For this assignment, we will be analyzing 6 sorting methods and comparing the efficiency (primarily from a time standpoint) of each in several different scenarios. We will also track the time taken as the number of elements N in the list rises and experimentally deduce big-O time of each algorithm.

We will go from N = 1 all the way to N = 50,001, by increments of 5, and get the total number of comparisons, moves, and comparison+moves. For each N, We will generate 10 random lists, and perform the algo on each one and average the number of comparison+moves. Then, we will plot average number of comps+moves vs N and see what the graph looks like. If I have time, I will also run a correlation against the predicted O(N) for each graph and see how good it is.

*the lists will have random integers from 0 to 200, to keep it simple. I could have done doubles or even a mixture of both, but conceptually this won't change the algorithms and instead would add unnecessary complexity to the assignment.

As you can tell, my approach to this project is largely experimental and deals mostly with empirically (rather than analytically) deriving the prowess of each algo.

Arguably the simplest algorithm, let's start with Insertion Sort. This algo simply goes left-to-right in the list, and inserts the element it's on to the correct spot in the section to the right of it.

Selection Sort

BucketSort (similar to radix sort)

HeapSort

Mergesort

Quicksort.

CODE:

Runs for 10-20 seconds, generates counts matrix, then asks for a number of elements and which sorting method.

Sort.java

```
import java.util.Arrays;
import java.io.*;
import java.util.Scanner;
import java.lang.Math;


public class Sort {


	public static void main(String[] args) {
		int min = 0;
		int max = 200;
		int range = max-min;
		int [] counts = new int[2];
		int reps = 10;
		int [] movesC = new int[1000];
		int [] compsC = new int[1000];
		int sumc; int sumb;

		//for each number of elements in list
		for(int i = 1; i < 5001; i = i+5) {
			int [] list = new int[i];


			//do it ten times to average out results
			int Mavgs = 0;
			int Cavgs = 0;
			for(int b = 0; b < reps; b++) {
				//for each element in the list
				for(int j = 0; j < i; j++) {
					list[j] = (int)Math.round(Math.random()*range +
min);
```

```
                        }
                        counts = hellaSort.insertionSort(list);
                        Cavgs += counts[0];
                        Mavgs += counts[1];
                }

                movesC[(i-1)/5] = Mavgs/reps;
                compsC[(i-1)/5] = Cavgs/reps;

        }

        System.out.print("\n" + Arrays.toString(movesC));
        System.out.print("\n" + Arrays.toString(compsC));




        System.out.print("How many elements in list?\n");
        Scanner input = new Scanner (System.in);

        int numElements = input.nextInt();
        boolean random = true;
        int[] list = new int[numElements];
        if(random == true) {
                double coef; int num;
                min = 0;
                max = 25;
                range = max-min;
                for(int k = 0; k < numElements; k++) {
                        coef = Math.random();
                        num = (int)Math.round(coef*range + min);
                        list[k] = num;
                }

        }
        System.out.print("InsertionSort, SelectionSort, HeapSort, BucketSort,
MergeSort, or QuickSort?");
        String select = input.next();
        char cselect = select.charAt(0);
        System.out.print("Old List:" + Arrays.toString(list));
        int newList[] = new int[numElements];
        if(cselect == 'i') {
                newList = hellaSort.insertionSort(list);
        }
        else if(cselect == 's') {
                newList = hellaSort.selectionSort(list);
        }
        else if(cselect == 'q') {
                newList = hellaSort.quickSort(list);
        }
        /* this method does hard copy, other don't (bc technicality) */
        else if(cselect == 'h') {
                //create Integer version of list
                Integer[] list1 = new Integer[list.length];
```

```
                for(int i = 0; i < list.length; i++)
                        list1[i] = list[i];
                //create Integer version of newList
                Integer[] newList1 = new Integer[list.length];
                newList1 = HeapSort.<Integer>heapSort(list1);
                //convert Integer version back to int version
                for(int i = 0; i < list.length; i++)
                        newList[i] = newList1[i];
        }
        else if(cselect == 'm') {
                hellaSort.mergeSort(list);
                newList = list;
        }
        else if(cselect == 'b') {
                hellaSort.bucketSort(list);
                newList = list;
        }
        System.out.print("\nNew List: " + Arrays.toString(newList));

    }



}
```

hellaSort.java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class hellaSort {

    public static int [] insertionSort(int[] list) {
        //set the counting variables
        int comps = 0;
        int moves = 0;

        //begin alg
        for(int i = 1; i < list.length; i++) {
                int currentElement = list[i];
                int k;
                for (k = i-1; k>=0 && list[k] > currentElement; k--) {
                        list[k+1] = list[k]; moves++; comps++;
                }

                list[k+1] = currentElement;
        }
        int [] counts = {comps, moves};
        return counts;
    }

    public static int [] selectionSort(int[] list) {
        for(int i = 0; i < list.length-1; i++ ) {
```

```
			int currentMin = list[i];
			int currentMinIndex = i;

			for(int j = i+1; j < list.length; j++) {
				if(currentMin > list[j]) {
					currentMin = list[j];
					currentMinIndex = j;
				}
			}

			if(currentMinIndex != i) {
				list[currentMinIndex] = list[i];
				list[i] = currentMin;
			}
		}
		return list;
	}

	public static int [] quickSort(int [] list) {
		list = quickSortSection(list, 0, list.length - 1);
		return list;
	}

	public static int [] quickSortSection(int [] list, int first, int last) {
		if(last > first) {
			int pivotIndex = partition(list, first, last);
			list = quickSortSection(list, first, pivotIndex-1);
			list = quickSortSection(list, pivotIndex + 1, last);
		}
		return list;
	}

	public static int partition(int [] list, int first, int last) {
		int pivot = list[first];
		int low = first + 1;
		int high = last;

		while (low < high) {
			while(low <= high && list[low] <= pivot)
				low++;
			while(low <= high && list[high] > pivot)
				high--;

			if(high > low) {
				int temp = list[high];
				list[high] = list[low];
				list[low] = temp;
			}
		}

		while (high > first && list[high] >= pivot)
			high--;

		if(pivot > list[high]) {
			list[first] = list[high];
```

```
                        list[high] = pivot;
                        return high;
                }
                else {
                        return first;
                }
        }

        public static void mergeSort(int[] list) {
                if(list.length > 1) {
                        //first half
                        int[] firstHalf = new int[list.length/2];
                        System.arraycopy(list,  0,  firstHalf, 0, list.length/2);
                        mergeSort(firstHalf);

                        //second half
                        int secondHalfLength = list.length - list.length/2;
                        int[] secondHalf = new int[secondHalfLength];
                        System.arraycopy(list, list.length/2, secondHalf, 0,
secondHalfLength);
                        mergeSort(secondHalf);

                        //merge first and second half into list
                        int current1 = 0;
                        int current2 = 0;
                        int current3 = 0;

                        int [] list1 = firstHalf;
                        int [] list2 = secondHalf;
                        int [] temp = list;
                        while(current1 < list1.length && current2 < list2.length) {
                                if(list1[current1] < list2[current2])
                                        temp[current3++] = list1[current1++];
                                else
                                        temp[current3++] = list2[current2++];
                        }

                        while(current1 < list1.length)
                                temp[current3++] = list1[current1++];

                        while(current2 < list2.length)
                                temp[current3++] = list2[current2++];


                }
        }
        public static void bucketSort(int[] list) {
                //mod number
                int n = 10; //decimal system
                int m;
                ArrayList<Integer> [] buckets = new ArrayList[n];

                //add all elements of list into buckets
                for(int i = 0; i < list.length; i++) {
                        //get key
```

```
                m = list[i]%n;
                //create list at key if not yet done
                if(buckets[m] == null) {
                        buckets[m] = new ArrayList<Integer>();
                }
                //add value to key
                buckets[m].add(list[i]);
        }

        int k = 0;
        //now we insertionsort every bucket
        for(int i = 0; i < n; i++) {
                if(!buckets[i].isEmpty()) {
                        int [] temp = new int[buckets[i].size()];
                        for(int j = 0; j < buckets[i].size(); j++) {
                                temp[j] = buckets[i].get(j);
                        }
                        insertionSort(temp);
                        java.lang.System.arraycopy(temp, 0, list, k, temp.length);
                        k = k + temp.length;
                }

        }
    }
}
```

Heap.java

```
public class Heap<E extends Comparable<E>> {
    private java.util.ArrayList<E> list = new java.util.ArrayList<>();

    public Heap() {

    }

    public Heap(E[] objects) {
        for (int i = 0; i < objects.length; i++) {
            add(objects[i]);
        }
    }
    public void add(E newObject) {
        list.add(newObject);
        int currentIndex = list.size() - 1;

        while (currentIndex > 0) {
            int parentIndex = (currentIndex-1)/2;
            if(list.get(currentIndex).compareTo(
                        list.get(parentIndex)) > 0) {
                E temp = list.get(currentIndex);
                list.set(currentIndex,  list.get(parentIndex));
                list.set(parentIndex,  temp);
            }
```

```
                else
                        break;

                currentIndex = parentIndex;
        }
    }

    public E remove() {
        if (list.size() == 0) return null;

        E removedObject = list.get(0);
        list.set(0,  list.get(list.size() - 1));
        list.remove(list.size() - 1);

        int currentIndex = 0;
        while (currentIndex < list.size()) {
            int leftChildIndex = 2*currentIndex + 1;
            int rightChildIndex = 2*currentIndex + 2;

            //Find the maximum between two children
            if(leftChildIndex >= list.size()) break;
            int maxIndex = leftChildIndex;
            if (rightChildIndex < list.size()) {
                if(list.get(maxIndex).compareTo(
                                list.get(rightChildIndex)) < 0) {
                    maxIndex = rightChildIndex;
                }
            }

            if (list.get(currentIndex).compareTo(
                            list.get(maxIndex)) < 0) {
                E temp = list.get(maxIndex);
                list.set(maxIndex,  list.get(currentIndex));
                list.set(currentIndex, temp);
                currentIndex = maxIndex;
            }
            else
                break;
        }

        return removedObject;
    }

    public int getSize() {
        return list.size();
    }
}
```

Heapsort.java

```
public class HeapSort {
    /*Heap sort method*/
    public static <E extends Comparable<E>> E[] heapSort(E[] list) {
        //Create a Heap of integers
```

```
		Heap<E> heap = new Heap<>();

		//Add elements to the heap
		for(int i = 0; i < list.length; i++) {
			heap.add(list[i]);
		}

		//Remove elements from the heap
		for(int i = list.length - 1; i >= 0; i--)
			list[i] = heap.remove();

		return list;
	}



	public static Integer[] hehe(Integer[] list) {
		heapSort(list);
		for (int i = 0; i < list.length; i++)
			System.out.print(list[i] + " ");
		return list;
	}

}
```