Investigation 1:

For our investigation we decided to pick two array sizes: 100,000 and 2,000,000. For the smaller size, we chose MIN_SIZE(the size when the algorithm switches from quicksort to insertion) values that were spread out to discern broader trends. For the larger array, we chose MIN_SIZE values that were grouped closer to try and pinpoint the optimal value for MIN_SIZE. For each array size we created a new array, set specific MIN_SIZE values to test, and used a loop to run through the following process 100 times: use fillAndShuffle() to fill array with random numbers, call quicksort(), and time the amount of time it took to complete. After the loop completed, we calculated the average time it took to run by dividing the total time all iterations took to run by 100. We calculated the average for each combination of test values to minimize the inevitable variation in runtime due to both fluctuation in computer speed and the exact permutation in the arrays when fillAndShuffle() is called(some arrays would be easier/harder to sort than others and thus take a shorter/longer amount of time). Below are two charts that show the data we collected.

Array of size 100,000:

MIN_SIZE	**Runtime**
10	~ 12.5 ms
100	~ 11.5 ms
1000	~ 20 ms
10,000	~ 135 ms
100,000	~ 1300 ms

Array of size 2,000,000:

MIN_SIZE	**Runtime**
10	~ 265 ms
100	~ 225 ms
200	~ 260 ms
500	~ 300 ms
1,000	~ 430 ms

We found that in general, the smaller MIN_SIZE is, the faster the overall algorithm performs. Because quicksort() had the lowest runtimes when the MIN_SIZE was small(and thus only uses insertion sort when the array becomes small), we believe that insertion sort works faster on small arrays and that quicksort works faster on large arrays. This is evident if one sees how long the sort took when we set the MIN_SIZE to the same size as our smaller array(and thus forcing quicksort() to use insertion sort right away); it took over a 100 times longer than when MIN_SIZE was 100! So optimally, one would want to set MIN_SIZE to be such that it is the exact size when insertion sort becomes faster than quicksort(for that problem size).

Interestingly, we found that no matter how large the original array is, the ideal MIN_SIZE seems to be around the same. For both array sizes, we estimated the optimal size for MIN_SIZE to be around 100. Since insertion sort works faster than quicksort on smaller arrays, it makes sense why the optimal size would be the same; no matter the size of the original array, quicksort should only use insertion sort once insertion sort would be faster. We also noticed that each time we ran a test of MIN_SIZE with the array of size 100,000, we noticed that the first few iterations of the 100 runs of quicksort always took significantly longer than the rest of the runs. We hypothesize that when the loop is run, the computer must allocate some resources for the overhead of running quicksort and thus increases runtime.

We also wanted to find out differences in runtime depending on if we inputted a randomized list, an sorted list, or a list in reverse sorted order. We ran a similar procedure as the one we used to test different values of MIN_SIZE and array size, running a sort with each type of list 100 times and calculating the average. To make sure we were just testing the difference between the input lists, we chose the same array size and MIN_SIZE for all runs of our experiment: 2,000,000 and 100 respectively. Here is the data we collected.

Sorted list	**List in reverse sorted order**	**Randomized**
~ 30 ms	~ 55 ms	~ 225 ms

Inputting a list in either sorted or reverse sorted order significantly reduced the runtime of quicksort. The difference was very large between a randomized list and the sorted lists, which suggests that there is a big difference between quicksort's best case and most cases.

Investigation 2:

For this investigation, RadixSort was the focus. We created two different RadixSort methods to observe differences in how the two methods function and the runtime of each. We hypothesized that the msdRadixSort runtime would be slightly faster than RadixSort's runtime. To prepare words for sorting in both methods, we added asterisks to the end of every word shorter than the longest word until all words were the same length to ensure no out of bounds errors and so that RadixSort could work. Our RadixSort sorted using a list of twenty-seven queues that represented each letter of the alphabet, asterisk being the first queues. When sorted with the least significant letter, a huge amount of words would be added to the linked list that contained words ending in an asterisk. Sorting through all these words in the asterisk queue would take longer than if the words were more spaced out between the queues. This does not happen for msdRadixSort because the method initially sorts the words by first letter and thus more evenly spaces the words before sorting.

For each type of sort, we ran a loop 100 times with each of the problem sizes(for each size, we loaded the entire words.txt file, randomized it, and then took the sublist from 0 to the specific number) and found the average runtime of those trials. We calculated the average for each combination of test values to minimize the inevitable variation in runtime due to both fluctuation in computer speed and the randomization in the lists when Collections.shuffle() is called(some lists would be easier/harder to sort than others and thus take a shorter/longer amount of time). Below are two charts that show the data we collected.

Problem size	**MSD RadixSort**	**Radix Sort**
1000	~ 10.5 ms	~ 6 ms
10,000	~ 28 ms	~ 40 ms
100,000	~ 180 ms	~ 430 ms
117943(half the words in words.txt)	~ 245 ms	~ 550 ms
235,886(words in words.txt)	~ 520 ms	~ 1100 ms

The size of the list of words definitely affects the runtime since our data is composed of many operations that rely on the size of the input list of words, such as for-loops that iterate through every word. This is evident if one views the table above; as size of the inputted list increases, total runtime does also. Data with little variation between words (for example: if the longest word is monologue, a******** and I******** or dialogue* and analogue*) would be

hard to sort for the sorting method that sorts from the least significant letter because these words will accumulate in one linked list group and require more iterations of the nested for-loop to sort, the operation that requires the most operations in the sort method. Another factor that deviates the runtimes between the methods is the state of the list. We can only hypothesize about why a sorted list is faster to sort using both RadixSort methods. When the list is already sorted, the runtime is almost cut in half for RadixSort (from ~1100 ms to ~620 ms). When we sort the words in the queue, we transport all the words into a temporary arraylist afterwords. Perhaps the pre-existing sorted nature of the words that the the sort methods try to achieve aids the sorting methods by resulting in less operations

We also wanted to test whether inputting a already sorted list would impact the performance of both sorts versus inputting a randomized list. We ran the previous experiment with the complete words.txt file but without shuffling it. We calculated the average runtimes for the sorts to sort an already sorted list and have put them below(along with the previously calculated runtimes for randomized lists). Both types of sorts ran much faster when sorting an already sorted list.

RadixSort

Sorted list	**Randomized**
~ 620 ms	~1100 ms

MSD RadixSort

Sorted list	**Randomized**
~330 ms	~ 520 ms