JProfile102: A System for Experimental Analysis of Algorithms

Tanin Krajangthong and Somchai Prasitjutrakul

Department of Computer Engineering
Chulalongkorn University
Bangkok, Thailand

***Abstract*— This paper presents a system called JProfile102 used for experimental analysis of algorithms. The system accepts algorithms implemented as Java methods along with experiment parameters specifying characteristics and sizes of input data. The objective is to count the number of times each source code instruction gets executed during the experiments. Source-code instrumentation technique is used by parsing the source code to obtain its associated abstract syntax tree, traversing the tree, inserting extra counting instructions at instruction nodes and finally transforming the tree back into an instrumented source code ready for experiments. To correctly handle method calls in the code (especially recursive calls), a separate run-time call stack has to be implemented to keep non-duplicated counter IDs in the call chain. All profiling data obtained from the experiments are summarized and reformatted into an HTML page with two views. One shows a histogram of execution counts. The other shows line plots of selected instruction counts vs. input data size to visualize efficiency behavior of the algorithm. The system is embedded into a Java IDE and effectively used as a teaching aid in several algorithm analysis courses.**

Keywords- analysis of algorithm, experimental analysis

I. Introduction

An algorithm is a step by step instruction describing a procedure for solving a problem. We analyze an algorithm in order to determine the amount of computational resources (usually time and memory space) needed by the algorithm to solve its corresponding problem. Algorithms are designed to work with any size of problem instances. Therefore, algorithms are analyzed to show the relationship of instance sizes, instance characteristics and the amount of computational resources expressing the algorithm efficiencies [1]. Because a problem can normally be solved by different algorithms, algorithm analysis must be done to compare their efficiencies and choose the most efficient one.

Since algorithms are usually not machine-specific, the analysis of time complexity is carried out by counting the number of executions of basic operations (rather than measuring the execution time) required as a function of input sizes. This counting can be done mathematically or experimentally [2]. Sums, recurrence relations and generating functions are among mathematical tools widely used to mathematically model and analyze algorithms [1]. Some algorithms can be analyzed to have the exact closed form of the functions, some algorithms can only be analyzed asymptotically to determine tight growth rate bounds of the functions however, some algorithms are hard to analyze without using advanced mathematical tools [3].

Another approach to analyze algorithms is to do it experimentally. This is done by implementing the algorithm, executing it using specific set of inputs and measuring computational resources under interests. Not only the approach can be used to verify the mathematical analysis result (as a posteriori) and to compare efficiencies of actual algorithm implementations, it can also be used to gain insight behavior of the algorithm when it is hard to analyze mathematically [4].

In this paper, we present a system called JProfile102 that facilitates the process of experimental analysis of algorithms. The system accepts an algorithm implemented in a Java method along with some parameters specifying experimental behavior. JProfile102 addresses two problems in algorithm analysis. First, counting the number of execution steps of *all* operations is too time-consuming for human. Therefore, one normally needs to specify “hotspot” (most-frequently executed) operations before doing the analysis (as our previous system JProfile101 [5] requires users to manually specify operations of interest) but this is not always an easy task. Hence, JProfile102 simply inserts instrumenting instructions to each operation to monitor all operations in the algorithm automatically. Using all instruction count data, we are able to determine the hotspot operations and further analyze the algorithm behavior.

Second, there are some housekeeping routines that have to be done repeatedly when one experimentally analyzes algorithms, for examples, setting and updating instrumenting counters, feeding various input data to the algorithm and, a very important task, summarizing and presenting profiling information in graphical forms. These routines are built into JProfile102 to facilitate the experiments by using experimental annotation code to instruct experimental engine to do as specified.

This paper is organized as follows. Section II presents related work in experimental analysis tools and methodologies. Section III elaborates the internal structure, source code instrumentation and output results of JProfile102. Some usage examples that we have effectively used are presented in Section IV and the paper concludes in Section V.

II. Related Work

One of the widely used tools for detail analysis of resource consumption during program execution is Profiler [6]. It helps developers to identify program hotspots in order to optimize the part of program that affects overall performance. In Java platform, there are JVMPI [7] and JVMTI [8] providing services for profiling Java execution. In [9], they present Timing API that helps develop tools for doing experimental analysis. It measures the actual execution times of various algorithms with the same input data set in order to compare their efficiencies. The system provides modules that require some integration work in order to perform experiments.

Rather than measuring the execution time, profiling can be done by code instrumentation that inserts counting instructions in codes to count the number of times the codes are executed. JP [10] and ByCounter [11] do the work in bytecode level. JP instruments the bytecodes by performing basic block analysis to compute control graph and then inserting bytecode counting instructions within the basic blocks. Since different bytecode instructions have different execution time, ByCounter counts the number of executed bytecodes separately for each opcode to better estimates the total running time (each bytecode is one byte in length, so there are at most 256 counters). These profilers provides timing statistics for total execution and for each method (or function), but not at the specific instruction within the method.

Another approach is to instrument at source-code level used in JProfile101 [5]. Users are required to manually insert counting instructions at the source code along with experiment parameters. Fig. 1 shows an example of instrumenting Bubble sort algorithm written in Java using a **`//@Profiler.count++`** line to count the number of comparisons done in the following line. The method is tagged with the system's **`@Profile`** annotation to specify an input data generator and data sizes used during experiments. The `Profiler.begin()` at the main method starts the JProfile101 controller engine which automatically performs experiments. Fig. 2 is the line graph showing the counting results and their associated input data sizes. Since JProfile101 requires users to manually tag specific instructions to be instrumented, this is not an easy task to do for novices and sometimes hard to do for complicated algorithms.

III. JProfile102

JProfile102, presented in this paper, utilizes a core module of JProfile101 by adding pre- and post-processing steps as shown in Fig. 3. In the pre-processing step (Instrument), the provided source code of the algorithm is instrumented by automatically insert counting instructions for all instructions in the source code. The instrumented code is then fed to JProfile101 to perform experiments according to the specified input data generators and sizes. All the profiling data obtained from JProfile101 are read back to the post-processing step (Present) to summarize and present results in two different views (Fig. 4). One is a count histogram for instructions of the source code. (The histogram is shown as horizontal bars overlaid on each line of source codes.) This view lets the user identify the hotspot instruction in the algorithm that can mathematically be analyzed in details, if desired. The other view shows line plots of execution counts for any selected instructions as a function of input sizes. (Fig. 4 shows two line plots of the number of data comparisons and the number of data movements in BubbleSort algorithm.) This gives us the growth rate of functions to better understand behavior of the algorithm efficiency.

```
import jprofile101.*;
public class BubbleSort {
  @Profile(
    inputs = {Util.RandomIntArray.class},
    from=0, to=100, step=2, repeat=30
  )
  static void bubbleSort(int[] d){
    for (int k=d.length-1; k>0; k--) {
      for (int i = 0; i < k; i++) {
        Profiler.count++;    // <---------
        if (d[i] > d[i+1]) {
          int t = d[i];
          d[i] = d[i+1];
          d[i+1] = t;
        }
      }
    }
  }
  public static void main(String[] args) {
    Profiler.begin("BubbleSort");
  }
}
```

Figure 1. BubbleSort with counting instruction and experiment parameters

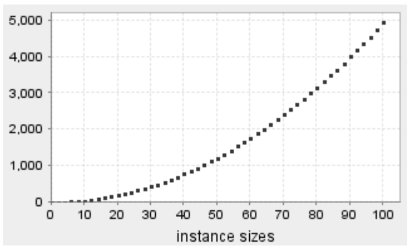


Figure 2. Couting results from the BubbleSort experiments in Fig. 1

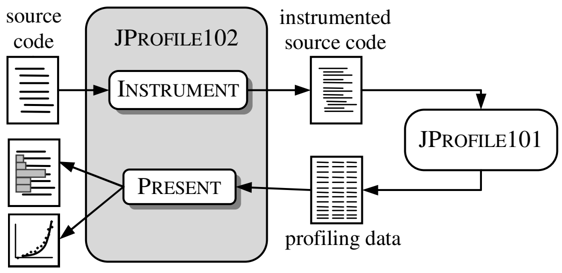
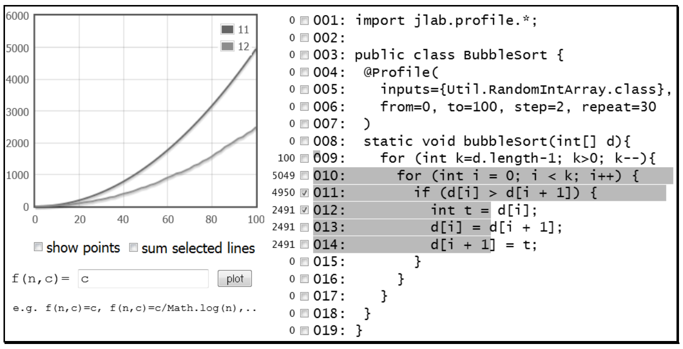


Figure 3. JProfile102 Structure.



```
   0 ☐ 001: import jlab.profile.*;
   0 ☐ 002:
   0 ☐ 003: public class BubbleSort {
   0 ☐ 004:  @Profile(
   0 ☐ 005:    inputs={Util.RandomIntArray.class},
   0 ☐ 006:    from=0, to=100, step=2, repeat=30
   0 ☐ 007:  )
   0 ☐ 008:  static void bubbleSort(int[] d){
 100 ☐ 009:    for (int k=d.length-1; k>0; k--){
5049 ☐ 010:      for (int i = 0; i < k; i++) {
4950 ☑ 011:        if (d[i] > d[i + 1]) {
2491 ☑ 012:          int t = d[i];
2491 ☐ 013:          d[i] = d[i + 1];
2491 ☐ 014:          d[i + 1] = t;
   0 ☐ 015:        }
   0 ☐ 016:      }
   0 ☐ 017:    }
   0 ☐ 018:  }
   0 ☐ 019: }
```

Figure 4. Two output views of JProfile102.

A. Source Code Instrumentation

The big advantage of JProfile102 over JProfile101 is that users do not have to specify the hotspot instruction in the algorithm. JProfile102 just instruments every instruction and the most-frequently executed instruction after experiments will become the hotspot. Source code instrumentation is done by parsing the source code into an *abstract syntax tree* (AST) [12], traversing the AST, inserting a counting instruction at each instruction node of the tree along the traversal path and turning the AST back to an instrumented-version of the original source code. Fig. 5 shows `heapSort` method along with its associated AST (a simplified version of AST is shown here). Each node in the AST represents a construct in the source code. For example, the node labeled `12:for` represents the `for` construct at line #12. Each child node represents each instruction in its block, e.g., the `12:for` node has two children, `13:swap` and `14: heapify`.

```
08: static void heapSort(int[] d) {
09:   int n = d.length;
10:   for (int k=n-1; k>=0; k--)
11:     heapify(d, k, size);
12:   for (int k=n-1; k>0; k--) {
13:     swap(d, 0, k);
14:     heapify(d, 0, --n);
15:   }
16: }
17: static void swap(int[] d, int a, int b) {
18:   int t = d[a]; d[a] = d[b]; d[b] = t;
19: }
20: static void heapify(int[] d, int p, int n) {
      ...
28: }
```

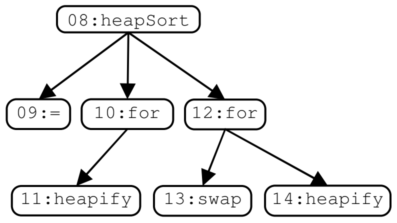


Figure 5. An abstract syntax tree of the `heapSort` method.

Counting instructions are easily inserted to the left of each node as traversing the AST in preorder fashion. We keep all instruction counters in an array and use the preorder numbers of the original AST nodes as indices of the array. Fig 6 shows the instrumented AST where nodes colored in gray and labeled with *k*++ represent the increment of the counter #*k*.

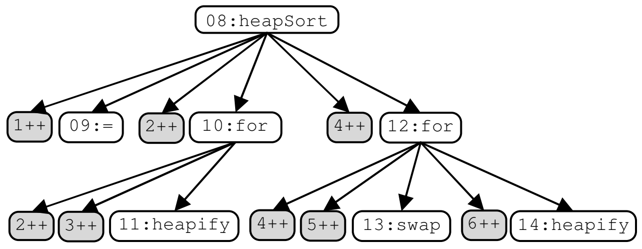


Figure 6. The instrumented AST.

Two special cases must be taken into consideration, First is the `for` and `while` statements. We need to increment its counter both before and after the statements to correctly instrument the conditions in `for` and `while`. Otherwise the `for` and `while` will be counted only once when entering the loop but not when testing the condition (for examples, see the `2++` and `4++` nodes in Fig. 6).

Another special case is method calls. Any instruction executed in a method has to increase both its associated counter and the counter of the instruction calling that method. For `heapSort` shown in Fig. 5, executing instructions inside `heapify` method must also increment the counter of `11:heapify` (which is `3++`) if this `heapify` has been called from line #11, or increment the counter of `14:heapify` (which is `6++`) if called from line #14. Actually, we need to increment all counters in the caller chain. This is done by having a call stack to keep the caller counter IDs, pushing the current caller counter ID to the stack at method entry and popping the stack at all method exits. Fig. 7 shows the instrumented `heapSort` method. The `profiler.inc(k)` shown in the code is to increment counter #*k*, `profiler.enterMethod()` is inserted at the beginning of method and `profiler.exitMethod()` is inserted before every `return` statement including the closed brace of any return-void method.

```
08: static void heapSort(int[] d) {
      profiler.enterMethod();
09:   profiler.inc(1);     int n = d.length;
10:   profiler.inc(2);     for (int k=n-1; k>=0; k--)
      { profiler.inc(2);
11:     profiler.inc(3);     heapify(d, k, size);
      }
12:   profiler.inc(4);     for (int k=n-1; k>0; k--)
      { profiler.inc(4);   {
13:     profiler.inc(5);     swap(d, 0, k);
14:     profiler.inc(6);     heapify(d, 0, --n);
15:                        }
      }
      profiler.exitMethod();
16: }
17: static void swap(int[] d, int a, int b) {
      profiler.enterMethod();
      ...
      profiler.exitMethod();
19: }
20: static void heapify(int[] d, int p, int n) {
      profiler.enterMethod();
      ...
      profiler.exitMethod();
28: }
```

Figure 7. Instrumented `heapSort` method.

Fig. 8 shows detail operations of the `inc`, `enterMethod` and `exitMethod` methods. We keep the lastest counter ID which just gets incremented (in `inc`) so that it will be pushed to the stack (in `enterMethod`) if the statement is a method call. Special care must be taken for recursive calls to avoid multiple counting the recursive call statements. In `enterMethod`, we do not push in the stack any caller counter ID that is already in the stack. However, something must be pushed so that it will get popped in `exitMethod`. In this case, we have a dummy counter ID to get pushed. And also in `inc`, we need to avoid double increment counter of the statement whose counter is in the stack.

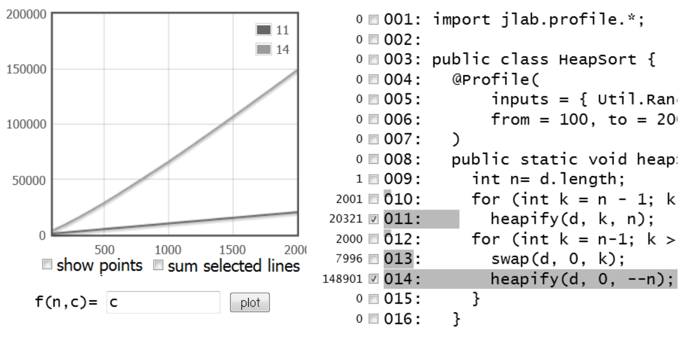
```
class Profiler {
 private static final int DUMMY = 0;
 private int [] stack;
 private int tos = -1;
 private int curCounterID = DUMMY;
 private int [] counters;
 // ... constructors are not shown here ...
 public void inc(int k) {
   // increment all counters in stack
   curCounterID = k;
   boolean kInStack = false;
   for (int i = 0; i <= tos; i++) {
     counters[stack[i]]++;
     if (curCounterID == stack[i]) kInStack = true;
   }
   // if k is in stack, don't count twice
   if (!kInStack) counters[k]++;
 }
 public void enterMethod() {
   boolean recursiveCall = false;
   for (int i=0; !recursiveCall && i <= tos; i++) {
     recursiveCall = (stack[i] == curCounter);
   }
   // push DUMMY ID, if it's a recursive call
   stack[++tos]=recursiveCall ? DUMMY :curCounterID;
 }
 public void exitMethod() {
   tos--; // pop the stack
 }
 ...
```

Figure 8. Details of methods used in profiling.

B. *Presenting Profiling Data*

After source-code-instrumentation phase, the code is then compiled and executed repeatedly using different input data and sizes as specified by experiment parameters using `@Profile` method annotation. All the profiling data of all instructions in the source code are summarized and reformatted into an HTML page with two views, one is an instruction count histogram for each instruction of the source code, the other is line plots of execution counts for selected instructions as a function of input sizes. Users can select specific source-code lines of interest on the histogram view to update the line-plot view for comparison.

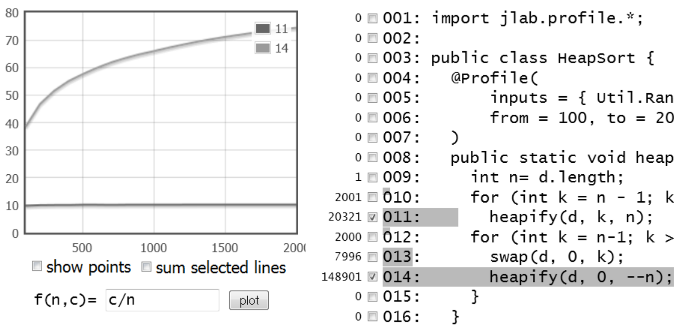
Fig. 9 shows the two result views from an experiment of `heapSort` algorithm. The algorithm consists of two main steps, building a heap (lines 10 – 11) and repeatedly removing the max (lines 12 – 15). Both steps utilize the `heapify` method and look almost identical in the code. However, we can observe either the histogram or the line-plots to conclude that the second step takes significantly more time than the first one.



```
     0 ☐ 001: import jlab.profile.*;
     0 ☐ 002:
     0 ☐ 003: public class HeapSort {
     0 ☐ 004:   @Profile(
     0 ☐ 005:       inputs = { Util.Ran
     0 ☐ 006:       from = 100, to = 20
     0 ☐ 007:   )
     0 ☐ 008:   public static void heap
     1 ☐ 009:     int n= d.length;
  2001 ☐ 010:     for (int k = n - 1; k
 20321 ☑ 011:       heapify(d, k, n);
  2000 ☐ 012:     for (int k = n-1; k >
  7996 ☐ 013:       swap(d, 0, k);
148901 ☑ 014:       heapify(d, 0, --n);
     0 ☐ 015:     }
     0 ☐ 016:   }
```

Figure 9. The two result pages from JPROFILE102.

By looking at the two line plots in Fig. 9, we seem to believe that both lines grow linearly with different slopes. In this case, JPROFILE102 lets us enter formulas to re-compute the values for *y*-axis so that the lines may look differently. Initially, the system uses execution counts as values for the *y*-axis. This is specified in a text box at the lower left of Fig. 9 as `f(n,c) = c` where `n` is data size (*x*-axis) and `c` is execution count. If we change the formula, e.g., to `f(n,c) = c/n`, Fig.9 becomes Fig. 10. The lower line for `c/n` is constant which means that the original line in Fig. 9 must be linear whereas the upper line grows like log function in Fig.10. This leads us to believe that the original one in Fig. 9 grows as an *n* log *n* function (which can be verified analytically to be true for `heapSort`).



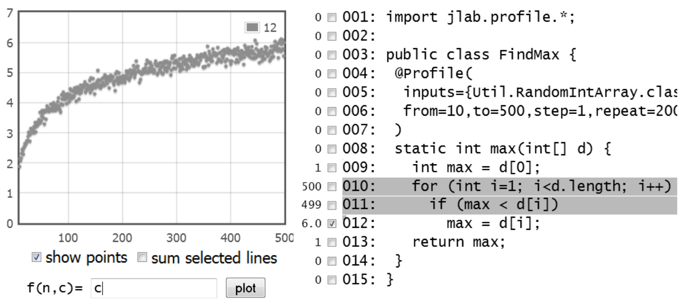
```
     0 ☐ 001: import jlab.profile.*;
     0 ☐ 002:
     0 ☐ 003: public class HeapSort {
     0 ☐ 004:   @Profile(
     0 ☐ 005:       inputs = { Util.Ran
     0 ☐ 006:       from = 100, to = 20
     0 ☐ 007:   )
     0 ☐ 008:   public static void heap
     1 ☐ 009:     int n= d.length;
  2001 ☐ 010:     for (int k = n - 1; k
 20321 ☑ 011:       heapify(d, k, n);
  2000 ☐ 012:     for (int k = n-1; k >
  7996 ☐ 013:       swap(d, 0, k);
148901 ☑ 014:       heapify(d, 0, --n);
     0 ☐ 015:     }
     0 ☐ 016:   }
```

Figure 10. Users can supply a formula to recompute values of the y-axis.

IV. USAGE EXAMPLES

We have integrated JPROFILE102 into JLAB [13], an integrated Java program development environment used for teaching programming. Users can edit, test and profile Java programs implementing algorithms of interest. By pressing CTRL + SHIFT + F5, the system will automatically activate JPROFILE102. If no `@Profile` annotation is provided, the program will run and be profiled starting at its `main` method and only a histogram of instruction execution counts is presented. Otherwise the method annotated with `@Profile` will be called and profiled with different input data as specified. This section presents some small-size examples that we have used effectively in teaching algorithm analysis and led to more class discussions.

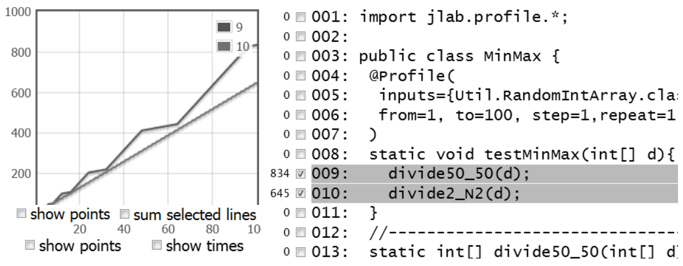
FindMax: It is easy to show that finding a maximum value in an array of size *n* requires *n* – 1 comparisons. However, it is not obvious that how many times the `max` variable in the program changes its value. It can range from 1 to *n* depending on the input array. Fig. 11 shows a plot of the number of such changes when finding the maximum in random-value arrays that grows like a log *n* function.



```
  0 ☐ 001: import jlab.profile.*;
  0 ☐ 002:
  0 ☐ 003: public class FindMax {
  0 ☐ 004:   @Profile(
  0 ☐ 005:     inputs={Util.RandomIntArray.clas
  0 ☐ 006:     from=10,to=500,step=1,repeat=20
  0 ☐ 007:   )
  0 ☐ 008:   static int max(int[] d) {
  1 ☐ 009:     int max = d[0];
500 ☐ 010:     for (int i=1; i<d.length; i++)
499 ☐ 011:       if (max < d[i])
6.0 ☑ 012:         max = d[i];
  1 ☐ 013:     return max;
  0 ☐ 014:   }
  0 ☐ 015: }
```

Figure 11. The `max` variable changes its value about log *n* times on average.

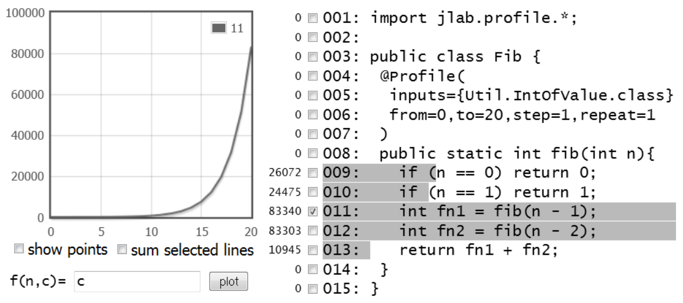
FindMinAndMax: Finding both min and max values of an array can be done by dividing the array into two parts, recursively finding min and max of each part and then comparing the results from both parts to obtain the min and max of the bigger part. We usually divide input instances in half when using divide and conquer techniques. However, Fig. 12 shows that dividing the array in half (the upper zigzag line) takes more comparisons than dividing it into 2 elements on the left part and the rest on the right part (the lower straight line).



```
  0 ☐ 001: import jlab.profile.*;
  0 ☐ 002:
  0 ☐ 003: public class MinMax {
  0 ☐ 004:  @Profile(
  0 ☐ 005:   inputs={Util.RandomIntArray.cla
  0 ☐ 006:   from=1, to=100, step=1,repeat=1
  0 ☐ 007:  )
  0 ☐ 008:  static void testMinMax(int[] d){
834 ☑ 009:    divide50_50(d);
645 ☑ 010:    divide2_N2(d);
  0 ☐ 011:  }
  0 ☐ 012:  //--------------------------------
  0 ☐ 013:  static int[] divide50_50(int[] d
```

Figure 12. Dividing in half does more work than dividing in 2 and *n*-2 (details of the two methods are not shown here).

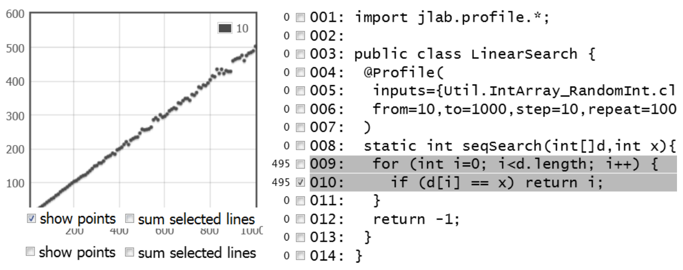
FibonacciNumbers: Writing a recursive code for the well-known Fibonacci recurrence is very straightforward. However, due to the nature of overlapping subproblems, the code runs very slowly. Fig. 13 shows an exponential growth-rate of the execution time when finding Fibonacci numbers recursively.



```
    0 ☐ 001: import jlab.profile.*;
    0 ☐ 002:
    0 ☐ 003: public class Fib {
    0 ☐ 004:  @Profile(
    0 ☐ 005:   inputs={Util.IntOfValue.class}
    0 ☐ 006:   from=0,to=20,step=1,repeat=1
    0 ☐ 007:  )
    0 ☐ 008:  public static int fib(int n){
26072 ☐ 009:    if (n == 0) return 0;
24475 ☐ 010:    if (n == 1) return 1;
83340 ☑ 011:    int fn1 = fib(n - 1);
83303 ☐ 012:    int fn2 = fib(n - 2);
10945 ☐ 013:    return fn1 + fn2;
    0 ☐ 014:  }
    0 ☐ 015: }
```

Figure 13. Computing Fibonacci numbers recursively takes exponential times.

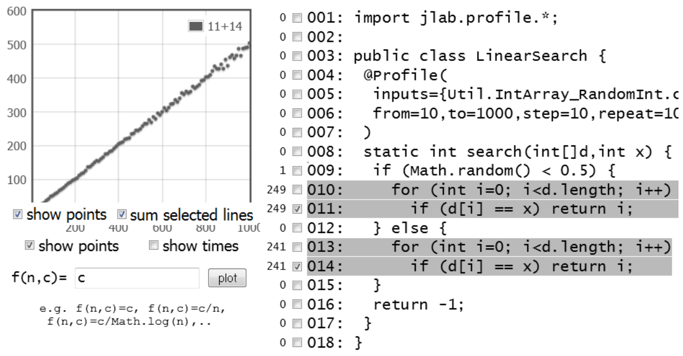
RandomizedLinearSearch: Most programmers seem to search sequentially from left to right in an array without hesitation. JPROFILE102 shows that on average, it compares about a half of array elements in successful search as shown in Fig. 14.



```
  0 ☐ 001: import jlab.profile.*;
  0 ☐ 002:
  0 ☐ 003: public class LinearSearch {
  0 ☐ 004:  @Profile(
  0 ☐ 005:   inputs={Util.IntArray_RandomInt.cl
  0 ☐ 006:   from=10,to=1000,step=10,repeat=100
  0 ☐ 007:  )
  0 ☐ 008:  static int seqSearch(int[]d,int x){
495 ☐ 009:   for (int i=0; i<d.length; i++) {
495 ☑ 010:     if (d[i] == x) return i;
  0 ☐ 011:   }
  0 ☐ 012:   return -1;
  0 ☐ 013:  }
  0 ☐ 014: }
```

Figure 14. Simple linear search compares about a half of the list

Questions may arise, e.g., "why don't we search it the other way around?" or "why don't we search it either way, depending on the outcome of a coin toss?" A quick analysis shows that with the randomized linear search, the worst expected number of comparisons is *n*/2 when the target element is at the middle of the array. This might lead us to believe that the randomized version may improve the number of comparisons on average. However, a quick experiment with JPROFILE102 showed (in Fig. 15) that a randomly choosing between a left-to-right and right-to-left searches compares about the same average number of probes needed in the simple left-to-right search. (In this example, we need to sum the number of data comparison instructions at line #11 and #14. This can be done by selecting both lines at the source code and checking the "sum selected lines" checkbox on the left as shown in Fig. 15.)



```
  0 ☐ 001: import jlab.profile.*;
  0 ☐ 002:
  0 ☐ 003: public class LinearSearch {
  0 ☐ 004:  @Profile(
  0 ☐ 005:   inputs={Util.IntArray_RandomInt.c
  0 ☐ 006:   from=10,to=1000,step=10,repeat=10
  0 ☐ 007:  )
  0 ☐ 008:  static int search(int[]d,int x) {
  1 ☐ 009:   if (Math.random() < 0.5) {
249 ☐ 010:     for (int i=0; i<d.length; i++)
249 ☑ 011:       if (d[i] == x) return i;
  0 ☐ 012:   } else {
241 ☐ 013:     for (int i=0; i<d.length; i++)
241 ☑ 014:       if (d[i] == x) return i;
  0 ☐ 015:   }
  0 ☐ 016:   return -1;
  0 ☐ 017:  }
  0 ☐ 018: }
```

Figure 15. Randomized linear search probes about the same as the simple one.

V. CONCLUSION

We have presented JPROFILE102, a system that can be used in experimental analysis of algorithms. Rather than let users manually specify operations of their interest and then perform the instruction profiling as done in our previous work in JPROFILE101. JPROFILE102 automatically instruments every source-code level instructions and then performs the profiling of all the instructions. These profiling data are useful not only for identifying hotspots of the algorithm, but also for studying algorithm behavior. Experiments can be set to execute the code repeatedly with different sizes of input data. The execution counts are plotted against data sizes to see the growth rate of execution times of each instruction in the code. This gives us an estimate of algorithm efficiency so that we can further mathematically analyze. JPROFILE102 is embedded into JLAB, a simple and small Java IDE, to quickly do experimental analysis after correctly implementing algorithms of interest. We have effectively used the system as a teaching aid in several algorithm analysis courses. The system is available for download at http://www.cp.eng.chula.a.th/~somchai/JLab

REFERENCES

[1] R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[2] D. Johnson, A Theoretician's Guide to the Experimental Analysis of Algorithms, 1-36, November. 25, 2001.

[3] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2003.

[4] G. Brassard and P. Bratley, Fundamentals of Algorithmics, Prentice Hall, 1996.

[5] W. Nilla-or and S. Prasitjutrakul, *JProfile101 : Controller for Experimental Analysis of Algorithms*, NCIT2009, National Conference on Information Technology, 2009.

[6] S. Liang and D.Viswanathan, Comprehensive Profiling Support in the Java Virtual Machine, The USENIX Association, 1999.

[7] Oracle Inc. Java Virtual Machine Profiler Interface [Online]. Available: http://docs.oracle.com/javase/1.5.0/docs/guide/jvmpi/jvmpi.html, 2011

[8] Oracle Inc. Java Virtual Machine Tool Interface [Online]. Available: http://docs.oracle.com/javase/1.5.0/docs/guide/jvmti/jvmti.html, 2011

[9] A. Duffy and T. Dowling, *Algorithm Analysis,* National University of Ireland, Maynooth, Ireland, Dept. of CS. Technical Report, 2003.

[10] W. Binde and J. Hulaas, *Exact and Portable Profiling for the JVM Using Bytecode Instruction Counting*, ENTCS (Electronic Notes in Theoretical Computer Science), 164(3), 45-64, 2006.

[11] M. Kuperberg, M. Krogmann and R. Reussner, *ByCounter : Portable Runtime Counting of Bytecode Instructions and Method Invocations*, ETAPS 11th Euro. Joint Conf. on Theory and Practice of Software, 2008.

[12] A. Tucker, *Programming Languages: Principles and Paradigms*, 2nd ed., McGraw-Hill, 2007.

[13] S. Prasitjutrakul, JLab : An Integrated Java Program Development Tool [Online]. Avaiable http://www.cp.eng.chula.ac.th/~somchai/JLab