Client.java

```
import java.util.Random; // lots of pseudo-random numbers to generate

/**
 * @author joe
 */
public class Client { // ... what if EVERYTHING was an object?
   public static Random r           = new Random();
   public static Student students[] = new Student[1000000];
   public static boolean[] intFlag  = new boolean[10000000]; // flags for the integers to ensure unique
ID
   public static String[] sList     = new String[10];
   public static float[] fList      = new float[20]; // split by 5% chunks

   // times for individual tests
   public static long merge_sort_ascending_id       = -1; // benchmark 1
   public static long quick_sort_ascending_gpa      = -1; // benchmark 2
   public static long bubble_sort_descending_id     = -1; // benchmark 3
   public static long insertion_sort_descending_gpa = -1; // benchmark 4
   public static long selection_sort_standing       = -1; // benchmark 5
   public static long radix_sort_complex            = -1; // benchmark 6

   public static void main(String[] args) {
      try {
         for(int i = 0; i < 1000000; i++)
            students[i] = new Student();

         fillStandingList();
         fillGpaList();

         System.out.println("Generating random student info");
         fillStudentArray(1000000);

         Student[] working_set;

         // BENCHMARK 1
         working_set = copyElementsTo(students, 1000000);
         System.out.println("\nTesting merge sort w/ ascending ID...");
         merge_sort_ascending_id = System.nanoTime();
         Sort.mergeSort(working_set, new IdCompAscending());
         merge_sort_ascending_id = System.nanoTime() - merge_sort_ascending_id;
         System.out.println("    Delta time: " + (merge_sort_ascending_id / 1000000) + " ms");
         working_set = null; // free for garbage collector

         // BENCHMARK 2
         working_set = copyElementsTo(students, 1000000);
         System.out.println("\nTesting quick sort w/ ascending GPA...");
         quick_sort_ascending_gpa = System.nanoTime(); // start time
         Sort.quickSort(working_set, new GpaCompAscending());
```

```
        quick_sort_ascending_gpa = System.nanoTime() - quick_sort_ascending_gpa; // delta time
        System.out.println("    Delta time: " + (quick_sort_ascending_gpa / 1000000) + " ms"); // nano
seconds to milliseconds
        working_set = null;

        // BENCHMARK 3
        working_set = copyElementsTo(students, 100000);
        System.out.println("\nTesting bubble sort w/ descending ID...");
        bubble_sort_descending_id = System.nanoTime();
        Sort.simpleBubbleSort(working_set, new IdCompDescending());
        bubble_sort_descending_id = System.nanoTime() - bubble_sort_descending_id;
        System.out.println("    Delta time: " + (bubble_sort_descending_id / 1000000) + " ms");
        working_set = null;

        // BENCHMARK 4
        working_set = copyElementsTo(students, 100000);
        System.out.println("\nTesting insertion sort w/ descending GPA...");
        insertion_sort_descending_gpa = System.nanoTime();
        Sort.insertionSort(working_set, new GpaCompDescending());
        insertion_sort_descending_gpa = System.nanoTime() - insertion_sort_descending_gpa;
        System.out.println("    Delta time: " + (insertion_sort_descending_gpa / 1000000) + " ms");
        working_set = null;

        // BENCHMARK 5
        working_set = copyElementsTo(students, 100000);
        System.out.println("\nTesting selection sort w/ standing...");
        selection_sort_standing = System.nanoTime();
        Sort.selectionSort(working_set, new StandingComp());
        selection_sort_standing = System.nanoTime() - selection_sort_standing;
        System.out.println("    Delta time: " + (selection_sort_standing / 1000000) + " ms");
        working_set = null;

        // BENCHMARK 6
        working_set = copyElementsTo(students, 1000000);
        System.out.println("\nTesting radix sort with 5 keys...");
        radix_sort_complex = System.nanoTime();
        Sort.radixSort(working_set,
                new StandingComp(),
                new GpaCompDescending(),
                new LastNameCompAscending(),
                new FirstNameComp());
        radix_sort_complex = System.nanoTime() - radix_sort_complex;
        System.out.println("    Delta time: " + (radix_sort_complex / 1000000) + " ms");
        working_set = null;

        //Sort.mergeSort(students, new IdCompAscending());
        //Sort.quickSort(students, new GpaCompAscending());
        //Sort.simpleBubbleSort(students, new IdCompDescending());
        //Sort.insertionSort(students, new GpaCompDescending());
```

```
            //Sort.selectionSort(students, new StandingComp());

            printResults();

        } catch(Exception e) {
            System.out.println(e.toString());
            e.printStackTrace();
        }
    }

    /**
     * @param s array to copy data from
     * @param num number of elements to copy from source array (s)
     * @return deep-copied array
     */
    public static Student[] copyElementsTo(Student[] s, int num) {
        Student[] tmp = new Student[num];
        for(int i = 0; i < num; i++) // manual array copy
            tmp[i] = s[i].getCopy(); // deep copy of each Student to ensure
        return tmp;                  // the same working set for each algorithm
    }

    /**
     * print results of benchmarks
     */
    public static void printResults() {
        String vertical_seperator = "-----------------------------------------";

        System.out.println(vertical_seperator);
        System.out.println("|    Sort       |   N       |     Time    |");
        System.out.println(vertical_seperator);
        System.out.printf("| Merge         | 1,000,000 | %8d ms |\n", merge_sort_ascending_id/1000000);
        System.out.println(vertical_seperator);
        System.out.printf("| Quick Sort    | 1,000,000 | %8d ms |\n", quick_sort_ascending_gpa/1000000);
        System.out.println(vertical_seperator);
        System.out.printf("| Bubble Sort   | 100,000   | %8d ms |\n",
bubble_sort_descending_id/1000000);
        System.out.println(vertical_seperator);
        System.out.printf("| Insertion Sort | 100,000   | %8d ms |\n",
insertion_sort_descending_gpa/1000000);
        System.out.println(vertical_seperator);
        System.out.printf("| Selection Sort | 100,000   | %8d ms |\n", selection_sort_standing/1000000);
        System.out.println(vertical_seperator);
        System.out.printf("| Radix Sort    | 1,000,000 | %8d ms |\n", radix_sort_complex/1000000);
        System.out.println(vertical_seperator);
    }

    public static void printFirstElements() {
        for(int i = 0; i < 10; i++)
```

```
        System.out.println(students[i]);
    System.out.println("\n\n");
}

/**
 * @param num number of entries to refill
 *
 */
public static void fillStudentArray(int num) {
    for(int i = 0; i < num; i++) {
        students[i].setFname(getRandomString());
        students[i].setLname(getRandomString());
        students[i].setId(getUniqueRandomInt());
        students[i].setStanding(getDistributedStanding());
        students[i].setGpa(getDistributedGpa());
    }
}

/**
 * @return properly distributed GPA
 */
public static float getDistributedGpa() {
    int tmp = Math.abs(r.nextInt()) % 20; // unsigned ints ftw
    float fTmp = fList[tmp] + (r.nextFloat() - 0.01f);
    if(fTmp > 4.0f)
        return 4.0f;

    if(fTmp < 0.0f)
        return fTmp+0.01f;

    return fTmp;
}

/**
 * @return String representation of standing
 * access standing LUT with randomly generated unsigned integer
 */
public static String getDistributedStanding() {
    int tmp = Math.abs(r.nextInt()) % 10;
    return sList[tmp];
}

/**
 * reset ID LUT to false
 */
public void resetIdFlags() {
    for(int i = 0; i < 10000000; i++)
        intFlag[i] = false;
}
```

```
/**
 *
 * @return integer that is unique as of most recent LUT reset
 */
public static int getUniqueRandomInt() {
    int tmp = 0;
    do {
        tmp = Math.abs(r.nextInt()) % 10000000;
    } while(intFlag[tmp]); // ..lack of static local variables in Java
    intFlag[tmp] = true;
    return tmp;
}

/**
 *
 * @return string with proper formatting
 */
public static String getRandomString() {
    String tmp = "";

    // random length 10 .. 15
    int rLength = 10 + (Math.abs(r.nextInt()) % 5);

    // first character is UPPERCASE
    tmp += (char)(65 + (Math.abs(r.nextInt()) % 26));

    // rest are lowercase
    for(int i = 0; i < rLength-1; i++) {
        tmp += (char)(97 + (Math.abs(r.nextInt()) % 26));
    }

    return tmp;
}

/**
 * fill standing LUT with properly distributed values
 */
public static void fillStandingList() {
    // use as few memory-allocated references as possible

    sList[0] = "freshman";  // 40% freshman
    sList[1] = sList[0];
    sList[2] = sList[0];
    sList[3] = sList[0];

    sList[4] = "sophomore"; // 30% sophomores
    sList[5] = sList[4];
    sList[6] = sList[4];
```

```
        sList[7] = "junior";    // 20% juniors
        sList[8] = sList[7];

        sList[9] = "senior";    // 10% seniors
    }

    /**
     * fill GPA LUT with properly distributed values
     */
    public static void fillGpaList() {
        fList[0] = 0.0f;

        for(int i = 1; i < 5; i++)
            fList[i] = 1.0f;

        for(int i = 5; i < 15; i++)
            fList[i] = 2.0f;

        for(int i = 15; i < 19; i++)
            fList[i] = 3.0f;

        fList[19] = 4.0f;

    }

}
```

Sort.java

```
import java.util.Arrays;

/**
 * @author joe
 */
public class Sort {
   // used with Bubble, Insertion, and Selection sort methods
   private static final int bubbleSortAmt = 100000; // lower than 100000 for testing purposes

   public static void radixSort(Object[] o, Comparator c1, Comparator c2) {
      quickSort(o, c2);
      quickSort(o, c1);
   }

   public static void radixSort(Object[] o, Comparator c1, Comparator c2, Comparator c3) {
      radixSort(o, c2, c3);
      quickSort(o, c1);
   }

   public static void radixSort(Object[] o, Comparator c1, Comparator c2, Comparator c3, Comparator
c4) {
      radixSort(o, c2, c3, c4);
      quickSort(o, c1);
   }

   private static void quickSortInPlace(Object[] o, Comparator comp, int a, int b) {
      if(a >= b)
         return;

      int left = a;
      int right = b-1;
      Object pivot = o[b];
      Object tmp;

      while(left <= right) {
         while(left <= right && comp.compare(o[left], pivot) < 0)
            left++;
         while(left <= right && comp.compare(o[right], pivot) > 0)
            right--;

         if(left <= right) {
            tmp = o[left];
            o[left] = o[right];
            o[right] = tmp;

            left++;
            right--;
         }
```

```
    }

    tmp = o[left];
    o[left] = o[b];
    o[b] = tmp;

    // recursive function calls
    quickSortInPlace(o, comp, a, left-1);
    quickSortInPlace(o, comp, left+1, b);
}

public static void quickSort(Object[] o, Comparator comp) {
    quickSortInPlace(o, comp, 0, o.length-1);
}

// for very large arrays of data, the working stack will tend to grow pretty large
public static void mergeSort(Object[] o, Comparator comp) {
    int n = o.length;
    if(n < 2)
        return;

    int mid = n/2;

    Object[] S1 = Arrays.copyOfRange(o, 0, mid);
    Object[] S2 = Arrays.copyOfRange(o, mid, n);

    mergeSort(S1, comp);
    mergeSort(S2, comp);

    merge(S1, S2, o, comp);
}

private static void merge(Object[] S1, Object[] S2, Object[] S, Comparator comp) {
    int i = 0, j = 0;
    while(i+j < S.length) {
        if(j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
            S[i+j] = S1[i++];
        else
            S[i+j] = S2[j++];
    }
}

public static void selectionSort(Object[] o, Comparator comp) {
    for(int i = 0; i < bubbleSortAmt-1; i++) {

        int j = i;
        for(int k = i+1; k < bubbleSortAmt; k++) {
            if(comp.compare(o[k], o[j]) < 0)
                j = k;
```

```
            }

            // swap object references
            Object tmp = o[j];
            o[j] = o[i];
            o[i] = tmp;
        }
    }

    public static void insertionSort(Object[] o, Comparator comp) {
        Object key;
        int j;
        for(int i = 0; i < bubbleSortAmt; i++) {
            key = o[i];
            j = i-1;

            while(j >= 0 && comp.compare(o[j], key) < 0) {
                o[j+1] = o[j];
                j--;
            }

            o[j+1] = key;
        }
    }

    // bubble sort is not required to run entire array
    public static void simpleBubbleSort(Object[] o, Comparator comp) {

        Student[] s = (Student[])o;

        for(int i = 0; i < bubbleSortAmt; i++) {
            for(int j = 0; j < bubbleSortAmt-1; j++) {
                if(comp.compare(s[j], s[j+1]) > 0) {
                    // swap by using temp variable
                    Student s_tmp = s[j];
                    s[j] = s[j+1];
                    s[j+1] = s_tmp;
                }
            }
        }
    }
}
```

Comparator.java

```
/**
 * @author joe
 * @param <E> type used in this Comparator
 */
public interface Comparator<E> {
    public int compare(E t1, E t2);
}
```

FirstNameComp.java

```
/**
 * @author joey
 * lexicographically text first names of students
 */
public class FirstNameComp implements Comparator<Student> {
    /**
     * @param t1 Student 1
     * @param t2 Student 2
     * @return comparison
     */
    @Override
    public int compare(Student t1, Student t2) {
        return t1.getFname().compareTo(t2.getFname());
    }
}
```

LastNameComp.java

```
/**
 * @author joey
 * lexicographically test last names of students, ascending order
 */
public class LastNameComp implements Comparator<Student> {
    /**
     * @param t1 Student 1
     * @param t2 Student 2
     * @return comparison
     */
    @Override
    public int compare(Student t1, Student t2) {
        return t1.getLname().compareTo(t2.getLname());
    }
}
```

StandingComp.java

```
/**
 * @author joey
 */
public class StandingComp implements Comparator<Student> {
    private int getIntFromString(String standing) {
        if(standing.equals("senior")) {
            return 0;
        } else if(standing.equals("junior")) {
            return 1;
        } else if(standing.equals("sophomore")) {
            return 2;
        } else if(standing.equals("freshman")) {
            return 3;
        }

        return -1;
    }

    /**
     * @param t1 Student 1
     * @param t2 Student 2
     * @return comparison
     */
    @Override
    public int compare(Student t1, Student t2) {
        int t1_num = getIntFromString(t1.getStanding());
        int t2_num = getIntFromString(t2.getStanding());

        if(t1_num < t2_num)
            return -1;
        if(t2_num < t1_num)
            return 1;
        return 0;
    }
}
```

IdComp.java

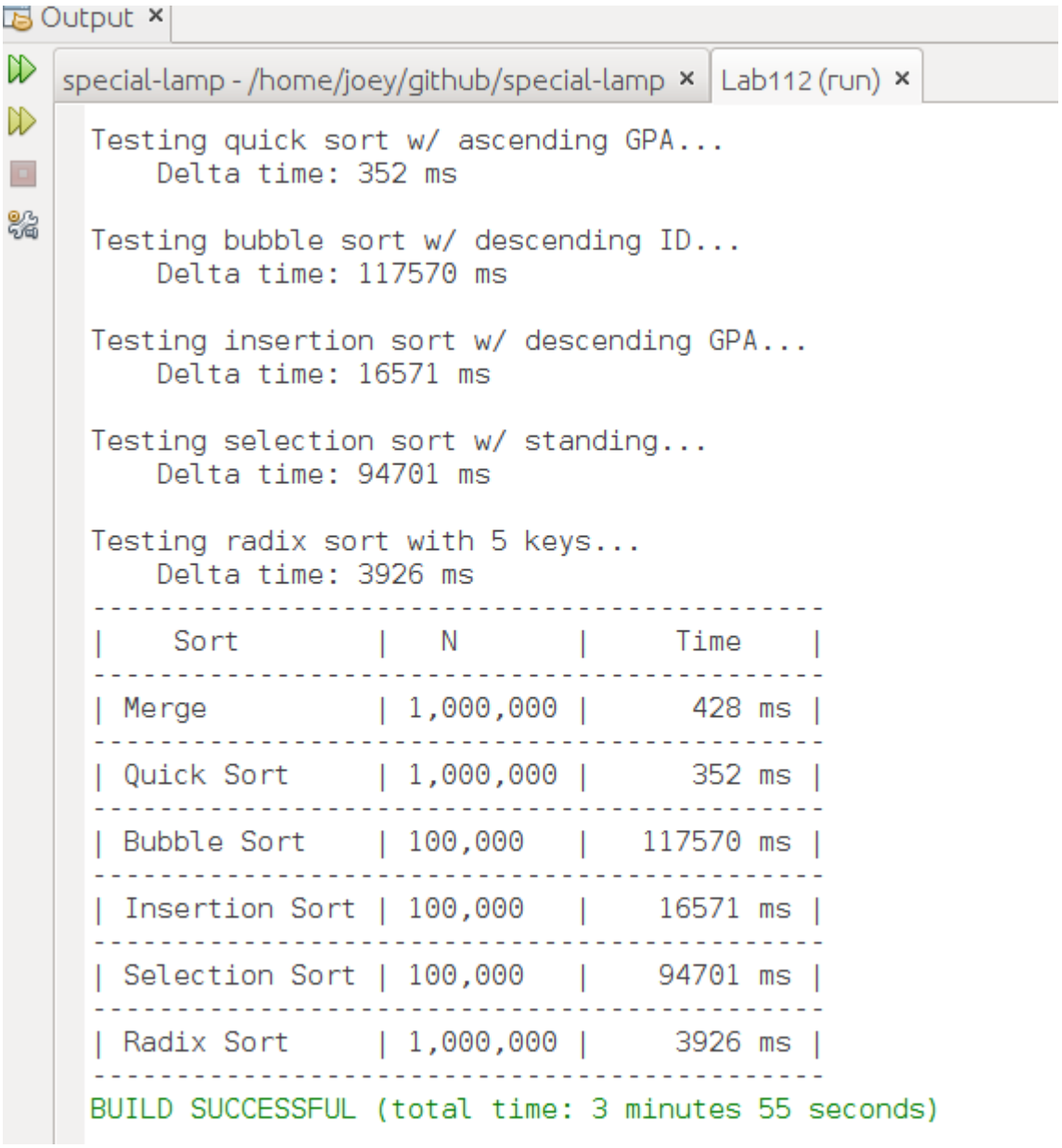
```
/**
 * @author joe
 */
public class IdComp implements Comparator<Student> {

    /**
     * @param t1 Student 1
     * @param t2 Student 2
     * @return comparison
     */
    @Override
    public int compare(Student t1, Student t2) {
        if(t1.getId() < t2.getId())
            return -1;
        if(t1.getId() > t2.getId())
            return 1;
        return 0;
    }

}
```

GpaComp.java

```
/**
 *
 * @author joey
 */
public class GpaComp implements Comparator<Student> {
    /**
     * @param t1 Student 1
     * @param t2 Student 2
     * @return comparison
     */
    @Override
    public int compare(Student t1, Student t2) {
        if(t1.getGpa() > t2.getGpa())
            return 1;
        if(t2.getGpa() > t1.getGpa())
            return -1;
        return 0;
    }
}
```


Output
special-lamp - /home/joey/github/special-lamp
Lab112 (run)
Testing quick sort w/ ascending GPA...
Delta time: 352 ms
Testing bubble sort w/ descending ID...
Delta time: 117570 ms
Testing insertion sort w/ descending GPA...
Delta time: 16571 ms
Testing selection sort w/ standing...
Delta time: 94701 ms
Testing radix sort with 5 keys...
Delta time: 3926 ms
Sort	N	Time
Merge	1,000,000	428 ms
Quick Sort	1,000,000	352 ms
Bubble Sort	100,000	117570 ms
Insertion Sort	100,000	16571 ms
Selection Sort	100,000	94701 ms
Radix Sort	1,000,000	3926 ms
BUILD SUCCESSFUL (total time: 3 minutes 55 seconds)