Sorting Algorithms

Project Report

student Bogdan Kovch

e-mail: kovch@pdx.edu

class: CS 350

Overview of Sorting Algorithms

Three sorting algorithms, insertion sort, quick sort and merge *sort* were selected for analysis within the scope of this project. Why particularly these algorithms were selected for the research? They are conceptually different from each other in their approach to solving the problem, complexity, memory utilization, time and space efficiency.

- ***Insertion sort*** is a simple compact algorithm which doesn't require extra memory, but does a lot of memory shift operations within the array. Its time efficiency varies between the best-case linear $O(n)$ for already sorted arrays and the average-case and worst-case quadratic $O(n^2)$ efficiency for random and sorted in descending order arrays respectively. This is a typical range of efficiencies covered by the majority of known sorting algorithms. It demonstrates efficiency close to its linear best-case efficiency for sorted or almost sorted arrays. It is also good for small arrays.
- ***Quick sort*** using Hoare partitioning is an algorithm of higher complexity. It partitions the array and swaps elements within this array without allocating any additional memory for its job. However it uses recursion technique uses memory stack when executed on a computer. Its average-case and best-case efficiencies are linear-logarithmic $O(n \log n)$, and its worst-case efficiency is quadratic $O(n^2)$ for arrays already sorted either in ascending or descending order. This makes the quick sort algorithm to be very efficient for sorting big completely unsorted arrays.
- ***Merge sort*** recursively halves an array and copies its partitions into new arrays allocated in memory. Then it goes the opposite way and recursively merges these arrays back into one bigger array by placing their elements in sorted order. This algorithm heavily uses memory in order to keep multiple copies of array partitions. This becomes a critical problem when this algorithm attempts to sort a big array on a computer which has very limited amount of fast memory. Its efficiency class is the same for best-, average- and worst-case efficiencies which is linear-logarithmic $O(n \log n)$.

All the algorithms discussed and implemented in this project are taken from book *Design and Analysis of Algorithms* by Ananiy Levitin. Translated into the program code, these algorithms look almost identical to those in the textbook. We will return to these algorithms later to discuss how the program written for this research helped us to test and analyze their behavior.

Program Description

The program written for this project aimed to serve as an efficiency analysis tool for algorithms implemented. This program analyzes one of the sorting algorithms selected by the user via the program's command line arguments. The program consists of two major parts. The first part is responsible for generating test cases for algorithms and their efficiency analysis. The second part is a collection of search algorithms with built in statistics gathering code. Both parts are more or less independent, so new algorithms can be added easily if needed.

Generating Test Cases

The best, average and worst efficiency cases for the sorting algorithms implemented can be tested by generating three types of test cases. These test cases are sequences of elements stored in random, descending and ascending orders. For every algorithm, the program automatically generates a set of sequences for every of these three test cases.

- ***Random*** order sequences of integers are generated for the first test case using a random number generator. This way, every generated array with random numbers is different from the another array with random numbers generated for this test case. This test case is the best way to test the average-case efficiency for all sorting algorithms.
- ***Descending*** order sequences of integers are generated by simply decreasing the value of every next element of the sequence by one. A series of such data sequences form the second test case. This test case is used to make the algorithms implemented to demonstrate their worst-case efficiency.
- ***Ascending*** order sequences of integers are generated by simply increasing by one the value of every consecutive element in the array. A series of such sequences form the third test case. As we will see from program analysis, some algorithms demonstrate their best-case efficiency while others demonstrate their worst-case efficiency.

A singe test case represents a series of arrays of integers in which every consequent array length grows exponentially. For example, for base 10, the length of arrays in a single test case will be $10^1=10,\ \ 10^2=100$, $10^3=1000$, etc. For base 2, the arrays of length 2, 4, 8, 16, 32, etc. will be generated. The base for the array size can be defined by the user via the second command line argument of the program. If it is not defined, it is 10 by default.

The number of sequences generated for every test case depends on the array length base value, time efficiency demonstrated by the test case, or by a handled out of memory exception. The program would generate big sequences faster for big base values and therefore will sooner feed the big array into a sorting algorithm. If the efficiency being demonstrated by the algorithm for a test case is quadratic, than big arrays would slow down the program execution sooner than the linear-logarithmic and linear test cases. This will make the program stop from generating new arrays and then finish the test cases. By default, if test case took more than 10 seconds, the program would finish generating new array sequences for this test case and stop after the last array was sorted. For a test case, approximately 6-8 base-10 length arrays or 16-29 base-2 length arrays will be generated.

Results Validation

The generated arrays are passed into an algorithm for sorting. The algorithm sorts the sequence by updating the elements in the array provided. Once sorting is finished, the program passes the array to the method which checks whether the array is actually sorted. In case if the algorithm was implemented with mistakes, and didn't sort the array, the program would display a message about the problem. If no problems are detected and the array was sorted correctly, then no other message is displayed.

Statistics Gathering

As mentioned above, every algorithm implemented has built in routines for gathering statistics about algorithm execution. This statistics collects the following information:

- number of elements in the array
- number of comparisons of array element
- number of memory operations manipulating with array elements
- total time taken to sort this array

Statistics collected while sorting individual arrays of one test case is stored into a list which thereby stores all statistics about one test case. After the test case finishes, its statistics is displayed in the form of table which shows the statistics gathered. The first four columns correspond to those four statistics fields mentioned above plus one column which shows computed average time taken to process one element of the array.

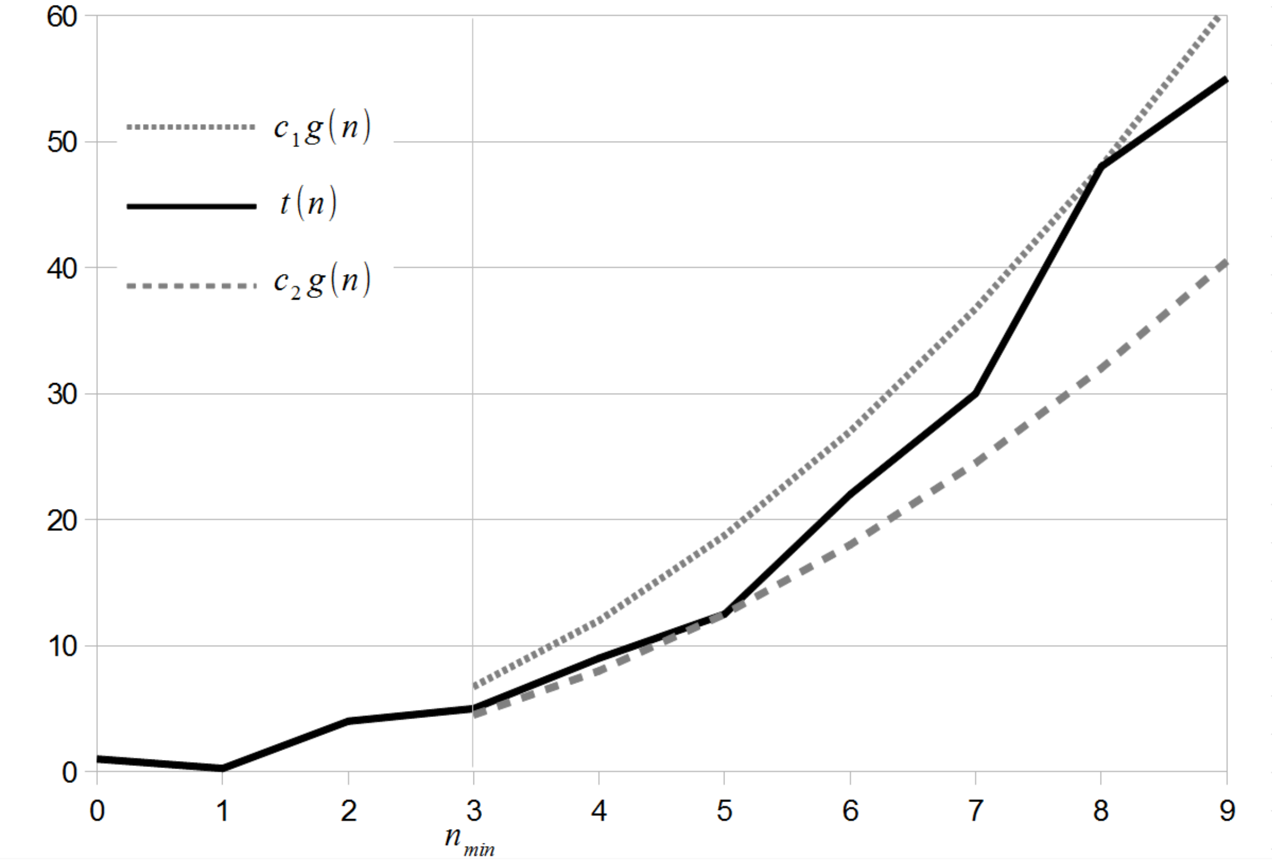
The number of comparisons and memory operations describe the algorithm efficiency in terms of its basic operations. These numbers are not affected by other processes executed in parallel in the operating system or hardware performance characteristics. Total time and time per element describe real behavior of an algorithm implemented in a program being executed which shares hardware resources with other processes in the operating system.

Statistics Processing

After test case statistics is collected, it is matched to different mathematical models describing possible efficiency classes in method *matchEfficiencyClass*. For every efficiency class model, the program computes percentage how well this statistics matches a particular efficiency class. The efficiency model which matched the given statistics with the highest percentage is then selected to describe the behavior of the given test case efficiency for the given algorithm.

Computations are done as follows. For every test case, let S_{tc} be the set which contains statistics S_i about sorting arrays A_i where $0<i<|S_{tc}|$. Let n_i be the length of array A_i , and $t(n_i)$ be the number of comparisons the algorithm did in order to sort A_i . Let $c_2 g(n_i)$ be the lower bound and $c_1 g(n_i)$ be the upper bound such that for for some constants c_1 , c_2 , n_{min} and function $g(n_i)$, $c_2 g(n_i)\leq t(n_i)\leq c_1 g(n_i)$ for all $n\geq n_{min}$. Then $t(n_i)=x_i g(n_i)$ for some variable x_i such that $c_2\leq x_i\leq c_1$. (See example chart on the next page)

These definitions can be applied to estimate the efficiency class of an algorithm for a particular test case. The program starts matching collected statistics data to a particular known function $g(n_i)\in\{1,\log_2 n, n, n\log_2 n, n^2\}$. This allows to compute the value of x_i for every array of length n_i which required $t(n_i)$ comparison to be sorted. Now equation $t(n_i)=x_i g(n_i)$ can be solved for x_i , $x_i=g(n_i)/t(n_i)$. In order to find c_1 and c_2 which would satisfy the condition $c_2\leq x_i\leq c_1$ mentioned above, we find $c_1=max(x_i\geq x_{min})$ and $c_2=min(x_i\geq x_{min})$ among all x_i computed. Once c_1 and c_2 , the upper and lower bound constants, are known, we can compute how well test case S_{tc} fits the offered efficiency model $g(n)$ by computing the ratio $r=c_2/c_1$ where $0\leq r\leq 1$. For values of r approaching to 1, the program can determine when $t(n)\in\Theta(g(n))$.



Understanding Program Output

For each of the sorting algorithms, three different types of test cases were generated and executed. These test cases allowed to demonstrate average, best and worst-case efficiencies of the algorithms. Please see attached statistics tables and efficiency analysis results produced by the program.

Base-10 exponential increase of array lengths makes the table human-readable and concise and allows to observe how each of the algorithm behaves while sorting arrays of very different sizes sizes, small ones with 10 elements and huge ones with million and more elements. Processing randomly generated arrays of big sizes allows the algorithms to demonstrate their pure average-case efficiency. On the other hand, the length and the number of comparisons grow so fast from 10^1 to 10^6 or 10^8 that it is hard to plot them on a chart even using logarithmic scales. However, from the table, we can observe the order of growth of comparisons as a function of array length and see how it fits the efficiency class defined by program analysis.

From program input, it is also possible to notice that time/element ratio for linear-logarithmic test cases changes from decreasing to increasing when array length grows from 10^6 and 10^7. This could be partially explained with the fact that the program was executed on a processor with 8 MB cache which can store approximately 2×10^6 integers. As we can observe, our array sizes often exceeds this memory capacity.

Algorithm Analysis

In the overview of sorting algorithms section we discussed expected average, best and worst case efficiency for every algorithm in the scope of this research. The program used for algorithm testing allowed us to receive practical evidence of what was expected. Test results can be summarized by the table as follows.

Test Case	Insertion Sort	Quick Sort	Merge Sort
Random Sequence	average-case $O(n^2)$ (98.5%) 5.6 min for $n=10^6$	average-, best-case $O(n \log_2 n)$ (89.9%) **136.5 *ms*** for $n=10^6$	average-, worst-case $O(n \log_2 n)$ (91.7%) 203.1 ms for $n=10^6$
Descending Sequence	worst-case $O(n^2)$ (99.9%) 11.2 min for $n=10^6$	worst-case $O(n^2)$ (99.7%) 12.7 min for $n=10^6$	best-case $O(n \log_2 n)$ (97.2%) **101.6 ms** for $n=10^6$
Ascending Sequence	best-case $O(n)$ (99.9%) **3.8 *ms*** for $n=10^6$	worst-case $O(n^2)$ (99.7%) 13.7 min for $n=10^6$	best-case $O(n \log_2 n)$ (98.0%) 99.9 ms for $n=10^6$

Every sorting algorithms demonstrated its own strengths and weaknesses. Test cases executed and analyzed by the program allowed to observe many interesting details about sorting algorithms..

- ***Insertion sort*** is very inefficient for big unsorted arrays but very efficient for small unsorted arrays. As we can see from the program output, this is the most efficient algorithm for sorting small random sequences 10 elements when compared to other two algorithms. For example, it showed to be 31 times faster than the asymptotically superior merge sort algorithm for such small arrays. Small arrays like this can easily fit in the CPU registers or its highest level cache, and compact insertion sort algorithm which doesn't need extra memory benefits from it. At the same time, the merge sort needs to make multiple copies of array partitions and push them into memory stack due to its recursive nature. Insertion sort also showed the best efficiency among other algorithms when given an already sorted array of any size. It correctly detects elements which are sorted already and simply skips them. It can be very practical if there is a need to resort a previously sorted array in which just one element was replaced.

- ***Quick sort*** demonstrated the best efficiency among other algorithms when given a big unsorted array. Descending and ascending sequences demonstrated very bad quadratic efficiencies. At the same time, linear logarithmic efficiency of the quick sort starts very slow but doesn't slow down much for bigger *n* values. Two examples which demonstrate worst-case behavior of the quick sort are attached to this paper. Known solutions which can improve the algorithm and fix its worst-case behavior exist. For example, median-of-three partitioning algorithm can offer some improvement over Hoare partitioning implemented for this project.

- ***Merge sort*** algorithm demonstrated the most balanced high results for all test cases given. This can be explained by the fact that for any input, the algorithm copies and them merges array partitions. For random arrays, it demonstrated slightly slower efficiency than the quick sort. However it showed the best efficiency among other algorithms while sorting sequences in descending order. Its balanced efficiency makes it the most universal among these three algorithms. It may show good results on computers which have fast memory of sufficient capacity.

Conclusion

The program with built in statistics collection and mathematical analysis tool developed for this project helped a lot to analyze the algorithms in this project. The efficiency estimation tool in the program helped to evaluate the efficiency of the algorithms for every test cases and demonstrated very high accuracy of its estimates. If I had more time, I would have spent it making more helpful tools like this one. A good programmer should sometimes write such programs like the one developed for this project in order to examine various (not only sorting) algorithms to make design decisions about their application based on its results.

Deep study of search algorithms demonstrates that nearly every sorting algorithm has its own strengths which deserve attention, and at the same time, there is no single perfect universal sorting algorithm. A programmer making a decision about which sorting algorithm would work better for a particular set of data should consider the specifics of the data sets to be sorted. Whether these are arrays of length up to 10 elements, almost sorted arrays, big unsorted arrays or arrays sorted in descending order, all of these issues should determine which algorithm to choose. It is often good to have multiple algorithms implemented in one program and make this program flexible to decide which of the algorithms to use.

```
insertion sort; input order: random
    length     comparisons      memory ops      total time    time/element
        10              36              36        0.005 ms          533 ns
       100            2460            2460        0.183 ms         1828 ns
      1000          253795          253795        9.050 ms         9050 ns
     10000        25029240        25029240       27.924 ms         2792 ns
    100000      2502808870      2502808870     3317.517 ms        33175 ns
   1000000    250076901421    250076901421   336514.671 ms       336514 ns
quadratic: 98.5%

insertion sort; input order: descending
    length     comparisons      memory ops      total time    time/element
        10              54              54        0.000 ms           33 ns
       100            5049            5049        0.008 ms           79 ns
      1000          500499          500499        0.683 ms          682 ns
     10000        50004999        50004999       66.094 ms         6609 ns
    100000      5000049999      5000049999     6669.391 ms        66693 ns
   1000000    500000499999    500000499999   669565.085 ms       669565 ns
quadratic: 99.9%

insertion sort; input order: ascending
test case terminated due to "OutOfMemoryError" exception
    length     comparisons      memory ops      total time    time/element
        10               9               9        0.000 ms            0 ns
       100              99              99        0.000 ms            3 ns
      1000             999             999        0.004 ms            3 ns
     10000            9999            9999        0.038 ms            3 ns
    100000           99999           99999        0.378 ms            3 ns
   1000000          999999          999999        3.811 ms            3 ns
  10000000         9999999         9999999       38.210 ms            3 ns
 100000000        99999999        99999999      381.331 ms            3 ns
linear: 99.9%
BUILD SUCCESSFUL (total time: 16 minutes 57 seconds)
```

quick sort; input order: random

length	comparisons	memory ops	total time	time/element
10	32	9	0.009 ms	899 ns
100	733	174	0.041 ms	409 ns
1000	11875	2402	0.665 ms	664 ns
10000	156472	31940	6.076 ms	607 ns
100000	2088129	393869	13.667 ms	136 ns
1000000	25138467	4725584	136.518 ms	136 ns
10000000	304956810	54579476	1538.725 ms	153 ns
100000000	3406259189	626734840	17244.630 ms	172 ns

linearlogarithmic: 89.8%

quick sort; input order: descending

length	comparisons	memory ops	total time	time/element
10	63	9	0.001 ms	133 ns
100	5148	99	0.010 ms	99 ns
1000	501498	999	0.830 ms	829 ns
10000	50014998	9999	77.253 ms	7725 ns
100000	5000149998	99999	7708.426 ms	77084 ns
1000000	500001499998	999999	764333.652 ms	764333 ns

quadratic: 99.7%

quick sort; input order: ascending

length	comparisons	memory ops	total time	time/element
10	63	9	0.002 ms	166 ns
100	5148	99	0.011 ms	106 ns
1000	501498	999	0.853 ms	852 ns
10000	50014998	9999	79.987 ms	7998 ns
100000	5000149998	99999	7857.703 ms	78577 ns
1000000	500001499998	999999	819236.483 ms	819236 ns

quadratic: 99.7%

BUILD SUCCESSFUL (total time: 26 minutes 59 seconds) merge sort; input order: random

length	comparisons	memory ops	total time	time/element
10	23	68	0.166 ms	16557 ns
100	543	1344	0.157 ms	1569 ns
1000	8709	19952	1.958 ms	1957 ns
10000	120526	267232	14.039 ms	1403 ns
100000	1536335	3337856	22.073 ms	220 ns
1000000	18673777	39902848	203.068 ms	203 ns
10000000	220099394	466445568	2230.303 ms	223 ns
100000000	2532917967	5331564544	24700.060 ms	247 ns

linearlogarithmic: 91.7%

merge sort; input order: descending

length	comparisons	memory ops	total time	time/element
10	19	68	0.002 ms	199 ns
100	356	1344	0.007 ms	73 ns
1000	5044	19952	0.073 ms	73 ns
10000	69008	267232	0.795 ms	79 ns
100000	853904	3337856	8.934 ms	89 ns
1000000	10066432	39902848	101.577 ms	101 ns
10000000	118788160	466445568	1214.713 ms	121 ns
100000000	1351335168	5331564544	12902.411 ms	129 ns

linearlogarithmic: 97.2%

merge sort; input order: ascending

length	comparisons	memory ops	total time	time/element
10	15	68	0.002 ms	233 ns
100	316	1344	0.007 ms	73 ns
1000	4932	19952	0.076 ms	75 ns
10000	64608	267232	0.813 ms	81 ns
100000	815024	3337856	9.033 ms	90 ns
1000000	9884992	39902848	99.909 ms	99 ns
10000000	114434624	466445568	1148.436 ms	114 ns
100000000	1314447104	5331564544	13729.146 ms	137 ns

linearlogarithmic: 98.0%

BUILD SUCCESSFUL (total time: 58 seconds)

```java
/**
 * @author      Bogdan Kovch
 * @class       CS 350
 * @assignment  final project
 * @file        SortAlgs.java
 * @date        May 29, 2013 - June 6, 2013
 */

/**
 * This program analyzes one of the sorting algorithms specified by the user. The program
 * automatically generates a series of arrays of integers (sequences) for every test case and
 * feeds them into the algorithm. Every algorithm has built in routines for gathering statistics
 * about algorithm's work while processing a particular sequence of integers. Statistics for
 * different individual sequences is combined into the statistics of an individual test case. Then
 * every test case statistics is passed to a procedure which matches this statistics to various
 * mathematical models of efficiency classes and computes percentage how well this data matches
 * every model. The efficiency model which matched the given statistics with the highest percentage
 * is then selected to describe the behavior of the given test case efficiency for the given
 * algorithm. For more details, please see the project report and comments within code.
 *
 * Compilation notes: use flags -Xss1024m and -Xmx4096m to deal with possible memory limitations.
 */

package sortalgs;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Random;

public class SortAlgs {

    private static Random rand;           // random number generator for random sequences
    private static int base = 10;         // base for sequence sizes in base^i for 0 < i
    private static int exp_incr = 1;      // exponent increase of i in base^i
    private static final double MAX_TIME = Math.pow(10, 10); // 10^9 ns = 1 s
    private static final int LENGTH_MIN = 1000; // minimum length required for an array to affect
                                                // efficiency class match.

    public enum Order {RANDOM, DESCENDING, ASCENDING} // order of elements in a generated sequence
    public enum Algorithm {INSERTION, QUICK, MERGE}   // sorting algorithms implemented
    public enum EfficiencyClass {NONE, CONSTANT,       // possible efficiency classes
                    LOGARITHMIC, LINEAR, LINEARLOGARITHMIC, QUADRATIC, EXPONENTIAL}

    // method:  program entry
    // input:   at least one command line argument
    public static void main(String[] args) {
        rand = new Random((System.currentTimeMillis()));
        if(args.length < 1) {       // at least one argument is required
            System.out.println("command-line argument expected: 1. 'i', 'm', or 'q'");
            System.out.println("                                2. array length base");
            return;                 // no arguments given, terminate
        }
        if(args.length >= 2)        // array length base for [length = base^exponent]
            base = Integer.decode(args[1]);
        if(args.length >= 3)        // array length exponent increment for [length = base^exponent]
            exp_incr = Integer.decode(args[2]);

        Algorithm algorithm;
        switch(args[0]){            // parse command line argument to define algorithm for testing
            case "i":               // "i" stands for insertion sort
                algorithm = Algorithm.INSERTION; break;
```

```java
                case "q":               // "q" stands for quick sort
                    algorithm = Algorithm.QUICK; break;
                case "m":               // "m" stands for merge sort
                    algorithm = Algorithm.MERGE; break;
                default:                // unknown argument provided
                    System.out.println("invalid 1st argument - use 'i', 'm', or 'q'.");
                    return;
            }
            doAlgorithmAnalysis(algorithm); // do the analysis of the selected algorithm
        }

        ///////////////////////////////////// Analysis Methods /////////////////////////////////////

        // method:  manages the analysis process of the algorithm provided
        //          generates test cases one for each sequence order type
        //          generates a series of sequences for every test case
        //          feeds each sequence into a specified sorting algorithm
        //          double checks whether the algorithm actually sorted the sequence
        //          gathers algorithm performance statistics for every sequence in every test case
        //          analises and defines efficiency class for every test case for the algorithm
        //          displays statistics and analysis results
        // input:   sorting algorithm to be analysed
        public static void doAlgorithmAnalysis(Algorithm algorithm) {
            for(Order order: Order.values()) {  // generate test cases for every sequence order type
                System.out.printf("\n%s sort; input order: %s\n", // print desciption of
                        algorithm.toString().toLowerCase(),       // algorithm and
                        order.toString().toLowerCase());          // sequence order type
                List<Statistics> stats = new ArrayList<>();       // stores test case statistics
                double time = System.nanoTime();                // time when this test case started
                try {                                           // handle possible memory exceptions
                    for(int i = exp_incr;                       // generate sequences for sorting
                            System.nanoTime()-time < MAX_TIME;  // until time is gone
                            i += exp_incr) {                    // increase next sequence multiple times
                        int[] sequence = generateSequence(order, (int)Math.pow(base, i));
                        Statistics s;                           // contains statistics for this sequence
                        switch(algorithm) {                     // select requested sorting algorithm
                            case INSERTION:
                                s = insertionSort(sequence); break;
                            case QUICK:
                                s = quickSortInit(sequence); break;
                            case MERGE:
                                s = mergeSortInit(sequence); break;
                            default:
                                return;
                        }
                        if(!sequenceIsSorted(sequence)) // check if the algorithm sorted the sequence
                            System.out.println("Error! Sequence is not sorted! Algorithm failure!" );
                        stats.add(s); // add sequence sorting statistics to test case statistics
                    }
                } catch (OutOfMemoryError e) {
                    System.out.println("test case terminated due to \"OutOfMemoryError\" exception");
                }
                printStats(stats); // display current test case statistics
                evaluateEfficiencyClass(stats); //evaluate test case efficiency class
            }
        }

        // method:  generates a sequence of specified size with elements in given order
        // input:   order (random, descending or ascending) of elements in the sequence to be generated
        //          size of the sequence to be generated
        // output:  sequence (array of integers) generated
        public static int[] generateSequence(Order order, int size) {
```

```
        if(size <= 0)                               // require nonnegative size
            return null;
        int[] sequence = new int[size];         // allocate a new array of needed size
        switch(order){
            case RANDOM:
                for(int i = 0; i < size; i++)
                    sequence[i] = rand.nextInt(Integer.MAX_VALUE);
                return sequence;
            case DESCENDING:
                for(int i = 0; i < size; i++)
                    sequence[i] = size - i;
                return sequence;
            case ASCENDING:
                for(int i = 0; i < size; i++)
                    sequence[i] = i;
                return sequence;
            default:                                // unknown order type
                return null;
        }
    }

    // method:  checks whether the given sequence is sorted in ascending order
    // input:   sequence (array of integers) to be checked
    // output:  true if sequence is sorted or false otherwise
    public static boolean sequenceIsSorted(int[] sequence) {
        if(sequence == null || sequence.length <= 1) // validate sequence passed
            return true;
        int length = sequence.length;
        for(int i = 0, j = 1; j < length; i++, j++)  // compare every pair of consecutive elements
            if( sequence[i] > sequence[j] )
                return false;
        return true;
    }

    // method:  displays test case statistics in the form of table
    // input:   list containing statistics about processing a particular test case
    // output:  display output visualizing the statistics
    public static void printStats(List<Statistics> stats) {
        System.out.printf("%10s %15s %15s %15s %15s\n",      // table header
                         "length", "comparisons", "memory ops", "total time", "time/element");
        Iterator<Statistics> iterator = stats.iterator();
        while(iterator.hasNext()) {                          // display every entry in the table
            Statistics s = iterator.next();
            if(s == null || s.length <= 0)                   // validate entry and data
                continue;
            long timePerElement = s.time / s.length;         // calculate field "time/element"
            System.out.printf("%10d %15d %15d %12.3f ms %15s\n",
                              s.length, s.comparisons, s.memoryOps,
                              s.time *  Math.pow(10, -6),
                              timePerElement +  " ns" );
        }
    }

    // method:  defines efficiency class for processing the test case described in given statistics
    // input:   list containing statistics about processing a particular test case
    // output:  displays to which efficiency class processing of this test case belongs
    public static void evaluateEfficiencyClass(List<Statistics> stats) {
        EfficiencyClass bestEfficiency = EfficiencyClass.NONE;  // efficiency with the best match
        double bestMatch = 0.0;                                 // best match score
        for(EfficiencyClass efficiency: EfficiencyClass.values()) { // try every efficiency class
            double match = matchEfficiencyClass(stats, efficiency);
            if(match > bestMatch) {                                 // store the best efficiency
```

```
                bestMatch = match;
                bestEfficiency = efficiency;
            }
        }
        System.out.printf("%s: %3.1f%%\n",                          // print the best match info
                          bestEfficiency.toString().toLowerCase(),  bestMatch * 100.0);
    }

    // method:  computes the score describing how well the test case statistics matches the
    //          efficiency class offered
    // input:   list containg statistics about a test case
    //          efficiency class to which the statistics needs to be matched and evaluated
    // output:  the score within [0.0, 1.0] describing how the statistics mathes the offered
    //          efficiency class: 0.0 - no match, 1.0 - parfect match
    public static double matchEfficiencyClass(List<Statistics> stats, EfficiencyClass effClass) {
        int size = stats.size();
        double lower = Double.MAX_VALUE;    // lower bound constant of Theta(n^2)
        double upper = Double.MIN_VALUE;    // upper bound constant of Theta(n^2)

        // Find potential min and max constants for lower and upper bounds of Theta(n^2).
        // If all constants in the statistics within the lower and upper bound constants are very
        // similar in terms of the offered efficiency model then the statistics trend matches this
        // efficiency class well.
        for(int i = 0; i < size; i++) {     // ignore statistics of the first smallest sequences
            Statistics s = stats.get(i);    // retrieve an individual statistics entry
            if(s == null)                   // check if entry is valid
                continue;
            double length = (double)s.length;
            if(length < LENGTH_MIN )        // ignore statistics from small arrays
                continue;
            double x;                       // potential upper/lower bound constant
            switch(effClass) {              // select math model for offered efficiency class
                case CONSTANT:
                    x = s.comparisons; break;
                case LINEAR:    // solve (x * length = s.comparisons) for x
                    x = s.comparisons / length; break;
                case LOGARITHMIC:// solve (x * log_ length = s.comparisons) for x
                    x = s.comparisons / (Math.log10(length)/Math.log10(2)); break;
                case LINEARLOGARITHMIC:     // solve (x * length log_2 length = s.comparisons) for x
                    x = s.comparisons / (length * Math.log10(length)/Math.log10(2)); break;
                case QUADRATIC: // solve (x * length^2 = s.comparisons) for x
                    x = s.comparisons / (length * length); break;
                default:
                    return 0.0; // unknown efficiency class
            }
            if(x < lower)
                lower = x;
            if(x > upper)
                upper = x;
        }
        if( upper != 0)
            return lower/upper;
        else
            return 0.0;
    }

    ////////////////////////////////// Insertion Sort Algorithm //////////////////////////////////

    // method:  sorts a given sequence using insertion sort algorithm
    // input:   sequence (array of integers) to be sorted
    // output:  statistics describing the process of sorting the sequence given
    //          sequence sorted in ascending order
```

```java
    public static Statistics insertionSort(int[] sequence) {
        if(sequence == null)                        // check if sequence is valid
            return null;
        int length = sequence.length;
        Statistics s = new Statistics(length);      // object for storing algorithm's statisitcs
        s.time = System.nanoTime();                 // take start time
        for(int i = 1; i < length; i++ ) {          // traverse all sequence
            int value = sequence[i];                // currently selected value to be inserted
            int idx = i;                            // index of the currently selected value
            s.comparisons++;                        // gathering statistics
            while(idx > 0 && value < sequence[idx-1]) { // traverse unsorted part of the sequence
                sequence[idx] = sequence[idx - 1]; // shift an element to the right
                idx--;                              // select position to the left
                s.memoryOps++;                      // gathering statistics
                s.comparisons++;                    // gathering statistics
            }
            sequence[idx] = value;                  // insert the value
            s.memoryOps++;                          // gathering statistics
        }
        s.time = System.nanoTime() - s.time;        // take finish time and compute total time
        return s;
    }

    ////////////////////////////////// Quick Sort Algorithm //////////////////////////////////

    // method:  prepares statistics before invoking the quicksort algorithm
    // input:   sequence (array of integers) to be sorted
    // output:  statistics describing the process of sorting the sequence given
    //          sequence sorted in ascending order
    public static Statistics quickSortInit(int[] sequence) {
        if(sequence == null)                         // check if sequence is valid
            return null;
        int length = sequence.length;
        Statistics s = new Statistics(sequence.length);
        s.time = System.nanoTime();                  // take start time
        quickSort(sequence, 0, length-1, s);         // do the actual quicksort
        s.time = System.nanoTime() - s.time;         // take finish time and compute total tome
        return s;
    }

    // method:  sorts a subsequence using quicksort algorithm
    // input:   sequence (array of integers)
    //          left and right indices which define the subsequence of sequence
    //          object for storing algorithm's statistics
    // output:  subsequence sorted in ascending order
    //          updated statistics
    public static void quickSort(int[] sequence, int left, int right, Statistics s) {
        if(left < right) {
            int split = hoarePartition(sequence, left, right, s); // sort this partition
            quickSort(sequence, left, split-1, s);                // sort left partition
            quickSort(sequence, split+1, right, s);               // sort right partition
        }
    }

    // method:  partitions and sorts a subarray using Hoare's partitioning algorithm
    // input:   sequence (array of integers)
    //          left and right indices which define the subsequence of sequence
    //          object for storing algorithm's statistics
    // output:  pivot index for further splitting into partitions
    //          updated statistics
    public static int hoarePartition(int[] sequence, int left, int right, Statistics s) {
        int pivot, i, j;
```

```java
        pivot = sequence[left];
        i = left;
        j = right + 1;
        while(true) {
            s.comparisons += 2;
            while(i < right && sequence[++i] < pivot)
                s.comparisons++;
            while(sequence[--j] > pivot)
                s.comparisons++;
            if(i >= j)
                break;
            s.memoryOps++;
            swap(sequence, i, j);
        }
        s.memoryOps++;
        swap(sequence, left, j);
        return j;
    }

    //////////////////////////////////// Merge Sort Algorithm ////////////////////////////////////

    // method:  prepares statistics before invoking the mergesort algorithm
    // input:   sequence (array of integers) to be sorted
    // output:  statistics describing the process of sorting the sequence given
    //          sequence sorted in ascending order
    public static Statistics mergeSortInit(int[] sequence) {
        if(sequence == null)
            return null;
        int length = sequence.length;
        Statistics s = new Statistics(sequence.length);
        s.time = System.nanoTime();                // take start time
        mergeSort(sequence, s);        // do the actual quicksort
        s.time = System.nanoTime() - s.time;       // take finish time and compute total tome
        return s;
    }

    // method:  sorts a sequence using recursive mergesort algorithm
    // input:   sequence (array of integers) to be sorted
    //          statistics describing the process of sorting the whole sequence
    // output:  sequence sorted in ascending order
    //          updated statistics
    public static void mergeSort(int[] sequence, Statistics s) {
        int length = sequence.length;
        if(length > 1) {
            int split = (int)Math.floor((double)length/2.0);
            s.memoryOps += length;
            int[] left = copyToNew(sequence, 0, split);
            int[] right = copyToNew(sequence, split, length);
            mergeSort(left, s);
            mergeSort(right, s);
            merge(left, right, sequence, s);
        }
    }


    // method:  merges left and right sorted sequences into one sorted sequence.
    // input:   left and right sorted sequences
    //          statistics describing the process of sorting the whole sequence
    // output:  sorted sequence
    //          updated statistics
    public static void merge(int[] left, int [] right, int[] sequence, Statistics s) {
        int i=0, j=0, k=0;
```

```java
        int leftLength = left.length;
        int rightLength = right.length;
        int sequenceLength = sequence.length;
        while(i < leftLength && j < rightLength) {
            s.comparisons++;
            s.memoryOps++;
            if(left[i] <= right[j]) {
                sequence[k] = left[i];
                i++;
            }
            else {
                sequence[k] = right[j];
                j++;
            }
            k++;
        }
        s.memoryOps += sequenceLength - k;
        if(i == leftLength)
            copyFromIdx(right, sequence, j, k);
        else
            copyFromIdx(left, sequence, i, k);
    }

    // method:  copies a subsequence of the source sequence into a new sequence
    // input:   source sequence (array of integers)
    //          left and right indices defining a subsequence of the source sequence
    // output:  the copy of subsequence of source sequence
    public static int[] copyToNew(int[] source, int left, int right) {
        int length = right-left;
        int[] destination = new int[length];
        for(int i = left, j=0; i<right; i++, j++)
            destination[j] = source[i];
        return destination;
    }

    // method:  copies elements from the source sequence into the destination sequence starting
    //          from indices specified for both of the sequences.
    // input:   source and destination sequences (arrays of integers)
    //          source index to start copy from
    //          destination index to start copy into
    // output:  destination sequence with some elements copied from the source sequence
    public static void copyFromIdx(int[] source, int[] destination, int srcIdx, int dstIdx) {
        int srcLen = source.length;
        int dstLen = destination.length;
        for(; srcIdx < srcLen && dstIdx < dstLen; srcIdx++, dstIdx++)
            destination[dstIdx] = source[srcIdx];
    }

    ///////////////////////////////// Generel Purpose Methods //////////////////////////////////

    // method:  swaps two values in the sequence
    // input:   sequence (array of integers) in which the elements have to be swapped
    //          two indices fo elements which values must be swapped
    // output:  sequence with swapped elements
    public static void swap(int[] sequence, int i, int j) {
        int buffer = sequence[i];
        sequence[i] = sequence[j];
        sequence[j] = buffer;
    }
}
```

```java
/**
 * @author      Bogdan Kovch
 * @class       CS 350
 * @assignment  final project
 * @file        Statistics.java
 * @date        May 29, 2013 - June 6, 2013
 */

/**
 * This file defines the class used to populate instances encapsulating statistics data about
 * sorting algorithm performance.
 */

package sortalgs;

public class Statistics {
    public int length = 0;           // number of elements in the sequence
    public long comparisons = 0;     // number of element comparisons an algorithm performed
    public long memoryOps = 0;       // number of memory operations; that is the number of
                                     // writes into sequence elements
    public long time = 0;            // total time taken to sort the sequence in nanoseconds

    Statistics(int length){          // constructor defines the length of the sequence
        this.length = length;
    }
}
```

```
Quick Sort Worst-Case Efficiency Examples.
In these examples, array partitions are always unbalanced with one element in one partitiotn and
the remaining elements in another partition. Pointers i and j need to move through the whole
bigger partition and finaly do and undo one single unnecesary swap. Such behavior causes
quadratic efficiency.

Example 1. Array Sorted in Ascending Order: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

    0 1 2 3 4 5 6 7 8 9      -  array indices
      i               j
    1 2 3 4 5 6 7 8 9 10   -  subarray to be partitioned (l=0, r=9); l<r
    j i
    2 1 3 4 5 6 7 8 9 10   -  swap i and j
    1 2 3 4 5 6 7 8 9 10   -  undo last swap
    1 2 3 4 5 6 7 8 9 10   -  swap pivot and j
    1                      -  single element subarray (l=0, r=-1); l>r
        i             j
      2 3 4 5 6 7 8 9 10   -  subarray to be partitioned (l=1, r=9); l<r
      j i
      3 2 4 5 6 7 8 9 10   -  swap i and j
      2 3 4 5 6 7 8 9 10   -  undo last swap
      2 3 4 5 6 7 8 9 10   -  swap pivot and j
      2                    -  single element subarray (l=1, r=0); l>r
          i           j
        3 4 5 6 7 8 9 10   -  subarray to be partitioned (l=2, r=9); l<r
        j i
        4 3 5 6 7 8 9 10   -  swap i and j
        3 4 5 6 7 8 9 10   -  undo last swap
        3 4 5 6 7 8 9 10   -  swap pivot and j
        3                  -  single element subarray (l=2, r=1); l>r
            i         j
          4 5 6 7 8 9 10   -  subarray to be partitioned (l=3, r=9); l<r
          j i
          5 4 6 7 8 9 10   -  swap i and j
          4 5 6 7 8 9 10   -  undo last swap
          4 5 6 7 8 9 10   -  swap pivot and j
          4                -  single element subarray (l=3, r=2); l>r
              i       j
            5 6 7 8 9 10   -  subarray to be partitioned (l=4, r=9); l<r
            j i
            6 5 7 8 9 10   -  swap i and j
            5 6 7 8 9 10   -  undo last swap
            5 6 7 8 9 10   -  swap pivot and j
            5              -  single element subarray (l=4, r=3); l>r
                i     j
              6 7 8 9 10   -  subarray to be partitioned (l=5, r=9); l<r
              j i
              7 6 8 9 10   -  swap i and j
              6 7 8 9 10   -  undo last swap
              6 7 8 9 10   -  swap pivot and j
              6            -  single element subarray (l=5, r=4); l>r
                  i   j
                7 8 9 10   -  subarray to be partitioned (l=6, r=9); l<r
                j i
                8 7 9 10   -  swap i and j
                7 8 9 10   -  undo last swap
                7 8 9 10   -  swap pivot and j
                7          -  single element subarray (l=6, r=5); l>r
                    i j
                  8 9 10   -  subarray to be partitioned (l=7, r=9); l<r
                  j i
                  9 8 10   -  swap i and j
                  8 9 10   -  undo last swap
                  8 9 10   -  swap pivot and j
                  8        -  single element subarray (l=7, r=6); l>r
                      ij
                    9 10   -  subarray to be partitioned (l=8, r=9); l<r
                    j i
                    10 9   -  swap i and j
                    9 10   -  undo last swap
                    9 10   -  swap pivot and j
                    9      -  single element subarray (l=8, r=7); l>r
                      10   -  single element subarray (l=9, r=9); l=r
    1 2 3 4 5 6 7 8 9 10   -  array has been sorted!
```

```
Example 2. Array Sorted In Descending Order: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

    0 1 2 3 4 5 6 7 8 9   -  array indices
      i               j
    10 9 8 7 6 5 4 3 2 1   -  subarray to be partitioned (l=0, r=9); l<r
                      ij
    10 9 8 7 6 5 4 3 2 1   -  swap i and j
    10 9 8 7 6 5 4 3 2 1   -  undo last swap
    1 9 8 7 6 5 4 3 2 10   -  swap pivot and j
      i             j
    1 9 8 7 6 5 4 3 2     -  subarray to be partitioned (l=0, r=8); l<r
    j i
    9 1 8 7 6 5 4 3 2     -  swap i and j
    1 9 8 7 6 5 4 3 2     -  undo last swap
    1 9 8 7 6 5 4 3 2     -  swap pivot and j
    1                     -  single element subarray (l=0, r=-1); l>r
        i           j
      9 8 7 6 5 4 3 2     -  subarray to be partitioned (l=1, r=8); l<r
                    ij
      9 8 7 6 5 4 3 2     -  swap i and j
      9 8 7 6 5 4 3 2     -  undo last swap
      2 8 7 6 5 4 3 9     -  swap pivot and j
        i         j
      2 8 7 6 5 4 3       -  subarray to be partitioned (l=1, r=7); l<r
      j i
      8 2 7 6 5 4 3       -  swap i and j
      2 8 7 6 5 4 3       -  undo last swap
      2 8 7 6 5 4 3       -  swap pivot and j
      2                   -  single element subarray (l=1, r=0); l>r
          i       j
        8 7 6 5 4 3       -  subarray to be partitioned (l=2, r=7); l<r
                  ij
        8 7 6 5 4 3       -  swap i and j
        8 7 6 5 4 3       -  undo last swap
        3 7 6 5 4 8       -  swap pivot and j
          i     j
        3 7 6 5 4         -  subarray to be partitioned (l=2, r=6); l<r
        j i
        7 3 6 5 4         -  swap i and j
        3 7 6 5 4         -  undo last swap
        3 7 6 5 4         -  swap pivot and j
        3                 -  single element subarray (l=2, r=1); l>r
            i   j
          7 6 5 4         -  subarray to be partitioned (l=3, r=6); l<r
                ij
          7 6 5 4         -  swap i and j
          7 6 5 4         -  undo last swap
          4 6 5 7         -  swap pivot and j
            i j
          4 6 5           -  subarray to be partitioned (l=3, r=5); l<r
          j i
          6 4 5           -  swap i and j
          4 6 5           -  undo last swap
          4 6 5           -  swap pivot and j
          4               -  single element subarray (l=3, r=2); l>r
              ij
            6 5           -  subarray to be partitioned (l=4, r=5); l<r
              ij
            6 5           -  swap i and j
            6 5           -  undo last swap
            5 6           -  swap pivot and j
            5             -  single element subarray (l=4, r=4); l=r
                7         -  single element subarray (l=6, r=5); l>r
                  8       -  single element subarray (l=7, r=6); l>r
                    9     -  single element subarray (l=8, r=7); l>r
                      10   -  single element subarray (l=9, r=8); l>r
                          -  single element subarray (l=10, r=9); l>r
    1 2 3 4 5 6 7 8 9 10   -  array has been sorted!
```