**CHAPTER ONE**

**INTRODUCTION**

**OVERVIEW**

Algorithms are essential to our ability to solve problems with computing. Algorithms are a conception in mathematics. An algorithm is a list of instruction specifying a sequence of operation, which will give the answer to any problem of a given topic in mathematics, a class of problems is considered solved when an algorithm for solving them is found. The discovery of such algorithms is a natural aim of mathematics.

Algorithms do have their characteristics and they are: been deterministic, been general, and been finite, should have one entry and exit, and should act at least on one impute and it should produce at least one output.

Sorting is crucial operation usually performed by the computers. It is very important for presentation of data extracted from database as it is generally preferred that data be sorted out in a particular manner before being used for any computations.

In many cases several different algorithms yield correct results, but the amount of time required to derive those results can vary by orders of magnitude. The factors that affect the algorithm’s run-time can be complex, and often are not known at implementation time—consider the situation of somebody who has to compile a software library, with no knowledge of how the library’s functions may be used in the future. Analysts need a way to quantify run-time performance so they can choose between different algorithms. One measure that’s widely used is the “big-O” notation. It expresses the amount of work for a given algorithm as being proportional to a bounding function, independent of implementation details such as the choice of computer hardware, operating system, programming language, compiler, etc. Despite its widespread use, big-O notation does not always yield results that are consistent with what knowledgeable programmers choose to do in practice.

In this project, we propose “robustness” as an alternate measure of algorithmic performance. Robustness measures overall performance in terms of a “loss function,” which incorporates both average performance and consistency of performance into a single measure—smaller loss means better overall performance. With robustness, we are willing to trade off a small amount of the average performance for greater consistency across a broad range of inputs, particularly when those inputs are outside of our control.

We tested our theories on sorting algorithms—one of the best-studied areas in the field of computer science. The use of experimental design techniques allowed us to measure the impact of factors that would be uncontrollable in the real world, by measuring how those factors affect the loss.

The robust analysis yielded results that are consistent with actual usage: practitioners prefer quick sort over heap sort, despite the fact that under big-O analysis, heap sort dominates quick sort. Robust analysis captures this real-world preference.

Algorithms are an essential part of programming methodology. In many cases, a given problem can theoretically be solved by more than one algorithm.

Therefore, OR analysts need to be aware of the efficiency of the algorithms under consideration as, in practice, the choice of algorithm can mean the difference between being able to solve a problem in a timely fashion or not. However, traditional approaches to assessing algorithm efficiency do not always capture the real-world trade-offs involved. In this thesis, we will explore the use of a new measure of algorithm efficiency, and contrast it with “big O” analysis, one of the most widely used analysis techniques. We will illustrate our points with sorting algorithms, arguably one of the best understood and most analyzed topics within the field of computer science.

Sorting algorithms are one of the most commonly used classes of algorithms employed in computers today. These algorithms arrange data by means of some well-defined ordering property, such as lexicographic ordering (for strings) or cardinality (for numerical data). The efficiency of any search is significantly enhanced through the use of sorting.

When we consider the complexity of an algorithm, we are usually concerned with properties such as the running time, or the amount of memory or intermediate storage required. This thesis will focus exclusively on running time T, but the reader should bear in mind that any other measure of interest could be used instead. T is determined by many factors, including:

• The number of elements to be processed;

• The choice of algorithm;

• The specific computing platform (combination of hardware and operating system) used;

• Other tasks which may be active on the system;

• Choice of programming language;

• Choice of compiler; and

• The exact distribution of the input data.

Complexity is the measure of determining the reliability and maintainability of a software. There are different types of complexities.

1. Time complexity: which is the measure of the number of computational steps used by algorithm of programming languages as the case may be, when solving a problem of size?
2. Space complexity: is the measure of the number of memory words used by an algorithm or programming language as the case may be, when solving a problem of size.
3. Computation complexity: however provides a means for comparing the characteristics of different algorithm solution or coding styles to a given problem algorithm.
4. Asymptote complexity: is the rate of growth of time (or memory) required to solve the problem as its size becomes large.

Asymptotic time complexity is the number of basic steps taken by an algorithm as a function of the size of its input. The input size expressed in terms of number of bits nodes, elements, integers and so on.

However, the best-way to quantity complexity in a program are :

1. Metrics which provides McCabe’s cyclomatic and
2. Hallstead’s Cyclomatic complexity is a measure of soundness and confidence for a program Thomas McCabe introduced it in 1976 and it measures the number of linearly independent path through a program module. This measure provides a single ordinal number that can be compared to the complexity of the program. The cyclomatic complexity of software is calculated from connected graph of the module, which shows the topology of control flow a connected graph of the module, which shows the topology of control flow within the program.

Halstead measure was developed to measure a programs module complexity directly from source code with emphasis in computational complexity. The measure was developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module. Due to the fact that they are applied to codes; they often are used as maintenance measurement.

1.2 **PROBLEM OF STATEMENT**

The problem is calculating the performance evaluation of the implementation of some programming language on merge sort algorithm.

Performance evaluation of system (computer systems) deals with the investigation of computer components, both hardware and software, with a view if establishing the level of their performance and efficiency.

Therefore, performance evaluation of implementation language, is the process by which the investigation of certain programming languages ( such as Pascal, C, Visual Basic and Java) written to code the merge sort algorithm with a view to establish and determine the level of complexity, performance and efficiency of all the languages.

Knowing the performance evaluation is of utmost importance to both the application developers and also the end users.

It enables the application developers choose the coding programming styles that leads to the best and optimum Program execution. It also enables the ends users have an upper hand by having a variety and choosing the best effective/fastest application of any of coding styles.

**1.3 RESEARCH OBJECTIVES**

The aims of this project are as follows:

1. To compare the implementation of certain programming language used to code the algorithm.
2. To calculate the Halstead values of each of these implementation languages
3. To compare the complexities of each of these implementation languages using the calculated value as a measure of their complexities.
4. To conclude which implementation language is fastest to give an accurate and optimal result.

**1.4 RATIONALE FOR THE RESEARCH WORK**

It is ultimate important that progress be made in ones are of study. This is the reason why the complexity of the implementation of an algorithm merge sort algorithm be made so as to choose which one gives the optimal performance, as it is essential that an algorithm needs to be fast accurate as well as easy to implement, test and maintain.

**1.5 SCOPE OF STUDY**

There are numerous ways of measuring the complexity of algorithms the respect to implementation languages in which in which they would been written.

The implementation languages considered in the project are: Pascal, C, Visual Basic and Java.

Pascal is general purpose high level programming language which was first derived from Algal – 60. Its construction are made up of algebra like expressions and certain English words such as BEIGIN, END, read, write, It, THEN, REPEAT, WHILE, DO.

Visual Basic is an object oriented/high level programming language. An object oriented/high programming language simplifies the test of programming application for windows.

Java is an object oriented, platform – neutral secure language that was designed to be easier to lear than C++ and to measure than C and C++. Its object oriented program is that a program in that a program is conceptualized as a grip of objects that work together.

**1.6 METHODOLOGY**

These are method by which the knowledge, information and material are acquired for this research work.

The liberty: study of essays, publications, books, journals related to algorithms and their complexities.

The internet: visitation of websites related to algorithm, complexity measures and the likes.

Resource person: enquiring from supervisor and lecturers that have knowledge or related knowledge on the topic.

**1.7 ARRANGEMENT OF WORK**

The project starts with chapter one which include the introduction and the general overview of what the project is all about. The two which includes the literature review and the explicit explanation of some terms succeeds the chapter one. The three which contains the bulk of the work follows the; it contains the calculation of the measures of the implement programs. The chapter five has the recommendation and conclusion, the reference follow after these.

**CHAPTER TWO**

* 1. **BACKGROUND CONCEPT AND LITERATURE REVIEW**
  2. **INTRODUCTION**

The study of algorithm is at the art of computer science. In the recent year a number of significant advances in the field of algorithm have been made and these range from the development of faster algorithms to the discovery of certain natural problems for which certain algorithms are inefficient. These results are kindled considerable interest in the study of algorithm, and the area of algorithm design and analysis has blossomed into a field of interest.

Algorithm thus can be evaluated by a variety of criteria, which are: the rate of growth of the time and space required to solve larger and larger instance of problem. A programmer usually has a choice of the data structure and algorithm to use. Choosing the best involves the measure through the time complexity and space complexity. As complexity becomes faster and we can handle larger problems, it is the complexity of an algorithm that determines the increase in problem size that can be achieved with increase in computer space. Sorting algorithms are one of the most commonly used classes of algorithms employed in computers today. These algorithms arrange data by means of some well-defined ordering property, such as lexicographic ordering (for strings) or cardinality (for numerical data). The efficiency of any search is significantly enhanced through the use of sorting.

**Merge Sort**

According to Mehlhorn (2013), Merge sort has a complexity of O(n log n). The O(n log n) worst case upper bound on merge sort stems from the fact that merge is O(n). The application of the Merge sort produces a stable sort, which means that the applied preserves the input order of equal elements in the sorted output. Merge sort, invented by Von Neumann and Morgenstern (1945) works by divide and conquer method and it is based on the division of the array into two halves at each stage and then goes to a compare stage which finally merges these parts into one single array. This is also a comparison-based sorting algorithms such as Bubble sort, Selection sort and Insertion sort. In this method, the array is divided into two halves. Then recursively sort these two parts and merge them into a single array. When working with small array, Merge sort is not a good choice as it requires an additional temporary array to store the merged elements with O (n) space.

**Merge sort** is an iterative version of the traditional binary merge sort. Merge sort

initially forms \_(N) sorted runs of constant size, and then makes \_(logN) merge-passes over the data forming successively larger runs (the run size at each pass doubles until all keys are in a single run). Each pass touches all the data, resulting in many cache misses.

**Merge sort** is a divide and conquer algorithm .It's Divide the list into two approximately equal sub lists, Then Sort the sub lists recursively[19]. It has an O (n log n) Time complexity .merge sort is a stable sort, parallelizes better, and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list.

**Tiled merge sort** is a tiled version of the iterative binary merge sort. Cache loads (or tiles) are sorted using merge sort, and then merged using an iterative binary merge routine. Tiling increases locality, and improves the cache performance of merge sort, but is effective only over the initial \_(log(BC)) passes, when merge sort is operating in-cache.

**Multi-merge sort** uses the iterative binary merge sort to form cache-sized runs (as in tiled merge sort), and a single merge or order \_(N/BC) to complete the sort. The merge of large order reduces cache misses as compared to tiled merge sort by reducing the number of passes over the data. We confirm the prediction of LaMarca and Ladner [1997] that multi-merge sort is faster than tiled merge sort for large input sizes.

**2.2 COMPUTATIONAL COMPLEXITY**

Computational complexity provides a means for comparing the characteristics of different algorithm solutions to given problem. It is usually expressed as an asymptotic complexity, i.e. the rate of growth of time or memory required to solve the problem as its size becomes large i.e. for increasing N

It studies:

* The efficiency of algorithms
* The inherent “difficulty” of problem of practical and/or theoretical importance.

Computational complexity was originally defined in term of the natural entities of time and space, and the term complexity was to donate the time of space used in the computation. Rather than checking whether an input satisfies a property S, a more natural question might be what is the complexity of expressing the property S? These two issues - checking and expressing – are closely related. It is starting how closely tied they are when the latter refers to expressing the property is first-order logic of finite and ordered structures.

A major discovery in the area was that computational problems can vary tremendously in the effort required to solve them precisely. The technical term for a hard problem is ‘NP-complete’ which essentially means: abandon all hope of finding an efficient algorithm for the exact solution of this problem’. Proving or knowing that a problem is NP-complete is not at all that negative of course. Knowing such limitations, experts do not waste time in impossible projects and instead turn to less ambitious approaches, for example to find approximate solutions, to solve special cases or to alter the problem a little so that it becomes tractable (even at the loss of some fit to the real – life situation).

Regarding the efficiency of algorithms, the rapid advances in computer technology make physical measure (run-time, memory requirements) irrelevant. A more standardized measure is the number of elementary computer operations it takes to solve the problem in the worst case. Average performance is not safe. There may be particular cases (technically called instance of a problem) that behave much worse than the average and nobody can afford to rely on his/her lurk.

The number of elementary operations depends of course on the “size’ of the problem data. Sorting 1 billion numbers is quite different from 10 ones, so we express the number of elementary operations (expected in the worst-case scenario) as a function of some characteristic number(s) that suffice to capture the volume of the work to be done. For a sorting algorithm, this number is simply the number n of numbers to be sorted.

Now, suppose an algorithm solves a problem of size n in at most 7n3 + 5n2 + 27 operations.

For such functions, we are primarily interested in their rate of growth as n increases. We want to distinguish between mild and “exploding” growth rate, therefore differences as that between 7n3 and n3 are really important ( besides, large differences in the constants do not arise in practice). We can also discard the lower order terms, because at large sizes it is the complexity of this algorithm can be sufficiently described by the function g(n) = n3 formally, we say that this algorithm is “of order O (n3)”. It is also usual to say that this algorithm “takes time O (n3)”. This symbolism is a reminder that this function expresses the worst-case behavior at sufficiently large sizes.

***Note: the for sufficiently large n: logn < n < nlogn < n2< n3<2n***

**2.2.1 ASYMPTOTIC COMPLEXITY**

This is the rate at which the storage of the time it takes an algorithm to solve a problem N grow as a function of the problem size. The absolute growth depends on the machine used to execute the program, the compiler used to construct the program and many other factors.

The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity. The limiting behavior of the complexity as the size increase is called the asymptotic time complexity.

The memory words used be an algorithm expressed as the function of the size of a problem are known as the space complexity as the size increase is called the asymptotic space complexity.

It is the asymptotic of an algorithm, which ultimately determines the sizes problems that can be solved by the algorithm. If an algorithm processes inputs of size n in term cn2 for some constant c, then we say that the time complexity of that algorithm is 0(n2).

In asymptotic analysis the problem grows, the complexity is described as a simple proportionality to some known function. This idea is incorporated on the “Big O” notation for asymptotic performance.

T (n) = 0 (f(n)) if and only if there are constants Co and no such that

T (n) < = Co (f(n)) and for all n > =no

The expression “T (n) = 0 (f(n))” is read as “T of n is in Big O of f of n”. Big O is assumed to describe to describe an ‘upper-bound’ on the complex.

Other forms of asymptotic analysis are “Big O mega”, “Little O”, Theta”. Big O is the commonly used in specifying asymptotic complexity.

Other forms of asymptotic analysis are “Big O mega”, “Little O”, Theta”. Big O is the commonly used in specifying asymptotic complexity.

Since “Big-O” Is an upper bound, it is often used to state the complexity of a worst-case analysis. Thus insertion sort is (n2). The worst case bound of (n2) is not a bound on the running time for every input.

The functions are often encountered in computer science Big O analysis:

T (n) = 0(1). This is called the constant growth. T (n) does not grow at all as a function of n, it is said to be constant, array access has this characteristics. A[i} takes the same time independent of the size of the array.

T (n) = 0(1g (n)). This is called the logarithm growth. T (n) grows proportional to the base 2 logarithm of n. Binary search has characteristic.

T (n) =) (n). This is called the linear growth. T (n) grows linearly with n. it is for looping overall the elements in a one – dimensional array.

T (n) = ) (nlogn). This is called the “nlogn” growth. T (n) grows proportional to n times base 2 logarithm of n. merge sort has this characteristic, and no other sorting algorithm that uses comparison between elements can be faster than n log n.

T (n) = 0 (nk). This is called the polynomial growth. T (n) grows proportional to the k the power of n. selection sort is an 0(n2) algorithm.

T (n) = 0(2n). This is called the exponential growth.

The growth patterns are in the size which is : 0(1), (1g (n)), (nlogn), 0(nk), 0(2n).

For using Big O:

1. For a single statement S, whose execution time does not depend on n:

S (S) c 0(1)

1. For sequence of statement: S1, S2, ……..Sk:

T (S1, S2, ……..Sk) = T (S1) + ………+ T (Sk)

1. For 1 = 1 to f(n)
2. S
3. T (for loop) = f(n) \* T(S)
4. While (condition)
5. S
6. No fixed rule: must analyses to find a bound on the number of interaction
7. If C then S1 else S2
8. T (conditional) = T (C) + max (T(S1), T(S2))
9. O (F(n) + O (log(n)) = O (f(n) + g(n) = 0(1g(n))
10. If (n) € O(1g(n)) then O(f(n) + g(n) = 0(1g(n))
11. (1g(n) \* 0(1g(n)) = 0 (f(n) g(n))

**2.3 SOFTWARE MEASURE**

Software complexity is one branch of software matrices that is focused on direct measurement of software attributes, as opposed to indirect software measures such as project milestones status and reported system failure. Current military metrics programs emphasize non-complexity metrics that track project management information about schedules, cost and defects, while such tracking measures are necessary to any substantial software engineering effort; they lack predictive power and are thus inadequate for risk management.

Complexity measure can be used to predict critical information about reliability and maintainability of software system form automatic analysis of the source code. Complexity measure also provides continuous feedback during a software project to help control the development process. During testing and maintenance, they provide detailed information about software modules to help pinpoint area of potential instability. Many of the factors affecting software quality that have been identified by researchers can be seen in part as functions of the complexity and the size of the program, and the capabilities of the programmers and managers. This will include, but is not limited to, testability, efficiency, legibility and structuredness.

There are a number of ways to qualify complexity in a program. They are: McCabe’s cyclomatic number/measures and the Hallstead’s Volume.

**2.3.1 HALSTEAD COMPLEXITY MEASURES**

The late Maurice Halstead developed these measures. It is a means of determining a qualitative measure complexity directly from the operators and the operands in the module.

Halstead argued that algorithms have measurable characteristics analogous to physical laws. His model is based on four different parameters: the number of distinct operators (instruction Type, Keyword, etc) in program, called n1; the number of distinct operands (variables and constants), n2; the total number of occurrence of the operators, N1; and, the total number of occurrences of the operands, N2. The sum of n1 and n2 is denoted as n while the sum of N1 and N2 is called N.

From those four counts, a number of useful measures can be obtained.

The number of bit required to specify the program is called the volume of the program is called the volume of the program and is obtained through the equation: V = Nlon2n.

The program level, which is the difficulty of understanding a program, is calculated by: L = (2n2) / (n1N2)

The intelligence content of a program is given by I = L \* V

In an attempt to include the psychological aspects of complexity in the measures Halstead studied the cognitive process related to the perception and retention of simple stimuli. Research by Stroud (1996) had shown that the means number of mental discriminations per second in an average human being, also called the Stroud number is between 3 and 20. Halstead uses 18 as a reference point for his studies. In his model, the number of discriminations made in the preparation of a program, called effort, is given by: E = V/L

The programming time, T, is an estimate of the number of mental discriminations necessary to complete a program divided by the average number of discrimination per second of Stroud number, S. it is important to note that this estimate assumes that the programmer is devoting all of its discriminations to the programming would use some mental effort on non-related tasks. All of these measures are valid under the assumption that the program is “pure” i.e. free of so – called “poor programming practices.”

**2.3.2 CYCLOMATIC COMPLEXITY MEASURES**

Thomas McCabe introduce these measure in 1976, it is also referred to as program complexity or McCabes complexity. It measure the number of linearly – independent paths through a program module. Cyclomatic complexity is considered a measure of confidence and soundness of a program. These measure provide a single ordinal number that can be compared to the complexity of other programme. It is independent of language and language format. It also encompasses the design and structural complexity of a system.

For cyclomatic complexity measures, ranges of complexity would help the software engineer determine a program’s inherent risk and stability. The resulting measures could be used in development, maintenance and reengineering solutions to develop estimates of risk, cost, or program stability.

Cyclomatic complexity can be applied in the area of code development risk analysis, change risk analysis in maintenance, test planning, reengineering.

* Code development risk analysis: while code is under development, it could be measures for complexity to assess inherent risk of risk buildup.
* Change risk analysis in maintenance: code complexity tends to increase as it is maintained over time. By measuring the complexity before and after a proposed change, this buildup can be monitored and used to help decide how to minimize the risk of the change.
* Test planning: mathematical analysis has shown that cyclomatic complexity gives the exact number of test needed to test every decision point in a program for each outcome. Thus, the analysis can be used as test planning. An excessively complex module will require a prohibitive number of test steps; that number can be reduced to a practical size by breaking the module into smaller, less – complex sub – modules.
* Reengineering: cyclomatic complexity analysis provides knowledge of the structure of operational code of a system. The risk involved in reengineering a piece of code is related to its complexity. Therefore, cost and risk analysis can benefit from the proper application of such an analysis.

A low cyclomatic complexity contributes to a program’s understandability and it indicates as it is amendable to modification at lower risk than a more complex program. A module’s cyclomatic complexity is a strong indicator of stability. The cyclomatic complexity of a software module is calculated from the connected graph of the module (that shows the topology) of control flow within the program.

Cyclomatic complexity (CC) = E – N + P

Where E is the number of edges of the graph, N is the number of nodes of the graph; p is the number of connected components.

To actually count these elements requires establishing a counting convention (tools to count cyclomatic complexity contain these convention). The complexity number is generally considered to provide by counting the lines of codes. Nodes are the numbered locations, which correspond to logic branch points; edges are the lines between the nodes.

Cyclomatic complexity is a mathematically rigorous approach to defining complexity and using identifiable properties of programs to define how modules of code will be tested. Although not bound to any language, the metric has been associated with the C Programming language and many examples in the literature are based on function implemented in C. For automated graphing and complexity calculation, the technology is language sensitive; there must be a front – end source parse for each language, with variants for dialectic differences.

The structured testing methodology is based on cyclomatic complexity, in the sense that the cyclomatic complexity is the number of test require. Given the correlation of complexity with errors, this is a desirable result since we want testing effort to be proportional to complexity. Many other coverage-based testing techniques, from the simple ones such as statement coverage to the complicate ones such as testing all data definition – usage association, do not have this property. You could have arbitrary complex software with lots of statements and data associations and still satisfy those other testing criteria with one or two tests, or you might require lots of tests. With cyclomatic complexity and structured testing. You know in advance exactly how many test you’ll need, so you can do do detailed test planning and manage the schedules, costs, and risks associated with unit testing. Design complexity provides similar benefits for integration testing.

Numerous studies and general industry experience have shown that the cyclomatic complexity measure correlates with error in software modules. Other factors being equal, the more complex a module is, the more likely it is to contain error. Also, beyond a colliding threshold of complexity, the likelihood that a module contains error increase sharply. Given this information, many organizations limit the cyclomatic complexity of their software modules in an attempt to increase overall reliability.

Complexity can be used directly to allocate testing effort by leveraging the connection between complexity and error to concentrate testing effort on the most error-prone software. In the structured testing methodology, this allocation is precise the number of test paths required for each software module is exactly the cyclomatic complexity. Other common white box testing criteria have the inherent anomaly that can satisfied with a small number of test for arbitrarily complex (by any reasonable sense of “complexity”) software.

**2.3.3 LINE OF CODES:**

The simplest way to measure the size of a project is to count the lines. This is the oldest and most widely used size metric. Line of code, or LOC, Looks like a simple concept. However, it’s not.

The source lines of code Count (SLOCC) computer program count the number of source line of a code, estimation of cost, and satisfying miscellaneous code metric needs.

Depending on the languages, SLOCC count whichever of the following is or are applicable: source-code lines, continuations of source lines, comment lines, side comment lines (comments on the same lines with source code), end lines, blank lines, and lines of such other types as are appropriate to the target language.

There are several ways to count the lines. Depending on what is counted, one gets a low or a highline count. In the table below are various alternatives.

**Table 2.1 Counting lines of codes.**

|  |  |
| --- | --- |
| **Metric**  **Simple line count**  (not supported) | **Support as description**  This type of metric counts all the lines in source files. Some simple programs may calculate this, but this is not a useful figure for VB developers because some classic VB files include more than just the code. |
| **Physical lines** (LINES) | This metric counts the physical lines, but excludes classic VB from definitions and attributes. |
| **Physical line of code** (not supported) | This type of a metric counts the lines but excludes empty line and comments. This is something referred to as the source lines of codes (SLOC) metric. |
| **Logical lines of code (LLOC)** | A logical code line is the same as a physical code line, except that two or more lines joined with the line continuation character “-“ are counted as one logical line. |
| **Statement STMT** | This is not a line count, but a statement count. Visual Basic programs typically contain one statement per line of code. However, It’s possible to put several statements on one line by using the colon “.” Or writing single-line If. Then statement |

The typical line count (LINE) is a simple but not a perfect way to measure code size. Since a logical line can expand over several lines, the physical line count exaggerates code size. A common problem in line counts is also that empty (or white space) lines, as well as comments, included in the count with improper line count, you can appear really productive by hitting the enter Key, or alternatively, pretend that you are writing tighter code by deleting all comments.

The logical line of code metric (LLOC) has both advantages and disadvantages. It is a simple measure, easy to understand, and widely used. It can be used to measure productivity, although care is needed, because programming style can have an impact on the value.

Line counts are notorious in that they can vary between programming languages and coding styles. A line of VB code is not the same as a line of C++ code. Implementing a feature in VB6 may require more effort (or maybe less) than what it would take in VB.NET. Especially when measuring programmer’s performance the line counts aren’t perfect. One programmer may produce a large number of lines, while the other spends a long time and succeeds in squeezing the same function in a small space. And developer work on other things than just producing more and more code, such as documentation, planning, testing etc.

Summing sum up all the logic lines of code, comment and white space, one gets the total number of logical lines.

Lines = Lloc + Lloc’ + Llow

**2.3.4 FAN IN FAN OUT COMPLEXITY.**

Sallie Henry and Dennis Kafura put forth something in early 1980s called the fan in – fan out complexity measure. What this does in essence is count of the number of data flows from a given unit or component plus the number of global data structure that the program updated procedure parameters and procedures called from with a modeule. This is the equation that they came up with: Complexity = length \* (fan in fan out)2.

Structural fan-in (SFIN) and fan-out (SFOUT) values measure the relationships between files and between procedures. They measure the complexity of the static (design-time) structure of code.

Fan metric for files and procedures is calculated from the procedure call tree SFIN (procedure) = number of procedures that call this procedure SFOUT

(Procedure) = number of procedure calls

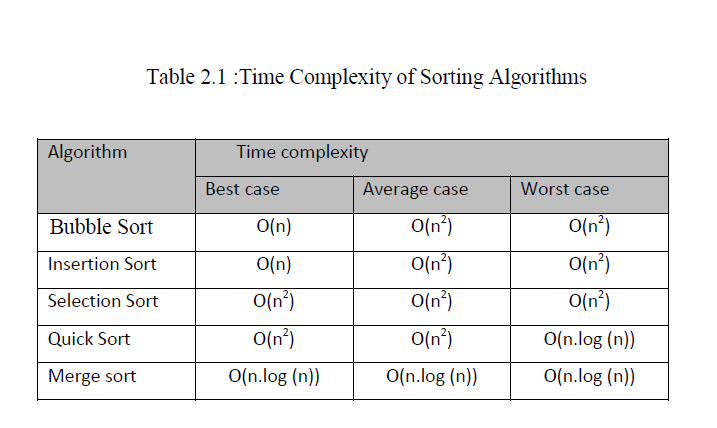
Structural fan-in/fan-out for file is calculated from the file dependency tree.

SFIN (file) = number of files that require this file to complier or run SFOUT (file) = number of file.

The best way to implement a complexity measurement program is to start small. Collect data on a wide variety of metrics, but pick a small, validated, intuitive set of metrics to actually apply. Continue to use lines of code, and add cyclomatic complexity and essential complexity. Train the developers to calculate complexity by hand, and use tools to automate the process. Start using the complexity threshold of 10 immediately to improve software reliability. Start evaluating test plans in terms of complexity to make sure that error-prone code gets the attention that it needs. Then, once the operational benefits of complex analysis have been widely experienced, risk management models can be refined with measures such as the Halstead Metrics and data complexity.

**2.4 Time Complexity**

According to Estakhr (2013), the time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function with the length of a string representing the input. The time complexity of an algorithm is commonly expressed using Big(O) notation, which excludes coefficients and lower order terms. When expressed this way, the time complexity is said to be described asymptotically as the input size goes to infinity. The time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor (Michael, 2006). Time can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take. The research tries to keep this idea of time separated from clock time, since many factors unrelated to the algorithm itself can affect the real time such as the language used, type of computing hardware, the proficiency of the programmer and optimization used by the compiler. If the choice of the units is wise, all of the other factors will not matter to get an independent measure of the efficiency of the algorithm. The time complexities of the algorithms studied are shown in the table below:



**2.4.1Worst-Case Analysis**

The worst case analysis anticipates the greatest amount of running time that an algorithm needed to solve a problem for any input of size n. The worst case running time of an algorithm gives us an upper bound on the computational complexity and also guarantees that the performance of an algorithm will not get worse (Szirmay & Márton, 1998).

**2.4.2 Best-Case Analysis**

The best case analysis expects the least running time the algorithm needed to solve a problem for any input of size n. The running time of an algorithm gives a lower bound on the computational complexity. Most of the analysts do not consider the best case performance of an algorithm because it is not useful (Szirmay & Márton, 1998).

**2.4.3 Average Case Analysis**

Average case analysis is the average amount of running time that an algorithm needed to solve a problem for any input of size n. Generally, the average case running time is considered approximately as bad as the worst case time. However, it is useful to check the performance of an algorithm if its behaviour is averaged over all potential sets of input data. The average case analysis is much more difficult to carry out, requiring tedious process and typically requires considerable mathematical refinement that causes worst case analysis to become more prevalent (Papadimitriou, 2003).

**2.5 SORTING ALGORITHMS**

Sorting means arranging items in a predetermined order. There are dozen of algorithms, the choice of which depends on factors such as the number of items relative to working memory, knowledge of the orderliness of the items.

In computer science and mathematics, a sorting algorithm is one that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and selection of algorithms) that require sorted list to work correctly; it is also often useful for canonicalizing data and for producing human-readable output.

In lay terms, a sorting algorithm is a stepwise instruction used to record data into a new sequence. Like all complicated problems, there are many solutions that can achieve the same results. One sort algorithm can re-sequence data faster than another. Most algorithms can be implemented as an in-place sort, and many can be implemented so they are stable to use.

There are several numbers of sorting algorithms which are: quick sort, heap sort, shell sort, radix sort, bucket sort, insertion sort, selection sort, merge sort, bubble sort mentioning a few of them.

**Quick sort** is an in-place sort algorithm that uses the divide and conquer paradigm. It picks an element from the array (the pivot), partition the remaining elements into those greater time (n2) in the worst case, but it is typically O(nlogn).

**Heap sort** is an algorithm, which builds heap, then repeatedly extracts the maximum item. Its run time is 0(nlogn).

**Shell Sort** is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shellSort is to allow exchange of far items. In shell Sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sub-lists of every hath element is sorted.

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **interval**.

**Radix Sort** The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort. etc) is Ω(nLogn), i.e., they cannot do better than nLogn.

Counting sort is a linear time sorting algorithm that sort in O(n+k) time when elements are in range from 1 to k.

**Bucket sort** is mainly useful when input is uniformly distributed over a range. For example, consider the following problem. Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?

**Insertion Sort** A simple way is to apply a comparison based sorting algorithm. The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(n Log n), i.e., they cannot do better than nLogn.

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms

**Selection sort** algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two sub-arrays in a given array.

**Bubble sort** is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2) where **n** is the number of items.

The project discussion is based majorly on merge sort which will be discussed in details in the next chapters.