Aug 20, 12 7:56 **BubbleSort.java** Page 1/1

```
package sorting;

public class BubbleSort implements Sorting {

    public void sort(int[] A, int n, SortStatistics stats) {
        // Pre: A == A0
        for (int i = n - 1; i >= 1; i--) {
            for (int j = 1; j <= i; i++) {
                stats.incrCompare();
                if (A[j - 1] > A[j]) {
                    // Swap A[j-1] and A[j]
                    stats.incrMove();
                    int temp = A[j - 1];
                    A[j - 1] = A[j];
                    A[j] = temp;
                }
            }
        }
        /* Post: ordered(A) && (items(A) = items(A0)) */
    }

}
```

Aug 04, 14 9:23 **InsertionSort.java** Page 1/1

```
package sorting;

public class InsertionSort implements Sorting {

    /* Insertion sort based on CLRS p18 (3rd) */
    public void sort(int[] A, int n, SortStatistics stats) {
        // Pre: A == A0
        for (int j = 1; j < n; j++) {
            int key = A[j];
            stats.incrMove();
            // Insert A[j] into the sorted sequence A[0..j-1]
            int i = j - 1;
            while (i >= 0 && A[i] > key) {
                stats.incrCompare();
                stats.incrMove();
                A[i + 1] = A[i];
                i = i - 1;
            }
            stats.incrCompare();
            A[i + 1] = key;
            stats.incrMove();
        }
        /* Post: ordered(A) && (items(A) = items(A0)) */
    }

}
```

```
package sorting;

import java.util.LinkedList;
import java.util.List;

public class MergeSort implements Sorting {

    SortStatistics stats;

    private void merge(int[] A, int lo, int mid, int hi) {
        int i, j;
        List<Integer> L = new LinkedList<Integer>();
        /* A = A0 && ordered(A[lo..mid]) && ordered(A[mid+1..hi]) */
        i = lo;
        j = mid + 1;
        /*
         * While both segments are non-empty select the least element and add to
         * the list.
         */
        while ((i <= mid) && (j <= hi)) {
            stats.incrCompare();
            stats.incrMove();
            if (A[i] <= A[j]) {
                L.add(A[i]);
                i = i + 1;
            } else {
                L.add(A[j]);
                j = j + 1;
            }
        }
        /* Move any elements left in the first segment to the list. */
        while (i <= mid) {
            stats.incrMove();
            L.add(A[i]);
            i = i + 1;
        }
        /*
         * Move any elements left in the second segment to the list. Note that
         * only one of this while loop and the one above actually do anything.
         */
        while (j <= hi) {
            stats.incrMove();
            L.add(A[j]);
            j = j + 1;
        }
        /* Place the merged elements in the list back into the array. */
        for (i = lo; i <= hi; i++) {
            stats.incrMove();
            A[i] = L.remove(0);
        }
        /*
         * ordered(A[lo..hi]) && items(A) = items(A0) && (A[0..lo-1] =
         * A0[0..lo-1]) && (A[hi+1..HIGH(A)] = A0[hi+1..HIGH(A)])
         */
    }

    private void mSort(int[] A, int lo, int hi) {
        int mid;
        /* (0 <= lo <= hi <= HIGH(A)) && (A = A0) */
        if (lo < hi) {
            mid = (lo + hi) / 2;
            mSort(A, lo, mid);
```



```
            /*
             * ordered(A[lo..mid] && items(A) = items(A0) && (A[0..lo-1] =
             * A0[0..lo-1]) && (A[mid+1..HIGH(A)] = A0[mid+1..HIGH(A)])
             */
            mSort(A, mid + 1, hi);
            /*
             * ordered(A[lo..mid]) && ordered(A[mid+1..hi]) && items(A) =
             * items(A0) && (A[0..lo-1] = A0[0..lo-1]) && (A[hi+1..HIGH(A)] =
             * A0[hi+1..HIGH(A)])
             */
            merge(A, lo, mid, hi);
            /*
             * ordered(A[lo..hi]) && items(A) = items(A0) && (A[0..lo-1] =
             * A0[0..lo-1]) && (A[hi+1..HIGH(A)] = A0[hi+1..HIGH(A)])
             */
        }
    }

    public void sort(int[] A, int n, SortStatistics stats) {
        /* Pre: A == A0 */
        this.stats= stats;
        mSort(A, 0, n - 1);
        /*
         * Post: ordered(A[0..N-1]) && items(A) = items(A0) && (A[N..HIGH(A)] =
         * A0[N..HIGH(A)])
         */
    }


}
```

```
package sorting;
public class QuickSort implements Sorting {

    SortStatistics stats;

    private void swap(int[] A, int i, int j) {
        int t;
        t = A[i];
        A[i] = A[j];
        A[j] = t;
        stats.incrMove();
    }

    private int partition(int[] A, int lo, int hi) {
        int x;
        /*
         * Pre: (0 <= lo < hi < A.length) && (A = A0)
         * Partitioning on the middle element of the array ensures quick sort
         * works efficiently for already sorted arrays. To achieve that with
         * this partitioning algorithm we swap the middle and last elements
         * before beginning the partition proper.
         *
         * If this swap is omitted, the partition will still work correctly, but
         * the algorithm will have its worst-case behaviour for the already
         * ordered array.
         */

        swap(A, (lo + hi) / 2, hi);

        /* Partition on A[hi] -- was the middle element */
        x = A[hi];
        int part = lo;
        for (int j = lo; j < hi; j++) {
            /* A[lo..part-1] <= x < A[part..j-1] */
            stats.incrCompare();
            if (A[j] <= x) {
                swap(A, part, j);
                part = part + 1;
            }
            /* A[lo..part-1] <= x < A[part..j] */
        }
        /* A[lo..part-1] <= x < A[part..hi-1] */
        swap(A, part, hi);
        /*
         * lo <= part <= hi && A[lo..part-1] <= A[part] < A[part+1..hi] &&
         * items(A) = items(A0) && (A[0..lo-1] = A0[0..lo-1]) &&
         * (A[hi+1..] = A0[hi+1..])
         */
        return part;
    }

    public void qSort(int[] A, int lo, int hi) {
        /* (0 <= lo <= hi < A.length) && (A = A0) */
        if (lo < hi) {
            /* 0 <= lo < hi < A.length */
            int part = partition(A, lo, hi);
            /*
             * lo <= part <= hi && A[lo..part-1] <= A[part] <= A[part+1..hi] &&
             * items(A) = items(A0) && (A[0..lo-1] = A0[0..lo-1]) &&
             * (A[hi+1..] = A0[hi+1..])
             */
            qSort(A, lo, part - 1);
            /*
             * ordered(A[lo..part-1] && items(A) = items(A0) && A[lo..part-1]
             * <= A[part] <= A[part+1..hi] && (A[0..lo-1] = A0[0..lo-1]) &&
             * (A[hi+1..] = A0[hi+1..])
             */
            qSort(A, part + 1, hi);
            /*
             * ordered(A[lo..part-1]) && ordered(A[part+1..hi]) &&
             * A[lo..part-1] <= A[part] <= A[part+1..hi] && items(A) = items(A0)
             * && (A[0..lo-1] = A0[0..lo-1]) && (A[hi+1..] = A0[hi+1..])
             */
            /*
             * Therefore: ordered(A[lo..hi]) && items(A) = items(A0) &&
             * (A[0..lo-1] = A0[0..lo-1]) && (A[hi+1..] = A0[hi+1..])
             */
        }
    }

    public void sort(int[] A, int n, SortStatistics stats) {
        /* Pre: n <= A.length && A = A0 */
        /*
         * Post: ordered(A[0..n-1]) && items(A) = items(A0) && (A[n..] =
         * A0[n..])
         */
        this.stats = stats;
        qSort(A, 0, n - 1);
        /*
         * Post: ordered(A[0..n-1]) && items(A) = items(A0) &&
         * (A[n..] = A0[n..])
         */
    }

}
```

Aug 20, 12 7:55 **Sorting.java** Page 1/1

```
package sorting;
public interface Sorting {

    public void sort(int[] A, int n, SortStatistics stats);

}
```

Aug 04, 14 9:21 **SortStatistics.java** Page 1/1

```
package sorting;

public class SortStatistics {
        int size;
        long moveCount;
        long compareCount;
        long startTime;
        long executionTime;

        public SortStatistics() {
            super();
            reset();
        }
        public void setSize( int n ) {
            size = n;
        }
        public void incrMove() {
            moveCount++;
        }
        public void incrCompare() {
            compareCount++;
        }
        public void startTime() {
            startTime = System.nanoTime();
        }
        public void finishTime() {
            executionTime = System.nanoTime() - startTime;
        }
        public void reset() {
            moveCount = 0;
            compareCount = 0;
            startTime = 0;
            executionTime = 0;
        }
        public void printStats( String sortType ) {
            System.out.printf( sortType + " sort for %,6d takes %,14d compares %,14d moves %8
d milliseconds%n",
                    size, compareCount, moveCount, executionTime/1000000 );
        }
}
```

```
package sorting;

public class SortTest {

    int A[];

    private SortTest(int n) {
        A = new int[n];
    }

    private void inOrder(int n) {
        for (int i = 0; i < n; i++) {
            A[i] = i;
        }
    }

    private void reverseOrder(int n) {
        for (int i = 0; i < n; i++) {
            A[i] = (n - 1) - i;
        }
    }

    private SortStatistics runTest(Sorting sort, int n, String sortType ) {
        SortStatistics stats = new SortStatistics();
        stats.setSize(n);
        stats.startTime();
        sort.sort(A, n, stats);
        stats.finishTime();
        for (int i = 0; i < n; i++) {
            if (A[i] != i) {
                System.out.println("Sort failed at " + i);
            }
        }
        stats.printStats( sortType );
        return stats;
    }

    private void inOrderTest(Sorting sort, int n) {
        inOrder(n);
        SortStatistics stats = runTest(sort, n, "In order" );
    }

    private void reverseOrderTest(Sorting sort, int n) {
        reverseOrder(n);
        SortStatistics stats = runTest(sort, n, "Reverse order" );
    }

    private void allSizes(Sorting sort, int baseSize) {
        inOrderTest(sort, 1*baseSize);
        inOrderTest(sort, 2*baseSize);
        inOrderTest(sort, 4*baseSize);
        inOrderTest(sort, 8*baseSize);
        reverseOrderTest(sort, 1*baseSize);
        reverseOrderTest(sort, 2*baseSize);
        reverseOrderTest(sort, 4*baseSize);
        reverseOrderTest(sort, 8*baseSize);
    }

    public static void main(String args[]) throws java.lang.Exception {
        int baseSize =10000;
        SortTest test = new SortTest(16*baseSize);
        System.out.println("Tests for quick sort:");
```



```
        test.allSizes(new QuickSort(), baseSize);
        System.out.println();
        System.out.println("Tests for quick sort:");
        test.allSizes(new QuickSort(), baseSize);
        System.out.println();
        System.out.println("Tests for insertion sort:");
        test.allSizes(new InsertionSort(), baseSize);
        System.out.println();
        System.out.println("Tests for merge sort:");
        test.allSizes(new MergeSort(), baseSize);
    }
}
```