms utilize divisions by two. In binary search, half of the data is eliminated – after we divide the data in two – with every operation.

Exercise 11

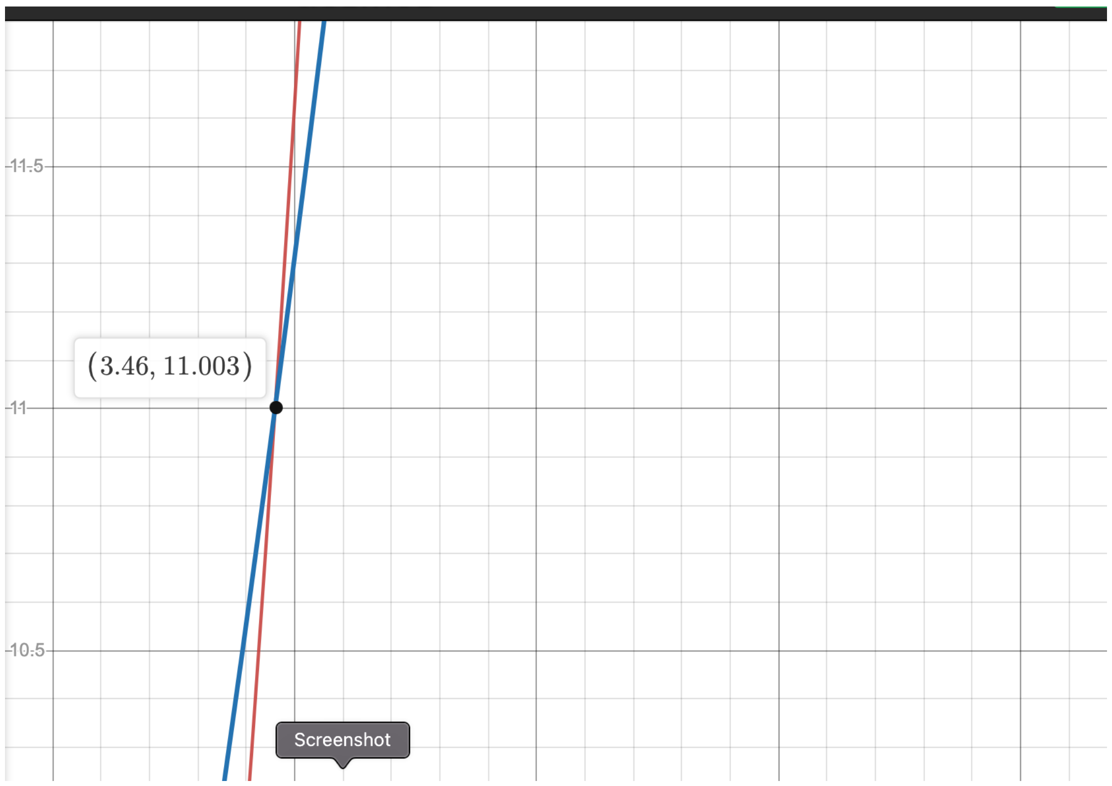
$$(n-1)(\frac{n(n-1)}{2} + (n-1)) = (n-1)\cdot(\frac{n^2-n}{2} + n - 1) = \frac{n^3-2n^2+n+2n^2-4n+2}{2} = \frac{n^3-3n+2}{2}$$

The final time that it takes for the selectionSort algorithm above can be described as $O(n^3)$, because the largest degree of power is n^3 , and it is unaffected by the coefficients.

Exercise 13
In order to argue that factorials are worse than exponentials, it can first be helpful to look at a graph of a factorial function (f(x) = x!) compared to an exponential function (f(x) = 2^x):
**Note: We will use 2^x specifically because it appears commonly in computer science as the inverse of log base 2.



Zooming in, we can see that the graph of f(x) = x! (outlined in red) will surpass the graph of f(x)=2^x (outlined in blue) around 3.5.



Exercise 14

f(r) is a lot smaller than r! or 2^r, even though f(r) still grows relatively large. Because the function is recursive, the time and memory cost are both larger than a non-recursive function, mainly because each recursive call will execute the countPaths() method twice. .