1 Selection Sort

Selection sort is a n squared algorithm used to sort a list by finding the smallest or largest element in the array with an inner for loop. The other for loop is keeping track of the current element that we will replace it with or in other words, swap

```
//this is an o(n2) algorithm
//This finds the next mallest element in the array
//and puts it in correct postion according to the "i"th place in the array.
    public static int selectionSort(String[] myArray) {
        int comparisions = 0;

        int arrayLength = myArray.length;
        for (int i = 0; i < arrayLength - 2; i++) {
            int smallPosition = i;
            for (int j = i + 1; j < arrayLength; j++) {

        if (myArray[j].replaceAll("\\s", "").compareTo(myArray[smallPosition].replaceAll("\\s", ""))

                    smallPosition = j;
                }
                comparisions++;
            }
//this is the swapping work that is done. it finds the next
//smallest element and swaps its position with i
            String temp = myArray[smallPosition];
            myArray[smallPosition] = myArray[i];
            myArray[i] = temp;
        }
        return comparisions;
    }
```

2 Insertion Sort

Insertion sort is still a n squared algorithm but it's different from selection sort. The difference is it will take the current element and keep comparing it to the next element in line. if the next element in the line is smaller then it will swap the 2. This is in a while loop which it won't stop until it's in the correct position.

```
public static int insertionSort(String[] myArray) {
        int comparisions = 0;

        for (int j = 1; j < myArray.length; j++) {

            String key = myArray[j].replaceAll("\\s", "");
            int i = j - 1;
//This while loop will keep swapping
//the current element with the next element until the conidtion isn't met anymore
            while (i >= 0 && (myArray[i].replaceAll("\\s", "").compareTo(key) > 0)) {

                myArray[i + 1] = myArray[i];
                i--;
                comparisions += 1;
            }
            myArray[i + 1] = key;

        }
        //This returns the comparison count
        return comparisions;
    }
```

3 Merge Sort

This is where the algorithm gets better. The algorithm's time will now take n log n. Merge sort isn't iterative, it is recursive. It is a divide and conquer algorithm which means it will divide the problem into sub-problems all of array size 1. This technically means all of the arrays are sorted. The conquering part is the part where the sorting is completed.

```
public static void mergeSort(String[] myArray, int p, int r) {
//This is like the base case for the function calls
        if (p < r) {
            int q = (p + r) / 2;
//This calls merge sort for the first half of the array and //the second half of the array
//This will keep going until the length of the arrays are 1
            mergeSort(myArray, p, q);
            mergeSort(myArray, q + 1, r);
            //This is used to merge everything together
            merge(myArray, p, q, r);
        }
    }


    public static void merge(String[] nums, int p, int q, int r){
        int left = p;
        int right = q + 1;
        String[] sentinal = new String[r-p+1];
        int t = 0;

        while (left <= q && right <= r){
            if (nums[left].compareTo(nums[right]) < 0){
                sentinal[t++] = nums[left++];
            }else{
                sentinal[t++] = nums[right++];
            }
            mergeSortCount+=1;
        }

        while (left <= q) {
            sentinal[t++] = nums[left++];}

        while (right <= r)
        {
        sentinal[t++] = nums[right++];
        }


        for (int i = p; i <= r; i++){
            nums[i] = sentinal[i-p];
        }
    }
```

4 Quick Sort

Quick sort is known to be the best sorting algorithm out of all 4 of these. It is still n log n time like merge sort, but it can be done in place rather than taking up more space in the algorithm. Quick sort is another recursive technique in conquering the problem. Quick sort picks a pivot value in the array and will put the values that are less than that pivot value on the left and values that are less than it on the right. If the algorithm keeps doing this every time, then the algorithm will be sorted.

```
public static void quickSort(String[] myArray, int smallIndex, int bigIndex) {

//This is the base case for the quickSort because we
//dont want the small index to be equal to the big index.
        if (smallIndex < bigIndex) {
//The pivot value is the element we will pivot around
```

```
            int pivotValue = getPivotIndex(myArray, smallIndex, bigIndex);

            //Recursively calls it self untill the arrays are sorted
            quickSort(myArray, smallIndex, pivotValue - 1);
            quickSort(myArray, pivotValue + 1, bigIndex);

        }
    }

    public static int getPivotIndex(String[] myArray, int smallIndex, int bigIndex) {


        String pivotIndex = myArray[bigIndex].replaceAll("\\s", "");

        int smallerElement = smallIndex - 1;
        for (int j = smallIndex; j <= bigIndex - 1; j++) {

//Im comparing the elements in the array to the current pivot //value If the element is smaller than
            if ((myArray[j].replaceAll("\\s", "").compareTo(pivotIndex)) < 0) {
                smallerElement += 1;
                String temp = myArray[smallerElement];
                myArray[smallerElement] = myArray[j];
                myArray[j] = temp;
            }
            //quicksort count goes up for every comparison
            quickSortCount += 1;

        }
        //This
        String temp = myArray[smallerElement + 1];
        myArray[smallerElement + 1] = myArray[bigIndex];
        myArray[bigIndex] = temp;
        smallerElement +=1;
        return smallerElement;
    }
```

5 Number of comparisons

Quick sort	Merge sort	Selection sort	Insertion sort
7236	5445	221444	114317