ANALYSIS AND COMPARISON OF SORTING ALGORITHMS--A NEW APPROACH

UMESH NAIR

ABSTRACT. This paper, prepared for the CSE532 course in OGI, discusses the implementation of a test bed for sorting algorithms and introduces a method by which different sorting algorithms can be compared using statistical techniques.

1. INTRODUCTION

Internal sorting is one of the main fields of interests in the theory of algorithms ever since the first digital computer came into existence. For a long time, the lower limit of time complexity for sorting algorithms was proportional to the square of the number of records. The \( N^2 \) barrier was beaten in late sixties, and many new innovative sorting algorithms were discovered.

It has been proved that the lower bound for any comparison-based sorting algorithm is \( O(N \ln N) \), even though only a handful of algorithms really achieve this. Even when a particular algorithm has a worst-case or average complexity of \( O(N \ln N) \), it may be found slower than many comparatively “inefficient” algorithms for all practical ranges of data. The analysis based on purely theoretical grounds and misleading big-oh notation has made it difficult to compare sorting algorithms for practical purposes.

This work gives emphasis to measure the complexity of sorting algorithms by running the algorithms on different sets of practical data, and counting the various resources consumed, thereby providing a reliable comparison of various sorting algorithms for practical purposes. An object-oriented framework is developed to test and measure various sorting algorithms. Ten of the most popular comparison-based sorting algorithms (including some variants) are implemented for illustration. Data has been collected for various values of \( N \) (10, 25, 50, 75, 100, 250, 500, 750, 1000, 2500, 5000, 7500, 10000, 25000 and 50000) and for different types of input (Random, Sorted, Reverse Sorted, Almost Sorted) to cover both average and worst-case scenarios and tabulated. The results are compared with already established results, to validate the authenticity of this approach. Various conclusions are drawn based on these observations.

This work should be treated as the starting point for a larger project emphasizing the analysis of sorting algorithms based on experimental data. Due to the limited scope of this project, certain assumptions are taken to ease the experiments and analysis, and they require more accurate treatment in a more serious project.

Date: June 15, 2001.
However, the framework and the tools developed for this project is adequate for conducting a more robust and complete analysis.

2. Scope

Only comparison-based algorithms are considered in this work. The generalized framework provides a mechanism to hook up any comparison-based sorting algorithm to the testbed. The purpose of this paper is to illustrate a technique and introduce the tools that can be used to analyze sorting algorithms. Since this was a short-term project, detailed experimentation was not done, and is left to future researchers.

3. Design of the Testbed Framework

The list of classes and interfaces is given below.

3.1. Overview.

3.1.1. Language used. Java is used to implement these tools. The following list shows the classes in the system.

3.1.2. Classes and interfaces. Figure 1 shows the class hierarchy of the system. It defines only one interface, DataElement, implemented by the TestDataElement class. Details of each class is described in section 3.2.

Each component is described below.

3.2. Utility classes.

3.2.1. Interface DataElement. An interface DataElement is defined for using data with the testbed. A DataElement is an entry in the array. It consists of a key and any amount of satellite data. The classes implementing this interface should define the following methods:

1. public boolean isLessThan(DataElement other) : Returns true if the current element is less than other (by comparing the keys appropriately). Note that defining less than operator is sufficient to define the order completely. This idea is borrowed from the Standard C++ library[1].
2. public boolean wasOriginallyBefore(DataElement other) : This is used to determine whether the sort was stable or not. It is used by the testbed to compare satellite data also and determine whether two elements are in the original order after sorting. Return true if this is irrelevant.
3. public DataElement makeCopy() : Creates a copy of the element and returns it.
3.2.2. Class DataArray. A DataArray object encapsulates the array (input and output of sorting) that holds data elements. In addition to holding the elements, it keeps track of various statistics for the array. These statistics include

1. Number of data reads
2. Number of data writes
3. Number of new element allocations
4. Number of comparisons
5. Number of swaps

Not all of these statistics are used in the present work. However, these are provided in order to give the future users of this library the ability to use those measurements.
The following are the important methods of this class. Function of most of the methods is clear from the name.

(1) Constructors:
   public DataArray();
   public DataArray(Vector vec);

(2) Methods to manipulate individual elements:
   public void set(int i, DataElement element);
   public DataElement get(int i);
   public boolean insert(int i, DataElement element);
   • Inserts an element before the specified location. It is an expensive operation, pushing all the remaining elements one space towards the end.
   public void append(DataElement element);

(3) Methods to compare/manipulate two elements:
   public boolean areAscending(int i, int j);
   • Equivalent to this[i] < this[j]
   public boolean areNonDescending(int i, int j);
   • Equivalent to this[i] <= this[j]
   public boolean areInOriginalSequence(int i, int j);
   • Used to check whether a sort was table. See the description of TestDataElement for details.
   public void swapElements(int i, int j);

(4) Methods to manipulate the entire array:
   public void reverse();

(5) Methods to check the status of the array:
   public boolean isSorted();
   public boolean isStableSorted();
   • Returns true if a conflict is found, false otherwise. A true return value doesn’t guarantee that the sort was stable. It may return true if there were no pair of elements with equal keys. This method makes sense only if the corresponding DataElement object returns the right value by examining the satellite data.
   public int getStableSortStatus();
   • Returns an integer specifying whether the sort was stable, not stable or could not be determined (i.e., there were no duplicate elements). This method makes sense only if the corresponding DataElement object returns the right value by examining the satellite data.
   public boolean isReverseSorted();

(6) Methods to get various statistics:
   public int size();
   public int nAllocations();
   public int nAccesses();
   public int nSwaps();
   public int nComparisons();
   public int nAssigns();

(7) Miscellaneous methods:
   public String toString();
   public void resetCounters();
public DataArray makeCopy();

3.2.3. Class SortingAlgorithm. This is the abstract base class of all sorting algorithms. All derived classes should implement the following method.

public abstract DataArray sort(DataArray inputData);

- Takes an input DataArray, and sorts it into another array and returns it. The input array is not affected. A derived class, InPlaceSortingAlgorithm handles in-place sorting.

public SortingAlgorithm(String name);

- Constructor. Assigns a name to the algorithm.

public DataArray sortAndReport(DataArray inputData, String title);

- Calls sort(), collects statistics about the input and output arrays, and reports the details.

3.2.4. Class InPlaceSortingAlgorithm (derived from SortingAlgorithm). This abstract class declares an abstract method

public abstract void sortInPlace(DataArray data);

which sorts the input array in place, and overrides SortingAlgorithm.sort() as follows:

public DataArray sort(DataArray inputData) {
    DataArray outputData = inputData.makeCopy();
    sortInPlace(outputData);
    return outputData;
}

thereby allowing in-place sorting algorithms to provide both sort() and sortInPlace() functions.

All sorting algorithms considered in this project - except MergeSort, which is a direct subclass of SortingAlgorithm - are subclasses of this class.

3.3. Sorting Algorithms. Various sorting algorithms implemented are listed here. For brevity and clarity, only the sortInPlace() method (sort() for MergeSort) is listed, which should be self-explanatory. All these algorithms are adopted from [2], with appropriate modifications suggested in [3].

3.3.1. Class SimpleBubbleSort. Sorting by exchanging adjacent elements. It is known to give same performance in all cases, so only the next variation is considered for experiment.

public void sortInPlace(DataArray data) {
    int upperBound = data.size();
    while (--upperBound > 0) {
        for (int j=0; j<upperBound ; ++j) {
if(data.areAscending(j+1, j)) {
    data.swapElements(j, j+1);
}
}
}

3.3.2. Class SmartBubbleSort. Modification of bubble sort, detecting the end early. This variant is used in the experiments.

    public void sortInPlace(DataArray data) {
        int upperBound = data.size() - 1;
        while (upperBound > 0) {
            int lastElementSwapped = 0;
            for (int j=0; j<upperBound ; ++j) {
                if (data.areAscending(j+1, j)) {
                    data.swapElements(j, j+1);
                    lastElementSwapped = j;
                }
            }
            upperBound = lastElementSwapped;
        }
    }

3.3.3. Class InsertionSort. Inserts each element at the right place.

    public void sortInPlace(DataArray data) {
        for (int i=1; i<data.size(); ++i) {
            DataElement value = data.get(i);
            int j = i;
            DataElement tempValue;
            while ((j > 0) &&
                    (value.isLessThan(tempValue = data.get(j-1)))) {
                data.set(j, tempValue);
                --j;
            }
            data.set(j, value);
        }
    }

3.3.4. Class SelectionSort. Finds the minimum element, and puts it in the right place.

    public void sortInPlace(DataArray data) {
        for (int i=0; i<data.size(); ++i) {
            DataElement currElem = data.get(i);
            DataElement minimum = currElem;
            int minimumIndex = i;
            for (int j=i+1; j<data.size(); ++j) {
                DataElement tempElem = data.get(j);
                if (tempElem.isLessThan(minimum)) {
                    minimum = tempElem;
                    minimumIndex = j;
                }
            }
            data.set(i, minimum);
        }
    }
ANALYSIS AND COMPARISON OF SORTING ALGORITHMS — A NEW APPROACH

DataElement elem = data.get(j);
if (elem.isLessThan(minimum)) {
    minimum = elem;
    minimumIndex = j;
}
}
if (minimum != currElem) {
    data.swapElements(i, minimumIndex);
}
}

3.3.5. Class HeapSort. Since heap sort involves the manipulation of the heap data structure, and the heap operations are more straightforward if the array index starts at 1 instead of 0, a separate class Heap is implemented to do the heap sort. The full source for heap sort is given below.

public class HeapSort extends InPlaceSortingAlgorithm {
    public HeapSort() {
        super("Heap Sort");
    }
    public void sortInPlace(DataArray data) {
        new Heap(data).doHeapSort();
    }
}

class Heap {
    private DataArray d_data;
    private int d_heapSize;
    public Heap(DataArray data) {
        d_data = data;
        d_heapSize = data.size();
    }
    public void doHeapSort() {
        buildHeap();
        for (int i=d_data.size(); i>1; --i) {
            swapElements(1, i);
            --d_heapSize;
            heapify(i);
        }
    }
    private void buildHeap() {
        d_heapSize = d_data.size();
        for (int i=d_data.size()/2; i>0; --i) {
            heapify(i);
        }
    }
    private void swapElements(int i, int j) {
        DataElement temp = d_data.get(i);
        d_data.set(i, d_data.get(j));
        d_data.set(j, temp);
    }
    private void heapify(int i) {
        int l = left(i);
        int r = right(i);
        int largest = i;
        if (l < d_heapSize && d_data.get(l).isGreaterThan(d_data.get(largest))) {
            largest = l;
        }
        if (r < d_heapSize && d_data.get(r).isGreaterThan(d_data.get(largest))) {
            largest = r;
        }
        if (largest != i) {
            swapElements(i, largest);
            heapify(largest);
        }
    }
    private int left(int i) {
        return 2*i;
    }
    private int right(int i) {
        return 2*i + 1;
    }
}

3.4.2. Class MedianSort. Since median sort is generally more efficient than the other sorting algorithms, it can be implemented in a different way. The class MedianSort is implemented as follows:

public class MedianSort extends InPlaceSortingAlgorithm {
    public MedianSort() {
        super("Median Sort");
    }
    public void sortInPlace(DataArray data) {
        new Median(data).doMedianSort();
    }
}

class Median {
    private DataArray d_data;
    private int d_medianIndex;
    public Median(DataArray data) {
        d_data = data;
        d_medianIndex = data.size() / 2;
    }
    public void doMedianSort() {
        // Median sort algorithm
    }
}

3.4.3. Class QuickSort. Quick sort is a divide-and-conquer algorithm that works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

public class QuickSort extends InPlaceSortingAlgorithm {
    public QuickSort() {
        super("Quick Sort");
    }
    public void sortInPlace(DataArray data) {
        new Quick(data).doQuickSort();
    }
}

class Quick {
    private DataArray d_data;
    public Quick(DataArray data) {
        d_data = data;
    }
    public void doQuickSort() {
        quickSort(d_data, 0, d_data.size()-1);
    }
    private void quickSort(DataArray data, int left, int right) {
        int i = left;
        int j = right;
        int pivot = data.get((left + right) / 2);
        while (i <= j) {
            while (data.get(i).isLessThan(pivot)) {
                i++;
            }
            while (data.get(j).isGreaterThan(pivot)) {
                j--;
            }
            if (i <= j) {
                swapElements(i, j);
                i++;
                j--;
            }
        }
        if (left < j) {
            quickSort(data, left, j-1);
        }
        if (i < right) {
            quickSort(data, i, right);
        }
    }
    private void swapElements(int i, int j) {
        DataElement temp = d_data.get(i);
        d_data.set(i, d_data.get(j));
        d_data.set(j, temp);
    }
}
private void heapify(int i) {
    int l = left(i);
    int r = right(i);
    int largest;
    if (l <= d_heapSize && areAscending(i, l)) {
        largest = l;
    } else {
        largest = i;
    }
    if (r <= d_heapSize && areAscending(largest, r)) {
        largest = r;
    }
    if (largest != i) {
        swapElements(i, largest);
        heapify(largest);
    }
}

private static int parent(int i) {
    return i / 2;
}

private static int left(int i) {
    return 2 * i;
}

private static int right(int i) {
    return 2 * i + 1;
}

private boolean areAscending(int i, int j) {
    return d_data.areAscending(i - 1, j - 1);
}

private boolean areNonDescending(int i, int j) {
    return d_data.areNonDescending(i - 1, j - 1);
}

void swapElements(int i, int j) {
    d_data.swapElements(i - 1, j - 1);
}

3.3.6. Class QuickSort. Based on a recursive partition algorithm.

public void sortInPlace(DataArray data) {
    sortPartition(data, 0, data.size() - 1);
}

/**
 * Sorts the partition from element first to element (last-1)
 */
protected void sortPartition(DataArray data, int first, int last) {
if (first < last) {
    int pivot = doPartition(data, first, last);
    sortPartition(data, first, pivot);
    sortPartition(data, pivot+1, last);
}
}

protected int doPartition(DataArray data, int first, int last) {
    DataElement value = data.get(first);
    int i = first-1;
    int j = last+1;
    for(;;) {
        while (value.isLessThan(data.get(--j)));
        while (data.get(++i).isLessThan(value));
        if (i < j) {
            data.swapElements(i, j);
        } else {
            return j;
        }
    }
}

3.3.7. Class RandomizedQuickSort (derived from QuickSort). This variant of QuickSort attempts to improve the $N^2$ worst-case complexity of Quick Sort by selecting the pivot randomly at each pass.

Here is the overridden sortPartition() method of QuickSort.

protected void sortPartition(DataArray data, int first, int last) {
    if (first < last) {
        int randomIndex = d_random.nextInt() % (last - first + 1);
        if (randomIndex < 0) {
            randomIndex = -randomIndex;
        }
        if (0 != randomIndex) {
            data.swapElements(first, first+randomIndex);
        }
        int pivot = doPartition(data, first, last);
        sortPartition(data, first, pivot);
        sortPartition(data, pivot+1, last);
    }
}

4. Implementation of a Testbed

Using the testbed framework described in previous sections, a sample testbed is implemented and used to analyze the sorting algorithms considered. This section
gives guidelines for using the testbed framework. The testbed implemented may be used to test sorting algorithms without modifications generally.

4.1. **Representation of test data.** A class `TestDataElement`, derived from `DataElement`, is used to represent one data element. The class is defined as follows:

```java
public class TestDataElement implements DataElement {
    // Static counter
    private static long lastCounter = 0;
    // Key
    private int d_key;
    // Satellite Data
    private int d_copyOfKey;
    private long d_counter;
    ...
    ...
};
```

The member variable `d_key` is an integer that acts as the key for the record. The array is sorted based on this key. The variable `d_copyOfKey` is a copy of `d_key`. This is used to verify whether the satellite data is copied properly. The variable `d_counter` is used to keep track of the original order for checking the stability of sort. Using the static variable `lastCounter`, every object is assigned a new value of `d_counter`, which is one more than the value of the previous object.

The constructor takes a key value and constructs a `TestDataElement` object as follows:

```java
public TestDataElement(int key) {
    d_key = d_copyOfKey = key;
    d_counter = ++lastCounter;
}
```

The `isLessThan()` method, used for sorting and also by methods `isSorted()`, `isReverseSorted()` etc. of the `DataArray` class, is defined as follows:

```java
public boolean isLessThan(DataElement element) {
    if (element instanceof TestDataElement) {
        return (d_key < ((TestDataElement)element).d_key);
    } else {
        throw new IncompatibleDataException("TestDataElement expected");
    }
}
```

The `wasOriginallyBefore()` method, used to check whether a sort is stable, is defined by comparing the `d_counter` variables, as follows:

```java
public boolean wasOriginallyBefore(DataElement element) {
    if (element instanceof TestDataElement) {
```
return (d_counter < ((TestDataElement)element).d_counter);
} else {
    throw new IncompatibleDataException("TestDataElement expected");
}
}

The class has a few other trivial member functions.

4.2. Testing Methodology. A DataArray object consisting of several TestDataElement objects is given to a sorting algorithm as input. The sort() method of the respective SortingAlgorithm class returns a new DataArray object containing the sorted elements. After the sorting operation, both the input and output DataArray objects have the statistics counters set during the sort procedure. (The copy operation in the InPlaceSortingAlgorithm.sort() before calling sortInPlace() is not considered for gathering statistics.) This data is recorded for analysis.

For example, here is an example of output using the input array 76, 13, 27, 18, 100, 814, 25, 18, 1000, 325, 18, 18, 100, 74, 13, using a few sorting algorithms considered.

DATA DESCRIPTION : Sample data
DATA SIZE : 15
SORTING ALGORITHM: Smart Bubble sort
Did it sort? : Yes
Stable sort? : No
# of accesses : 0 + 102 = 102
# of comparisons : 0 + 105 = 105
# of swaps : 0 + 51 = 51
# of allocations : 0 + 0 = 0
# of assignments : 0 + 102 = 102
TIME COMPLEXITY = 204
SPACE COMPLEXITY = 0

DATA DESCRIPTION : Sample data
DATA SIZE : 15
SORTING ALGORITHM: Insertion Sort
Did it sort? : Yes
Stable sort? : No
# of accesses : 0 + 78 = 78
# of comparisons : 0 + 0 = 0
# of swaps : 0 + 0 = 0
# of allocations : 0 + 0 = 0
# of assignments : 0 + 65 = 65
TIME COMPLEXITY = 143
SPACE COMPLEXITY = 0

DATA DESCRIPTION : Sample data
DATA SIZE : 15
SORTING ALGORITHM: Selection Sort
Did it sort? : Yes
Stable sort? : No
# of accesses : 0 + 142 = 142
# of comparisons : 0 + 0 = 0
# of swaps : 0 + 11 = 11
# of allocations : 0 + 0 = 0
# of assignments : 0 + 22 = 22
TIME COMPLEXITY = 164
SPACE COMPLEXITY = 0

DATA DESCRIPTION : Sample data
DATA SIZE : 15
SORTING ALGORITHM: Quick Sort
Did it sort? : Yes
Stable sort? : No
# of accesses : 0 + 144 = 144
# of comparisons : 0 + 0 = 0
# of swaps : 0 + 18 = 18
# of allocations : 0 + 0 = 0
# of assignments : 0 + 36 = 36
TIME COMPLEXITY = 180
SPACE COMPLEXITY = 0

DATA DESCRIPTION : Sample data
DATA SIZE : 15
SORTING ALGORITHM: Randomized Quick Sort
Did it sort? : Yes
Stable sort? : No
# of accesses : 0 + 176 = 176
# of comparisons : 0 + 0 = 0
# of swaps : 0 + 29 = 29
# of allocations : 0 + 0 = 0
# of assignments : 0 + 58 = 58
TIME COMPLEXITY = 234
SPACE COMPLEXITY = 0

DATA DESCRIPTION : Sample data
DATA SIZE : 15
SORTING ALGORITHM: Merge Sort
Did it sort? : Yes
Stable sort? : Yes
# of accesses : 15 + 73 = 88
# of comparisons : 0 + 0 = 0
# of swaps : 0 + 0 = 0
# of allocations : 0 + 74 = 74
# of assignments : 0 + 0 = 0
TIME COMPLEXITY = 88
SPACE COMPLEXITY = 74

DATA DESCRIPTION : Sample data
DATA SIZE : 15
SORTING ALGORITHM: Heap Sort
Did it sort? : Yes
Stable sort? : No
# of accesses : 0 + 92 = 92
# of comparisons : 0 + 73 = 73
# of swaps : 0 + 46 = 46
# of allocations : 0 + 0 = 0
# of assignments : 0 + 92 = 92
TIME COMPLEXITY = 184
SPACE COMPLEXITY = 0

In order to test the functionality and the complexity, the experiment should be repeated for different input sizes and different type of input data for all the sorting algorithms. In order to do this, an abstract class SortTester is defined. It is described below:

4.2.1. Class SortTester. A SortTester object conducts tests for a particular type of input data of arbitrary size (provided by the user as a command line argument). It keeps two member variables - d_inputData and d_outputData - to hold the input and output DataArray objects. The method testSorts() runs various sorts. The default implementation in the base class is as follows:

```java
public void testSorts() {
    SortingAlgorithm[] algorithms = {
        new SimpleBubbleSort(),
        new SmartBubbleSort(),
        new InsertionSort(),
        new SelectionSort(),
        new QuickSort(),
        new RandomizedQuickSort(),
        new MergeSort(),
        new HeapSort(),
        new ShellSort()
    };
    prepareInputData();
    for (int i=0; i<algorithms.length; ++i) {
        algorithms[i].sortAndReport(d_inputData, typeOfData());
    }
}
```
The method `prepareInputData()` is an abstract method, to be overridden by the child classes. It is defined as follows:

```java
protected abstract void prepareInputData();
```

It also provides the following protected method for subclasses to add data from `preparedData()` method:

```java
protected void addData(int x) {
    d_inputData.append(new TestDataElement(x));
}
```

To consider one type of test data, the `SortTester` class is subclassed and `prepareInputData()` is implemented. There are several subclasses of `SortTester` implemented for the project. They are described below:

4.2.2. **Class SampleSortTester.** A simple class to test the basic functionality. It generates 15 elements, with duplicates. The following is its implementation of `prepareInputData()`:

```java
protected void prepareInputData() {
    int[] values = { 76, 13, 27, 18, 100, 814, 25, 18,
                    1000, 325, 18, 18, 100, 74, 13 };
    for (int i=0; i<values.length; ++i) {
        addData(values[i]);
    }
}
```

4.2.3. **Class RandomSortTester.** Takes the number of elements (and an optional seed value) as a parameter to the constructor, and generates those many elements containing random integers. The Java library class `Random` and the member function `nextInt()` are used to generate the next random integer.

```java
protected void prepareInputData() {
    for (long i=0; i<d_nElements; ++i) {
        addData(d_random.nextInt());
    }
}
```

4.2.4. **Class SortedSortTester.** Takes the number of elements (and an optional seed value) as a parameter to the constructor, and generates those many elements containing sorted integers. It does that by generating those many random numbers, and sorting them using the `QuickSort` class.

```java
protected void prepareInputData() {
    for (long i=0; i<d_nElements; ++i) {
        addData(d_random.nextInt());
    }
    // Sorts the data, assuming our QuickSort works fine
    new QuickSort().sortInPlace(d_inputData);
```
if (!d_inputData.isSorted()) {
    throw new UndefinedOrderException();
}

4.2.5. Class ReverseSortedSortTester. Takes the number of elements (and an optional seed value) as a parameter to the constructor, and generates those many elements containing integers sorted in the descending order. It does that by generating those many random numbers, and sorting them using the QuickSort class, and calling DataArray.reverse().

protected void prepareInputData() {
    for (long i=0; i<d_nElements; ++i) {
        addData(d_random.nextInt());
    }

    // Sorts the data, assuming our QuickSort works fine
    new QuickSort().sortInPlace(d_inputData);
    d_inputData.reverse();
    if (!d_inputData.isReverseSorted()) {
        throw new UndefinedOrderException();
    }
    d_inputData.resetCounters();
}

4.2.6. Class AlmostSortedSortTester. Takes the number of elements (and an optional seed value) as a parameter to the constructor, and generates those many elements containing almost integers. The caller can specify a percentage value, by which the sorting order should be disturbed. It does that by generating those many random numbers, and sorting them using the QuickSort class, and then doing the specified amount of random swaps. In the project, an AlmostSortedSortTester object with 10% noise was used, but didn't yield any significant results, and hence excluded from this report. It will be interesting to try different levels of noise and study the variation.

5. Experiments and statistical analysis

Using the framework and tools described in previous sections, eight different testing algorithms were tested using three type of input test data (Using AlmostSortedSortTester didn't give any more information than SortedSortTester) covering the average, best and worst cases of most of the algorithms. The big-Oh notation is of great theoretical importance, but it is practically almost impossible to arrive at the most suitable big-Oh expression from experimental data. Hence, a new approach was experimented in this work. It is described below. The algorithm is executed for different input data size.

(1) The metrics (total reads + total writes) is taken as a measure for running time. Database gurus [4] consider 15 milliseconds for a read or write (assuming that read/write happens on raw data, it is fair to assume that
they take the same time, but it may be different when disk pages and cache are involved), and 100 nanoseconds for comparing a key value with a value in memory. Internal sorting also can assume similar values. Since reads/writes are the most expensive operations with a considerably large record size, only these are considered to contribute to the running time, and all constant time operations are ignored. For a comprehensive theoretical analysis, experimental data and comparison of sorting algorithms, refer to [3].

(2) Using least-square linear regression technique, the following equation is obtained from the data:

\[ T = aN^2 + bN \ln N + cN + d \]

where
- \( T \) = Running time (data obtained as the sum of reads and writes)
- \( N \) = size of the array

(3) The coefficients \( a, b, c \) and \( d \) are obtained for each case. By inspecting these values, the behavior is studied. For example, if \( a \) is dominating, we can conclude that the algorithm is more towards an \( N^2 \) algorithm.

The derivation of the least square equations is given in the appendix. SquareMatrix is a class written for solving simultaneous linear equations, implementing Gauss-Jordan elimination.

The appendix contains the following:

1. Source listing of SquareMatrix class.
2. Derivation of the least square equations.
3. Data used (with some charts)
4. The following algorithm is used for the statistical analysis.

```java
public static double[] analyze(double[] yValues) {
    int n = yValues.length;
    double[] coeff = new double[16];
    double[] values = new double[4];
    for (int i=0; i<n; ++i) {
        double x = (double)xValues[i];
        double y = (double)yValues[i];
        double logX = Math.log(x);
        double x2 = x * x;
        double x3 = x2 * x;
        double x4 = x2 * x2;
        double xLogX = x * logX;
        double x2LogX2 = xLogX * xLogX;
        double x2LogX = x * xLogX;
        double x3LogX = x2LogX * x;
        coeff[0] += x4;
        coeff[1] += x3LogX;
        coeff[2] += x3;
        coeff[3] += x2;
    }
    // Further processing...
}
```
values[0] += (y * x2);
coeff[4] += x3LogX;
coeff[5] += x2LogX2;
coeff[6] += x2LogX;
coeff[7] += xLogX;
values[1] += (y * xLogX);
coeff[8] += x3;
coeff[9] += x2LogX;
coeff[10] += x2;
values[2] += (y * x);
coeff[12] += x2;
coeff[13] += xLogX;
coeff[14] += x;
coeff[15] += 1.0;
values[3] += y;
}
SquareMatrix coeffMatrix = new SquareMatrix(4, coeff);
return coeffMatrix.solve(values);

Noted expert in the theory of algorithms observes [5] that one of the most common mistakes in experimental analysis of algorithms is “Misusing statistical tools”. This is true in general, and this work doesn’t boast a reliable technique to analyze algorithms. However, from the results obtained, it can be concluded that statistical techniques can be used as the starting point for many similar analysis, especially when theoretical analysis can be very cumbersome.

5.1. Comparison of sorting algorithms using least-square fitting. Tables 1, 2, 3 list the equations obtained from least-square fitting for various sorting methods using random data, sorted data and reverse-sorted data. These results remarkably illustrate the following already established facts. (The big-Oh notation is deliberately not used, as these results doesn’t imply anything for Big-Oh)

(1) MergeSort, HeapSort and Randomized QuickSort (in almost all cases) give $N\ln N$ performance in all cases.
(2) Quicksort give $N\ln N$ performance for random data, but $N^2$ performance for sorted and reverse sorted data.
(3) Bubble sort, insertion sort and selection sort give $N^2$ performance in general. Bubble sort gives $N$ performance on sorted data.
(4) The effect of randomization in quicksort is not much for random data, but it is quite obvious for sorted and reverse sorted data.

These results conclude that this technique of using statistical analysis can be used as an effective tool for analysis of algorithms.
6. Future Scope

The testbed and the framework developed can be used for further experimental analysis of sorting algorithms. A few variants are considered in this work. However, it is not comprehensive. The following are worth considering.

1. Analysis of Shell sort, with different strategies of choosing the increments.
2. Analysis of quicksort, studying a suggestion by R.C. Singleton (see [3], 5.2.2.), which suggests to use equality also when determining the exchanges. This will make the algorithm efficient when there are a lot of records with the same key.
3. Analysis of hybrid sort algorithms. For example, Sedgewick suggests (see [3], 5.2.2.) that using insertion sort in the subsets of a quicksorted array when the number of elements is less than a particular value M, will be much...
faster than the original quicksort. It will be an interesting experiment to study the effect of this parameter $M$ on running time.

7. Conclusions

In this work, an effective and convenient framework for testing sorting algorithm is discussed and implemented. A statistical technique is used to compare the performance of the algorithms. Eight sorting algorithms are analyzed using this technique, and the results are very much in line with the already established results.
References