Assignment 4 – Andre Godinez - 15460718

```
package assignment_4;

import java.util.Comparator;
import java.util.Random;


public class CompareAlgorithms {
	//member data
	//insertion , quick sort, shell sort
	public static int move1,move2,move3 = 0;
	public static int comp1,comp2,comp3 = 0;

	/**
	 * THIS IS MY OWN CODE.
	 */

	//method to do Insertion sort by passing in an array.
	private static void insertionSort_temp(String[] temp){
	//prints out size of the array we are passing in
	System.out.println("\nSize : "+ temp.length +"\nInsertion Sort : ");
	//prints out the array before sorting
	//System.out.println("Array before Insertion Sort : "+array2String(temp));

	//taking the start time here because the sorting only starts here.
	long start = System.nanoTime();
	insertionSort(temp);
	long finish = System.nanoTime();
	long time = finish - start;

	//printing out the array after the sort. No. of comparisons and moves and the time taken to sort
	//System.out.println("Array after Insertion Sort : "+array2String(temp));
	System.out.println("No. of Comparisons = "+comp1+"\nNumber of moves = "+move1);
	System.out.println("Time taken to Sort = "+ (time/1000000) + " ms");
	comp1 = 0;
	move1 = 0;
	}

	//method to do Quick sort by passing in an array.
	private static void quickSort_temp(String[] temp){
	//prints out size of the array we are passing in
	System.out.println("\nSize : "+ temp.length+"\nQuicksort : ");
	//prints out the array before sorting
	//System.out.println("Array before Quick Sort : "+array2String(temp));

	//taking the start time here because the sorting only starts here.
	long start = System.nanoTime();
	quickSort(temp, new StringComparator());
	long finish = System.nanoTime();
	long time = finish - start;

	//printing out the array after the sort. No. of comparisons and moves and the time taken to sort
	//System.out.println("Array after Quick Sort : "+array2String(temp));
	System.out.println("SNo. of Comparisons = "+comp2+"\nNumber of moves = "+move2);
	System.out.println("Time taken to Sort = "+ (time/1000000) + " ms");
	comp2=0;
	move2=0;
	}

	//method to do Shell sort by passing in an array.
	private static void shellSort_temp(String[] temp){
	//prints out size of the array we are passing in
	System.out.println("\nSize : "+ temp.length+"\nShell sort : ");
	//prints out the array before sorting
	//System.out.println("Array before Shell Sort : "+array2String(temp));

	//taking the start time here because the sorting only starts here.
	long start = System.nanoTime();
	shellSort(temp);

	/* THIS IS THE CODE FOR TASK 7.
	 * shellSort_task7(temp);
	 */

	long finish = System.nanoTime();
	long time = finish - start;

	//printing out the array after the sort. No. of comparisons and moves and the time taken to sort
	//System.out.println("Array after Shell Sort : "+array2String(temp));
```

```
System.out.println("No. of Comparisons = "+comp3+"\nNumber of moves = "+move3);
System.out.println("Time taken to Sort = "+ (time/1000000) + " ms");
comp3=0;
move3=0;
}

//method that creates a random array given a size
	private static String[] createRandArray(int size){
		String[] temp = new String[size];
		String LettersAndNo = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";
		Random rnd = new Random();

		for(int i=0; i<size;i++){
			StringBuilder tempstr = new StringBuilder();

		int rand = (int) (rnd.nextFloat() * LettersAndNo.length());
		tempstr.append(LettersAndNo.charAt(rand));
		String superString = tempstr.toString();

		temp[i]=superString;
		// System.out.println("Generated "+i+" string");
		}

		return temp;
	}

/*
 * END OF MY OWN CODE
 */


	/*
	 * THIS IS THE CODE FROM LECTURE NOTES WITH SOME STUFF EDITED WHICH I WILL HIGHLIGHT
	 */
	private static void insertionSort(Comparable[] a, int first, int last, int gap)
	{
		int index;     // general index for keeping track of a position in array
		int toSort;  // stores the index of an out-of-place element when sorting.

		// NOTE: Instead of considering a full array of adjacent elements, we are considering
		// a sub-list of elements from 'first' to 'last, separated by 'gap'. All others are ignored.
		//
		// Work forward through the list, starting at 2nd element,
		// and sort each element relative to the ones before it.

		for (toSort = first+gap; toSort <= last; toSort += gap)
		{
			Comparable toSortElement = a[toSort];

			// Go back through the list to see how far back (if at all)
			// this element should be moved.
			// Note: we assume all elements before this are sorted relative to each other.
			boolean moveMade = false;
			index = toSort - gap;
			while ((index >= first) && (toSortElement.compareTo(a[index]) < 0))
			{
				// Shuffle elements over to the right, put firstUnsorted before them
				a[index+gap] = a[index];
				index = index - gap;
				moveMade = true;
				comp1++;
			}
			if (moveMade) {
				//System.out.println("Inserting " + toSortElement + " at pos " + (index+1));
				a[index+gap] = toSortElement;
				move1++;
			}
		}

	}


	/** Version of insertionSort method that uses the correct settings to sort an entire array
	 *  from start to end in steps of 1.
	 *
	 *  This version uses the 'helper function' version, but is a direct substitute for the
	 *  quickSort() method call.
	 */
	public static void insertionSort(Comparable arr[])
	{
		insertionSort(arr, 0, arr.length-1, 1);
	}
```

```java
 /** QuickSort method:
  * Sorts the elements of array arr in nondecreasing order according
  * to comparator c, using the quick-sort algorithm. Most of the work
  * is done by the auxiliary recursive method quickSortStep.
  **/
public static void quickSort (Object[] arr, Comparator c) {
    if (arr.length < 2) return; // the array is already sorted in this case
    quickSortStep(arr, c, 0, arr.length-1); // call the recursive sort method
}

/** QuickSortStep method:
  * Method called by QuickSort(), which sorts the elements of array s between
  * indices leftBound and rightBound, using a recursive, in-place,
  * implementation of the quick-sort algorithm.
 **/
private static void quickSortStep (Object[] s, Comparator c,
                                   int leftBound, int rightBound )
{


   if (leftBound >= rightBound) return; // the indices have crossed
   Object temp;  // temp object used for swapping

   // Set the pivot to be the last element
   Object pivotValue = s[rightBound];

   // Now partition the array
   int upIndex = leftBound;     // will scan rightward, 'up' the array
   int downIndex = rightBound-1; // will scan leftward, 'down' the array
   while (upIndex <= downIndex)
   {
       // scan right until larger than the pivot
       while ( (upIndex <= downIndex) && (c.compare(s[upIndex], pivotValue)<=0) )
           upIndex++;
                comp2++;
       // scan leftward to find an element smaller than the pivot
       while ( (downIndex >= upIndex) && (c.compare(s[downIndex], pivotValue)>=0))
           downIndex--;
                comp2++;
       if (upIndex < downIndex) { // both elements were found
          temp = s[downIndex];
              s[downIndex] = s[upIndex]; // swap these elements
              s[upIndex] = temp;
              move2++;
       }
   } // the loop continues until the indices cross

   int pivotIndex = upIndex;
   temp = s[rightBound]; // swap pivot with the element at upIndex
   s[rightBound] = s[pivotIndex];
   s[pivotIndex] = temp;
   move2++;

   // the pivot is now at upIndex, so recursively quicksort each side
   quickSortStep(s, c, leftBound, pivotIndex-1);
   quickSortStep(s, c, pivotIndex+1, rightBound);
}

/** Task: Sorts equally spaced elements of an array into ascending order.
 *  The paramater arr is an array of Comparable objects.
 */


public static void shellSort (Comparable[] arr)
{
        int last = arr.length-1;

        // Begin with gap = half length of array; reduce by half each time.
        for (int gap = arr.length/2; gap > 0; gap = gap/2)
        {
                if (gap % 2 == 0) gap++; // if gap is even, move to next largest odd number

                // Apply Insertion Sort to the subarrays defined by the gap distance
                for (int first = 0; first < gap; first++) {
                        insertionSort_shell(arr, first, last, gap);
                }
        } // end for
} // end shellSort
```

```java
/*
 * COMMENTED OUT CODE THAT MADE GAP ALWAYS ODD.
 */

public static void shellSort_task7 (Comparable[] arr)
{
	int last = arr.length-1;

	// Begin with gap = half length of array; reduce by half each time.
	for (int gap = arr.length/2; gap > 0; gap = gap/2)
	{
		//if (gap % 2 == 0) gap++; // if gap is even, move to next largest odd number

		// Apply Insertion Sort to the subarrays defined by the gap distance
		for (int first = 0; first < gap; first++) {
			insertionSort_shell(arr, first, last, gap);
		}
	} // end for
} // end shellSort







/*
 * MADE SEPARATE INSERTIONSORT FOR SHELLSORT SO I CAN INCREMENT OPERATIONS INDEPENDENTLY
 */
private static void insertionSort_shell(Comparable[] a, int first, int last, int gap)
{
	int index;     // general index for keeping track of a position in array
	int toSort;  // stores the index of an out-of-place element when sorting.

	// NOTE: Instead of considering a full array of adjacent elements, we are considering
	// a sub-list of elements from 'first' to 'last, separated by 'gap'. All others are ignored.
	//
	// Work forward through the list, starting at 2nd element,
	// and sort each element relative to the ones before it.

	for (toSort = first+gap; toSort <= last; toSort += gap)
	{
		Comparable toSortElement = a[toSort];

		// Go back through the list to see how far back (if at all)
		// this element should be moved.
		// Note: we assume all elements before this are sorted relative to each other.
		boolean moveMade = false;
		index = toSort - gap;
		while ((index >= first) && (toSortElement.compareTo(a[index]) < 0))
		{
			// Shuffle elements over to the right, put firstUnsorted before them
			a[index+gap] = a[index];
			index = index - gap;
			moveMade = true;
			comp3++;
		}
		if (moveMade) {
			//System.out.println("Inserting " + toSortElement + " at pos " + (index+1));
			a[index+gap] = toSortElement;
			move3++;
		}
	}

}
```

```
private static String array2String(Object[] a)
{
        String text="[";
        for (int i=0; i<a.length; i++) {
                text += a[i];
                if (i<a.length-1)
                        text += ",";
        }
        text += "]";
        return text;
}
```

```
public static void main(String args[]){

        /*String arr[] = {"a", "hello", "x", "w", "q", "h", "d", "p", "a1", "x2", "w2", "q1", "2h", "2d", "3p", "1a",
                        "3x", "2w", "3q", "h4", "d4", "4p", "3a", "3x", "5w", "q5", "h5", "d5", "p5",
                        "n3x", "2bw", "n3q", "nh4", "nd4", "n4p", "b3a", "b3x", "c5w", "xq5", "kh5", "dj5", "jp5",
                        "3x", "2w", "3q", "h4", "d4", "4p", "3a", "3x", "5w", "q5", "h5", "d5", "p5",
                        "q3x", "q2w", "3qq", "qh4", "ad4", "4ap", "3aa", "3ax", "a5w", "q5", "fh5", "fd5", "fp5",
                        "n3x", "2bw", "n3q", "nh4", "nd4", "n4p", "b3a", "b3x", "c5w", "xq5", "kh5", "dj5", "jp5",
                        "3x", "2w", "3q", "h4", "d4", "4p", "3a", "3x", "5w", "q5", "h5", "d5", "p5",
                        "n3x", "2bw", "n3q", "nh4", "nd4", "n4p", "b3a", "b3x", "c5w", "xq5", "kh5", "dj5", "jp5",
                        "q3x", "q2w", "3qq", "qh4", "ad4", "4ap", "3aa", "3ax", "a5w", "q5", "fh5", "fd5", "fp5",
                        "n3x", "2bw", "n3q", "nh4", "nd4", "n4p", "b3a", "b3x", "c5w", "xq5", "kh5", "dj5", "jp5",
                        "3x", "2w", "3q", "h4", "d4", "4p", "3a", "3x", "5w", "q5", "h5", "d5", "p5",
                        "n3x", "2bw", "n3q", "nh4", "nd4", "n4p", "b3a", "b3x", "c5w", "xq5", "kh5", "dj5", "jp5",
                        "3x", "2w", "3q", "h4", "d4", "4p", "3a", "3x", "5w", "q5", "h5", "d5", "p5",
                        "q3x", "q2w", "3qq", "qh4", "ad4", "4ap", "3aa", "3ax", "a5w", "q5", "fh5", "fd5", "fp5",
                        "n3x", "2bw", "n3q", "nh4", "nd4", "n4p", "b3a", "b3x", "c5w", "xq5", "kh5", "dj5", "jp5",
                        "n3x", "2bw", "n3q", "nh4", "nd4", "n4p", "b3a", "b3x", "c5w", "xq5", "kh5", "dj5", "jp5",
                        "3x", "2w", "3q", "h4", "d4", "4p", "3a", "3x", "5w", "q5", "h5", "d5", "p5",
                        "3x", "2w", "3q", "h4", "d4", "4p", "3a", "3x", "5w", "q5", "h5", "d5", "p5",
                        "g3x", "g2w", "f3q", "fh4", "gd4", "f4p", "f3a", "d3x", "s5w", "sq5", "sh5", "dd5", "sp5"};

        String arr2[] = (String[])arr.clone();
        String arr3[] = (String[])arr.clone();



        /*
         * This was task 1
         */
        /*
        //Insertion Sort
        System.out.println("Array Length for All Sorting Algorithms = "+arr.length);
        System.out.println("Array before Insertion Sort : "+array2String(arr));
        is.insertionSort(arr);
        System.out.println("Array after Insertion Sort : "+array2String(arr));
        //Quicksort
        System.out.println("\nArray before Insertion Sort : "+array2String(arr2));
        qs.quickSort(arr2, new StringComparator());
        System.out.println("Array after Insertion Sort : "+array2String(arr2));
        //Shell Sort
        System.out.println("\nArray before Insertion Sort : "+array2String(arr3));
        ss.shellSort(arr3);
        System.out.println("Array after Insertion Sort : "+array2String(arr3));
        System.out.println("\nInsertion Sort - \nNo. of Comparisons = "+is.compare+"\nNumber of moves = "+is.move);
        System.out.println("\nQuick Sort - \nNo. of Comparisons = "+qs.compare+"\nNumber of moves = "+qs.move);
        System.out.println("\nShell Sort - \nNo. of Comparisons = "+ss.compare+"\nNumber of moves = "+ss.move);


        */

        int size_1 = 10000;
        int size_2 = 20000;
        int size_3 = 50000;

        //creating random array 1 and cloning them so each algorithm has separate array to sort.
        String[] randArr1 = createRandArray(size_1);
        String randArr1_1[] = (String[])randArr1.clone();
        String randArr1_2[] = (String[])randArr1.clone();
        String randArr1_3[] = (String[])randArr1.clone();

        //creating random array 2 ...
        String[] randArr2 = createRandArray(size_2);
        String randArr2_1[] = (String[])randArr2.clone();
```

```
                String randArr2_2[] = (String[])randArr2.clone();
                String randArr2_3[] = (String[])randArr2.clone();
                //creating random array 3 ...
                String[] randArr3 = createRandArray(size_3);
                String randArr3_1[] = (String[])randArr3.clone();
                String randArr3_2[] = (String[])randArr3.clone();
                String randArr3_3[] = (String[])randArr3.clone();


                /**
                 * I just commented out what outputs I didnt need
                 */
                //for task 5
                //sorting random array1...

                insertionSort_temp(randArr1);
                quickSort_temp(randArr1_1);
                shellSort_temp(randArr1_2);

                System.out.println("-------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------");
                //sorting random array 2...

                insertionSort_temp(randArr2);
                quickSort_temp(randArr2_1);
                shellSort_temp(randArr3_2);

                System.out.println("-------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------");
                //sorting random array 3...

                insertionSort_temp(randArr3);
                quickSort_temp(randArr3_1);
                shellSort_temp(randArr3_2);

                System.out.println("-------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------");
                // for task 7
                shellSort_temp(randArr1_3);
                shellSort_temp(randArr2_3);
                shellSort_temp(randArr3_3);
        }

}

/** Comparator class for case-insensitive comaprison of strings */
        class StringComparator implements Comparator
        {
                public int compare(Object ob1, Object ob2)
                {
                        String s1 = (String)ob1;
                        String s2 = (String)ob2;
                        //return s1.compareTo(s2); // use compareTo for case-sensitive
comparison
                        return s1.compareToIgnoreCase(s2);
                }
        }
```

Data with 3 Readings

	Insertion Sort		
ArraySize	**Compare**	**Move**	**Time Taken(ms)**
10000	24416857	9719	249
10000	24365782	9733	249
10000	24153169	9730	251
20000	97870677	19440	1006
20000	96583779	19464	1267
20000	97379602	19451	1024
50000	609916852	48651	10997
50000	610373301	48551	9473
50000	608272064	48633	9327

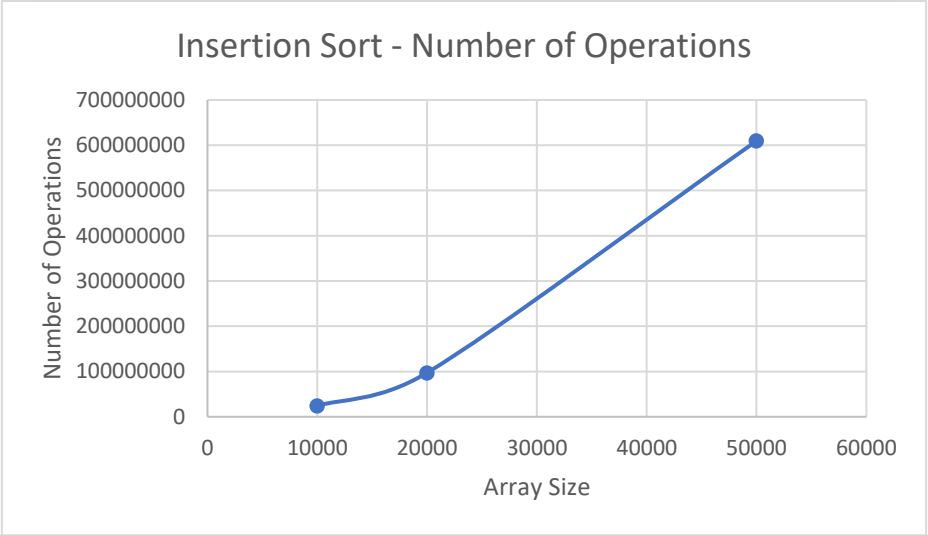
Quicksort		
Compare	**Move**	**Time Taken(ms)**
37642	18821	42
37308	18654	33
37322	18661	36
75660	37830	90
75048	37524	135
74206	37103	109
184796	92398	984
189242	94621	837
189986	94993	671

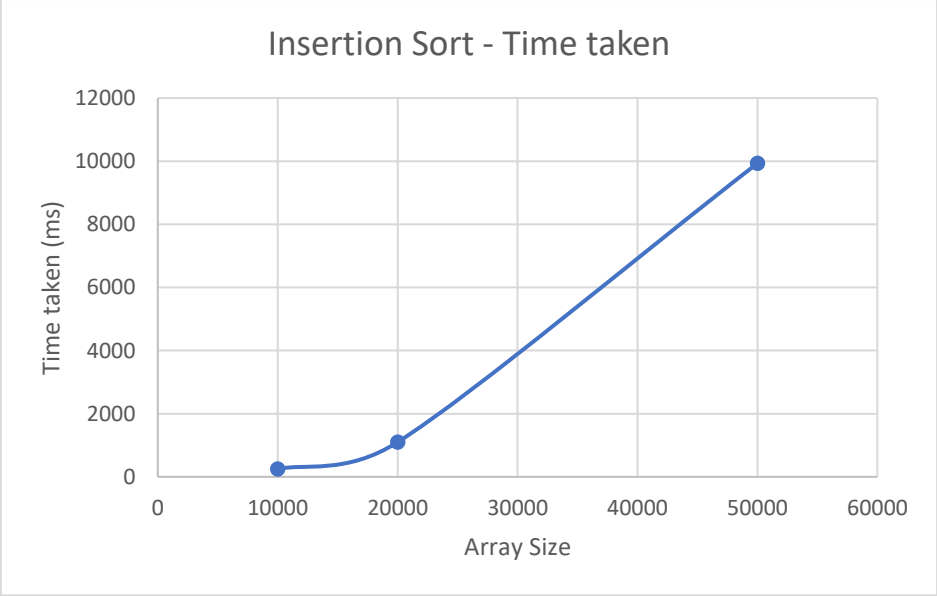
Shell Sort	(gap always odd)	
Compare	**Move**	**Time Taken(ms)**
62013	31256	19
54628	30538	17
55421	30694	23
104879	60101	17
104259	59784	30
115434	62456	11
290396	156365	29
297534	157802	27
282789	154491	31

Shell Sort	(w/o gap always odd	
Compare	**Move**	**Time Taken(ms)**
71845	32101	13
68585	31639	12
68412	31607	7
151660	65182	18
148324	64906	18
147835	64428	25
334705	158790	38
325762	157715	30
315924	157196	30

Averages & Analysis with Graphs

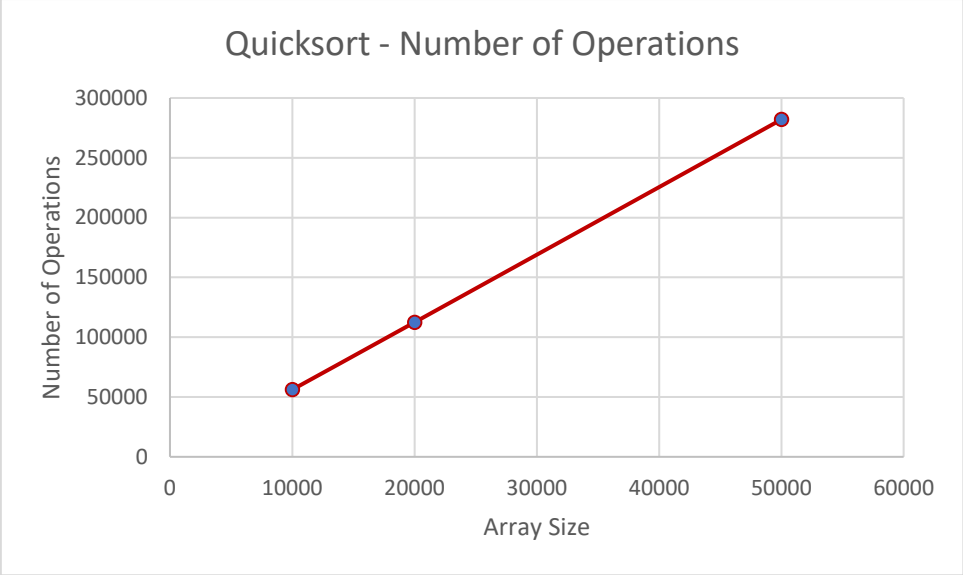
	Insertion Sort			
ArraySize	**Compare**	**Move**	**Operations**	**Time Taken(ms)**
10000	24311936	9727	24321663	250
20000	97278019	19452	97297471	1099
50000	609520739	48612	609569351	9932

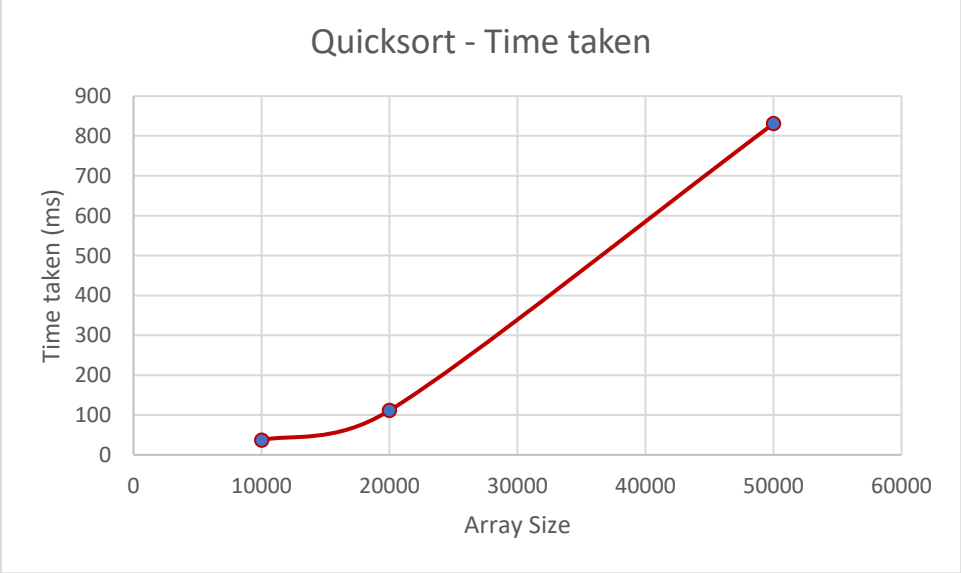




Analysis : Average complexity for insertion sort in notes is O(n^2) which is exactly what we see in the insertion sort time taken graph. We can see how the line is exponentially growing. We can also see how the number of operations increases as array size increases. This is because insertion sort compares each element and see which has a higher value.

Quicksort			
Compare	**Move**	**Operations**	**Time Taken(ms)**
37424	18712	56136	37
74971	37486	112457	111
188008	94004	282012	831



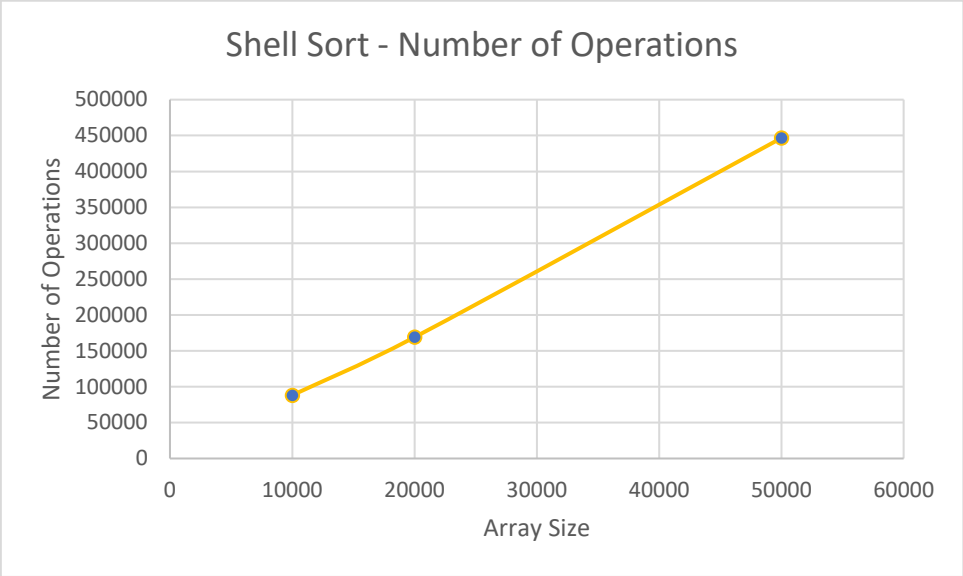


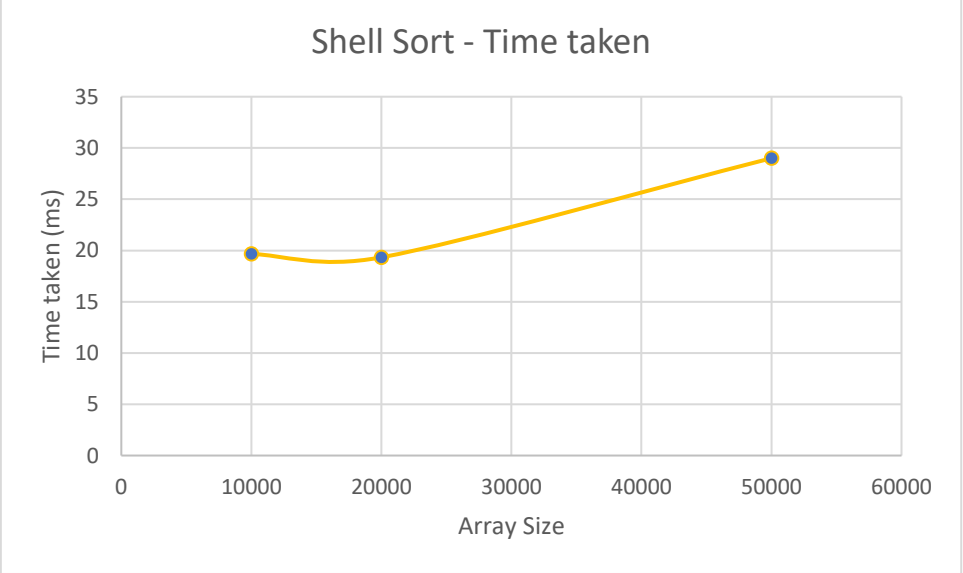
Analysis : Complexity of quick sort in notes = O(n log n)

From my graph we can see a similar result. The time taken doesn't go up as much as array size increases.

As for number of operations graph it is the same has the complexity for sorting algorithms.

Shell Sort (gap always odd)			
Compare	**Move**	**Operations**	**Time Taken(ms)**
57354	30829	88183	20
108191	60780	168971	19
290240	156219	446459	29



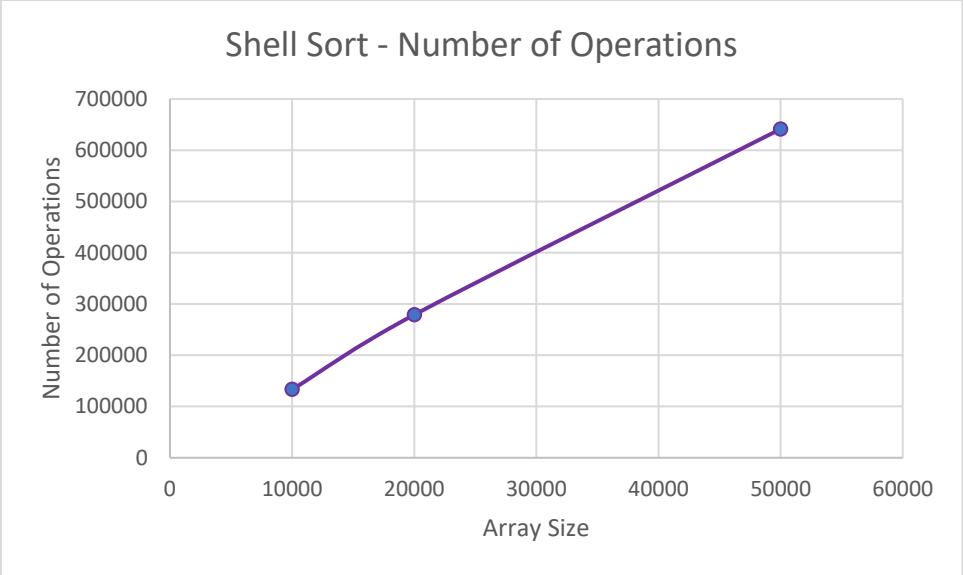


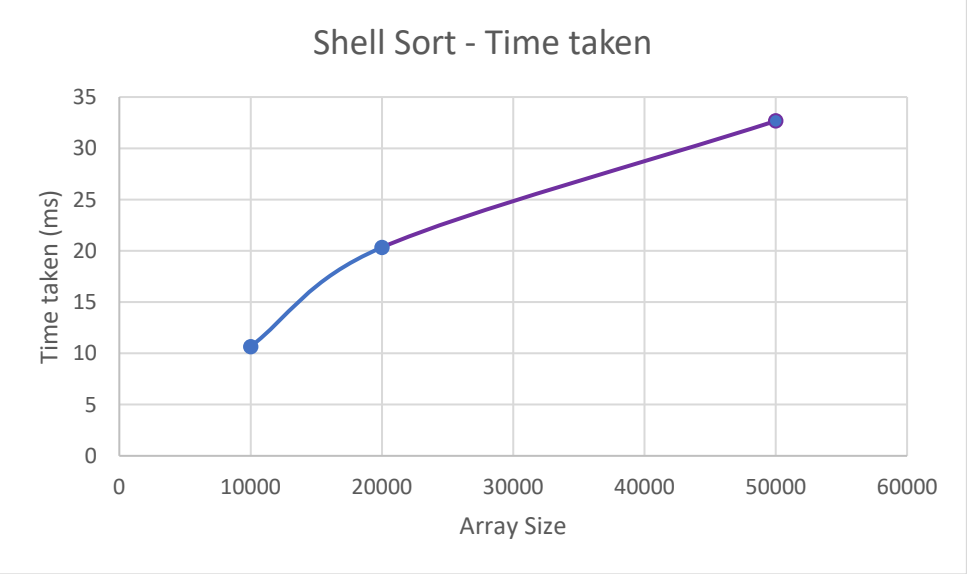
Analysis: Complexity of Shell sort in notes : O(n^1.5)

From my data I think we need to have bigger array sizes up 100000 - > 200000 or more to actually see the complexity of O(n^1.5).

However for the number of operations. It is the same result as the complexity graph of number of operations in the notes.

Shell Sort (task 7)			
Compare	**Move**	**Operations**	**Time Taken(ms)**
101396	31793	133189	11
214112	64859	278971	20
483364	157933	641297	33





Analysis :

When we commented out the code for the gap to always be odd the number of operations increased and although the time taken to sort didn't change as much, the changes will be seen if we test on array sizes around 100000+ which is the conclusion from the lecture slides that setting the gap to always be odd speeds up the sorting process.