SOCIETY FOR SCIENCE AND EDUCATION
UNITED KINGDOM

TMLAI TRANSACTIONS ON
MACHINE LEARNING AND ARTIFICIAL INTELLIGENCE

Volume 5 No. 2
ISSN 2054-7390

Performance Evaluation of Some Selected Sorting Algorithms by the Use of Halstead Complexity Metrics

A.O Afolabi
Department of Computer Science and Engineering,
Ladoke Akintola University of Technology, Ogbomoso. Nigeria
aoafolabi@lautech.edu.ng

ABSTRACT

Complexity is developed to demonstrate feasible metrics for process obtaining objectives and quantifiable measurement, which may have numerous valuable applications in schedule and budget planning, cost estimation and optimal personnel task assignments a particular software. Also it aids the developer and practitioners in evaluating the software complexity due to its simplicity, which serves both as an analyzer and as a predicator in quantitative software engineering. In this research work, the Halstead complexity metrics were applied to three sorting algorithms and C-Sharp programming language was used to implement them. The results shows the variant performances of sorting algorithms and the best algorithm that can perform better in a defined system of application.

Keyword: algorithm, Halstead Measure, software metrics, software complexity

1 Introduction

There are a number of important practical and theoretical reasons for analyzing algorithms. The principal reason is that we need to obtain estimates or bounds on the storage or run time which an algorithm will need to successfully process a particular input. Computer time and memory are relatively scarce resources which are often simultaneously sought by many users. It is advantageous to avoid runs that are aborted because of insufficient time. One would like to predict such things with pencil and paper in order to avoid disastrous runs. A good analysis is also capable of finding bottlenecks in our programs, that is, sections of a program where most of the time is spent. Computational complexity theory investigates the problems related to the amount of resources required for the execution of algorithms (e.g. execution time), and the inherent difficulty in providing efficient algorithms for specific computational problems. A typical question of the theory is, ‘‘as the size of the input to an algorithm increases, how do the running time and memory requirements of the algorithm change and what are the implications and ramifications of that change.’’ In other words, the theory, among other things, investigates the scalability of computational problems and algorithms. In particular, the theory places practical limits on what computers can accomplish. The time complexity of a problem is the number of steps that it takes to solve an instance of the problem as a function of the size of the input (usually measured in bits), using the most efficient algorithm. Intuitively, we consider the example of an instance that is n bits long that can be solved in n2 steps. In this case we say the problem has a time complexity of order n2.

DOI: 10.14738/tmlai.52.2793
Publication Date: 11th April, 2017
URL: http://dx.doi.org/10.14738/tmlai.52.2793

Most of the available metrics cover only certain features of a language for example, when line of code is applied, then only size will be considered, When McCabe complexity is applied, the control flow of the program will not be covered and also determined the complexity base on the number of control paths created by the code. While Halstead based is approached on the Mathematical relationship among the number of variables. Most of the available metric consider the cognitive characteristics in calculating the complexity of the code which directly affect the cognitive of the program.

1.1 Complexity Concepts

In system especially for software, the word complexity was probably first used for what is called computational or time complexity. For an example, the task of searching for a sorted list of length *n* for the single item has complexity 0 (log n) meaning that any logarithm giving a solution to the task will worst case needed and the order log n pair wise comparisons to solve the task for large n. The task to sort such a list has computational complexity *O (nlogn)* and is thus a more complex task. These complexities characterize the class of problems to be solved and give the least possible growth in computation times as a function of the growth in problem size. In addition, to this each method designed to solve problems belonging to some class has its own complexity which of course cannot be less than the complexity of the corresponding problem class.

Complexity has also been used to characterize software. According to (McCall, 1977) complexity "relates to data set relationship, data structures, data flow and the algorithm being implemented". It measures the degree of decision making logic within the system. Beizer states that "using only our intuitive notion of software complexity, we expect that more complex software will cost more to build and test and will have more bugs". (Beizer, 1984). (Tourlakis,1984) distinguish between two classes of complexity measure, that is dynamic and static. Dynamic complexity measures the amount of 'resources' consumed during a computation. Static complexity measures on the other hand may be size e.g. (program length) or the structural complexity e.g. (level of nesting of do-loops) of an algorithm's description.

2 Methodology

The technique used in this project is Halstead software metrics and three sorting algorithms were implemented with Microsoft C sharp programming language. The software science developed by M. H. Halstead principally attempts to estimate the rate of program errors and the effort invested in program maintenance. Halstead Metrics are used in project scheduling and reporting, in that they measure the overall quality of the program and rate the effort invested in its development. They are easy to calculate and do not require in-depth analysis of program structure. Halstead Metrics are based on the measurement and interpretation of tokens. A token is the smallest unit of text recognized by the compiler. The metrics analyzer considers the following tokens as operators of Halstead Metrics:

The following tokens are considered Halstead Operands:

- Identifiers,
- Typedef name types,
- Numerical constants,
- Strings.

A label and its terminating colon do not count, as they are comments according to Halstead. In addition, function headings, including the initializations included in them, do not count.

2.1.1 Halstead Parameters

The basic parameters are:

a) Unique operators ($n1$)the number of unique occurrences of Halstead Operators in the program,
b) Unique operands ($n2$) the number of unique occurrences of Halstead Operands in the program,
c) Total operators ($N1$) the total number of Halstead Operators,
d) Total operands ($N2$) the total number of Halstead Operands.

The *derived parameters* of Halstead Metrics are of great importance to the interpretation of code complexity. Basic parameters are used to calculate them.

2.1.2 Halstead Program Length

The total number of operator occurrences and the total number of operand occurrences.

$$N = N1 + N2 \quad (1)$$

2.1.3 Halstead Vocabulary

The total number of unique operator and unique operand occurrences.

$$n = n1 + n2 \quad (2)$$

2.1.4 Program Volume

Proportional to program size, represents the size, in bits, of space necessary for storing the program. This parameter is dependent on specific algorithm implementation. The parameters V, N, and the number of lines in the code are shown to be linearly connected and equally valid for measuring relative program size.

$$V = N * log2(n) \quad (3)$$

2.1.5 Program Difficulty

This parameter shows how difficult to handle the program is.

$$D = (\frac{n1}{2}) * (\frac{N2}{n2}) = \quad (4)$$

2.1.6 Programming Effort

Measures the amount of mental activity needed to translate the existing algorithm into implementation in the specified program language.

$$E = V * D \quad (5)$$

2.1.7 Language Level

Shows the algorithm implementation program language level. The same algorithm demands additional effort if it is written in a low level program language. For example, it is easier to program in Pascal than in Assembler.

$$L' = \frac{V}{D}/D = \tag{6}$$

2.1.8 Intelligence Content

Determines the amount of intelligence presented (stated) in the program This parameter provides a measurement of program complexity, independently of the program language in which it was implemented.

$$I = \frac{V}{D} \tag{7}$$

2.1.9 Programming Time

Shows time (in minutes) needed to translate the existing algorithm into implementation in the specified program language.

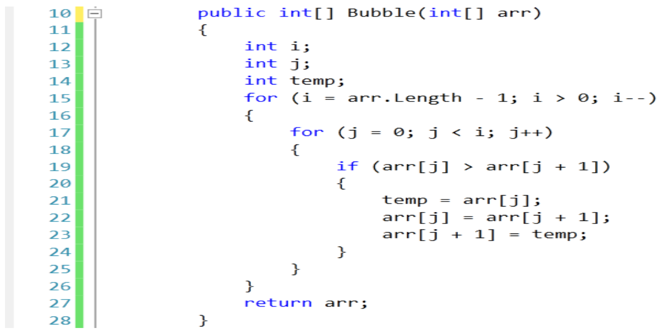
$$T = E/(F * S) \tag{8}$$

The concept of the processing rate of the human brain, developed by the psychologist John Stroud, is also used. Stoud defined a moment as the time required by the human brain requires to carry out the most elementary decision. The Stoud number S is therefore Stoud's moments per second with: 5 <= S <= 20. Halstead uses 18.

Stroud number S = 18 moments / second seconds-to-minutes factor f = 60

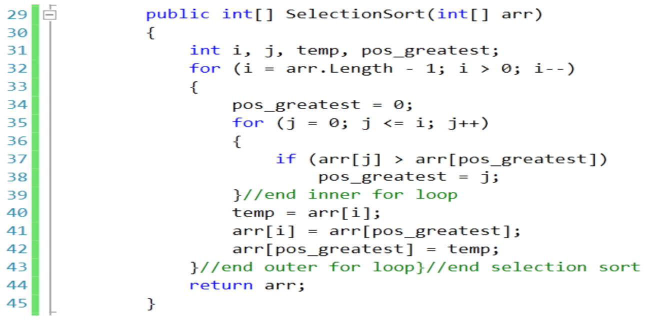
3 Results and Discussion

For the successful implementation of this project, three sorting algorithms were considered and these algorithms were implemented in Microsoft C# programming language.



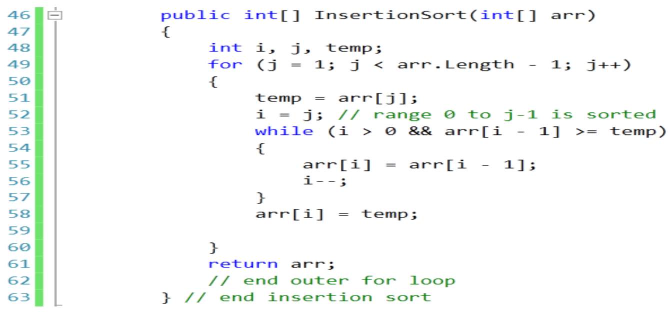
```
public int[] Bubble(int[] arr)
{
    int i;
    int j;
    int temp;
    for (i = arr.Length - 1; i > 0; i--)
    {
        for (j = 0; j < i; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    return arr;
}
```

Figure 1: Interface describing the implementation bubble sort



```
public int[] SelectionSort(int[] arr)
{
    int i, j, temp, pos_greatest;
    for (i = arr.Length - 1; i > 0; i--)
    {
        pos_greatest = 0;
        for (j = 0; j <= i; j++)
        {
            if (arr[j] > arr[pos_greatest])
                pos_greatest = j;
        }//end inner for loop
        temp = arr[i];
        arr[i] = arr[pos_greatest];
        arr[pos_greatest] = temp;
    }//end outer for loop}//end selection sort
    return arr;
}
```

Figure.2: Interface describing the implementation selection sort



```
        public int[] InsertionSort(int[] arr)
        {
            int i, j, temp;
            for (j = 1; j < arr.Length - 1; j++)
            {
                temp = arr[j];
                i = j; // range 0 to j-1 is sorted
                while (i > 0 && arr[i - 1] >= temp)
                {
                    arr[i] = arr[i - 1];
                    i--;
                }
                arr[i] = temp;

            }
            return arr;
            // end outer for loop
        } // end insertion sort
```

Figure 3. Interface describing the implementation insertion sort

Table.1: Table showing the metric value of bubble sort implementation

Metrics	Value
LOC	19
n1	5
n2	4
N1	13
N2	22
Program Length	35
Program Vocabulary	9
Control Density	13.75
Program Volume	33.43603

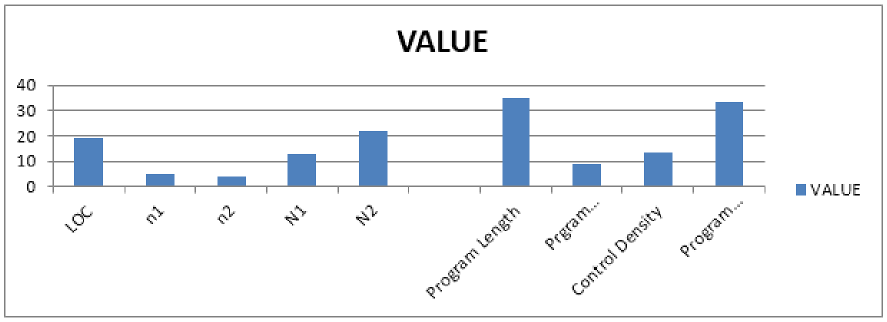


Figure4: Graphical Representation of Bubble sort algorithm

The diagram in figure1 is an excerpt of the code implementation of bubble sort. This was then analyzed with Halstead performance method and that led to table 1 which was then plotted to form the graph on figure 4

Table 2: Table showing the metric value of bubble sort implementation

METRICS	VALUE
LOC	18
n1	6
n2	5
N1	13
N2	13
Program Length	26
Program Vocabulary	11
Control Density	7.8
Program Volume	11.177

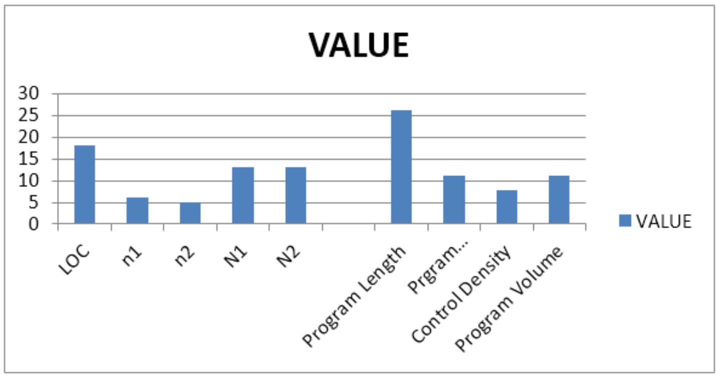


Figure 5: Graphical Representation of Selection sort algorithm

The interface displayed in figure 2 is a code excerpt of the implementation of selected sort algorithms in C#. This was then analyzed with Halstead method of performance metrics. Then table 3.2 was derived with respect to the list of equations described in the method, this was later used to plot the graph shown on figure 3.5.

Table 3.3: Table showing the metric value of bubble sort implementation

METRICS	VALUE
LOC	16
n1	6
n2	4
N1	10
N2	24
Program Length	38
Program Vocabulary	10
Control Density	36
Program Volume	38.04271

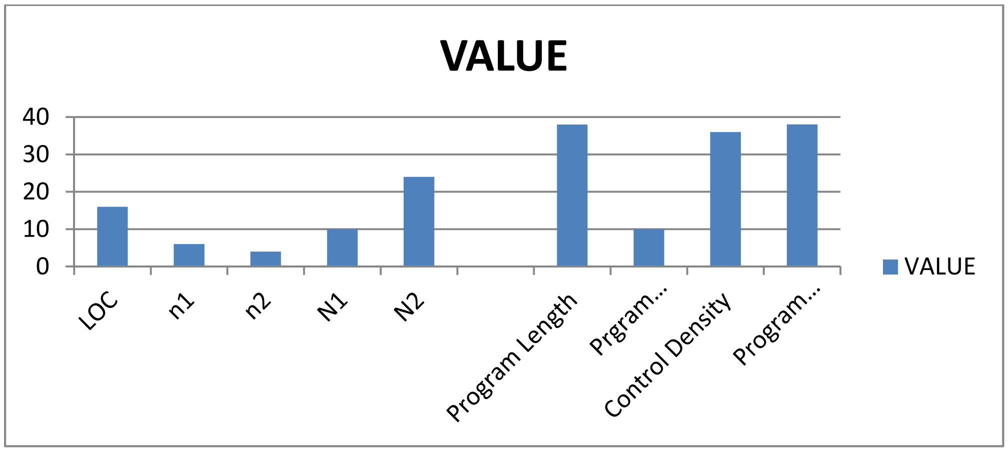


Figure 3.6: Graphical Representation of Insertion sort algorithm

The interface displayed in figure 3 is a code excerpt of the implementation of selected sort algorithms in C#. This was then analyzed with Halstead f performance metrics. Then table 3 was derived with respect to the list of equations described in section 3. This was later used to plot the graph shown on figure 6.

Table 3.4: Table showing the metric value of bubble sort, selection sort and insertion sort implementation

BUBBLE SORT	SELECTION SORT	INSERTION SORT
19	18	16
5	6	6
4	5	4
13	13	10
22	13	24
35	26	38
9	11	10
13.75	7.8	36
33.43603	11.177	38.04271

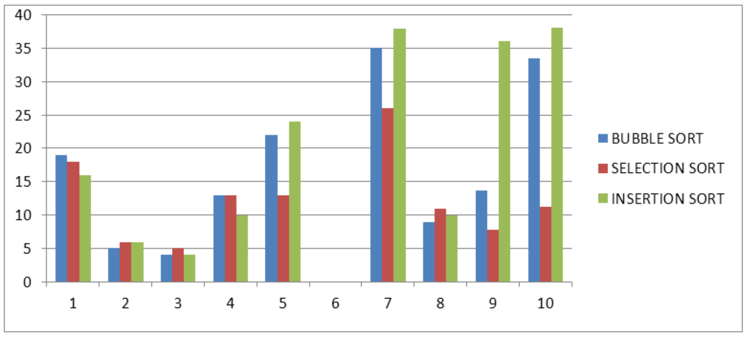


Figure 3.7: Graphical Representation of Insertion sort algorithm.

4 Conclusion

Software metrics are numerical data related to software development. Metrics strongly support software project management activities which help the developers to perform some manage options with respect to planning, organizing and controlling the basic functional aspect of their code. Therefore, in this research, Halstead method of software metrics was adopted on some selected sorting algorithms which were later implemented with C#.

Computer Resource Utilization indicators show whether the software is using the planned amount of system resources. The computer resources are normally CPU time, I/O, and memory. Computer resource utilization is planned during the requirements activity and reviewed during the design activity. Resources are monitored from the start of implementation activity to the end of the life cycle. However, It is recommended that performance metrics should be administered before the implementation of any software.

REFERENCES

[1] McCall, M.M (1977) :Art Without A.market: Creating value in a Provincial Artword Sym:Software System Testing and Quality Assurance. Van Nastraud

[3] Tourslakis. G, (1984) :Theory of Computation, John Wiley & Sons

[4] Ramoorthy, (1985) :Achieving Quality in Software Proceedings of the 3rd International Conference on Quality In Software.

[5] Zuse (1991): Practicing Software Engineering in the 21st Century, IRM Press, USA

[6] Fenton(1992): Sofware Metrics –A rigorous Approach. Chapman Hall

[7] Akiyama F, ''An example of software system debugging'', Inf Processing 71, 353379,1971.

[8] Cusumano, M. A., "Objectives and Context of Software Measurement, Analysis and Control," Massachusetts Institute of Technology Sloan School of Management Working Paper 3471-92, October 1992.

[9] Daskalantonakis, M. K., "A Practical View of Software Measurement and Implementation Experiences Within Motorola," IEEE Transactions on Software Engineering, Vol. SE-18,1992, pp. 998–1010.