# Benchmarking - Algorithms in a Nutshell

by Gary Pollice, George T. Heineman, Stanley Selkow

Each algorithm in this book is presented in its own section where you will find individual performance data on the behavior of the algorithm. In this bench marking chapter, we present our infrastructure to evaluate algorithm performance. It is important to explain the precise means by which empirical data is computed, to enable the reader to both verify that the results are accurate and understand where the assumptions are appropriate or inappropriate given the context in which the algorithm is intended to be used.

There are numerous ways by which algorithms can be analyzed. Chapter 2 presented the theoretic formal treatment, introducing the concepts of worst-case and average-case analysis. These theoretic results can be empirically evaluated in some cases, though not all. For example, consider evaluating the performance of an algorithm to sort 20 numbers. There are $2.43*10^{18}$ permutations of these 20 numbers, and one cannot simply exhaustively evaluate each of these permutations to compute the average case. Additionally, one cannot compute the average by measuring the time to sort all of these permutations. We find that we must rely on statistical measures to assure ourselves that we have properly computed the expected performance time of the algorithm.

## Statistical Foundation

In this chapter we briefly present the essential points to evaluate the performance of the algorithms. Interested readers should consult any of the large number of available textbooks on statistics for more information on the relevant statistical information used to produce the empirical measurements in this book.

To compute the performance of an algorithm, we construct a *suite* of *T* independent *trials* for which the algorithm is executed. Each trial is intended to execute an algorithm on an input problem of size *n*. Some effort is made to ensure that these trials are all reasonably *equivalent* for the algorithm. When the trials are actually identical, then the intent of the trial is to quantify the variance of the underlying implementation of the algorithm. This may be suitable, for example, if it is too costly to compute a large number of independent equivalent trials. The *suite* is executed and millisecond-level timings are taken before and after the observable behavior. When the code is written in Java, the system garbage collector is invoked immediately prior to launching the trial; although this effort can't guarantee that the garbage collector does not execute during the trial, it is hoped to reduce the chance that extra time (unrelated to the algorithm) is spent. From the full set of *T* recorded times, the best and worst performing times are discarded as being "outliers." The remaining *T*−2 time records are averaged, and a standard deviation is computed using the following formula:

$_{\Sigma}(x_i - x)^2$

*i*

σ = -----------------------------

*n* − 1

where *xi* is the time for an individual trial and *x* is the average of the *T*−2 trials. Note here that *n* is equal to *T*−2, so the denominator within the square root is *T*−3. Calculating averages and standard deviations will help predict future performance, based on Table A-1, which shows the probability (between 0 and 1) that the actual value will be within the range [*x*−*k**σ,*x*+*k**σ], where σ represents the standard deviation value computed in the equation just shown. The probability values become *confidence intervals* that declare the confidence we have in a prediction.

*Table A-1. Standard deviation table*

| k | Probability |
|---|---|
| **1** | 0.6827 |
| **2** | 0.9545 |
| **3** | 0.9973 |
| **4** | 0.9999 |
| **5** | 1 |

For example, in a randomized trial, it is expected that 68.27% of the time the result will fall within the range [*x*−σ, *x*+σ].

When reporting results, we never present numbers with greater than four decimal digits of accuracy, so we don't give the mistaken impression that we believe the accuracy of our numbers extends that far. When the computed fifth and greater digits falls in the range [0, 49,999], then these digits are simply truncated; otherwise, the fourth digit is incremented to reflect the proper rounding. This process will convert a computation such as 16.897986 into the reported number 16.8980.

## Hardware

In this book we include numerous tables showing the performance of individual algorithms on sample data sets. We used two different machines in this process:

*Desktop PC*

We used a reasonable "home office" personal computer. This computer had a Pentium(R) 4 CPU 2.8Ghz with 512 MB of RAM.

*High-end computer*

We had access to a set of computers configured as part of a Linux cluster. This computer had a 2x dual-core AMD Opteron™ Processor with 2.6 Ghz speed and 16 gigabytes of Random Access Memory (RAM).

The high-end computer was made available because of work supported by the National Science Foundation under Grant No. 0551584. Any opinions, findings, and conclusions or recommendations expressed in this book are those of the authors and do not necessarily reflect the views of the National Science Foundation.

We refer to these computers by name in the tables of this book.

## An Example

Assume we wanted to benchmark the addition of the numbers from 1 to *n*.An experiment is designed to measure the times for *n*=1,000,000 to *n*=5,000,000 in increments of one million. Because the problem is identical for *n* and doesn't vary, we execute for 30 trials to eliminate as much variability as possible.

The hypothesis is that the time to complete the sum will vary directly in relation to *n*. We show three programs that solve this problem—in Java, C, and Scheme— and present the benchmark infrastructure by showing how it is used.

## Java Benchmarking Solutions

On Java test cases, the current system time (in milliseconds) is determined immediately prior to, and after, the execution of interest. The code in Example A-1 measures the time it takes to complete the task. In a perfect computer, the 30 trials should all require exactly the same amount of time. Of course this is unlikely to happen, since modern operating systems have numerous background processing tasks that share the same CPU on which the performance code executes.

*Example A-1. Java example to time execution of task*

```
public class Main {

public static void main (String[]args) {

TrialSuite ts = new TrialSuite();

for (long len = 1000000; len <= 5000000; len += 1000000) {

for (int i = 0; i < 30; i++) {

System.gc();

long now = System.currentTimeMillis();

/** Task to be timed. */ long sum = 0; for (int x = 1; x <= len; x++) { sum += x; }
```

*Example A-1. Java example to time execution of task (continued)*

```
long end = System.currentTimeMillis(); ts.addTrial(len, now, end);

} } System.out.println (ts.computeTable());

} }
```

The TrialSuite class stores trials by their size. Once all trials have been added to the suite, the resulting table is computed. To do this, the running times are added together to find the total sum, the minimum value, and the maximum value. As described earlier, the minimum and maximum values are removed from the set when computing the average and standard deviation.

## Linux Benchmarking Solutions

For C test cases, we developed a benchmarking library to be linked with the code to test. In this section we briefly describe the essential aspects of the timing code and refer the interested reader to the code repository for the full source.

Primarily created for testing sort routines, the C-based infrastructure can be linked against existing source code. The timing API takes over responsibility for parsing the command-line arguments:

```
usage: timing [-n NumElements] [-s seed] [-v] [OriginalArguments] -n declares the problem size [default: 100,000] -v verbose output [default: false] -s # set the seed for random values [default: no seed] -h print usage information
```

The timing library assumes a problem will be attempted whose input size is defined by the [-n] flag. To produce repeatable trials, the random seed can be set with [-s seed]. To link with the timing library, a test case provides the following functions:

void problemUsage() Report to the console the set of [OriginalArguments] supported by the specific code. Note that the timing library parses the declared timing parameters, and remaining arguments are passed along to the prepareInput function.

void prepareInput (int size, int argc, char **argv)

Depending upon the problem to be solved, this function is responsible for building up the input set to be processed within the execute method. Note that this information is not passed directly to execute via a formal argument, but instead should be stored as a static variable within the test case.

void postInputProcessing()

If any validation is needed after the input problem is solved, that code can execute here.

void execute()

This method will contain the body of code to be timed. Thus there will always be a single method invocation that will be part of the evaluation time. When the execute method is empty, the overhead (on the high-end computer) is, on average, .002 milliseconds and is considered to have no impact on the overall reporting.

The test case in Example A-2 shows the code task for the addition example.

*Example A-2. Task describing addition of n numbers*

```
extern int numElements; /* size of n */ void problemUsage() { /* none */ } void prepareInput() { /* none */ } void postInputProcessing() { /* None */ }

void execute() { int x; long sum = 0; for (x = 1; x <= numElements; x++) { sum += x; }

}
```

Each execution of the C function corresponds to a single trial, and so we have a set of shell scripts whose purpose is to execute the code under test repeatedly in order to generate statistics. For each suite, a configuration file is constructed to represent the trial suite run. Example A-3 shows the *config.rc* for the value-based sorting used in Chapter 4.

*Example A-3. Sample configuration file to compare sort executions*

```
# configure to use these BINS BINS=./Insertion ./Qsort_2_6_11 ./Qsort_2_6_6 ./Qsort_straight

# configure suite TRIALS=10 LOW=1 HIGH=16384 INCREMENT=*2
```

This specification file declares that the set of executables will be three variations of QUICKSORT with one INSERTION SORT. The suite consists of problem sizes ranging from *n*=1 to *n*=16,384, where *n* doubles after each run. For each problem size, 10 trials are executed. The best and worst performers are discarded, and the resulting generated table will have the averages (and standard deviations) of the remaining eight trials.

Example A-4 contains the *compare.sh* script that generates an aggregate set of information for a particular problem size *n*.

```
#!/bin/bash
```

*Example A-4. compare.sh benchmarking script*

| | | |
|---|---|---|
| # | | |
| # | This script expects TWO arguments: | |
| # | $1 | -- size of problem n |
| # | $2 | -- number of trials to execute |
| # | This script reads its parameters from the $CONFIG configuration file | |
| # | BINS | set of executables to execute |
| # | EXTRAS | extra command line arguments to use when executing them |
| # | | |
| # | CODE is set to directory where these scripts are to be found | |

```
CODE=`dirname $0`

SIZE=20 NUM_TRIALS=10 if [ $# -ge 1 ] then

SIZE=$1 NUM_TRIALS=$2 fi

if [ "x$CONFIG" = "x" ]

then echo "No Configuration file (\$CONFIG) defined" exit 1

fi

if [ "x$BINS" = "x" ]

then if [ -f $CONFIG ] then

BINS=`grep "BINS=" $CONFIG | cut -f2- -d'='` EXTRAS=`grep "EXTRAS=" $CONFIG | cut -f2- -d'='` fi

if [ "x$BINS" = "x" ]

then echo "no \$BINS variable and no $CONFIG configuration " echo "Set \$BINS to a space-separated set of executables"

fi fi

echo "Report: $BINS on size $SIZE" echo "Date: `date`" echo "Host: `hostname`" RESULTS=/tmp/compare.$$ for b in $BINS do

TRIALS=$NUM_TRIALS

# start with number of trials followed by totals (one per line) echo $NUM_TRIALS > $RESULTS while [ $TRIALS -ge 1 ]
```

*Example A-4. compare.sh benchmarking script (continued)*

```
do $b -n $SIZE -s $TRIALS $EXTRAS | grep secs | sed 's/secs//' >> $RESULTS TRIALS=$((TRIALS-1))

done

# compute average/stdev RES=`cat $RESULTS | $CODE/eval` echo "$b $RES"

rm -f $RESULTS done
```

*compare.sh* makes use of a small C program, eval, which computes the average and standard deviation using the method described at the start of this chapter. This *compare.sh* script is repeatedly executed by a manager script, *suiteRun.sh*, that iterates over the desired input problem sizes specified within the *config.rc* file, as shown in Example A-5.

*Example A-5. suiteRun.sh benchmarking script*

```
#!/bin/bash CODE=`dirname $0`

# if no args then use default config file, otherwise expect it if [ $# -eq 0 ] then

CONFIG="config.rc"

else CONFIG=$1 echo "Using configuration file $CONFIG..."

fi

# export so it will be picked up by compare.sh export CONFIG

# pull out information if [ -f $CONFIG ] then

BINS=`grep "BINS=" $CONFIG | cut -f2- -d'='` TRIALS=`grep "TRIALS=" $CONFIG | cut -f2- -d'='` LOW=`grep "LOW=" $CONFIG | cut -f2- -d'='` HIGH=`grep "HIGH=" $CONFIG | cut -f2- -d'='` INCREMENT=`grep "INCREMENT=" $CONFIG | cut -f2- -d'='`

else echo "Configuration file ($CONFIG) unable to be found." exit -1

fi

# headers HB=`echo $BINS | tr ' ' ','` echo "n,$HB"
```

*Example A-5. suiteRun.sh benchmarking script (continued)*

```
# compare trials on sizes from LOW through HIGH SIZE=$LOW REPORT=/tmp/Report.$$ while [ $SIZE -le $HIGH ] do

# one per $BINS entry
```

```
$CODE/compare.sh $SIZE $TRIALS | awk 'BEGIN{p=0} \ {if(p) { print $0; }} \ /Host:/{p=1}' | cut -d' ' -f2 > $REPORT

# concatenate with , all entries ONLY the average. The stdev is # going to be ignored # -------------------------------------
----------------------VALS=`awk 'BEGIN{s=""}\

{s = s "," $0 }\ END{print s;}' $REPORT` rm -f $REPORT

echo $SIZE $VALS

# $INCREMENT can be "+ NUM" or "* NUM", it works in both cases. SIZE=$(($SIZE$INCREMENT)) done
```

## Scheme Benchmarking Solutions

The Scheme code in this section measures the performance of a series of code executions for a given problem size. In this example (used in Chapter 1) there are no arguments to the function under test other than the size of the problem to compute. First we list some helper functions used to compute the average and standard deviation for a list containing execution times, shown in Example A-6.

*Example A-6. Helper functions for Scheme timing*

```
;; foldl: (X Y -> Y) Y (listof X) -> Y ;; Folds an accumulating function f across the elements of lst. (define (foldl f acc lst)

(if (null? lst) acc (foldl f (f (car lst) acc) (cdr lst))))

;; remove-number: (listof number) number -> (listof number) ;; remove element from list, if it exists (define (remove-number nums x)

(if (null? nums) '() (if (= (car nums) x) (cdr nums) (cons (car nums) (remove-number (cdr nums) x)))))

;; find-max: (nonempty-listof number) -> number ;; Finds max of the nonempty list of numbers. (define (find-max nums)

(foldl max (car nums) (cdr nums)))
```

*Example A-6. Helper functions for Scheme timing (continued)*

```
;; find-min: (nonempty-listof number) -> number ;; Finds min of the nonempty list of numbers. (define (find-min nums)

(foldl min (car nums) (cdr nums)))

;; sum: (listof number) -> number ;; Sums elements in nums. (define (sum nums)

(foldl + 0 nums))

;; average: (listof number) -> number ;; Finds average of the nonempty list of numbers. (define (average nums)

(exact->inexact (/ (sum nums) (length nums))))

;; square: number -> number ;; Computes the square of x. (define (square x) (* x x))

;; sum-square-diff: number (listof number) -> number ;; helper method for standard-deviation (define (sum-square-diff avg nums)

(foldl (lambda (a-number total)

(+ total (square (- a-number avg)))) 0 nums))

;; standard-deviation: (nonempty-listof number) -> number ;; Calculates standard deviation. (define (standard-deviation nums)

(exact->inexact (sqrt (/ (sum-square-diff (average nums) nums) (length nums)))))
```

The helper functions in Example A-6 are used by the timing code in Example A-7, which runs a series of test cases for a desired function.

*Example A-7. Timing Scheme code*

```
;; Finally execute the function under test on a problem size ;; result: (number -> any) -> number ;; Computes how long it takes to evaluate f on the given probSize. (define (result f probSize)

(let* ((start-time (current-inexact-milliseconds)) (result (f probSize)) (end-time (current-inexact-milliseconds)))

(- end-time start-time)))

;; trials: (number -> any) number number -> (listof number) ;; Construct a list of trial results (define (trials f numTrials probSize)

(if (= numTrials 1) (list (result f probSize))
```

*Example A-7. Timing Scheme code (continued)*

```
(cons (result f probSize) (trials f (- numTrials 1) probSize))))

;; Generate an individual line of the report table for problem size (define (smallReport f numTrials probSize) (let* ((results (trials f numTrials probSize))

(reduced (remove-number (remove-number results (find-min results)) (find-max results))))

(display (list 'probSize: probSize 'numTrials: numTrials (average reduced)))

(newline)))

;; Generate a full report for specific function f by incrementing ;; one to the problem size (define (briefReport f inc numTrials minProbSize maxProbSize)

(if (>= minProbSize maxProbSize) (smallReport f numTrials minProbSize) (begin

(smallReport f numTrials minProbSize) (briefReport f inc numTrials (inc minProbSize) maxProbSize))))

;; standard doubler and plus1 functions for advancing through report (define (double n) (* 2 n)) (define (plus1 n) (+ 1 n))
```

The largeAdd function from Example A-8 adds together a set of n numbers. The output generated by (briefReport largeAdd millionplus 30 1000000 5000000) is shown in Table A-2.

*Example A-8. largeAdd Scheme function*

```
;; helper method (define (millionplus n) ( + 1000000 n))

;; Sum numbers from 1..probSize (define (largeAdd probSize) (let loop ([i probSize] [total 0])
(if (= i 0) total (loop (sub1 i) (+ i total)))))
```

*Table A-2. Execution time for 30 trials of largeAdd*

| n | Execution time (ms) |
|---|---|
| **1,000,000** | 382.09 |
| **2,000,000** | 767.26 |
| **3,000,000** | 1155.78 |
| **4,000,000** | 1533.41 |
| **5,000,000** | 1914.78 |

## Reporting

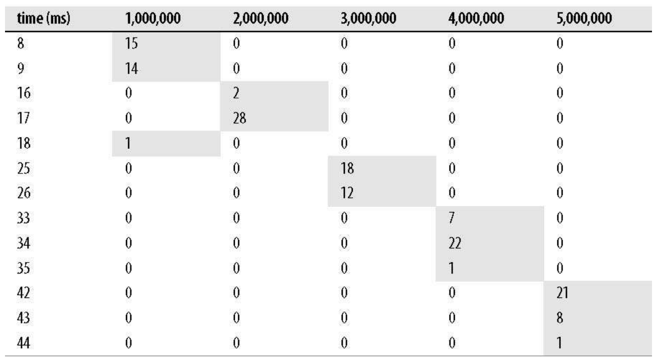
It is instructive to review the actual results when computed on the same platform, in this case a Linux 2.6.9-67.0.1.ELsmp i686 (this machine is different from the desktop PC and high-end computer mentioned earlier in this chapter). We present three tables (Tables A-3, A-5, and A-6), one each for Java, C, and Scheme. In each table, we present the millisecond results and a brief histogram table for the Java results.

*Table A-3. Timing results of 30 computations in Java*

| n | average | min | max | stdev | # |
|---|---|---|---|---|---|
| **1,000,000** | 8.5 | 8 | 18 | 0.5092 | 28 |
| **2,000,000** | 16.9643 | 16 | 17 | 0.1890 | 28 |
| **3,000,000** | 25.3929 | 25 | 26 | 0.4973 | 28 |
| **4,000,000** | 33.7857 | 33 | 35 | 0.4179 | 28 |
| **5,000,000** | 42.2857 | 42 | 44 | 0.4600 | 28 |

The aggregate behavior of Table A-3 is detailed in histogram form in Table A-4. We omit from the table rows that have only zero values; all nonzero values are shaded in the table.

*Table A-4. Individual breakdowns of timing results*



| time (ms) | 1,000,000 | 2,000,000 | 3,000,000 | 4,000,000 | 5,000,000 |
|---|---|---|---|---|---|
| 8 | 15 | 0 | 0 | 0 | 0 |
| 9 | 14 | 0 | 0 | 0 | 0 |
| 16 | 0 | 2 | 0 | 0 | 0 |
| 17 | 0 | 28 | 0 | 0 | 0 |
| 18 | 1 | 0 | 0 | 0 | 0 |
| 25 | 0 | 0 | 18 | 0 | 0 |
| 26 | 0 | 0 | 12 | 0 | 0 |
| 33 | 0 | 0 | 0 | 7 | 0 |
| 34 | 0 | 0 | 0 | 22 | 0 |
| 35 | 0 | 0 | 0 | 1 | 0 |
| 42 | 0 | 0 | 0 | 0 | 21 |
| 43 | 0 | 0 | 0 | 0 | 8 |
| 44 | 0 | 0 | 0 | 0 | 1 |

To interpret these results for Java, we turn to statistics. If we assume that the timing of each trial is independent, then we refer to the *confidence intervals* described earlier. If we are asked to predict the performance of a proposed run for *n*=4,000,000, then we can say that with 95.45% probability the expected timing result will be in the range [32.9499, 34.6215].

*Table A-5. Timing results of 30 computations in C*

| n | average | min | max | stdev | # |
|---|---|---|---|---|---|
| **1,000,000** | 2.6358 | 2.589 | 3.609 | 0.1244 | 28 |
| **2,000,000** | 5.1359 | 5.099 | 6.24 | 0.0672 | 28 |
| **3,000,000** | 7.6542 | 7.613 | 8.009 | 0.0433 | 28 |
| **4,000,000** | 10.1943 | 10.126 | 11.299 | 0.0696 | 28 |
| **5,000,000** | 12.7272 | 12.638 | 13.75 | 0.1560 | 28 |

In raw numbers, the C implementation appears to be about three times faster. The histogram results are not as informative, because the timing results include fractional milliseconds, whereas the Java timing strategy reports only integer values.

The final table contains the results for Scheme. The variability of the execution runs in the Scheme implementation is much higher than Java and C. One reason may be that the recursive solution requires more internal bookkeeping of the computation.

*Table A-6. Timing results of 30 computations in Scheme*

| n | average | min | max | stdev | # |
|---|---|---|---|---|---|
| **1,000,000** | 1173 | 865 | 1,274 | 7.9552 | 28 |
| **2,000,000** | 1921.821 | 1,824 | 2,337 | 13.1069 | 28 |
| **3,000,000** | 3059.214 | 2,906 | 3,272 | 116.2323 | 28 |
| **4,000,000** | 4040.607 | 3,914 | 4,188 | 81.8336 | 28 |
| **5,000,000** | 6352.393 | 6,283 | 6,452 | 31.5949 | 28 |

## Precision

Instead of using millisecond-level timers, nanosecond timers could be used. On the Java platform, the only change in the earlier timing code would be to invoke System.nanoTime() instead of accessing the milliseconds. To understand whether there is any correlation between the millisecond and nanosecond timers, the code was changed as shown in Example A-9.

*Example A-9. Using nanosecond timers in Java*

```
TrialSuite tsM = new TrialSuite(); TrialSuite tsN = new TrialSuite(); for (long len = 1000000; len <= 5000000; len += 1000000) {

for (int i = 0; i < 30; i++) { long nowM = System.currentTimeMillis(); long nowN = System.nanoTime(); long sum = 0; for (int x =
0; x < len; x++) { sum += x; } long endM = System.currentTimeMillis(); long endN = System.nanoTime();
```

*Example A-9. Using nanosecond timers in Java (continued)*

```
tsM.addTrial(len, nowM, endM); tsN.addTrial(len, nowN, endN); } }
```

Table A-3, shown earlier, contains the millisecond results of the timings, and Table A-7 contains the results when using the nanosecond timer. The clearest difference is that the standard deviation has shrunk by an order of magnitude, thus giving us much tighter bounds on the expected execution time of the underlying code. One can also observe, however, that the resulting timings still have issues with precision—note the large standard deviation for the *n*=5,000,000 trial. This large deviation corresponds with the "spike" seen in this case in Table A-3.

*Table A-7. Results using nanosecond timers*

| n | average | min | max | stdev | # |
|---|---|---|---|---|---|
| **1,000,000** | 8.4833 | 8.436 | 18.477 | 0.0888 | 28 |
| **2,000,000** | 16.9096 | 16.865 | 17.269 | 0.0449 | 28 |
| **3,000,000** | 25.3578 | 25.301 | 25.688 | 0.0605 | 28 |
| **4,000,000** | 33.8127 | 33.729 | 34.559 | 0.0812 | 28 |
| **5,000,000** | 42.3508 | 42.19 | 43.207 | 0.2196 | 28 |

Because we believe using nanosecond-level timers does not add sufficient precision or accuracy, we continue to use millisecond-level timing results within the benchmark results reported in the algorithm chapters. We also continue to use milliseconds to avoid giving the impression that our timers are more accurate than they really are. Finally, nanosecond timers on Unix systems are not yet standardized, and there are times when we wished to compare execution times across platforms, which is another reason why we chose to use millisecond-level timers throughout this book.

Why such variation among what should otherwise be a rather consistent behavior? Reviewing the data from Table A-3, there appear to be "gaps" of 15 or 16 milliseconds in the recorded trial executions. These gaps reflect the accuracy of the Java timer on the Windows platform, rather than the behavior of the code. These variations will appear whenever System.currentTimeMillis() is executed, yet the values are significant only when the base execution times are very small (i.e., near 16 milliseconds).

The Sun engineers who developed Java are aware of the problem of timers for the Windows platform, and have no immediate plans to resolve the issue (and this has been the situation for nearly six years now). See *http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4423429* for clarification.

If you enjoyed this excerpt, buy a copy of **Algorithms in a Nutshell** .

Benchmarking