

A METHODOLOGY FOR MAPPING PROGRAMMING LANGUAGES TO
PROGRAMMING PROBLEMS

THESIS

Presented to the Graduate Council
of Texas State University–San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Jason Lawrence Michlowitz, B.S.

San Marcos, Texas
August 2006

A METHODOLOGY FOR MAPPING PROGRAMMING LANGUAGES TO
PROGRAMMING PROBLEMS

Committee Members Approved:

Dr. Carol Hazlewood, Chair

Dr. Rodion Podorozhny

Dr. Xiao Chen

Approved:

J. Michael Willoughby
Dean of the Graduate College

ACKNOWLEDGEMENTS

First and foremost I would like to thank my advisors, Dr. Hazlewood, Dr. Podorozhny, and Dr. Chen of the Computer Science department of Texas State University-San Marcos. Their patience, guidance, and teaching have been a tremendous help throughout the course of this project.

Second, I would like to express my deepest gratitude to my wife, Michelle, who has stayed up late nights helping me to proof read this document. Also, she has been very understanding in the lack of time I have been able to spend with her so that I might complete this project. Without her love and encouragement, this project would not have been a success.

Finally, I want to thank my parents, Ralph and Barbara Michlowitz, who always taught me that doing my best was the only way to be in life. Because of this, I was able to complete a Bachelor's degree, and now a Master's in the field in which I have accomplished many things. Their love, devotion, and dedication as parents will be forever remembered in my professional career.

This manuscript was submitted on May 5, 2006.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS.....	iii
LIST OF FIGURES.....	vii
LIST OF TABLES.....	x
ABSTRACT.....	xi
CHAPTER	
I. INTRODUCTION.....	1
II. RELATED WORK.....	4
2.1 Introduction	
2.2 Studies in Software Engineering and Language Comparisons	
2.3 Using Principal Components Analysis	
2.4 Conclusions on Related Work	
III. EXPERIMENTAL DESIGN.....	19
3.1 Overview	
3.2 Independent Variables	
3.3 Dependent Variables	
3.4 Subjects	
3.5 Operation of Experiment	
3.6 Threats to Validity	
3.7 Project Scope	
3.8 Understanding .NET Metadata	
IV. THE PROGRAMMING LANGUAGES.....	31
4.1 The Environment	
4.2 The C Programming Language	
4.3 The C++ Programming Language	
4.4 The C# Programming Language	
4.5 The Java Programming Language	

4.6 The Visual BASIC Programming Language	
V. THE ALGORITHMS.....	44
5.1 Definition of Selection Criteria	
5.2 Definition of Implementation Criteria	
5.3 Searching	
5.4 Sorting	
5.5 String Matching	
5.6 Arithmetic Algorithms	
5.7 Order Statistics	
VI. METRICS AND THEIR DEFINITIONS.....	62
6.1 Definition of Selection Criteria	
6.2 Factors Present in the Measurement Environment	
6.3 Static Metric Definitions	
6.4 Dynamic Metric Definitions	
6.5 Metadata Metric Definitions	
VII. PRINCIPAL COMPONENTS ANALYSIS.....	78
7.1 Understanding Metric Data	
7.2 Understanding Sources of Variation	
7.3 Metric Domains	
7.4 The Relative Complexity Metric	
VIII. STATIC MEASUREMENT ANALYSIS.....	85
8.1 Introduction	
8.2 Individual Algorithm Results	
8.3 Evaluation of Results	
8.4 Conclusions	
IX. DYNAMIC MEASUREMENT ANALYSIS.....	99
9.1 Introduction	
9.2 Individual Algorithm Results	
9.3 Evaluation of Results	
9.4 Conclusions	
X. METADATA MEASUREMENT ANALYSIS.....	113
10.1 Introduction	
10.2 Individual Algorithm Results	

10.3 Evaluation of Results	
10.4 Conclusions	
XI. OVERALL MEASUREMENT ANALYSIS.....	126
11.1 Introduction	
11.2 Individual Algorithm Results	
11.3 Evaluation of Results	
11.4 Conclusions	
XII. RESEARCH PRODUCTS AND CONCLUSIONS.....	140
12.1 Lessons Learned	
12.2 Software Development Questions and Answer Guidelines	
12.3 Further Research	
12.4 General Conclusions	
APPENDIX A: SOURCE CODE.....	152
APPENDIX B: RAW MEASUREMENT DATA.....	221
APPENDIX C: PCA-RCM TOOL OUTPUT.....	228
REFERENCES.....	238

LIST OF FIGURES

5.1 Linear Search Pseudocode.....	48
5.2 Bubblesort Pseudocode.....	49
5.3 Quicksort Pseudocode.....	51
5.4 Naïve String Matching Pseudocode.....	52
5.5 KMP String Matching Pseudocode.....	54
5.6 Polynomial Addition Pseudocode.....	57
5.7 Gaussian Elimination Pseudocode.....	58
5.8 Minimum Pseudocode.....	59
5.9 Random Selection Pseudocode.....	60
6.1 Control Flow Diagram of a For Loop.....	69
8.1 Linear Search Static Measurement RCM Results.....	87
8.2 Bubblesort Static Measurement RCM Results.....	88
8.3 Quicksort Static Measurement RCM Results.....	89
8.4 Naïve String Matching Static Measurement RCM Results.....	90
8.5 KMP String Matching Static Measurement RCM Results.....	91
8.6 Polynomial Addition Static Measurement RCM Results.....	92
8.7 Gaussian Elimination Static Measurement RCM Results.....	93
8.8 Minimum and Maximum Static Measurement RCM Results.....	94
8.9 Random Selection Static Measurement RCM Results.....	95

9.1 Linear Search Dynamic Measurement RCM Results.....	101
9.2 Bubblesort Dynamic Measurement RCM Results.....	102
9.3 Quicksort Dynamic Measurement RCM Results.....	103
9.4 Naïve String Matching Dynamic Measurement RCM Results.....	104
9.5 KMP String Matching Dynamic Measurement RCM Results.....	105
9.6 Polynomial Addition Dynamic Measurement RCM Results.....	106
9.7 Gaussian Elimination Dynamic Measurement RCM Results.....	107
9.8 Minimum and Maximum Dynamic Measurement RCM Results.....	109
9.9 Random Selection Dynamic Measurement RCM Results.....	110
10.1 Linear Search Metadata Measurement RCM Results.....	115
10.2 Bubblesort Metadata Measurement RCM Results.....	116
10.3 Quicksort Metadata Measurement RCM Results.....	117
10.4 Naïve String Matching Metadata Measurement RCM Results.....	118
10.5 KMP String Matching Metadata Measurement RCM Results.....	119
10.6 Polynomial Addition Metadata Measurement RCM Results.....	120
10.7 Gaussian Elimination Metadata Measurement RCM Results.....	121
10.8 Minimum and Maximum Metadata Measurement RCM Results.....	122
10.9 Random Selection Metadata Measurement RCM Results.....	123
11.1 Linear Search Overall Measurement RCM Results.....	127
11.2 Bubblesort Overall Measurement RCM Results.....	128
11.3 Quicksort Overall Measurement RCM Results.....	129
11.4 Naïve String Matching Overall Measurement RCM Results.....	130
11.5 KMP String Matching Overall Measurement RCM Results.....	131

11.6 Polynomial Addition Overall Measurement RCM Results.....	133
11.7 Gaussian Elimination Overall Measurement RCM Results.....	134
11.8 Minimum and Maximum Overall Measurement RCM Results.....	135
11.9 Random Selection Overall Measurement RCM Results.....	136

LIST OF TABLES

11.1 Summary of Overall RCM Values.....	139
B.1 Static Raw Measurements.....	222
B.2 Dynamic Raw Measurements.....	223
B.3 Matadata Raw Measurements.....	226

ABSTRACT

A METHODOLOGY FOR MAPPING PROGRAMMING LANGUAGES TO PROGRAMMING PROBLEMS

by

Jason Lawrence Michlowitz, B.S.

Texas State University–San Marcos

August 2006

SUPERVISING PROFESSOR: CAROL HAZLEWOOD

Several algorithms that solve different types of problems are implemented, tested, and compared by applying a set of metrics. The results are analyzed using Principal Components Analysis to calculate a Relative Complexity Metric. The results of the study reveal that a programming language does have an effect on the simplicity, speed and other attributes of an implementation. The results of the study also reveal which languages are best suited for a particular type of programming technique, such as recursion.

CHAPTER I

INTRODUCTION

Within the computer science and software engineering communities, there has been much research on the subject of algorithms and the work that can be done through their use. Numerous discussions can be found on the speed, complexity, and effectiveness of different algorithms and how those that perform the same type of work measure against each other. Lacking, however, is extensive research on how a given algorithm's performance can be affected through the choice of a programming language. There are many different types of algorithms that perform many different tasks and it is one of the goals of this research to ease the task of finding a programming language that best suits the problem at hand. Finding the best language for that problem will ease the implementation task, which will allow for the production of better quality software. Aside from this goal, two important questions will need to be answered: Will algorithms perform differently when written in different languages? Which language offers the least complex algorithm implementation? Performance and suitability are defined in terms of software metrics.

To answer the above questions, the following hypotheses must be tested:

H1: An algorithm, when implemented in a set of programming languages, will perform differently in each language.

H2: Given a specific algorithm and a set of languages, it is possible to determine which language is best suited for the given algorithm.

In order to illustrate the concept that a programming language can have an effect on algorithm performance, several classical algorithms have been implemented in multiple languages. The algorithms were chosen from a variety of problem domains and the languages from a common platform. Each implementation has been tested and measurements have been applied statically, dynamically, and with respect to .NET metadata. The details of this experiment are described in Chapter 3. As a basis for comparison, these measurement results have been put through a statistical process described in Chapter 7 in which a greater understanding of language complexities becomes visible. Looking at these statistical data, it can be determined which languages are best suited for the given algorithms based on this data. The final goal of this research, however, is not to decide for the programmer which language to use in a particular situation. Due to the multitude of aspects inherent in a programming language, the constraints of the software being developed must be the factor in choosing an implementation language. There are three measurement categories, namely static measurements, dynamic measurements, and measurements on .NET metadata. As detailed in Chapters 8, 9, 10, and 11, it has been determined that languages that perform better in one category might not perform well in another, so instead the goal of this research is to provide the programmer with several questions that should be asked and how these questions should be answered before choosing an implementation language.

This work expands on the ideas of Munson (2003) in that he uses a set of metrics and statistical analysis in order to determine the most complex code modules in a large

software system. What is different in this work, however, is that this study does not compare program modules, but rather programming languages. In other words, the same program module is written several times, each in a different language, is measured and compared, and results are given. In the Munson work (2003), this same process is used, but he compares the individual modules, all written in one language. As a result of Munson's process, there is little to explain what complexity was introduced as a side effect of the language used. This study adds this dimension to what has already been done in Munson's process. Munson and Khoshgaftaar (1990) have also done a fair amount of work in the area of creating a one-valued representation for the complexity of a program module, and this concept is used here to determine how a programming language introduces additional, possibly unseen complexities. The methodology used in this work has not previously been applied to language comparisons or .NET metadata.

Within the constraints of this study, the final results establish that programming languages do in fact introduce possibly unseen complexities into program implementations. It is also possible from the results of this study to determine which language is best suited for a given problem. Again, suitability is defined in terms of software metrics and their analysis.

CHAPTER II

RELATED WORK

2.1 Introduction

In this chapter, articles and other publications on measurement analysis and language comparisons are reviewed and related to this work.

2.2 Studies in Software Engineering and Language Comparisons

The following works compare different aspects of software engineering using empirical studies. Also, this section lists articles and other publications that compare languages using various methods. The strengths and weakness of each are discussed as well as what this research project contributes to the literature. Taking a look at these articles can give validity to this project as other authors have done similar work.

2.2.1 An Empirical Comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a Search/String-Processing Program

The study involved language comparisons as implemented by a number of different programmers. The study compared for various properties, including run-time, memory constraints, reliability, etc. They concluded that scripting languages are more productive than compiled languages, given all such factors involved (Prechelt, 2005). This is similar to this study because it compared languages for efficiency. The differences

lie in the fact that while this study compares across a common platform for efficiency in language constructs, Prechelt's study focuses on human factors, such as various programming styles, and does not compare across a consistent framework. Also, due to the nature of the study, it does not (and can not) use a Relative Complexity Metric, a one valued representation of a program's metrics, to statistically compare the programming languages, as there is additional unseen variation as a result of using different programming styles and different frameworks. This study can use a Relative Complexity Metric as it avoids this problem by using the .NET framework, and one machine as the common approach for comparison.

2.2.2 Software Faults in Evolving a Large, Real-Time System: A Case Study

This study looks at the faults found in a large, real-time system and categorizes them by when they were found, what testing procedure was used in finding the fault, how difficult these were to fix, and perhaps what the underlying causes of these faults are. It is the hope of this study that by finding these causes, future projects may benefit from the knowledge gained about the faults in the system measured. All of the data in this study were gathered using a standard questionnaire in which the requirements, design, and code of the system are inspected. In a second questionnaire, the methods of testing and quality assurance are examined, as it is important to find at what point in the testing process the faults were found (Perry & Stieg, 1990).

This study does not use formal statistics, as it is more of an analysis of the software process used by the organization rather than a comparison of code modules; however it does contribute work in the areas of design and requirements, in which some

base faults may be found before coding begins. It is similar to this research project in that the faults are categorized and analyzed, much like the measurement categories of static, dynamic, and metadata metrics taken on each of the code modules written for this project. It is a similar goal of this study that future knowledge of programming languages might give insight into preventing faults, much like the fault prevention hope of the study in this article. One benefit of this research project over the one found in the article is that a major system did not need to be developed in which time and money were spent in order to learn anything new, and thus the data gained may perhaps be useful before a real system begins development.

2.2.3 A Comparison of the Programming Languages C and Pascal

In this article, the authors look at the language constructs and design patterns of C and Pascal. The authors believe that Pascal programs tend to be more reliable than C because of its richer set of data types, its strong typing, readability and portability. On the other hand, the authors believe that C is much more flexible and can be used effectively in more applications than Pascal as it gives the programmer more control. The authors list all of the strengths and weakness of each language in much the same way as this research project. They then go into all of the features and data types of each language with an in-depth look at all of the language aspects of C and Pascal. Once the languages are described in detail, the authors list which applications each language is best suited for (Feuer & Gehani, 1982). The main problem with this study is that there is no measurement data, or statistical analysis to give valid insight into the comparison of these two languages. It seems as though this article is more a collection of programmer

opinions rather than fact. The research performed in the study performed here, however, obtains data on running programs written in these languages, and gives insight into the complexities of each language using researched metrics, and formal statistical analysis.

2.2.4 Java as a Better C++

The author of this article was at the time of the writing learning the Java programming language. He believes that the use of Java as an alternative to C++ would make a better teaching tool in the classroom. He presents pros and cons to each language and how it would affect students in their learning process. There is a presentation of the different data types offered by each language, and a synopsis of the constructs in each language (Bergin, 1996). Sadly, however, this article is not backed with much scientific data, and is simply an opinion of the author that Java would make a better teaching tool. There are not code measurements and in turn, no statistical analysis. It is impossible from this article to gain any knowledge as to why one language behaves “better” than the other.

2.2.5 C# as a First Language: A Comparison with C++

The author of this article was at one time a student of computer science at a university. He discusses the pros and cons to C# being a better teaching tool. He gives some code examples and then asks thought-provoking questions, i.e. “What is a main() function, and why do I need it?” These questions serve to give the reader a sense of the author’s thought process and how it is he came to his conclusions (Bates, 2004). Again, as with the previous articles, the author may have written well, but there is no science

here. There again is no measurement data, no statistical analysis, and no conclusions drawn from these techniques. It again is not possible to come to a reasonable conclusion simply because the author believes that he is right

2.2.6 A Comparison of Ada and Java as a Foundation Teaching Language

This article once again discusses the benefits and drawbacks of one language over another as a teaching tool, in this case, Ada and Java. The author presents his case based on code examples, thought-provoking questions, and his personal opinions about each language. For each language, a list of the common data types, constructs, and modules in each language are compared and contrasted, and the author gives his conclusions based on these comparisons (Brosgol, 1998). Once again, as with the articles above, there is no science here. Again there is a lack of code measurement data, statistical analysis, and the like preventing the conclusions drawn from being of any use. Most of this article is opinion rather than fact. Even if this author's opinions are valid, there is no data to back these opinions.

2.2.7 The Effects of Using a Nonprocedural Computer Language on Programmer Productivity

This article looks at the differences of two languages, COBOL, a procedural language, and Focus, a nonprocedural language. The differences studied include programmer productivity and execution time by the CPU. Several programmers using both languages developed six "mid-sized" applications. There are several independent variables associated with the empirical study performed on these languages. These are

hardware, programming mode (all development online), organizational characteristics of the program development, the source languages, the types of applications, and programmer expertise. Associated with these independent variables, are several dependent variables, namely time to understand the applications, program design time, programming time, testing and debugging time, consulting time, and documentation time. In addition, several run-time factors were studied. These include total CPU time for compilation, total CPU time for execution, total clock time for execution, total number of I/O operations, the number and size of input and output files, and the total number of source lines. Each of these variables is measured and studied. Each set of measurements goes through several statistical processes. These include simple averages, standard deviation, and many others. Once all of the data is processed, the authors give their results, concluding that COBOL is faster and more efficient for the CPU, but that Focus is a more productive language from the perspective of the programmer (Harel & McLean, 1985).

The study in this article follows much the same process as this research project; however, there are some key factors to consider. Many of the variables in this study may contain noise. There is no mention anywhere, for example, of how programming time is handled with respect to coffee breaks, and other factors that might cause problems in the data. Also, there is little mention of why the programs developed are classified, as they are, i.e. complex, simple, etc. Also, another problem exists in that there is little mention of the type of hardware used, even though it is considered an independent variable. This research project hopes to look at these issues and remove the problems that might be

caused by them, i.e. noise in the measurements, providing a standard data set, and using a standard platform for all programming.

2.2.8 Are Visual Programming Languages Better? The Role of Imagery in Program Comprehension

This article looks at programming languages differently. The authors here wish to look at programming from a human-factors prospective. The experiment conducted in this article looks at program comprehension as seen by C programmers and spreadsheet users. The spreadsheet users have formulas at their disposal for computation, while programmers have many operations that may be used with inputs and outputs. The spreadsheet users tended to look at programs from a data flow representation in all situations. The C programmers, on the other hand, looked at control flow and logical construction before data was applied. The experiment conducted involved groups of users, divided by their preferred medium, looking at problems and solving them on paper. From the results, it is believed that visual programmers can create semantic information quicker than non-visual programmers based on the data flow approach. With this in mind, it is believed that programs can be developed quicker using visual tools (Navarro-Prieto & Canas, 2001).

The results of this study bring to the foreground interesting points regarding problem comprehension as viewed by groups of programmers. These programmers use different technologies and therefore see programs and their structure differently. The problem, however, is that these concepts are purely subjective. The article makes general assumptions with regard to programmers based on a small select group of individuals.

There really is no measurement data found in this article nor is there anything to base statistics upon. While the points in the article might be useful, it is not possible to prove anything true or false on how fast programs can be developed, or how correct and productive programmers can be simply based on the tool they use without actually developing applications. This study, while not focused on programmer development time, does focus on the objective, rather than the subjective. This is why measurement data and scientific analysis are performed in order to prove the hypothesis.

2.2.9 Towards More Natural Functional Programming Languages

The author of this article looks at programming languages from the human factors perspective. It is believed that programmers might be more productive and would use languages more effectively if more human factors concerns were taken into account during the language design process. With human factors considerations, language constructs would more closely match human thinking and capabilities. The data and background research in this article are taken from known information from empirical studies in software engineering and from programming psychology. Since much is known about what people find difficult in programming, languages can be designed to address issues with regard to syntax, and bug-prone constructs, making them easier for human thinking to comprehend programming and algorithms (Myers, 2002).

In this article, the author addresses human factors related issues in programming and algorithm understanding. The method used in this study involves review of code from programmers who have written several programs in several languages. Measurements are taken on each program's code in order to see which languages yielded

the most bugs. From the results in this study, it was decided that visual languages, such as Visual BASIC, gave programmers the chance to see their program visually allowing them the chance to better understand the problem before code was finished. This project uses a similar approach in that programs are written in several types of languages and an empirical study is performed to analyze the results. This author believes that languages that have syntax similar to Visual BASIC are superior to others when human factors considerations are taken into account.

2.3 Using Principal Components Analysis

The following is a list of articles from various fields, journals, and sources that use the technique of Principal Components Analysis (PCA) as the authors' mode of analysis of measurement results. Since a statistical tool is needed to process the raw metrics into meaningful data, PCA was chosen. It is clear from the work in the following publications that PCA is a valuable tool for many fields and using the tool for this study is valid and reproducible. In the following works, the authors use several of the same steps as they appear in this study. Measurements are taken, components are determined, and valuable data is returned. In all cases, the sources of variation that might be found in the works below are removed through the use of PCA for study. This variation is important but it is also important to look at variation without noise being introduced. PCA removes this noise and discards it, allowing for pure results, meaning that all sources of variation not associated with the independent variables are removed. It is these reasons that PCA was chosen as the analysis tool for this study. Use of these

articles in this study is only meant to illustrate the value PCA and most do not relate to the topic of this study directly.

2.3.1 Young Adolescents' Leisure Patterns

This article evaluates data collected on a study of pre-teen and adolescent leisure activities. The children were broken up into two groups: 10-12 year-olds, and 14-15 year-olds. The study involved variables such as class differentiation and sex, and involved both organized and unorganized activities. To evaluate relationships between these activities and the subgroups, the study used Principal Components Analysis for categorical data to explore nominal, ordinal, and interval data collected by the study (Zeijl, du Bois-Reymond, & Poel, 2001). The article relates to this study through its use of multiple variables that are combined to form a basis of comparison. PCA is the tool that is used and the authors are able to ascertain valuable and comparable results.

2.3.2 Water quantity and quality dynamics in high-elevation watersheds:

Developing a scientific approach to understanding ski area impacts in Vermont

This study explores the impact of ski area development on high-elevation watersheds in Vermont. The study evaluated several watersheds of Mt. Mansfield, similar in such things as geology, soil, and vegetation. Water samples were collected and evaluated to determine the solute concentrations in the water. Measurements were taken on the amounts of different chemicals and then these are logged for analysis. Principal Components Analysis was used to explore the variability of the chemicals in the solutes

as well as the seasonal chemical changes regarding runoff (Wemple, 2004). This is similar to the use of PCA in this study because it analyzes the individual effects of these chemicals in separate groups. This approach is almost exactly like this study, in that the separate groups here are the static, dynamic, and metadata measurements. Each of these groups is looked at separately. The article also points out that variation in the measurement results can be impacted by sources of variation found in each of the samples. PCA illustrates those sources of variation and highlights them for comparison.

2.3.3 Sex Differences in the Vocalizations and Syrinx of the Collared Dove (*Streptopelia Decaecto*)

This study involves the pitch and frequency of the voices of collard doves. It studies acoustic discrimination ratios as they relate to sex differentiation in the birds. The study concludes that the vocal and anatomical data demonstrate that physical differences contribute to sexual differentiation of their vocalizations. To gather these data, they measured the syrinx of the individual test subjects, and these measurements were taken in three groups: those taken after perfusion-fixation, those taken from cartilage-bone stained syringes, and those taken from horizontal sections. The authors used Principal Components Analysis to reduce the number of variables in these groups (Ballintjn & Ten Cate, 1997). In this way, their work relates to this study because its principal components are simplified using PCA. Again, it can be seen here in this article that PCA is used to determine sources of variation when the different categories of measurements are applied. These variations present themselves as the individual principal components and this new data can be compared.

2.3.4 Multivariate analysis of Mammalian Communities: Membership and Species Lineage ranges in the Tertiary of North America

This study discusses mammalian fossils in North America, using primarily localities in Wyoming and Nebraska. The study concentrates on using statistical tools to parse out differences among many localities simultaneously, rather than focusing on individual communities. The study was limited to fossils from the Wasatchian through the Arikareean periods. Principal Components Analysis was used to simplify the vast amount of data that was collected on the fauna of this period. Each fauna was described by a function of 600 variables indicating the presence or absence of each species (Dewar, 2003). This is similar to what this study focuses on, in that it simplifies the data for analysis using PCA, but there is another similarity in that the fossil study also groups the PCA results in order to measure degrees of similarity among various faunas.

2.3.5 Associations Between Perinatal Interventions and Hospital Stillbirth Rates and Neonatal Mortality

This study investigated the effects of various factors on hospital stillbirth and mortality rates. Data analyzed included staffing rates, facilities, and birth weight. The study concluded that higher staffing helped to neutralize birth weight factors in stillbirth rates. Principal Components Analysis was used in situations where the data was significantly related to the outcome and was highly correlated. They combined variables within the groups into more concise representations of their effects (Joyce, Webb & Peacock, 2003). The relation of this article to the work in this study stems from finding

variables that are highly correlated, as many software measurements tend to be (Munson, 2003).

2.3.6 Software Engineering Measurement

This textbook by Munson, referenced throughout this study, is a great tool for those learning the measurement and scientific processes in software engineering. His approach is used as the basis for the comparison of languages in this study. Munson (2003) looks at many of the measurements typically found in the software engineering world, analyses their usefulness, and uses this as a basis for creating a standard set of metrics. With a standard set of metrics, measurement becomes reproducible, and useful. But that is not all of what he describes. In addition to looking at raw measurement results, he goes on to say that measurements are nothing more than base data, and do not mean much without statistics. The tool he uses to create the statistics is PCA, from which a Relative Complexity Metric (RCM) can be found. This RCM value represents a one metric representation of a programming module's complexity (Munson, 2003). It is this process that will give valid and useful results for this project.

2.3.7 Applications of a Relative Complexity Metric for Software Project

Management

As in the above work, this article presents the concept of a Relative Complexity Metric in the use of software development. Munson (the author of the above work) and Khoshgoftaar (1990) discuss how such a one number representation may be used in software project management to the benefit of the team building the software. The

authors discuss that there is a problem with so many different metrics available that it is difficult to make sense of the data, and present the use of the one-number representation in the form of the Relative Complexity Metric on software projects. The calculation of the Relative Complexity Metric is done through the use of Principal Components Analysis and this process was completed on 27 pieces of software developed over several years by several developers. The authors find that there is an important correlation between writing-debugging time and the value of the Relative Complexity Metric, where higher numbers indicate seemingly more complex software and longer writing-debugging times (Munson & Khoshgoftaar, 1990). This research study uses the same approach as in this article as measurements are taken, and a Relative Complexity Metric is calculated giving meaning to the data collected. This article is an important piece of research that illustrates the validity of the approach in this project.

2.4 Conclusions on Related Work

From the first section of articles, it is clear that there is a large number of opinions and research in the study of languages. Many of the above articles lack scientific data and analysis to prove what the authors believe. Others, however, do, but these take different approaches that might introduce noise and other problems in the data that might not be visible. The point of using PCA as an approach and using the RCM that can be derived is to remove any possible noise or other useless variation that might be in the data. This study employs measurement data, analysis, and conclusions based on that analysis, and the work is objective and simply based on the scientific approach used. This study also hopes to serve as a future lesson in language comparisons.

In the second section of articles, those that use PCA as a scientific statistical approach, it is clear that it is a respected and useful tool. Many disciplines use this tool effectively and through the use of this tool, researchers are able to produce excellent, and useful results. In the software engineering world, it is possible the next phase in the evolution of software engineering is the use of PCA and RCM, producing a one-value representation of quality. Munson (2003) suggests this is possible, and has given plenty of evidence to support his claim. Even if this process does not take hold in the software engineering community, it still provides valid and reproducible statistical data from which future lessons can be learned.

CHAPTER III

EXPERIMENTAL DESIGN

3.1 Overview

Following is a description of each of the elements of the experiment. These are the independent variables, the dependent variables, the subjects, and the method of operation, each of which is an important piece that needs to be defined. Since the experiment described below requires funding, manpower, and resources beyond the scope of this study, the experiment actually conducted is a subset.

3.2 Independent Variables

In first describing this experiment, it is important to understand the independent variables in the study, those factors in which outside influence has no effect. These are actually very simple. The first is the programming languages themselves. All languages that are commonly used among students and industry professionals would be measured and studied. The second variable is the algorithms. Nine different algorithms, each performing different programming tasks, have been implemented in each language. Several problem spaces have been chosen for simulation in this research. The selection criteria for the languages and their compilers will come from market share reports from industry sources and the market share percentage must add to 70% of the general code

writing population. The algorithms chosen are frequently used and represent a cross-section of problem domains. Similarly, the languages and platforms are chosen for their popularity

3.3 Dependent Variables

For the design of this experiment, it is important to discuss the dependent variables that are associated with the above independent variables. Each language will need to be measured and compared, and these measurements depend on both the algorithm, and the programming language in which it was written. These measurements can also be individually looked at as their own variables, but regardless, in order for measurement to take place, it is necessary to have something to measure. Therefore, it is only possible for the measurements to be the dependent variables in this experiment. The most commonly used metrics among students and industry professionals will be used to describe the performance of each program. It is important to find metrics that can be reproducible, that are accepted by the general software engineering community, and that are valid and have meaning (Munson, 2003). There is, however, a second dependent variable: the statistical analysis that is performed on each set of measurements. In a way, this analysis is dependent on the measurements themselves, but transitively is still dependent on the programming languages, and the algorithms. Once again, only the most common statistical tools would be used. These tools are chosen from those commonly used in software projects as documented in the literature.

3.4 Subjects

As with almost all research studies, there must be a set of subjects that will be involved in conducting the experiment. In the case of this study, we have two very important subjects that must be discussed. The first is the programmers who write, test, and execute the code to ensure valid program execution. These programmers may each see a programming problem differently and therefore coding style might be a factor in measurement results. In order to account for the different types of coding styles that can appear a group of programmers is selected randomly at different levels. These levels include experienced professionals, graduate students, and undergraduate students, each with their own understanding of programming concepts. With this large range of skill level, it is possible to see how many different ways an algorithm can be coded, illustrating much of the way a particular programming language works.

The second subject is the set of compilers used. These are selected from the most commonly used sources both by students and in industry. Also, the compiler set includes work from both commercial development organizations as well as open source non-profit resources. The reason for several compilers is to compare the optimization techniques within each, as these may have an effect on dynamic measurement.

3.5 Operation of Experiment

3.5.1 Producing the Programs

Once compilers and developers have been chosen, programming can now begin. Developers will write each algorithm in each language given. Throughout the writing process, each program must be tested for correctness, ensuring that each program

produces the correct output. A program that is incorrect will introduce noise into the measurement data so it is important that each produce the intended results. Once all of the programs have been written, each is submitted to a set of measurement specialists that will produce all of the measurement data necessary for analysis. Once measurement has been completed, analysis can begin.

3.5.2 Performing Measurement

Each program is measured statically and dynamically, and with respect to the size and complexity of the resulting executable program (.NET metadata). Once all of these data has been gathered, it can be put through an intense statistical process. One thing must be clear, however, before beginning this analysis. This is a comparison of languages, not algorithms, and therefore, only programs written for one algorithm will be compared, rather than against all of the programs as a whole. It does not make sense, for example, to compare programs written to perform a string matching process and a sort. These are different problems and can therefore not be directly comparable. The measurements that are used must be chosen from research on the subject. So too must the statistical processes follow these same concepts. Since measurements are simply only raw data and have no meaning in and of themselves, statistics and analysis must be applied. The analysis must also be taken from research sources and must be generally accepted by the software engineering community. Chapter 6 defines one such practice as used by Munson (2003). Among the many statistical tools available, only ones that are relevant to the project are used.

3.5.3 Conducting Measurement Analysis

The first step in the analysis is to take all of the data on each program, and create simple averages of like units. This means that, for example, all of the Lines of Code measurements on each of the C language programs written to perform a string-matching algorithm will be made into a simple average as all of the individual measurements are a single value. This simple average will from this point forward represent the single measurement of Lines of Code, on C implementations of a string-matching algorithm. Now the statistical analysis can formally begin.

Several statistical tools will be used. These include finding the standard deviation, z-scores, and many other calculations. Also, a useful tool is Principal Components Analysis (PCA), in which a Relative Complexity Metric (RCM) can be found, a one number representation of all of the measurements taken on each language as applied to each algorithm (Munson, 2003). With all of this analysis data available, it is possible to determine that languages have an effect on algorithm performance, and which languages perform better given the problems presented to the developer writing the code. The higher the RCM value, the more complex and difficult writing the program becomes (Munson & Khoshgoftaar, 1990).

3.6 Threats to Validity

As with any empirical study, it is important to discuss any possible threats to validity. Following are definitions of the types of validity in question and a discussion of the possible threats.

3.6.1 External Validity

External validity refers to the degree to which the findings of the study can be replicated outside the context of the experiment. A research study is said to have external validity if the claims made from the results of the study can be generalized in other situations. The first threat to external validity is with regard to the choice of programmer, one of the subjects in this study. As discussed in the next section of this chapter, only one programmer will be writing the programs, testing, and performing the analysis. It is difficult to generalize any claims about programming languages from the abilities of one programmer. With only one programmer available, the variables of coding style and problem comprehension are over simplified. If several programmers completed the tasks of the experiment, it is likely that the results of this study may change and therefore be more general.

Another threat to external validity is with the choice of operating system. All of the programs of this study were run using Microsoft Windows XP. Each was executed several times under as close to the same conditions as possible to reduce measurement error. Since operating systems each have different specifications, requiring various amounts of background processes and memory usage, this can affect the dynamic attributes of the results. The choice of operating system also affects the method chosen to measure the complexity of the actual executable program itself. The use of .NET metadata, as one of the measurement categories, is only available from within the .NET environment and this is only found on Microsoft platforms. By changing operating system, .NET metadata is eliminated as a measurement category forcing the implementation of some other method. Another measurement method for executable

program measurement may allow for better generalization of the study's results. The choice of operating system is seen as a threat to external validity since the operating system is part of the environment in which the programs execute. To address this threat, additional operating systems might be considered for a fuller test of each program.

The choice of compiler also presents a threat to external validity. Compilers can have possibly unseen influence on the dynamic run-time attributes of a program. This is seen as a threat to external validity since the compiler is also seen as part of the programs' development environment. A different compiler might change the final results of the experiment's analysis. In order to better generalize the claims made from this study, additional compilers might be needed in order to test the programs more completely.

One last threat to external validity is with the choice of computer. Only one computer system was used to execute the programs. Computer systems each have different hardware specifications with different processor speeds and memory availability. Like the operating system, the actual hardware system is considered for this study as part of the execution environment. It is difficult to generalize claims having tested the programs on only one system. As with the operating system, to address this threat, additional computer systems would need to be used in order to more fully test each program.

3.6.2 Internal Validity

Internal validity refers to the relationships between the independent and dependent variables. A research study is said to have internal validity if there is evidence

to support that the independent variables cause the effects seen in the dependent variables. One threat to internal validity is with regard to measurement collection and analysis. It is possible that errors may have appeared in the general measurement collection process. This is seen as a threat to internal validity since errors can have an unwanted effect on the dependent variables, and should be as accurate as possible. To ensure proper accurate measurement data, tools were used with clear definitions for each metric. A tool was also used for the collection of analysis data. The measurement tools are described in detail in Chapter 6, and the analysis tool is described in detailed in Chapter 7.

A second threat to internal validity is concerned with dynamic measurement data. It is possible that errors may appear on the dynamic, run-time attributes of a program if something unexpected happens in the background processes of a given operating system. It is possible that these background activities within the operating system can have an effect on the final results. This is seen as a threat to internal validity since the measurement data should depend on the choice of algorithm and language, not the operations in the background of an operating system. To address this threat, each program was run several times and the measurements were taken on each run and then averaged together. This ensures that any values seen as outliers are removed before analysis begins.

Another threat to internal validity is with regard to algorithm implementation. It is possible that faults may be present in the source code itself. This is seen as a threat to internal validity given that unwanted noise can be introduced into the measurement data, and subsequently the analysis if incorrect output is discovered. Again, only the language

and algorithm choices should have an effect on the measurements taken in this experiment. To address this threat, randomized test cases were used and the output of each program was validated for correctness. By ensuring that each program returns correct output, errors in measurement data can be reduced.

3.6.3 Construct Validity

A study is said to have construct validity if what is measured actually supports or refutes the hypothesis. It is also concerned with ensuring what is measured is what actually should be measured in order to conduct a successful experiment. In order to remove threats to construct validity, measurements must be appropriate to the experiment itself. Since this is a study on programming languages, it is necessary to ensure that what is actually measured is the language and not its compiler. This is why static attributes on the source code itself are taken into account as part of the analysis of this experiment. Compilers do not affect the printed source code since a language has some form of standard syntax. All of the metrics that are used in this study, and why they were chosen, are explained in full detail in Chapter 6. Using the several measurement categories described will ensure that the languages are what is actually compared, allowing the experiment to support the hypotheses stated in Chapter 1.

3.7 Project Scope

The first of several major components for this research project is the programming languages themselves. Each algorithm chosen has been implemented in C, C++, C#, Java, and Visual BASIC. To ensure consistent results for later analysis, each

program is written using the Microsoft Visual Studio .NET Enterprise Edition environment. This gives the project a single tool, providing a common environment. Using compilers created in the open source software world might introduce variability into the measurement results since each of these compilers are engineered using different methods. The Microsoft tool offers one suite of compilers in which executable assemblies are created in the same format, a feature boasted by .NET developers (Petzold, 2001). In addition, accompanying this project is a discussion on each of the languages and how they evolved into what they are designed for today (Sebesta, 1999). Each programming language in this study has its own set of strengths and drawbacks causing differences in software performance (Pratt & Zelkowitz, 2001).

The second component to this project is the algorithms. Algorithms have been coded that do sorting, searching, mathematical calculation, string processing, and order statistic evaluations. For every algorithm implemented there is a discussion on why the algorithm was chosen, its important features, and a description of its time complexity. The algorithm and language discussions together will give the full scope of this research, providing the reasons why algorithms would perform differently from one language to another.

The third component for this project is the set of metrics and statistical analysis. Several metrics have been carefully chosen and defined using suggestions from Munson (2003). On each of the implementations, measurements have been taken and formatted so that the necessary statistical analysis can be performed. From this analysis, it can be determined which implementations had the best success (least complex measurement

results) for each of the algorithms, giving programmers a useful tool for choosing the best programming language for the implementations of various algorithms.

Each algorithm will be implemented by one developer using a specific coding style (Sedgewick, 1983) in each of the five .NET languages, and only the programs for a particular algorithm will be compared. There will not be a case where a program written that solves one algorithm will be compared to a program written in the same language, or any other language, that solves a second algorithm. This is not the purpose of this study. The purpose instead is to see how a particular algorithm behaves when a specific programming language is applied. The statistical tool used in this study to show the differences in behavior is PCA, producing the RCM value described earlier. The RCM values only relate to a single algorithm. Munson (2003) uses this approach in that he compares program modules by taking the same set of metrics on several program modules and compares them based on the RCM produced when PCA is used. The difference in this study is that languages are compared, not modules, and therefore it does not make sense to compare the programs written for different algorithms. The higher the RCM value, the more complex the program has become, and therefore, each program written to implement the same algorithm can be compared based on this value (Munson, 2003; Munson & Khoshgoftaar, 1990).

3.8 Understanding .NET Metadata

Microsoft has created an innovative approach to software development by allowing programs compiled in different languages to understand each other. While other areas of software development have utilized multiple languages in the same

project, the difference that Microsoft has introduced is that regardless of the language, the .NET Framework is available and uses the same function calls and the same set of classes creating a common interface. This common interface is contained in a set of dynamically linked libraries developed by Microsoft and these libraries are available for use on most Microsoft platforms. This cross language integration is done through the use of metadata (Petzold, 2001). The structure of .NET metadata is much like a database, containing tables of data that programs can search through and obtain information from regarding the way a program module functions. Each .NET assembly, be it an executable (EXE) or dynamically linked library (DLL), is compiled in the metadata format, allowing a module written in one language to be run from another language in the same suite. As an example, a portion of code or a class implementation written in C# may be used by Visual BASIC. This allows developers the choice of using a specific language better suited to the given problem with the ease of integration into a larger software project (Petzold, 2001). This project will serve to help developers take the best advantage of the languages offered for Windows platforms.

With the understanding of metadata within the .NET environment, measurements can be taken on the assemblies themselves. Assemblies in this context are defined as either dynamically linked libraries (DLL) or executable (EXE) files. These measurements will be independent of both the static and dynamic measurements that will be discussed later in this project. Understanding the complexities of .NET metadata will give a greater understanding of the performance of a specific algorithm when implemented in a particular language, although this can only be achieved when using .NET compilers found in the Microsoft tool set.

CHAPTER IV

THE PROGRAMMING LANGUAGES

4.1 The Environment

This project utilizes the Microsoft Visual Studio .NET Enterprise Edition environment. The .NET system provides several interesting features. Programs written in any of the .NET languages can take advantage of the Common Language Runtime Library (CLR). The CLR allows programmers to use the same set of classes across all of the languages that Visual Studio provides (Petzold, 2001). While each language is given the same class library as a tool for cross-language development, each individual language still retains its traditional approach. The Microsoft implementation of Java, for example, can use both the CLR and the standard Sun Microsystems set of functionality, and can use them both simultaneously (Petzold, 2001). The .NET environment also utilizes what is called Windows Forms, a method of generating source code through a design window where an actual application can be built visually. Prior to this approach, The Microsoft Foundation Class library (MFC) and Win32 API were used and were commonly coded by hand (Petzold, 2001). The .NET environment removes the repetitive programming tasks required of MFC and Win32 API. This project, however, does not take full advantage of these features. All of the algorithms are implemented to function solely on the command line. No graphical elements have been used to run the algorithms. It is important to note, however, that the .NET editor and environment provides a common

and identically engineered environment from one language to another without the use of multiple tools. Also, by using the Microsoft set of compilers, it is possible to take advantage of one engineering model. Each compiler associated with the .NET Framework is built to the same standard (Petzold, 2001), allowing the most common environment possible with the purest possible data collection.

4.2 The C Programming Language

The C Programming Language was developed by Bell Laboratories in 1972 and evolved from the ALGOL 68 project. It was originally developed for systems programming and was made for low-level architectures (Sebesta, 1999). The language was standardized for the first time in 1978 and has been used in a wide variety of areas. The programmer has control over much of the memory management. Variables and pointers are declared statically, forcing programmers to declare them before their use. Pointers are exceptionally interesting as the programmer has direct access to what is contained in the memory address, rather than through automatic dereferencing and can add and subtract memory during run-time (Pratt & Zelkowitz, 2001). C has an entry point in a “main” function in which the operating system passes control to the process created by the program.

C has many useful control constructs common to many other languages. These include features like loops, conditional jumping, and data control. In addition to these built-in controls, the C language can also take advantage of recursion in which routines may call themselves in order to decompose a larger problem into several pieces. Also, C allows for casting, which can change a data type from within memory that has already

been set-aside at compile time. For example, an integer can be changed “on the fly” to a floating-point representation. There are dangers with doing this, however, as precision can be lost and cast values can have unexpected results (Pratt & Zelkowitz, 2001). While C is not traditionally an object oriented language and provides a much more sequential approach to programming, data can be structured into records by using the “struct” keyword (Pratt & Zelkowitz, 2001). Using this keyword, a programmer has the option of structuring sets of data into manageable, reusable pieces. Linked lists are often made from pointers to “struct” type data sets.

The C Programming Language provides the programmer with several primitive data types for use in many areas. Included in these data types are 32 bit integers, 64 bit long integers, 32 bit floating-point real numbers, 64 bit double floating-point real numbers, characters, and bool values (true or false). Each of these primitive types can be structured into an array of one or more dimensions. This can create easy maneuvering through sets of values of the same type for inserting and retrieving. The only strange case among these is character strings. Character strings are actually represented in terms of arrays of characters, or “char” values. The strings can be traversed in much the same way as their numerical counterparts (Pratt & Zelkowitz, 2001). Programmers, in addition to these primitive types, have the option of creating their own in using the “struct” keyword, or using the “enum” keyword. The “enum” keyword allows a programmer to assign a numerical value to a set of characters and these characters can be used in place of the primitive types. Also, the “typedef” keyword gives the programmer the option of creating original new data types that are software specific.

C, however, does have its drawbacks, as do all programming languages. The language has no specific standard input or output built into the language. In order to take advantage of input and output, the programmer must include libraries contained in header files at the top of the source code (Pratt & Zelkowitz, 2001; Sebesta, 1999). It is here that the developers of the language create the streams for displaying or collecting data to and from standard sources respectively. Users of the language, however, can create their own header files that contain function prototypes, declared data structures, and other constructs that have been implemented in other source code files for use in programs (Pratt & Zelkowitz, 2001).

4.3 The C++ Programming Language

C++ as a language is almost exactly like C in many ways, and evolved from C directly (Sebesta, 1999). Its main feature and benefit is the introduction of Object Oriented Programming (OOP) to a C-like environment (Pratt & Zelkowitz, 2001; Sebesta, 1999). With OOP, concepts such as inheritance, polymorphism, and encapsulation, which typically define what an object oriented programming language offers, can now be used (Pratt & Zelkowitz, 2001). Given to the programmer is a new keyword, “class.” A class is a definition of an object that both performs work and retains data simultaneously. Instances of class objects can work independently of each other as well as communicate through message passing. Most typically, programmers of C++ create a header file containing the class’s definition, and another source file in which the methods in the class are implemented. The term “method” is used here to describe functions within class definitions. C++ has an external entry point, which means that

class instances must then be declared within the program's main function. The main function is called by the operating system's shell and is used to pass control from the operating system to the program. Typically in C++ programs, the main function exists in its own source code file (Pratt & Zelkowitz, 2001; Sebesta, 1999).

C++ has the same primitive data types as C, and data enumeration is also handled the same way. C++ also has at its disposal the "struct" keyword and can structure data into records much like C (Sebesta, 1999). Data structures can also be members of classes and can be incorporated into objects in much the same way as integers or character strings. Instances of class objects may also be members of other classes. Unlike C, however, C++ gains the benefit of using the Standard Template Library (STL). Through STL, constructs like maps (a type of array construct), iterators (a type of looping control), and standard strings (an STL character string object) become available. While C++ can still use an array of char type values to represent strings, the language also has the option of the standard string, which not only contains the string's value, but can operate on it as well (Pratt & Zelkowitz, 2001). Although not part of the original language, STL has become part of the ANSI standard for C++ implementations (Sebesta, 1999).

C++ can be used in a variety of situations. Although not designed for systems programming, it is possible to do such tasks with C++ (Pratt & Zelkowitz, 2001; Sebesta, 1999). It is more designed for encapsulation of data and routines (methods) and can easily create applications from within several environments (Pratt & Zelkowitz, 2001). The language has been used on Windows, Unix/Linux, Macintosh, and the IBM OS series (OS/2, OS/390, etc.). Implementations, however, differ from one platform to another and there can be times when different code is needed for the same program to

compile when ported to another platform. If the programmer stays with the ANSI standard it is possible to port programs to additional platforms with almost no changes in the source code (Sebesta, 1999).

C++ has many of the same drawbacks as C. An additional step is needed to link all of the source files together to create the executable program. After compiling the program, the compiler creates object files containing hexadecimal instructions matching that of the hardware architecture. In order for the executable to be created, these files must be linked together in a separate step and all external, non-static entries must be resolved. The potential for additional errors is possible and debugging can sometimes be a tedious process (Pratt & Zelkowitz, 2001; Sebesta, 1999).

4.4 The C# Programming Language

Microsoft has created a new language that can function with the .NET environment. This language, known as C#, is a hybrid of Java, C and C++ (Petzold, 2001). The entry point for a C# program is contained within a class definition and is statically bound to be used by the operating system. Unlike C and C++, C# uses a virtual machine architecture and has the potential to be completely platform independent (Petzold, 2001). What this means is that if another company were to create a new implementation of the C# virtual machine, programmers could write C# applications in the same way that they previously were able in Microsoft Windows. Currently there are several open source projects that focus on making C# available to Unix/Linux programmers. Microsoft does not affiliate themselves with these developers since its implementations are proprietary, and code is not shared with the general public. While

C# is primarily interpreted, it can be compiled to an executable program with the virtual machine included. This can cause a large amount of memory to be used but allows additional portability between machines (Petzold, 2001). The virtual machine is not platform independent, but the C# code potentially can be, and recompiling on another platform should warrant no change in the C# source.

C# is fully object oriented. It allows for all of the benefits of object orientation, meaning objects can take advantage of inheritance, polymorphism, and encapsulation (Petzold, 2001). C# has several primitive data types available for use. These include 32 bit signed integers, 32 bit unsigned integers, 64 bit long integers, 64 bit unsigned long integers, 8 bit bytes, 16 bit short integers, 16 bit unsigned short integers, 32 bit floating-point real numbers, 64 bit floating-point real numbers, characters, and bool (Petzold, 2001). Different from C and C++, coupled with the language are character string objects that are part of the standard library of classes. While STL was an add-on to the original ANSI Standard C++ language, the C# equivalents came as an original feature. These string objects both contain and can operate on the string value. These strings are class objects as any other in C#. Standard input and output stream classes are also available and no additional libraries are necessary (Petzold, 2001). C# also has many of the same program flow constructs that are a benefit in C and C++. Looping, conditional jumping, and object communication are all available within C#. An important language feature to note is that of C#'s memory management. Garbage collection and the handling of pointers is taken care of automatically by the language, freeing a programmer from having to do this manually as in C or C++ (Petzold, 2001).

C# source code has the interesting benefit of easy integration within web pages. HTML browsers can load small programs into these pages and the programs can perform a variety of tasks. These tasks range from user authentication, database look-up and insertion, and online gaming (Petzold, 2001). HTML browsers, however, must be equipped with a C# interpreter in order for this to be possible from within a web page. This research project does not take advantage of this feature in any way, but it is important to note as a feature of the language.

Although this research project does not take advantage of graphical user interfaces, it is important to note that as part of the language's environment, a window design editor is included. Programmers can easily build window prototypes of the application in order to see quickly what it will look like (Petzold, 2001). The code generated is included in the main libraries of C# and additional libraries are not always necessary. All of the standard Microsoft Windows controls are available to the programmer and through this method application development becomes much faster and less error prone (Petzold, 2001). Microsoft contends that C# is a perfect hybrid of Java, C, and C++, which creates a better, more efficient language for all Windows developers (Petzold, 2001).

C#, like any other language, also has its drawbacks. As a result of the virtual machine architecture and full interpretation, programs written in C# potentially suffer in performance. C# programs can be compiled directly to an executable, but the virtual machine is coupled within the executable, as it is necessary for these programs to run properly (Petzold, 2001). As will be seen from the measurement results, C# programs use large amounts of available memory for even the simplest of programming tasks.

Another drawback of C# is that not all web browsers come with C# interpreters, unlike Java, which tends to be more universal across almost every major computing platform (Sebesta, 1999). In order for web page embedded code to function, an interpreter must be present within the web browser. This can limit the programmer's browser choices, forcing the users of the application to be limited as well.

4.5 The Java Programming Language

Java was developed by Sun Microsystems in the mid 1990s and was used originally as a C++-like language for web page embedded programming (Sebesta, 1999). Much like C#, its entry point is coupled within a class object and is statically bound for use by the operating system. Java, as with C#, is a fully object oriented programming language and can also take advantage of inheritance, polymorphism, and encapsulation (Pratt & Zelkowitz, 2001). Java code is also run through a virtual machine architecture, and while the virtual machine's own implementation may be platform specific, code written in Java is not. In order to be certified by Sun as a standard Java implementation, the virtual machine must comply with all of Sun Java's features (Pratt & Zelkowitz, 2001).

Java has many of the same primitive data types as C#. These include 8, 16, 32, and 64 bit signed and unsigned integers, 32 and 64 bit floating-point real numbers, 16 bit Unicode characters, and bool. Java also has an object class available for character strings. These objects both contain and operate on the string much as C# does (Pratt & Zelkowitz, 2001; Sebesta, 1999). Standard input and output are handled through Java's extensive class library of objects and, in most cases, additional libraries are not necessary

for these tasks (Pratt & Zelkowitz, 2001). As with the other languages, the control constructs include looping, conditional jumping, and object communication. Memory management is also handled directly by the language structure, allowing programmers less worry in regard to pointers and their values, which may not be directly accessed by the programmer. Garbage collection is also handled automatically, leaving less room for memory leaks to occur (Pratt & Zelkowitz, 2001).

Java, like C#, has the additional benefit of being integrated into web pages. These programs, in the Java context, are known as applets (Pratt & Zelkowitz, 2001). Applets can take the form of games, database look-up and entry forms, user authentication and password protection, and many other types of applications (Pratt & Zelkowitz, 2001: Sebesta, 1999). Java is more universal than C# as more browsers are available with a Java interpreter. Java has been successfully implemented and certified by Sun on both open source and proprietary platforms. Although not used in this project, its “Swing” library can create standard graphical elements like check boxes, radio buttons, etc. that can be used in Unix/Linux, Microsoft Windows, and Macintosh, and can take on the motif of each (Pratt & Zelkowitz, 2001).

Java has several drawbacks. The language has very limited console application and is better suited for graphical environments. This can cause performance problems when Java programs are run over a command line interface (Pratt & Zelkowitz, 2001). Java, like C#, uses a virtual machine, and while Java code can be compiled to an executable, the virtual machine must be coupled within the code at the cost of extra memory (Sebesta, 1999). Also, while most browsers come equipped with Java

interpreters, this does not mean that all browsers do, limiting the choices of both the programmer and the user of the application (Pratt & Zelkowitz, 2001).

4.6 The Visual BASIC Programming Language

Visual BASIC (VB) started out as BASIC, the Beginners All Purpose Symbolic Instruction Code. It was designed for the liberal arts students at Dartmouth University in the 1960s (Sebesta, 1999). While science students had little trouble with ALGOL or FORTRAN, there was a need among liberal arts and other non-science students for a language that was easy to learn and friendly for a fast homework turnaround (Sebesta, 1999). In the beginning, BASIC had no way of accepting interactive input and, as a result, programs were written to be run in a batch, much like FORTRAN. Although Digital Equipment Corporation used BASIC to write one of its operating systems, the language was never really meant for large-size applications of great significance, which is why its greatest criticism is its poor program structure (Sebesta, 1999).

In the mid 1980s, development of Quick BASIC (QBASIC) by Microsoft enhanced BASIC for a greater range of use (Sebesta, 1999). Standard input and output became available and sub routines became easier to create. Unlike C or C++, the input and output systems were an integral part of the language, requiring no additional libraries. With the changes made to BASIC by Microsoft, users could still benefit from its ease of use while creating significantly large sized applications (Petzold, 2001: Sebesta, 1999). QBASIC does not, however, contain a library for use in creating graphical user interfaces.

In the early 1990s, Microsoft created another version, now known to its users as Visual BASIC. It was one of the first languages to incorporate a design window for application prototyping (Sebesta, 1999). After a window is created, programmers can then “attach” QBASIC code to the objects on screen, creating an event driven environment. Through this unique development model, programmers are able to produce full-size, quality applications in smaller amounts of time. With the innovations that Microsoft has made to this language, VB has become a fully functional language in its own right (Petzold, 2001).

VB, as with the other languages in this project, has a set of primitive data types. These include characters, character strings, integers, floating-point real numbers, and bool. The choices are much more limited in VB than in the other languages, as it was believed that not many of its users would gain benefit from more than this (Sebesta, 1999). VB behaves much like an object oriented language utilizing encapsulation, but lacks polymorphism and inheritance. Its entry point is a statically bound main routine from within an object (Petzold, 2001). Visual BASIC tends to be easier to read than most languages with its statements appearing in an almost English-like structure. The language has an extensive library of objects that may be used in graphical Microsoft Windows interfaces. It includes all of the components frequently found in Windows and can be used to develop applications quickly and with little need for repetitive and tedious programming tasks (Petzold, 2001).

The drawbacks of VB are numerous, as again it was never meant to be a language for significant application development (Sebesta, 1999). VB is also a language run through an interpreter. Although VB source code can be compiled to the form of an

executable, the interpreter must be coupled within causing large amounts of memory usage and significant performance problems. In addition, while the English-like structure allows for readability, it forces many more reserved words to be used, frequently making programs less readable (Sebesta, 1999). Visual BASIC, in its current form, lacks significant functionality for use within command line interfaces. It is much more suited for graphical environments, specifically Microsoft Windows. Few other operating systems can use the language making application portability extremely limited. The form of BASIC used with other operating systems resembles more the original form of the language with minimal input and output capability and is exclusively limited for applications on the command line interface (Sebesta, 1999).

CHAPTER V

THE ALGORITHMS

5.1 Definition of Selection Criteria

Before choosing algorithms for this research study, it was important to understand exactly what algorithms are. An algorithm can be simply defined as a computational process, in which input is given, work is performed, and a useful set of output is found (Cormen, Leiserson, Rivest, & Stein, 2001). Since algorithms are created to solve problems, it was imperative to decide which programming problems were most important before choosing the algorithms to solve those problems. Since it is impossible to examine every imaginable programming situation, it became important to find problems that would be useful for research purposes. The problems chosen in this study are used often in industry and are applied to many applications (Cormen et al., 2001). The criteria for choosing the problems in this study include how much research is available on each, and how often the problem is used in industry, while at the same time keeping them small enough so that they may fit into the time constraints involved in the course of this project.

It is next important to understand the differences between what simple and complex algorithms are. Simple algorithms tend to use the “brute force” approach. The simplest solution may be the easiest to implement but it may not be the best solution. Simple algorithms tend also to be slower and take more computational instructions to complete (Cormen et al., 2001). Complex algorithms, on the other hand, are usually

better solutions than the simple and tend to make better use of processor time and are typically faster. These algorithms might be more difficult to implement. Programmers must often decide between either creating software faster, or creating faster software (Cormen et al., 2001).

Five programming problems were chosen that often occur in software development and have been heavily researched. These problems are searching, sorting, mathematical computation, string processing, and order statistics. Each of these problems has several characteristics that are important. Searching and look-up is found very often in database programming as well as simpler applications. Sorting is another problem widely used in database applications as well as for the display of items on screen. The need to solve systems of equations arises frequently in mathematical, scientific, and engineering computational problems. String processing can be found in compilers for syntax highlighting, spell checking, and other systems and application programming (Cormen et al., 2001). The finding of order statistics was chosen so that each language can be measured on how well it performs when there is perhaps a more optimal algorithm available. This gives insight into how a language can perform when presented with an unusually lengthy problem. With the exception of searching, a complex and a simple algorithm was chosen to solve each of these problems.

5.2 Definition of Implementation Criteria

In writing the implementations for the algorithms presented in this chapter, an example was taken from Sedgewick (1983) with regard to the use of the languages. He states in his book that in order to best highlight the constructs of each language, only the

simplest of language constructs should be used, and also as few comments as possible (in this study's case, no comments). This study uses this approach so as to keep the code easy to read and in a common format. The variable names are small and perhaps non-descriptive but again this further highlights the control constructs contained in the algorithms by keeping them more visible. In addition, the function names are the same as in the pseudocode diagrams found below, simply describing what each function does. With this common format, the implementations from language to language can be compared on several different levels. Sedgewick (1983) explains that this method of algorithm implementation is a positive approach when the project is under time and resource constraints such as this project's scope dictates.

One thing that must be defined clearly before moving on to the algorithm descriptions is what is meant by simple vs. complex. For purposes of this study, algorithms considered to be simple use the "brute force" approach. Simple algorithms take a naïve look at problem solving. These algorithms are Linear Search, Bubble Sort, Naïve String Matching, Polynomial Addition, and Minimum / Maximum. Simple algorithms obtain a solution to a problem without regard to efficiency or elegance and are often easier to understand from a programmer's perspective. It is possible, however, for a simple algorithm to be the most efficient.

Complex algorithms, on the other hand, use more elegant techniques, such as recursion for example. These algorithms are not naïve, and can often be difficult to understand from the perspective of a programmer. Complex algorithms work to find problem solutions that are efficient, even though some simple algorithms perform faster

over smaller data sets. For purposes of this study, the complex algorithms implemented are Quicksort, KMP String Matching, Gaussian Elimination, and Random Selection.

5.3 Searching

The searching problem may be described as follows: given an array L indexed from 0 to n , and a key k , k is searched against in the array comparing each element of the array to k . The first index in which k is found in L is returned, otherwise -1 is returned, indicating that k is not an element of L .

5.3.1 Linear Search

Linear Search was chosen to solve the searching problem. It is one of the most widely used searching algorithms in computer science and uses important features common in most programs. This algorithm is often one of the first taught to computer science and programming students in high schools and colleges (Cormen et al., 2001). Linear Search, while not complex, is optimal. It has a worst case and average case time performance of $O(n)$, and it is one of the simplest algorithms to implement (Cormen et al., 2001). It uses a single loop to walk through the elements included in the search, looking for a key that perhaps does or does not exist. Each element of the searched area, an array, a string, or a linked list, is compared with the key. The algorithm returns the index in which the first instance of the key can be found. This is commonly known as the “brute force” approach, as is described by Cormen et al. (2001).

```

LinearSearch (L, k)
{
    i = 0

    while (i <= n and L[i] <> k)
        i = i + 1

    if (i > sizeof(L))
        i = -1
}

```

Figure 5.1 Linear Search Pseudocode.

While this algorithm may not seem complex, as noted by the figure above (Cormen et al., 2001), there are important things to note. The algorithm uses one of the most common programming constructs, a “for” loop. In the implementations used for this research, integers are used for the search criteria, so the best data structure for containing all of the data is an array of integers. Since the algorithm is not complex, each programming language used less memory than some of the algorithms’ more complex counterparts. All data was generated randomly so that average case performance could be measured over the course of running the program several times.

5.4 Sorting

The sorting problem can be described as follows: given a random sequence of n numbers (a_1, a_2, \dots, a_n) , find a permutation of the original sequence such that $(a'_1 \leq a'_2 \leq \dots \leq a'_n)$ (Cormen et al., 2001).

5.4.1 Bubblesort

The Bubblesort is one of the most commonly used algorithms in computer science and is another often taught first to computer science students (Cormen et al., 2001). It is one of the simplest yet slowest algorithms that solve the sorting problem and uses sorts in place (Martin, 1971). For small data sets this algorithm is sufficient as it is easy to understand and implement. Larger sets of data require something more advanced if performance is an issue (Cormen et al., 2001). Bubblesort runs in $O(n^2)$ time as every element must be compared to every other element. After two adjacent elements are compared, if one element is larger (or smaller depending on the sort), then the two elements will switch places (Martin, 1971). Once no change occurs, the data is sorted, and the algorithm terminates. The version used in this research study is that of Cormen et al. (2001).

```
BubbleSort (A)
{
    for (n = 1 to sizeof(A))
        for (m = sizeof(A) to n + 1 step -1)
            if A[m] < A[m - 1]
                exchange(A[m], A[m - 1])
}
```

Figure 5.2 Bubblesort Pseudocode.

Appearing in this algorithm is a nested loop since each item must be compared to every other item. This is why the time analysis for Bubblesort is $O(n^2)$ (Cormen et al., 2001). Again integers were used for the data in the program and so the most appropriate data structure was an array of integers. This algorithm, like Linear Search, is very simple and should not require large amounts of memory to run or large numbers of lines of code

to write. The results for Bubblesort vary from Linear Search in terms of the programming languages and their performance. Again, all data for this algorithm were randomly generated so that the average case could be measured. The worst case is still possible, however, and happens when the array is in the reverse order from that desired (Cormen et al., 2001).

5.4.2 Quicksort

Quicksort is probably the most used algorithm that solves the sorting problem and is considered to be one of the most complex. It has been used in many situations in both systems programming and application programming (Cormen et al., 2001). The algorithm was originally developed by C.A.R. Hoare in 1961 and was published in the Computer Journal in 1962 (Hoare (Algorithm 64), 1961, 1962). The idea was that a new algorithm could be created that solves the sorting problem by reducing large problems into trivial simple ones, thereby solving the large problem as a whole using known methods once the problem is reduced (Hoare, 1962). The version of the algorithm presented in this study is that of Cormen et al. (2001). With a timing of $O(n \log n)$, Quicksort is much faster than Bubblesort in the average case. Quicksort uses a divide and conquer approach commonly implemented through recursion. It can be implemented iteratively, but for this research, recursion was chosen to illustrate how programming languages handle this common practice. First, a partition procedure is used to place an element (the pivot) where it belongs in the final sorted set of elements (Hoare (Algorithm 64), 1961, 1962). All elements on the left are smaller than the pivot, and all elements on the right are larger than the pivot (Hoare (Algorithm 63), 1961). At this point, the

Quicksort procedure is called again on each side of the partition and the process repeats until there are no longer changes necessary and the elements are sorted. The algorithm has an implied loop created from its recursive structure (Cormen et al., 2001). Quicksort is ideal for large sets of data and can reduce processing time for its application when compared to Bubblesort.

```

Partition (A, p, r)
{
    k = A[r]
    j = p - 1

    for (n = p to r - 1)
        if (A[n] <= k)
            j = j + 1
            exchange(A[j], A[n])

    exchange(A[j + 1], A[r])

    return j + 1
}

Quicksort(A, p, r)
{
    if (p < r)
        q = Partition (A, p, r)
        Quicksort(A, p, q - 1)
        Quicksort(A, q + 1, r)
}

```

Figure 5.3 Quicksort Pseudocode.

The important aspect of Quicksort for this research is that the recursive implementation is used so that languages can demonstrate how they behave when this technique is applied. Quicksort's main procedure calls itself until the sort is complete. Only the pointers are passed and the sort is done in-place. The data structure once again was the array of integers, and all values were generated randomly so that the average case could be measured.

5.5 String Matching

The string matching problem can be described as follows: given a string of text $T[1 \dots n]$ and a string pattern $P[1 \dots m]$ of length $m \leq n$, we say that the pattern P occurs

within the text T at a valid shift s . If there is no valid shift s , then the pattern P does not exist in text T . Returned are all of the beginning indices of the pattern in the text (Cormen et al., 2001).

5.5.1 Naïve String Matching

The Naïve String Matching algorithm is a simple algorithm that is used to match patterns against larger sets of characters, also known as the string matching problem. It is slow over large data sets when compared to other string matching algorithms as a result of every element in the pattern having to be compared with every element in the main text. Naïve String Matching is timed at $\Theta(n \times m)$ where n is the size of the text and m is the size of the pattern (Cormen et al., 2001). This algorithm has often been described in terms of a slide rule, where the pattern is slid across the text and reports every time the pattern is found. Naïve String Matching is ideal for small sets of data but does not perform well over larger sets (Cormen et al., 2001). The algorithm terminates once the number of characters in the text left to compare is less than the number of characters in the pattern.

```
NaiveStringMatch (T, P)
{
    j = sizeof(T)
    k = sizeof(P)

    for (s = 0 to j - k)
        if (P[1...k] = T[s + 1...s + k])
            print ("Pattern occurs with shift %d", s)
}
```

Figure 5.4 Naïve String Matching Pseudocode.

Implementing Naïve String Matching presented several challenges. Results varied greatly from language to language and in some cases passing pointers to arrays of characters proved difficult. The algorithm uses a nested loop structure, however, the inner loop and the outer loop do not have the same termination value which is why the timing is not $O(n^2)$ (Cormen et al., 2001). Randomly generating the data for this algorithm did not prove economical in that randomly creating a data set and pattern does not guarantee that the pattern will be matched from within the text. Instead, the song “Take Me Out to the Ball Game” was used and the pattern searched was the word “ball.” Also used was the text of the United States Constitution, and the pattern “the.” In addition, another data set in which the pattern was not found was also used. For every run on this algorithm, each element in the text must be compared to every element in the pattern (Cormen et al., 2001) and therefore randomly generating data would not have proven useful. The profiler used to take dynamic run-time measurements was able to get accurate timings based on these problems.

5.5.2 KMP String Matching

Developed by Knuth, Morris, and Pratt, the KMP String Matching algorithm is a more efficient alternative over Naïve String Matching (Knuth, Morris & Pratt, 1977). Through the algorithm’s routine to compute prefixes, information can be gathered on both the text and the pattern so that unnecessary comparisons can be eliminated. With this prefix information, a non-match can be assumed right away once the prefix of the current position in the text does not match the prefix of the pattern (Knuth et al., 1977). This information can be assumed without testing the entire pattern, which is what Naïve

String Matching does. At this point, the algorithm moves the pattern along the text in the amount of the length of the pattern forward to try a new match in which the process begins again. The algorithm terminates once the entire text has been searched and all relevant matches have been found (Knuth et al., 1977). This makes KMP String Matching a much better candidate for large data sets (Cormen et al., 2001: Knuth et al., 1977).

With the elimination of the unnecessary comparisons of characters, the algorithm can be timed at $O(n \log m)$ in the average case where n is the size of the text and m is the size of the pattern (Knuth et al., 1977). This Algorithm is considered to be a complex algorithm when compared to Naïve String Matching and is used widely in many applications (Cormen et al., 2001). While this algorithm was created by Knuth, Morris, and Pratt (1977), the implementation used here is that from Cormen et al. (2001).

```

KMPMatch (T, P)
{
    n = sizeof(T)
    m = sizeof(P)
    Pi = ComputePrefixFunction(P)
    q = 0

    for (i = 1 to n)
    {
        while (q > 0 && P[q + 1] <> T[i])
        {
            q = pi[q]

            if (P[q + 1] = T[i])
                q = q + 1

            if (q == m)
                print ("Pattern occurs with shift %d", i - m)
                q = pi[q]
        }
    }
}

```

Figure 5.5 KMP String Matching Pseudocode.

As with Naïve String Matching, KMP String Matching presented many programming challenges. One of the larger challenges in implementing this algorithm was the passing of character strings as arguments as each language has its own format for doing such a thing (Pratt & Zelkowitz, 2001). The algorithm uses loops in order to compute the prefix of both the pattern and the text. Each language performed very differently with regard to this algorithm as will be seen from the data presented later. Again, as with Naïve String Matching, it was not economical to create random data for this algorithm. All of the same text and pattern combinations were used as was with Naïve String Matching. Because the performance of the algorithm is not based on random data, results were very consistent from one run of the program to the next.

5.6 Arithmetic Algorithms

Two mathematical problems are addressed in this section. The first is polynomial addition. The problem is defined by saying that there is a set of coefficients $C(x)$ such that $A(x) + B(x) = C(x)$ where A and B are arrays that represent coefficients in polynomials. A polynomial can be described as an algebraic expression consisting of one or more summed terms, each term consisting of a constant multiplier and one or more variables raised to integral powers. For example, $x^2 - 5x + 6$ is considered a polynomial. The new polynomial with coefficients in C will be of the same degree as A and B. Degree in this context is defined as the term raised to the highest power. It is possible to add two polynomials of different degrees, but the sum is expressed in terms of the highest degree (Cormen et al., 2001).

The second problem that is solved in this section is that of finding solutions to systems of linear equations. Given the equation $Ax = b$ where A is an n by n matrix containing the coefficients of each linear equation, b is the set of constants, and x is the set of n unknowns, we wish to find the set x so that $Ax = b$. If A is non-singular, there will be a unique solution to the system (Cormen et al., 2001).

5.6.1 Polynomial Addition

The addition of polynomials, the adding together of expression coefficients, is a simple illustration of arithmetic as done through computer programming (Cormen et al., 2001). Polynomial Addition is often used in applications that involve algebra and other uses in mathematics and science. It is not a very complex process and has a run-time of $O(n)$, where n is the number of coefficients in the polynomial, including zeros. This time performance is for all cases. Polynomial Addition is the simple pairing of coefficients and combining them into a new polynomial of the same power, such that $A(x) + B(x) = C(x)$ (Cormen et al., 2001). Written in the general form we can say that $A = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \dots + a_{n-1}x + a_n$ and $B = b_0x^n + b_1x^{n-1} + b_2x^{n-2} + \dots + b_{n-1}x + b_n$. Using this general form we can say that $C = (a+b)_0x^n + (a+b)_1x^{n-1} + (a+b)_2x^{n-2} + \dots + (a+b)_{n-1}x + (a+b)_n$. It is possible for a coefficient to be equal to zero, in which case a zero is placed in the array. The study of this algorithm allows for greater understanding of how each of the programming languages performs while completing simple mathematical calculations.

```

PolyAdd (A, B, C)
{
    i = sizeof(A)

    for (n = 0 to i)
        C[n] = A[n] + B[n]

    return C
}

```

Figure 5.6 Polynomial Addition Pseudocode.

In the implementations of this algorithm, random data was created to fill two arrays of integers. Each array element represents a coefficient of a polynomial. The i^{th} element in the first array is added to the i^{th} element in the second array, producing the i^{th} coefficient of the sum (Cormen et al., 2001). This was done through a “for” loop and the process terminates once all of the coefficients have been added together producing the new array. An important point to note is that even though random data were used, all cases perform the same, as each element in each array must be processed regardless of value (Cormen et al., 2001). This is important because each programming language yields easily comparable measurement results when the metrics are applied.

5.6.2 Gaussian Elimination

Gaussian Elimination is a mathematical algorithm that has been in use for more than 150 years (Cormen et al., 2001). It is used to solve for the unknowns in a system of linear equations. This algorithm is more complex than the addition of polynomials. As a result of its 3-level nested loop structure, Gaussian Elimination has a timing analysis of $O(n^3)$ (Cormen et al., 2001). The purpose of the algorithm is to solve for any number of unknowns as long as the number of unknowns is equal to the number of equations. The

product of this algorithm is the unknowns themselves, resulting in the solution to the system of equations (Cormen et al., 2001).

```

LUPDecomposition (A)
{
    k = Rows(A)
    Pi = 0
    p = 0

    for (x = 1 to k)
        Pi[x] = x

    for (y = 1 to k)
    {
        p = 0

        for (x = y to k)
        {
            if (A[x][y] > p)
                p = A[x][y]
                k' = x
        }

        if (p == 0)
            print("Error, Singular Matrix")

        exchange(Pi[y], pi[y'])

        for (x = 1 to k)
            exchange(A[y][x], A[y'] [x])

        for (x = y + 1 to k)
        {
            A[x][y] = A[x][y] / A[y][y]

            for (z = y + 1 to k)
                A[x][z] = A[x][z] - (A[x][y] * A[y][z])
        }
    }
}

LUPSolve (L, U, pi, b)
{
    n = Rows[L]

    for (i = 1 to n)
        y[i] = b[pi[i]] - sumof(i - 1, j = 1, (L[i][j] * y[j]))

    for (i = n to 1 setp - 1)
        x[i] = (y - (sumof(j = i + 1, n, (U[i][j] * x[j]))) / U[i][i])

    return x
}

```

Figure 5.7 Gaussian Elimination Pseudocode.

Gaussian Elimination uses two routines each of which performs distinct tasks.

The first routine factors a permutation of the matrix of coefficients into an upper triangular matrix and a lower triangular matrix. After this is complete, the second routine can perform the back substitutions needed to solve for the unknowns themselves (Cormen et al., 2001). Random integer data was used to generate the matrix coefficients. For the first time in this research study, floating-point arithmetic is used (Cormen et al., 2001).

5.7 Order Statistics

The i^{th} order statistic of a set of n elements is the i^{th} smallest value in the set. The minimum is the smallest element in an array. The maximum is the largest element in an array. If the minimum is desired, $i = 1$ and if the maximum value is desired then $i = n$ (Cormen et al., 2001).

5.7.1 Minimum and Maximum

In order to find the minimum and maximum values in a given data set, every value must be addressed. This produces results in the worst case and average case time of $O(n)$ and is optimal (Cormen et al., 2001). By using this algorithm, we can see how each language performs under the worst case using only elementary operations (Cormen et al., 2001). This is different from sorting in that we are not re-ordering the set of elements, only returning a single element from the set.

```
Minimum (A)
{
    m = A[1]

    for (x = 2 to sizeof[A])
        if (m > A[x])
            m = A[x]

    return m
}
```

Figure 5.8 Minimum Pseudocode.

The process for determining the minimum and maximum values in a data set is simple and both statistics use the same process with only a slight modification. In the case of the minimum value, the first element in the data set placed is in the “leader” variable, meaning that the first element in the array is the smallest (Cormen et al., 2001). This value is then compared with the next element in the array and if this new element is smaller then it is in turn placed in the “leader” variable. This process is repeated until the entire data set has been compared (Cormen et al., 2001). Finding the maximum value is similar. Random generation of integers is used to produce the data set.

5.7.2 Random Selection

The final algorithm chosen for this research study is Random Selection. This is an algorithm that performs in $O(n^2)$ for worst case time and $O(n)$ expected time (Cormen et al., 2001). This algorithm produces the i^{th} smallest number in the data set and returns this value to the calling routine. The reason this algorithm performs in $O(n^2)$ worst-case time is that it is compared to every element in the data set (Cormen et al., 2001). The difference between this algorithm and the minimum and maximum selections is the introduction of recursion. With both randomization and recursion, it is possible to see how well each language performs with these two techniques (Cormen et al., 2001).

<pre> RandomPartition (A, p, r) { x = Random(p, r) exchange(A[r], A[x]) return Partition(A, p, r) } </pre>	<pre> RandomSelect (A, p, r, i) { if (p == r) return A[p] q = RandomPartition(A, p, r) k = q - p + 1 if (i == k) return A[q] elseif (i < k) return RandomSelect(A, p, q - 1, i) else return RandomSelect (A, q + 1, r, i - k) } </pre>
--	---

Figure 5.9 Random Selection Pseudocode.

Data for this algorithm were generated randomly and integers were placed into an array. For research purposes, the smallest number in the array was selected, however any i^{th} smallest number can be selected, i.e. 2nd smallest or 3rd smallest (Cormen et al., 2001). Random Selection uses the Randomized Partition procedure to return a pivot where all of the elements are less than or equal to the pivot on one side of the array. Any element is as equally likely to be returned (Cormen et al., 2001). As a result of the partitioning, it is impossible to determine if the i^{th} smallest number in the data set is above or below the

pivot before hand and so this must be calculated. This condition is the determining factor on which side of the data set is to be compared (Cormen et al., 2001). The algorithm will continue to recursively decompose the data set using further partitions until the i^{th} smallest number in the set has been found (Cormen et al., 2001).

CHAPTER VI

METRICS AND THEIR DEFINITIONS

6.1 Definition of Selection Criteria

In order to gain a full understanding on how each language will perform with respect to each algorithm, measurements must be taken on each implementation. For this research, a comprehensive metrics suite has been defined from static, dynamic, and .NET metadata measurements. All of the metrics used for this research come from the work of experts in the field of software development and have been widely applied. Each metric chosen clearly defines what is being measured, is easy to reproduce, and represents important and valid attributes of each program (Munson, 2003; Wohlin, 1996).

These attributes fall into two categories: quantitative and qualitative. Quantitative metrics measure attributes related to the size of the program. Qualitative metrics measure attributes related to program complexity, writing difficulty, and readability (Munson, 2003). For example, a high Lines of Code metric might indicate that the program is large. This is a quantitative measurement. On the other hand, if the program's Cyclomatic Complexity is high, this indicates complex code. This is qualitative metric (Munson, 2003; Wohlin, 1996). These metrics are defined in detailed in later sections. All metrics, regardless of what they measure, can be classified as quantitative or qualitative (Wohlin, 1996). Both of these categories together can give the programmer a

wide range of knowledge on how fault-prone the program may be (Munson, 2003).

Programs that tend to be more fault-prone usually have more bugs, design flaws, and other problems associated with complex software (Munson, 2003).

Each program is measured both statically and dynamically, with qualitative and quantitative metrics provided as part of each. Static measurements are taken on the source code itself, while dynamic measurements are taken on the program at run-time (Munson, 2003). Each of these has a very important purpose. Static measurements give indications about how large the program is, how difficult it was to write, and how long it may have taken to finish. Dynamic metrics give insight into how well programs perform, how much memory is used during run-time, and how many routines are called (Wohlin, 1996). Quantity and quality can both be measured statically and dynamically, producing a full data set for developers to analyze for decision-making purposes (Wohlin, 1996). All of these factors must be considered in order for a set of metrics to have any meaning (Munson, 2003). While programs implementing the same algorithm might have similar values, the slight differences show variations in how each language performs with respect to a given algorithm. This can be seen through the process of PCA, described in detail in Chapter 7.

In addition to measuring each program both statically and dynamically, a look at the .NET assembly for each program proves useful as well. .NET assemblies use the Common Language Runtime library (CLR), which produces metadata that can be imported into other programs, even if those programs are created in different languages (Petzold, 2001). While both the metadata and static categories might seem similar in that the program is not needed to run in order to obtain measurement, they are different with

respect to what is actually measured. Static measurements are taken directly on the source code itself, rather than the executable program that it creates. The metadata looks at how the .NET environment puts together an executable program from the inside, showing the viewer the many attributes available with the proper data reader. Since the .NET tool is used to provide a common development tool, the metadata created for each program can be measured providing another tool in the understanding of how well each algorithm performs from language to language. Several metrics on this metadata are defined in the following sections of this chapter.

Measurements were taken on each implementation in the same way and under the same conditions to produce comparable results. Since any computer system can introduce useless noise into a measurement, each metric was taken several times to ensure that an accurate result was produced (Munson, 2003). All static measurements were taken using Resource Standard Metrics (RSM), a tool created by M Squared Technologies (M Squared Technologies, 2005). The tool enables the use of several static metrics that are described in detail in the next few sections. Dynamic measurements were taken using the tool “AQTime 4”, by AutomatedQA Corp (1996), a tool that measures .NET run-time attributes. This tool profiles .NET programs in many ways and without regard to programming language. A memory profile and an execution profile were used to produce the measurements.

The last portion of the measurement process is on the .NET metadata from each program’s assembly code. These measurements were taken using the “A .NET Assembly Viewer” (The Code Project, 2002), a tool used to break down and categorize the information contained within the .NET metadata. This format of the metadata will not

change from system to system and therefore any tool that can view this information can be used on any computer carrying the .NET Framework, making these measurements easier to reproduce. The data that is of interest comes from the tables contained within the metadata. An important fact to note is that the metadata will contain information outside the scope of the source file, meaning that all of the libraries imported into the assembly will be measured.

6.2 Factors Present in the Measurement Environment

First and foremost, the data in this study was collected using the .NET set of compilers. Each compiler has its own optimization methods and the .NET compiler suite is no different in this regard (Petzold, 2001: Pratt & Zelkowitz, 2001: Sebesta, 1999). The use of the .NET compiler suite ensures that each program is compiled to the same format (Petzold, 2001). When looking at the metadata created by .NET compilers, it is possible to see exactly how the .NET virtual machine creates the objects present in each program, showing the differences between implementations from an internal point of view. It is possible that conclusions drawn from the data collected here might not be valid for other compiler suites, such as the GNU suite under Unix based environments.

The dynamic measurements are also bound by the optimizations created by the .NET Framework as well as the specifics of the machine in which the runs are taken. It is important to run these programs for dynamic measurements under as close to the same conditions as possible. Several runs were done on each program and the data collected are averages of all of these runs, ensuring that if anything happens in the background of the operating system, it does not introduce unnecessary noise into the measurements

(Munson, 2003). Because of this unforeseen operating system noise (new processes, memory management, etc.) the dynamic data, while in theory useful on any system, is only tested and currently valid using Windows XP Professional, SP2. In addition to operating system noise, the machine itself can also cause measurement noise causing dynamic measurement variability. The machine noise can come in the form of processor speed, memory availability, etc., and thus in order for the data to be preserved, the measurements should be reproduced on the same type of machine as well.

Lastly, the static measurements are dependent on the particular implementation. It is possible, as was mentioned earlier, that there is any number of implementations that will satisfy an algorithm's requirements. Because of the Sedgewick (1983) guidelines used in writing the code for these programs, the static measurements can only be reproduced with validity using his approach. Each algorithm was written using this strict coding style, ensuring that only core language constructs were used (Sedgewick, 1983). Very little .NET specific code was used to ensure that the base language was all that was measured. It is of course possible to write certain lines in a combined fashion, but all operations were broken up into individual and easily readable pieces. A look at the code in Appendix A will give further insight into exactly how the coding was done for each program written for the use of this project. It is with this code that the static measurements have valid results as Munson suggests is necessary to conduct science (Munson, 2003). Again, as described in Chapter 3, in order to truly gain a full understanding of how the code may be influenced by other developers, many programs need to be written and many measurements need to be taken and averaged before statistical analysis may be done.

6.3 Static Metric Definitions

6.3.1 Physical Lines of Code

The first metric of importance is that of the infamous Lines of Code (LOC). Being one of the oldest metrics, it is commonly used to determine program size, effort in units of time, and other development related data (Fenton & Neil, 1999). This is a crude metric, often used to determine more than intended (Munson, 2003), and is often misused (Fenton & Neil, 1999). Physical Lines of Code, or pLOC, is the first of several LOC metrics that will be included in this study. The “Resource Standard Metrics” (RSM) tool defines this metric as the total number of lines contained within the file without regard to blank spaces, comments, and the like. The pLOC metric will be used as a representation of file size, a quantitative attribute. It is important to note that the pLOC metric does not take into account executable statements versus non-executable statements. After pLOC is taken, additional metrics, such as those described below, are needed to gain a fuller understanding of the meaning behind the pLOC value.

6.3.2 Effective Lines of Code

The second LOC metric will be the Effective Lines of Code, or eLOC. This metric determines the number of lines of code in which work is performed, including the executable statements, and decision-making Boolean checks. The RSM tool takes this metric by excluding comments, blank lines, lines with only braces used as scope delimiters, and the like. The eLOC value will offer a better understanding of the total lines of code contained within a file, defining which lines will actually perform useful work. Again, this is a measure of *functional* size, rather than *actual* size (Fenton & Neil,

1999), making this another quantitative attribute. This measurement is important in determining other metrics, such as McCabe's Cyclomatic Complexity (McCabe, 1976), which is discussed in section 6.3.4.

6.3.3 Code Statements

This is the last of the LOC metrics. The RSM tool defines code Statements, or ILOC as those lines that contain a statement separator. It is important to determine what a statement actually is in order to take this metric, meaning that a clear definition is necessary so that this value remains unambiguous (Fenton, 1994). For the C-style syntax, the semi-colon is used, and for VB, it is the carriage return, since there is no need for a semi-colon in the latter (Petzold, 2001). All of the LOC metrics used in this study are highly correlated (Weyuker, 1988). This means that with a high pLOC count, a high eLOC and ILOC count will most likely be the result. As a result of the high correlation between the LOC metrics, ILOC is classified as a quantitative size metric.

6.3.4 McCabe's Cyclomatic Complexity

McCabe created one of the most commonly used qualitative complexity metrics used in software engineering, the Cyclomatic Complexity Metric, commonly noted as $V(g)$ (McCabe, 1976). It is a measure of both size and program complexity, and generally is a good indication of program fault content. A higher $V(g)$ value results in a more complex code module (McCabe, 1976). The RSM tool uses the classical definition (there have been many enhancements to this metric (Zhao, Wohlin, Ohlsson & Xie, 1998), the total number of edges and nodes in a given program or program module

(McCabe, 1976; Munson, 2003). Edges are the conditional paths and nodes are the processing that takes place at the end of those paths (McCabe, 1976). In an “if” block, for example, a new path is created. If the block’s conditional holds true, then the code takes a new path, otherwise, the code continues as normal without performing the operations within the block. A conditional node is created determining the Boolean value of the “if” statement and a connection node is created joining the two possible paths together. Again, this can all be seen in a flow diagram much like the examples given by Munson (2003). It is these edges and nodes that define the $V(g)$ value.

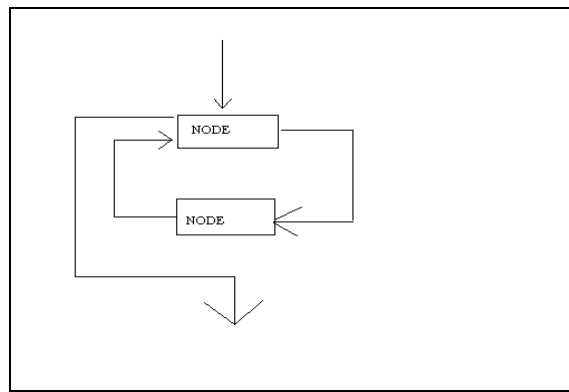


Figure 6.1 Control Flow Diagram of a For Loop.

This is much more of a qualitative metric than those that measure LOC. High LOC counts are not necessarily correlated to the number of edges and nodes contained within a program code module, however, in many cases high LOC counts indicate high $V(g)$ values. For example, a program can be written to perform one set of operations where there are no decision structures. The operations may take 1000 lines of code but the $V(g)$ value is small. On the other hand, many decision structures add to the LOC counts, giving both high LOC and $V(g)$. This metric can generally be computed even before coding begins if there is detailed design documentation. It is thus available to the

programmers at an early stage of development (Jung, Pikva & Kim, 2000; McCabe, 1976).

6.3.5 Compiler Directives

Each programming language has a standard set of libraries that must be included when compiling and running programs (Pratt & Zelkowitz, 2001; Sebesta, 1999). Since each language has different numbers of libraries that are needed in order for the program to run properly, counting the number of compiler directives and files included is a good measure of program complexity. With each header line, an additional library is loaded into the program by the compiler, producing larger executable files and additional chances for faults. Of course, these headers do not always have to be the standard ones included in the language. Programmers can create their own, which is why this metric is important for the discussion of how complex a program can become. Compiler Directives tends to be more of a qualitative metric, meaning that this measurement may not have a direct impact on program size but may impact its overall complexity. The RSM tool defines this metric as all of the *#include*, *import(s)* and *using* statements, used by C and C++, Java and Visual Basic, and C# respectively.

6.4 Dynamic Metric Definitions

6.4.1 Memory Usage

The amount of memory used by a program is one of the important factors in the program's performance (Ebert, 1995). Using the memory profiler of the AQTime tool, it is possible to take a snapshot of the memory used at its maximum point. The results are

often significantly different, showing that the libraries included by each language have complexities unseen by the programmer. This metric is both quantitative and qualitative. Large numbers in this measurement can lead to memory management issues, with memory leaks a possibility. One thing to note is that these programs are not interactive and memory size can change with different size inputs.

6.4.2 Total Objects Created

The total number of objects created can often describe how complex language libraries tend to be and how many are used (Ebert, 1995). Objects in this measurement refer to the number of items placed in memory. These do not have to be objects in the object oriented programming sense, but rather they can be things as simple as integers, strings, constants, etc. Larger numbers of objects created can show another factor of dynamic size and complexity. Memory usage is of course related to the numbers of objects created, but it does not accurately describe the size of the objects in memory. Memory usage values can be large for small numbers of objects, and it can be small for large numbers of objects. Knowing how many objects have been created gives insight into what memory usage actually means. The AQTime memory profiler can again take snapshots of the value of objects created on a given execution. This is more of a quantitative metric as it relates to program size.

6.4.3 Average Execution Time

Execution time is a direct measure of software performance (Munson, 2003). Execution time is affected by factors other than the complexity of the program.

Operations from within the operating system can stall the measured process for seconds at a time if another process takes priority. It is these extra factors that force the need for an average over several runs. The AQTime execution profiler times the program from start to finish. The programs written for this project are very simple and have no interaction with the user, so no user interferences are possible in slowing down the run. The average execution time is taken as the simple mean of ten consecutive timing measurements. This metric is qualitative in that faster software is usually desired. The programs were all run on a stand-alone machine and each run was performed under the same conditions. This variation leads to asking for more information, which is why the following two metrics were introduced.

6.4.4 Minimum Execution Time

Average run-time is not enough to describe exactly how long it takes each algorithm to run. Large outliers below the mean value can skew an average, making the average meaningless without additional information. Of the ten executions in which timing measurements were taken, the lowest of these ten is recorded for this metric. Again, since faster software is what is desired, this metric relates to software quality.

6.4.5 Maximum Execution Time

This metric is similar to the Minimum Execution Time. The difference here, however, is that among the ten values used in the average, the largest is recorded. Once again, quality is the attribute here since faster software usually is what's desired.

6.4.6 Total Routine Calls

Another measure of program complexity is the total number of routine calls made during execution. This can also be a factor in the execution time of the program (Munson, 2003). Using the AQTime execution profiler, it is possible to see how often routines are used to complete the programming objective. The profiler counts the number of function calls throughout each execution and records the value. Complexities unseen by the programmer can be brought to light as object definitions are often hidden.

6.4.7 Routines Executed

This complexity metric is a measure of all of the unique routine calls through a given run. Unlike the total number of routines, this metric only counts a given routine one time after it has been called and ignored if the routine is called subsequent times. Without this metric the total number of routine calls when running a program loses some of its meaning. Small numbers of unique routines can be called many times producing large values for Total Routine Calls. Also, large numbers of unique routines can be called only a few times, producing smaller numbers in Total Routine Calls. The AQTime execution profiler records this data at the end of each execution. This metric is qualitative.

6.4.8 Total Routines

This metric describes the total number of routines defined in the objects created. This does not necessarily mean that every routine defined will be used in the program's execution. The Total Routines metric can be another measure of the complexity that can

explain the variation of memory usage from program to program. The routines mentioned here are not only the ones defined in the source code, but those routines defined in header files and object files that are available to the programmer. The AQTime execution profiler records this data once a given run has completed.

6.5 Metadata Metric Definitions

6.5.1 Type References

This metric is used to describe the number of built-in data types referenced from within a given assembly. These include types such as int, char, float, double, and language specific types such as System or PrintStream (Petzold, 2001). This is useful in understanding the many different data types used to implement an algorithm. Within the metadata, there is a table known as TypeRef, which contains a listing of all of the types used within the assembly. Also included in this table is the information used to find where the data type is defined in the language libraries.

6.5.2 Type Definitions

This table lists all of the programmer defined data types. These include class definitions, *#define* references, arrays, and other objects not already available in the language libraries (Petzold, 2001). Languages suited for a given algorithm allow users to use built-in objects rather than defining their own. When programmers define their own objects, it becomes easier for faults to enter into code since the new types are not always fully tested.

6.5.3 Fields

This table contains a listing of all of the variables used within the assembly. These may be as simple as a single integer or as complex as an array of objects (Petzold, 2001). As more variables are declared, both the programmer and the operating system must undertake more memory management. If an assembly contains many fields, it is clear that the program is complex in terms of the number of objects that must be used in order to complete the algorithm.

6.5.4 Methods

This table contains the number of methods called from within the source files. This is different with respect to the number of methods called in the dynamic measurements section. This table only includes the methods called at the highest scope. This means that if a method calls another method, it is ignored, showing only the number of root calls (Petzold, 2001). This is important in understanding how many methods, both user defined and built-in, must be used in order to complete an algorithm. This is taken from the perspective of the programmer with regard to information hiding, commonly used in object oriented programming (Sebesta, 1999), not the system as a whole as is seen by the AQTime tool used to take dynamic measurements.

6.5.5 Member References

This table contains all of the members referenced in all of the classes and data structures used by a given assembly. These include items such as class member functions and variables at the scope of the entire assembly (Petzold, 2001). References from deep

within a language's structure may be found in this table. Also, user defined references may be found in this table, including the entry point called by the operating system. A large number of member references is a clear indication of memory management complexities both on the part of the programmer and the operating system.

6.5.6 Assembly References

Described within this table are the external references needed by a program to properly resolve all of the function calls. In some cases no assemblies are referenced, as the base definition of the programming language may be suitable enough on its own (Petzold, 2001). In other cases, however, pre-compiled objects must be imported into the assembly so that the program may find all of the objects needed to complete an operation. These operations include items such as I/O, arithmetic, and other operations that .NET programs may perform (Petzold, 2001). With many additional assembly references, a program's executable may become large and difficult to manage from the perspective of the operating system, causing slower performance and larger amounts of memory usage. This can be a major factor in the speed of .NET programs (Petzold, 2001).

6.5.7 Stand Alone Signatures

In .NET programs, data is described by its signature. Signatures are used to describe all of the references necessary for overloading data members externally by outside assemblies (Petzold, 2001). This means that a routine (i.e. function within a class) or operator (such as a '+' or '-') used in a C# program might be overloaded by a Visual BASIC program. The assembly that wishes to do the overloading must first read

the signature of the routine or operator to gain the location information of the given routine or operator, at which time it may gain access to the fields containing the data that is to be overloaded. The signature can be seen as a gateway to the fields section of the metadata (Petzold, 2001). The number of signatures is directly correlated with both the number of fields and the number of methods contained in other data, as these are the objects that may be overloaded (Petzold, 2001). Obviously, with more data that must be signed, there will be more items that must be managed in memory and more objects that must be created by both the operating system and by the programmer.

CHAPTER VII

PRINCIPAL COMPONENTS ANALYSIS

7.1 Understanding Metric Data

Defining primitive metrics and taking measurements is only the first part in understanding the size and complexity of each program written as part of this research study. The metrics defined in Chapter 6 are only the primitive data elements that describe certain aspects of each program but give little meaning without further analysis (Munson, 2003). This analysis can be found in the form of derived metrics; linear and non-linear composites of the primitive data sets created from taking measurements (Munson, 2003). Maurice Halstead was one of the original derived metric pioneers using what he called the Software Science Metrics to obtain additional information from his primitives (Halstead, 1977).

The Software Science Metrics used addition, multiplication, and logarithmic functions on the primitives to create new derived values without regard for measurement unit. What Halstead failed to understand was that simple mathematical computations on values do not give additional information from the primitive data, but in fact a loss of information is possible instead (Halstead, 1977; Munson, 2003). Munson suggests an example. If 5000 undergraduate students attend a university, and 1000 graduates attend the same university, then it is simple to assert that there are 6000 students at the university. The addition of these two groups does not lead to new information about the

relationships between graduates and undergraduates (Munson, 2003). Adding numbers together gives no new information and does not explain the variation between undergraduate students and graduate students. What is needed instead is a derived metric that will combine the primitives in such a way that sources of variation can be accounted for and understood. Software metrics tend to be highly correlated and so reasons for the differences in measurements must be made visible (Munson, 2003). This is the purpose of Principal Components Analysis.

7.2 Understanding Sources of Variation

Finding mean, median, and mode can certainly describe some of the central tendencies found in primitive metric data (Halstead, 1977; Jackson, 1991). While these statistics may be useful, it is important, however, to understand the relationships of what is being measured. Information on the sources of variation among different measurement values can be both intrinsic and systemic. Intrinsic variation can be as simple as saying that some programs have more lines of code than others. Systemic variation, on the other hand, is related to errors in measurement and can introduce noise into the analysis (Munson, 2003). It is hoped that the systemic variation can be eliminated from the measurement process.

If two related metrics share a common element of variation, they are said to have covariance. If the two metrics vary about the mean in much the same way, the shared variance is considered to be large. If, conversely, the two metrics do not vary about the mean in a similar way, the covariance is considered to be small (Jackson, 1991). Since

most of the metrics used in this study will tend to be highly correlated, it is important to understand that some may share common factors in variation (Munson, 2003).

Each metric for a given algorithm will have its own mean and variance and with this in mind it can become difficult to impossible to learn anything about a program (Munson, 2003). By adjusting each metric for the effect of its own mean and standard deviation then it is possible to look at the adjusted values and understand something more about the metric in question (Jackson, 1991). To do this, a value known as a z-score must be calculated. The z-scores will now have a mean of 0 and a standard deviation of 1. Positive z-scores indicate that the measurement was greater than the mean, while negative z-scores indicate that a measurement was less than the mean (Munson, 2003). By using z-scores, it is possible to determine which metrics share variation. Two metrics with similar z-score values will be covariate, and metrics with dissimilar z-score values will be non-covariate. Understanding covariance can lead to additional and useful information that raw metric data cannot give on its own (Jackson, 1991: Munson, 2003). Using the z-scores, it is now possible to begin the process of Principal Components Analysis.

A z-score is calculated simply by the following formula:

$$Z_i = (x_i - x'_i) / \delta$$

where subscript i represents the current measurement, x is the measurement value, x' is the mean of the measurement values, and the δ is the Standard Deviation (Jackson, 1991). Now, as was mentioned, the z-score values will have a mean of 0 and a Standard Deviation of 1 (Jackson, 1991: Munson, 2003). If z-score values are greater than 1.0, this

means that the measurement is larger by at least one Standard Deviation from the mean. If they are less than -1.0 , this means that the measurement is smaller by at least one Standard Deviation from the mean (Jackson, 1991).

The next step is to find the relationship coefficient, which will be based on the z-score values above (Jackson, 1991). Using the z-scores, it is possible to calculate the Pearson product moment correlation statistic (Munson, 2003). The formula is as follows:

$$r_{xy} = (1/n-1) * SUM[(from i = 1 to n) Z_x Z_y]$$

This yields a diagonal matrix of correlation coefficients showing how each value is related to every other value in the data set for a given algorithm (Jackson, 1991; Munson, 2003). With both the z-scores and the Pearson technique, it is possible to see how each variable (in this case measurements taken on a particular algorithm in a particular language) is related to every other variable and how they share variance.

7.3 Metric Domains

Metric data, as was already stated, are simply data and nothing more. It is very difficult to draw any useful conclusions from simply reading raw metric values (Munson, 2003). In Principal Components Analysis, it is necessary to transform the highly correlated raw values into a set of unrelated domains; metrics on specific attributes of the object measured, in this case, software (Munson, 2003). The main problem here is to determine exactly how many usable sources of variation can be identified in the original

metric values. The domain metrics are seen as principal components, each illustrating an underlying common attribute from among the raw values (Munson, 2003).

Principal Components Analysis is a straightforward process. We wish to transform our set of correlated values into a set of non-correlated values. Given a set of n metrics M indexed from 1 to n such that $M = (m_1, \dots, m_n)$ we wish to transform them into a set n domain metrics D such that $D = (d_1, \dots, d_n)$. Each measurement value will be mapped to the domain in which it is most correlated (Jackson, 1991: Munson, 2003). This is done by extracting the eigenvalues and the corresponding eigenvectors from the elements in the matrix created in the previous section. The complete mathematical basis for extracting the eigenvalues and eigenvectors can be found in Appendix 1 of Munson's textbook (Munson, 2003). Once the eigenvalues and eigenvectors have been found, calculating the product moment will yield all of the principal components. The principal components represent the orthogonal measurements in which there is no correlation to any other value. There is a point of diminishing returns, however, as the principal components will not yield any information about variation if the eigenvalues are too small. For this reason, the stopping point for eigenvalue extraction is an eigenvalue minimum of 1.0. In this way, only the orthogonal domains that are most correlated with the metric values will be visible (Jackson, 1991: Munson, 2003). A domain matrix results from the operations performed on the covariance matrix from the previous section.

Once the new orthogonal metric domains have been found and have been placed in their new matrix, new sources of variation will become apparent. This variation can be seen in the fact that often metrics will be highly correlated to one domain but not to another. Higher metric domain values indicate higher correlation to that domain and this

new source of variation is a direct artifact of the Principal Components Analysis process (Jackson, 1991). To clarify this new variation, a varimax rotation will be performed on the domain matrix (Jackson, 1991). The resulting rotated matrix will show factor patterns for the metric domains that have been extracted (Munson, 2003). Now raw metrics are shown to be highly related to certain domains without unclear sources of variation. All unseen possible noise has been removed from the data. All of the mathematical foundations of Principal Components Analysis can be found in Appendix 1 of Munson's textbook (2003).

7.4 The Relative Complexity Metric

Once Principal Components Analysis has been completed, there can still be an additional simplification of the metric data. Munson suggests that if it is possible to describe a program or program module in terms of a single complexity value, this value can be used in a linear function to describe how fault-prone a program or module might be (Munson, 2003; Munson & Khoshgoftaar, 1990). This value is known as a Relative Complexity Metric (RCM). This new metric is a weighted sum of a set of uncorrelated attribute domain metrics, the orthogonal domains found earlier (Munson, 2003). The sum is weighted against the eigenvalues that were also extracted earlier. In order to calculate the RCM, the following formula is used:

$$\text{RCM} = \text{SUM}(l_j d_{ji})$$

where l is the eigenvalue extracted from j th measurement and d is the ij th domain metric found in the last matrix created from the previous section after the varimax rotation is performed. From this RCM value, one measurement can describe a single program module.

Once the RCM has been found, it is much simpler to understand how complex a given program or module is. The programs being measured for this research can now be grouped and arranged by this single metric (Munson, 2003). The RCM provides a simple mechanism of aggregating the many similar complexity metrics into one single metric used to describe a set of programs (Munson, 2003). The RCM however is not a complete measure but rather a stand-in for aspects of software quality that are not measurable and it can be simply stated as a surrogate for software faults (Munson, 2003). The RCM value will be used in the following discussions on how each programming language performed with respect to a given algorithm. The higher the RCM value, the more complex the program and the more likely faults may be contained within the source code. An example of this entire process may be found in Munson's textbook, Chapter 6 (2003). Appendix C contains all of the process output, the domain metrics, and the RCM values for each algorithm if reference is needed. It must be made clear, however, that even though a program might have a higher RCM value, meaning that the program has a greater fault-prone nature, this does not mean that the program actually contains faults. It is a measure of how likely faults may appear when compared to other RCM values (Munson, 2003).

CHAPTER VIII

STATIC MEASUREMENT ANALYSIS

8.1 Introduction

Before looking at the overall results of the measurement process, the process must first be broken down into three parts: the static measurements, the dynamic measurements, and the metadata measurements. After the process of Principal Components Analysis, the static, dynamic, and metadata metrics will be broken into two domains representing the quantitative and qualitative software attributes. Understanding the three parts measured independently will allow for greater understanding of the entire process. In this chapter, the first part of the measurement process, the static measurements, will be analyzed and discussed in detail, revealing important trends that have surfaced while writing the programs. Appendix B contains all of the raw measurement values.

The static metrics used in this research are designed to show the difficulty of actually writing the code. When remembering the language descriptions from Chapter 4, some of the results may be surprising, while others may be what were expected. In either case, the measurements taken here will be a good indication of the overall difficulty of actually writing the programs. One thing that is important to remember when looking at these results is that coding style was maintained in all five languages whenever possible

(Sedgewick, 1983), and all of the implementations were structured using traditional approaches (Pratt & Zelkowitz, 2001: Sebesta, 1999).

8.2 Individual Algorithm Results

8.2.1 Linear Search

One of the simplest algorithms in this study is Linear Search. It produced different results from language to language, but the simplest language for this algorithm was Java. It produced the smallest LOC measurements and was small in the area of Cyclomatic Complexity. Next was C#, which is not surprising simply because the code syntax is similar. Third for this algorithm was Visual BASIC, which is a little unexpected given that Visual BASIC is designed for readability rather than with construct in mind. Even so, it still scored well (Pratt & Zelkowitz, 2001: Sebesta, 1999). The C implementation was fourth. It had a larger amount of LOC but scored well on $V(g)$. Finally, at the end of the list, was C++, which is expected, as it is a super set of C rather than its own language (Sebesta, 1999). It produced higher lines of code and higher $V(g)$. C++, with the way it structures class objects, adds what seems like a higher level of complexity.

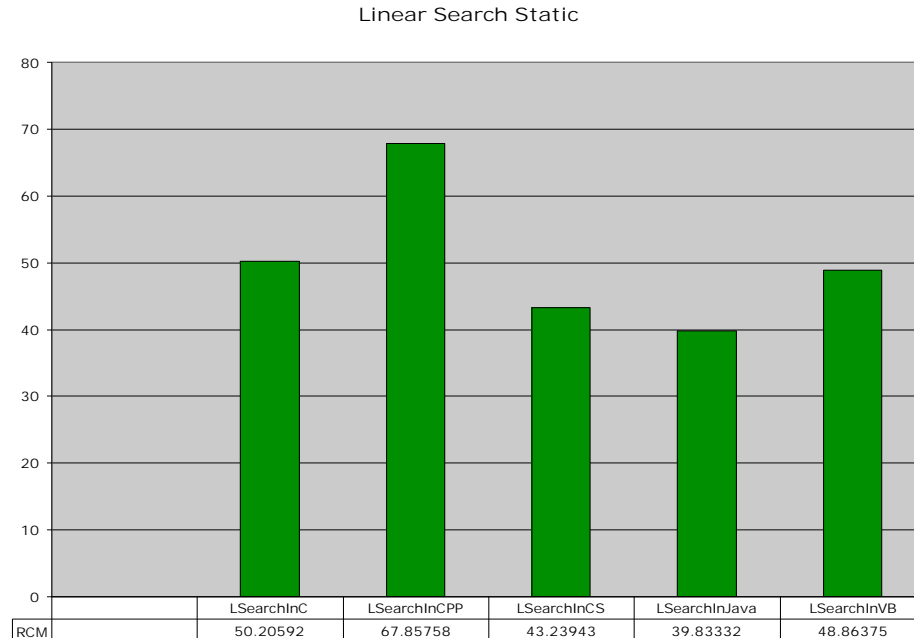


Figure 8.1 Linear Search Static Measurement RCM Results.

8.2.2 Bubblesort

For the second time, Java was the leader and proved to be the least complex solution. Again, its LOC measurements were the smallest. Second again was C#, but its values were still close to Java since its syntax structure is similar. Third this time was the C implementation; better than Visual BASIC, which is expected as Visual BASIC, with its English like structure tends to be more complex (Pratt & Zelkowitz, 2001). Visual BASIC was fourth, as it had higher LOC and V(g) measurements than all of the other implementations except for C++. C++, as before, proved the most complex as it is a super set of another language, rather than its own language. Again, the class implementation portion of the C++ language proved to be the main complexity factor, as it added counts to the LOC and V(g) values.

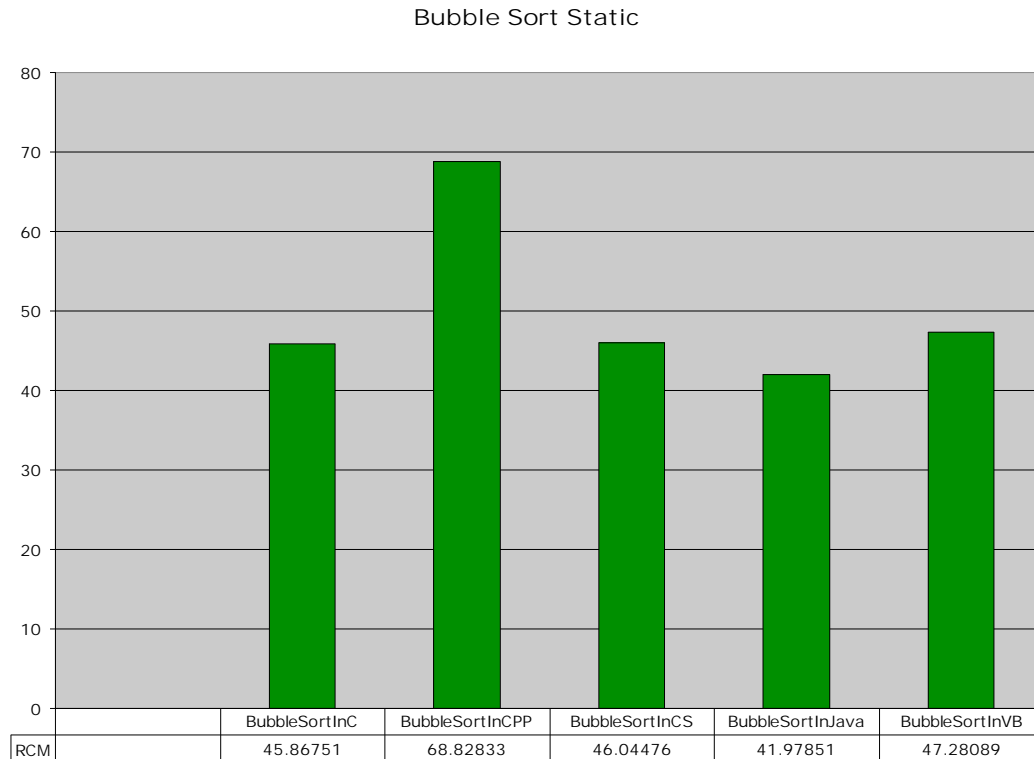


Figure 8.2 Bubblesort Static Measurement RCM Results.

8.2.3 Quicksort

In this set of implementations, things changed from the previous. The leader with the lowest RCM value this time was C#. It produced lower LOC and V(g) metrics than its counterparts. Java, while not the leader this time, was still close to the C# implementations with the same reasons as before, that its syntax and code structure are very much like Java. Java was developed first, and, as stated in Chapter 4, C# was developed to be based on Java and C++ together (Petzold, 2001: Pratt & Zelkowitz, 2001: Sebesta, 1999). Third again was the C program, which was a little higher for this algorithm than for the others when related to its competition. It had a value of 47.3, nearly six points higher than the leader. Visual BASIC was fourth once again as it produced a larger amount of LOC. C++ once again falls fifth, with its syntax again the

culprit. C++, as was stated in Chapter 4, was meant for object organization and readability, which is why these results are not surprising. With organizational features, additional lines of code are required (Pratt & Zelkowitz, 2001; Sebesta, 1999).

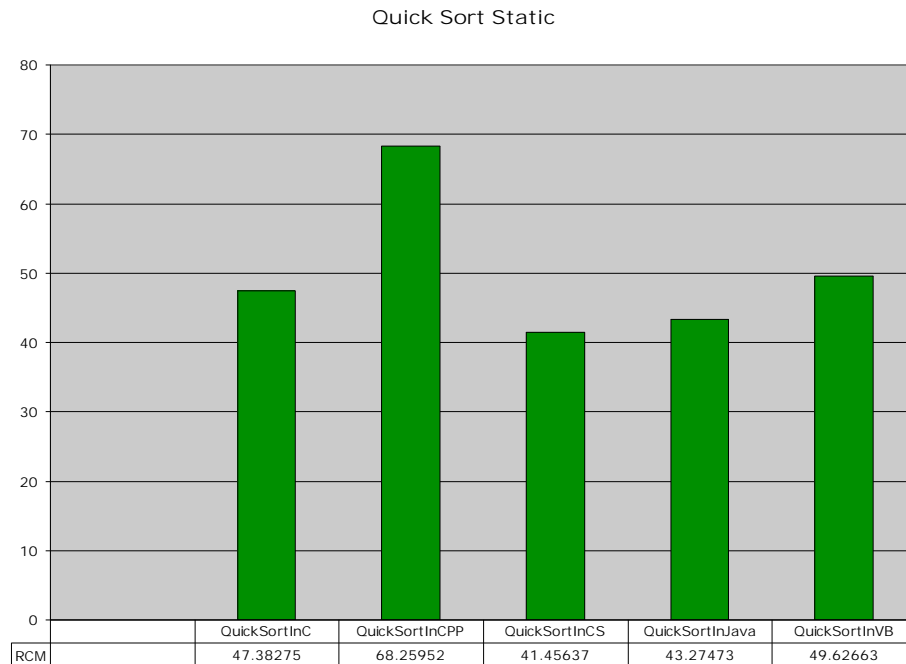


Figure 8.3 Quicksort Static Measurement RCM Results.

8.2.4 Naïve String Matching

In this algorithm, C was the clear leader with the lowest RCM value. As this is a string-matching algorithm, and since the processing of strings is necessary, C came out ahead with its use of arrays of characters to perform string operations. The other languages fell slightly behind due to the use of additional constructs and language features for the processing of strings. One surprising result is that Visual BASIC was second, even though the language is not designed for this purpose (Pratt & Zelkowitz, 2001). One thing that can be reasoned about Visual BASIC's results is that because Visual BASIC is designed to be simple, and since this is a simple algorithm by

comparison to KMP String Matching, it produced lower static measurement values (Cormen et al., 2001). Java and C# were third and fourth respectively, proving again that they are close in value because of their syntax structure. C++, while using the same convention as C for its string processing, still proved to be the most complex statically with its class object organization and implementation.

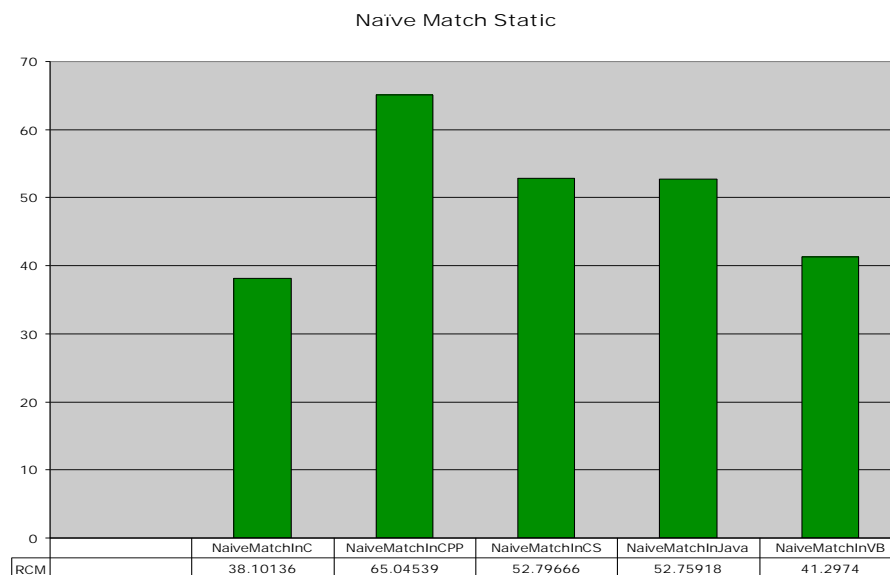


Figure 8.4 Naïve String Matching Static Measurement RCM Results.

8.2.5 KMP String Matching

For this algorithm, C was the leader producing the lowest RCM value. The C implementation had the fewest LOC and $V(g)$. C++, however, continues to be plagued by problems in the very same areas mentioned above. Again, the C++ implementation was the worst performer, posting an RCM value of nearly 68. Visual BASIC did not fare as well with this algorithm as it posted high measurement results, the highest it has done so far. This is a more complex algorithm and Visual BASIC was not designed for this kind of processing (Cormen et al., 2001; Sebesta, 1999). Java and C# had the same

problems with this algorithm that it did with Naïve String Matching. The addition and use of the String object proved difficult to parse and more lines of code were needed for this process. They were again close to each other for the same reasons as before, that their syntax is almost the same.

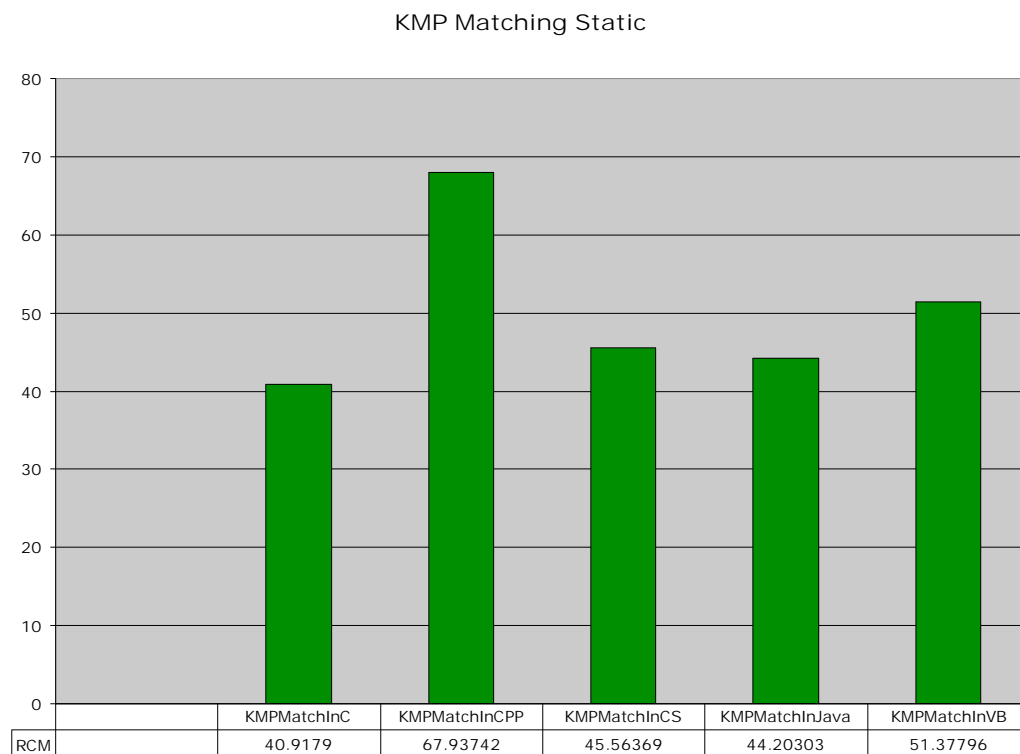


Figure 8.5 KMP String Matching Static Measurement RCM Results.

8.2.6 Polynomial Addition

The C programming language fared best with this algorithm posting the lowest RCM value. The factor that gave C the edge was the need for fewer LOC to write the program. C++ was once again the worst having the highest measurement data for all of the static metrics. The language features of C++, C#, and Java proved too complex for this simple algorithm (Cormen et al., 2001; Sebesta, 1999). These languages might have

been overkill for a program of this size and complexity. Java and C#, as all of the algorithms before, had similar values when compared to the others for their syntax is similar. C++, while closer to a competitor this time, Visual BASIC, still did not perform like the others, posting higher measurements, making this algorithm more complex. Again, a class implementation simply for this use of adding polynomials together, might have been more than was needed. It is important to illustrate, however, since C++ is so often used. Perhaps in a larger piece of software, a simple function for computing this process might have been a benefit, but for such a small scope, it was not.

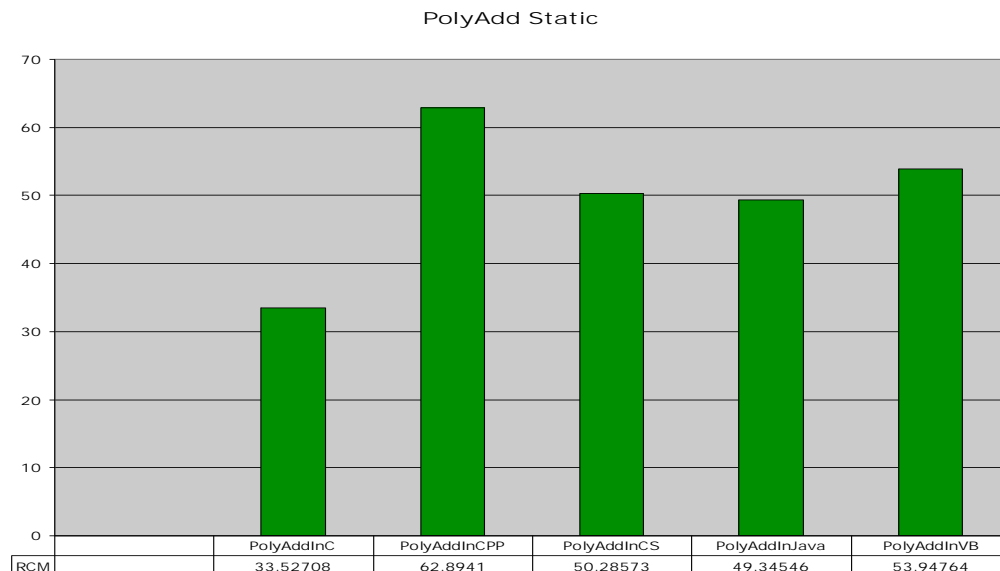


Figure 8.6 Polynomial Addition Static Measurement RCM Results.

8.2.7 Gaussian Elimination

The results for this algorithm were as expected. For the first time, C++ was not fifth in the list for an algorithm. Visual BASIC was the most complex with the highest RCM value. This makes sense, as Visual BASIC was not designed for this kind of processing. It had the highest LOC and V(g) measurements. The leader for this

algorithm was C with its simple structure, language constructs, and small values for the LOC measurements. This makes sense since C was designed for scientific programming, and while this is not scientific software, the calculations done in this algorithm might be compared with software that uses large amounts of mathematics. Close again were C# and Java, second and third respectively, and they were once again close for the reasons as stated above for all of the algorithms thus far. C++ was fourth, not last this time, but again, the way it handles its class of objects implementation proves to cause higher LOC and V(g) measurements.

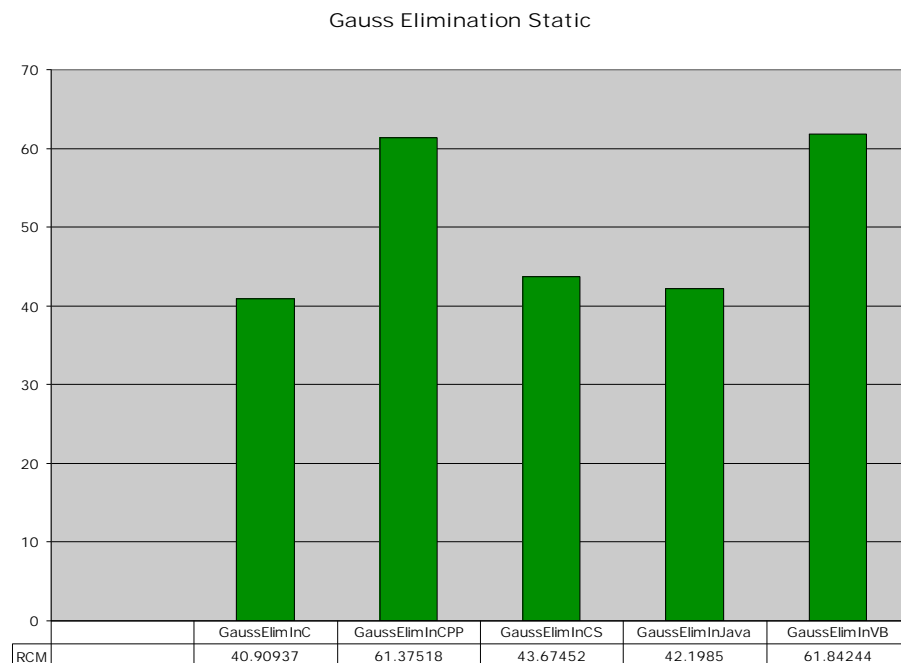


Figure 8.7 Gaussian Elimination Static Measurement RCM Results.

8.2.8 Minimum and Maximum

For this algorithm, the C implementation produced the least complex source code, followed very closely by C#. Since the algorithm used to find the minimum and maximum values in a given array was simple (Cormen et al., 2001), the features of

object-oriented programming may not have been necessary (Cormen et al., 2001: Sebesta, 1999). C++ was the most complex and its biggest problem was in high measurement values. For the first time, Visual BASIC has fallen further behind as it had high LOC counts. The Visual BASIC implementation required higher LOC than all of the other implementations (except C++). An odd result in this algorithm is that Java and C# are not as close as in the algorithms prior. There is a difference of over two RCM points, as C# was less complex than Java. This algorithm, while not complex and with simple array processing, produced different results as compared with the other algorithms.

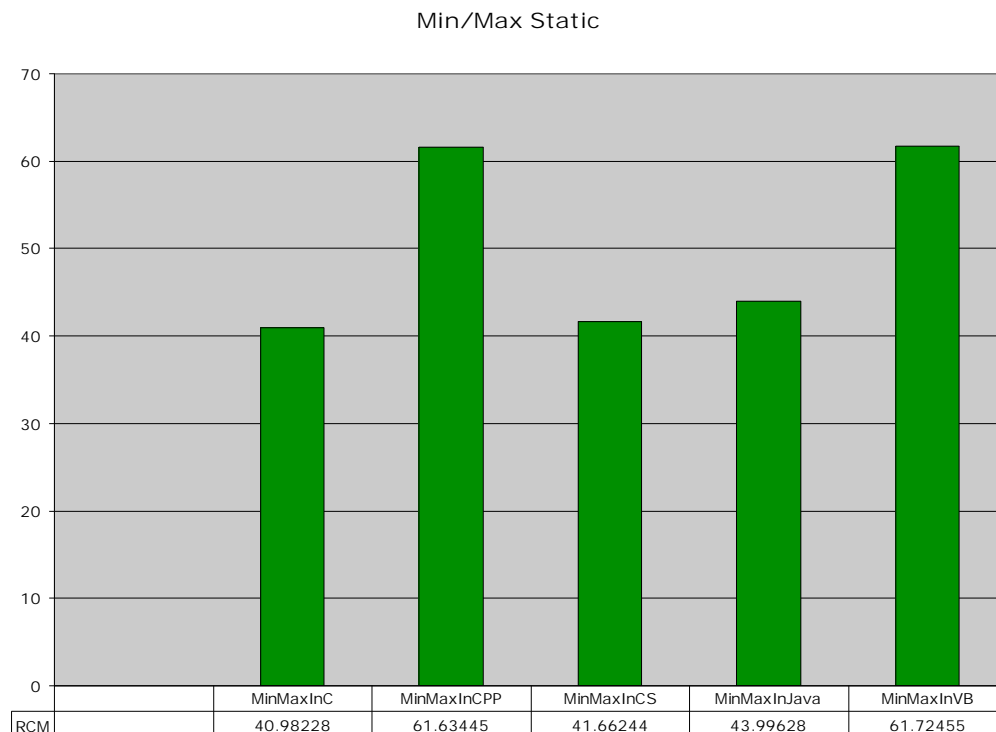


Figure 8.8 Minimum and Maximum Static Measurement RCM Results.

8.2.9 Random Selection

In this algorithm, Java was the language that produced the least complex solution. The highest RCM value was posted by C++ with its object implementation structure

again the culprit for its high complexity. Second was C#, again with a value close to Java for their syntax styles are similar. C was third, with its low LOC and V(g) counts when compared to Visual BASIC and C++. Visual BASIC was fourth with an RCM value of about 55. Visual BASIC had the expected results, as the use of recursion tends to be more complex. Rather than an explicit loop, this technique provides an implicit loop that can be hard to parse in a language with the design structure as Visual BASIC (Cormen et al., 2001: Pratt & Zelkowitz, 2001: Sebesta, 1999).

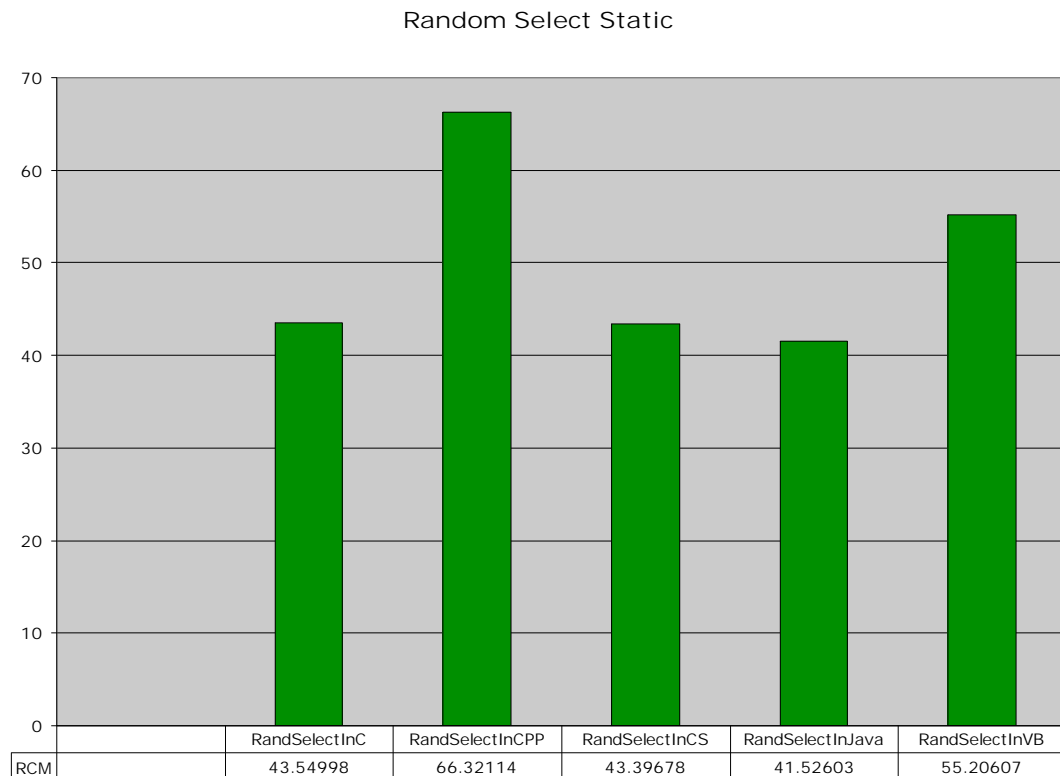


Figure 8.9 Random Selection Static Measurement RCM Results.

8.3 Evaluation of Results

Since the static measurements defined for this study are designed to measure the complexity on the source code itself for each implementation, it is important to discuss a

few trends. Since C is not an object-oriented language, some of the complexities of class definitions are removed, and as a result the C programs require fewer LOC to complete an implementation of each of these algorithms. Visual BASIC, as has been discussed, was designed to be a simple language and for the most part, it has achieved this goal as often enough its $V(g)$ was small and so were its LOC measurements (Sebesta, 1999).

One thing to note, however, is that Visual BASIC's results were a bit of a surprise. For some of the more complex algorithms, it did well as far as not being complex, going against what is expected based on its design (Pratt & Zelkowitz, 2001; Sebesta, 1999).

C++ was the worst performing language for each algorithm. In every case the C++ programs required more LOC and in most cases had a higher $V(g)$. C++ class definitions are typically written into a header file and these classes can be reused as libraries in the future. The implementations of these class definitions are usually found in corresponding source files. In class definitions, additional lines of code are needed as the function prototypes are declared in the header, and then written again in the implementation file. Source code then typically looks like the following in the implementation file:

```
ReturnType ClassName :: MethodName () { ... }
```

This line is repeated in the header file where it is declared. C++ is also different in that class definitions are done externally from the main running program. C# and Java main entry points are always as part of class definitions, so scope operators and additional declarations are not always necessary (Pratt & Zelkowitz, 2001; Sebesta, 1999). Also,

with this basic feature of both Java and C#, fewer lines of code are required to declare and implement a class and its members.

One interesting fact about the measurements is that the C and C++ Compiler Directives metric was variable, while the other languages each had a standard value. C and C++ require additional headers since the libraries that are coupled with the language are separated into logical parts. For example, there are libraries that define input and output, higher-level mathematical functions, and many others. For all of the other languages, one or two imported libraries gave to the compiler everything it needed. As will be seen in Chapter 9, where dynamic measurement results are discussed, this will have interesting effects on memory management.

Another interesting trend that can be observed is that both the C# and Java RCM values were often very close to one another. As has been stated by Microsoft, the design of C# was intended to match the syntax of Java with the power of C++ (Petzold, 2001). As a result, the programs look almost identical and tend to use the same control constructs and data structures. In almost all cases, the Java and C# measurements vary from each other only slightly which is the direct reason that the RCM values for each are, in most cases, very close.

8.4 Conclusions

Static metrics give developers a more functional understanding of how difficult programs are to implement. What is more important, however, is what can be learned from the results (Munson, 2003). Some languages will be more complex than others, forcing developers to make educated decisions about the tools that will be used in a

software life cycle. From the static measurement results and the trends presented here, developers will be able to gain better understanding of programming language semantics that can be applied to projects in the future. All static measurement results and analysis documents may be found in the appendix.

CHAPTER IX

DYNAMIC MEASUREMENT ANALYSIS

9.1 Introduction

The second part of the measurement analysis is the study of the dynamic metrics that have been obtained on all of the programs written for this study. As has been discussed, the dynamic measurements refer to the actual performance of the programs rather than the complexity of the source code (Munson, 2003). Here the speed, efficiency, and memory management of each language can be seen through the measurements of each algorithm program. With this information, developers will be able to best understand how programs will behave under specific language environments. The two principal components singled out are the qualitative and quantitative variations.

After Principal Components Analysis was performed on each algorithm's measurements, the results that were found tended to be consistent with the language descriptions in Chapter 4. In most cases, each language performed as expected with the exception of Visual BASIC, which had the most variable measurement data. This affected the outcome of the Principal Components Analysis process to some degree as it introduced some new sources of variation. Understanding this source of variation will be the most important factor in making sense of the raw data. All raw measurement results can be found in Appendix B.

9.2 Individual Algorithm Results

9.2.1 Linear Search

The C# implementation of Linear Search was the best performer over all. It was strong in the areas of declared routines, routine calls, and routines executed. Also, the run-times were better here than with the other programs. The C program had the second highest RCM value. Its strengths lied in memory size, routines executed, and total objects created. Visual BASIC was next and had some interesting results. While performing better in some areas than the other languages, its memory size became a weakness since this measurement value was the second highest. Also, execution times for Visual BASIC were among the highest. C++ was fourth and had some weak areas. The C++ implementation produced high measurements in the areas of total routines defined, routines executed, and total routine calls. Finally, Java was the worst performer producing the highest RCM value. The weakness in Java was found in its large size in memory, its slow execution times, and its large number of objects created. Each of these measurements was highest in the Java implementation.

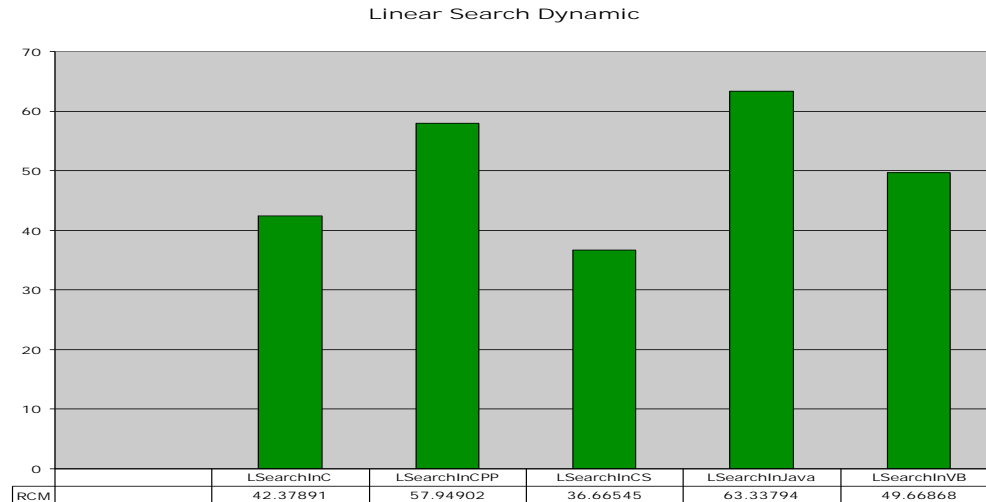


Figure 9.1 Linear Search Dynamic Measurement RCM Results.

9.2.2 Bubblesort

For the second time the C# implementation was the best performer, posting an RCM value of under 40. The implementation's strongest areas were found in objects created, executions times, and the metrics concerning routines. It was weak, however, in memory size. The second best performer was the C implementation with strengths in memory size, objects created, and execution times. C# and C were separated only by a point in their RCM values. Third in this algorithm was C++, which had strengths in objects created, execution times, and routines executed when compared to the other languages. It was weak, however, in memory size. Java was next although it was weak in many areas. The memory size was large, it had slow execution times, and the total routine calls were the highest. Visual BASIC was this time the worst performer with large size in memory, executions times and total routine calls. The main problem area for Visual BASIC was its total objects created measurement which was significantly higher when compared to the other implementations. The results for Visual BASIC make sense

since the algorithm was not designed for programs with the amount of operations that Bubblesort has (Cormen et al., 2001: Sebesta, 1999).

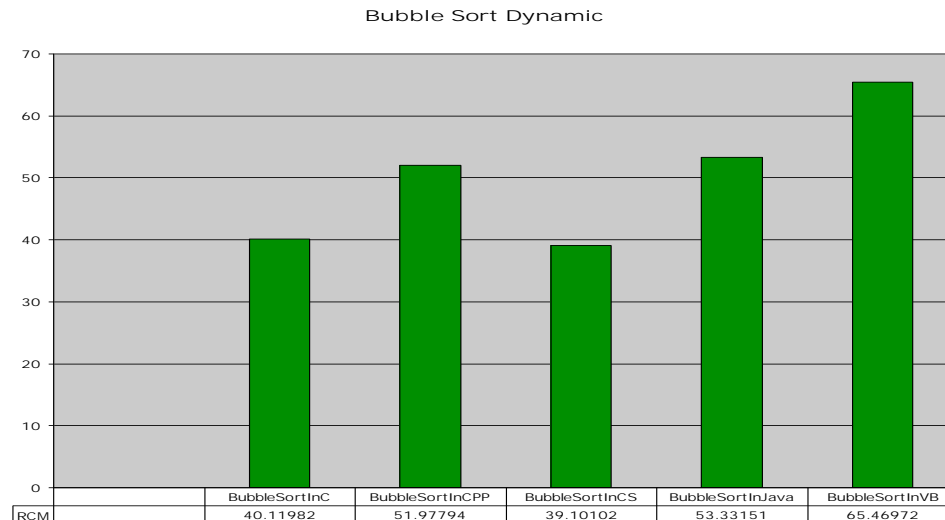


Figure 9.2 Bubblesort Dynamic Measurement RCM Results.

9.2.3 Quicksort

C was, for the first time, the best performing algorithm. The C implementation produced the smallest measurement values for routines executed, objects created, and was strong in execution times. The introduction of recursion may have been the reason since this application is often used in systems programming (Cormen et al., 2001: Pratt & Zelkowitz, 2001: Sebesta, 1999). C# was a close second producing the best execution time values and a small measurement for the number of routines executed. Also, the total number of objects created was among the smallest. C++ was third again, producing small values for the measurements of execution time and objects created. C++ was weak, however, in the memory size metric. Visual BASIC was fourth with weaknesses in memory size, objects created, and total routine calls. Quicksort is a complex algorithm

and Visual BASIC may not have been well suited for this implementation (Cormen et al., 2001; Sebesta, 1999). Java was again last, posting an RCM of over 60. The main weakness for Java once again was in its memory size, where it was the highest. Also, the executions times once again hurt the Java performance. An interesting result is that the Quicksort routine offered in the environment and used in each language did not change the results.

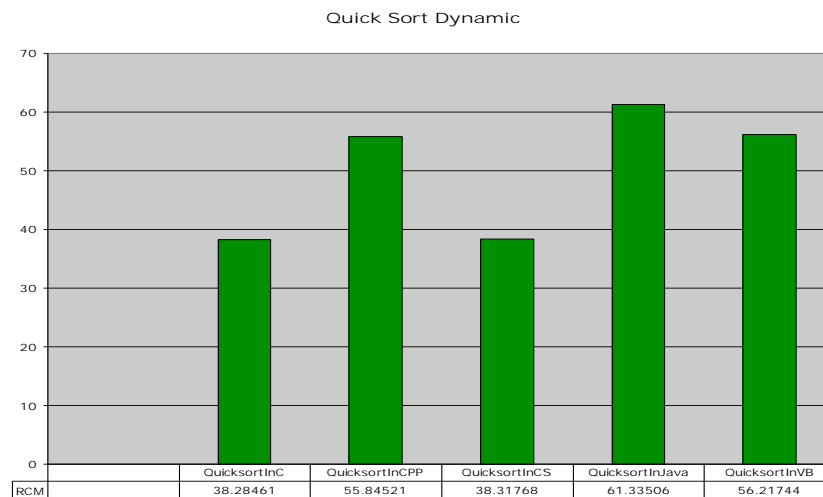


Figure 9.3 Quicksort Dynamic Measurement RCM Results.

9.2.4 Naïve String Matching

For Naïve String Matching, C was the best performer. The C implementation had the strongest values in memory size, objects created, and routines executed. The C# implementation was second from C with less than one point difference in the RCM values. C# showed strength in execution times, routines executed, and total routines. It was weak, however, in the objects created measurement. C++ was third again with strong measurements for execution times and objects created. It had weakness, however, in the total number of routine calls. Java was fourth with a major weakness in its

memory size. Also, since Java needed an additional String object for the data processing, higher numbers were found in the total objects created measurement (Sebesta, 1999). Visual BASIC was again the worst performer. It was weak in memory size and was worst in execution time. The algorithm was run a second time in the worst-case (no pattern match) and the results did not change, an interesting fact to observe.

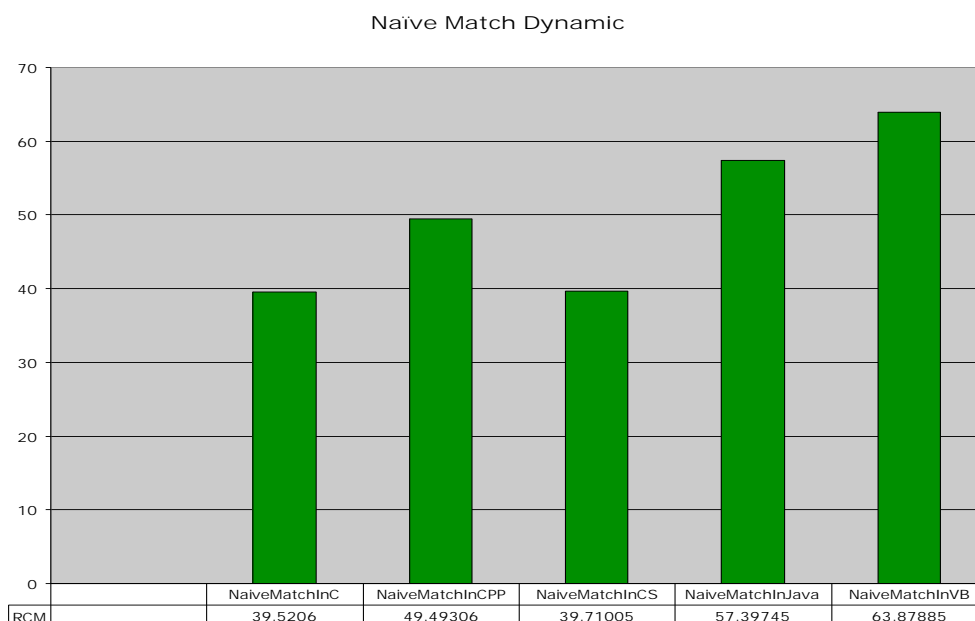


Figure 9.4 Naïve String Matching Dynamic Measurement RCM Results.

9.2.5 KMP String Matching

C# produced the best performing implementation for KMP String Matching. Its areas of strength were found in execution times, size in memory, and the measurements concerning the numbers of routines involved in the program. C was second with strengths in memory size and objects created. C was a little weaker for this algorithm with respect to execution times. This may have been caused by not having a specific object related to strings, since C uses arrays of characters that must be parsed (Pratt &

Zelkowitz, 2001; Sebesta, 1999). Visual BASIC was third this time, performing well in the areas of execution time and total routines. This was surprising since Visual BASIC was not designed for an algorithm with this much complexity (Cormen et al., 2001; Sebesta, 1999). C++ was fourth and tended to be weaker in memory size and execution times. Java was the worst performer producing high measurements in memory size, execution time, and total routine calls.

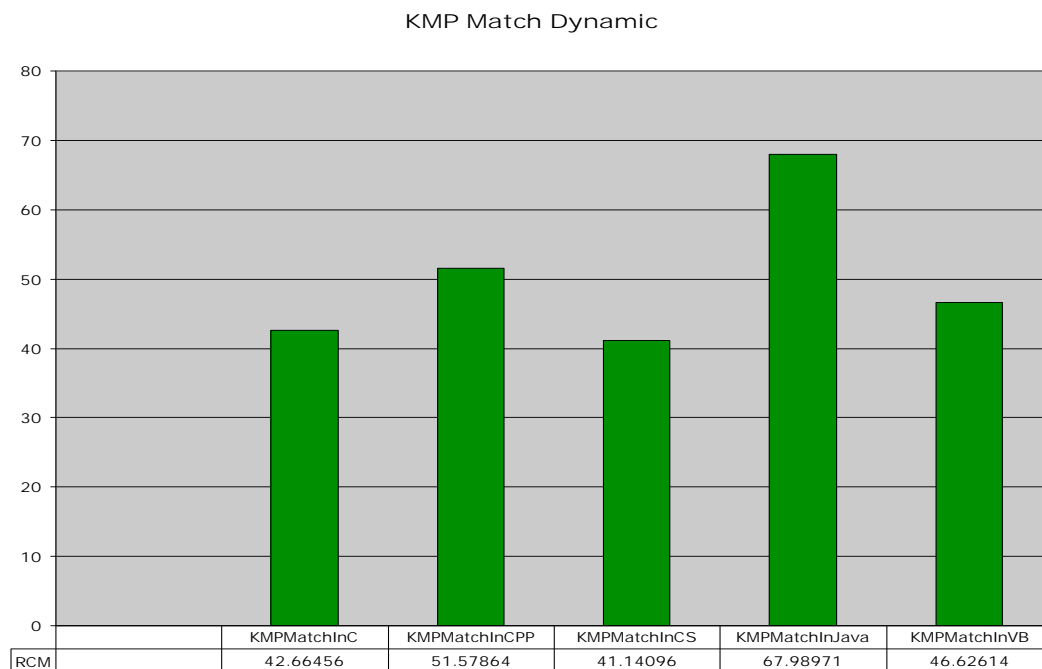


Figure 9.5 KMP String Matching Dynamic Measurement RCM Results.

9.2.6 Polynomial Addition

The C# implementation produced the only RCM value under 40. C# was once again strong in execution time, routines executed, and objects created. C was second with an RCM value only two points higher. C was strong in memory size, objects created, execution times, and routines executed. Visual BASIC was third with an RCM

value over 45. Although strong in execution time, Visual BASIC was weak in memory size and total routine calls. C++ was fourth with a clear weakness in the total routine calls, memory size, and in execution times. Since Polynomial Addition is a simpler mathematic algorithm, C++ may have been too complex (Cormen et al., 2001; Sebesta, 1999). Java was once again the worst performer and again the weakness lies in memory size, execution times, and the total number of routine calls.

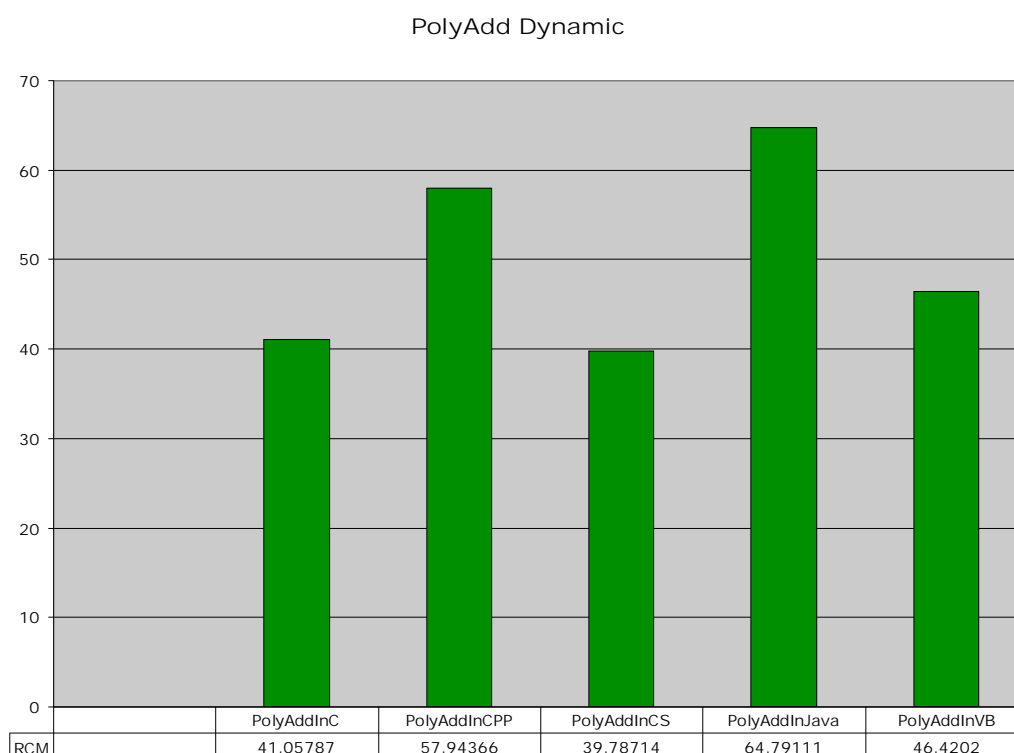


Figure 9.6 Polynomial Addition Dynamic Measurement RCM Results.

9.2.7 Gaussian Elimination

The results for this algorithm were surprising in that Gaussian Elimination is a much more complex algorithm than the others in this study (Cormen et al., 2001). C# proved the best performer with fast execution times and strong measurements in objects

created, total routines defined, and total routine calls. C was second, and this makes sense since C was developed to be a language for complex use (Pratt & Zelkowitz, 2001: Sebesta, 1999). C was strong in memory size, objects created, and execution times but was one of the worst in total routine calls. The most surprising result for this algorithm is that Visual BASIC was third. Visual BASIC was not designed for high-level operations such as this and yet had strong values for execution time, memory size, and routines executed (Pratt & Zelkowitz, 2001: Sebesta, 1999). C++ was fourth with respect to this algorithm. This implementation was strong in memory size but weak in most other areas. Java was once again the worst performer with extreme weakness in memory size and total routines called.

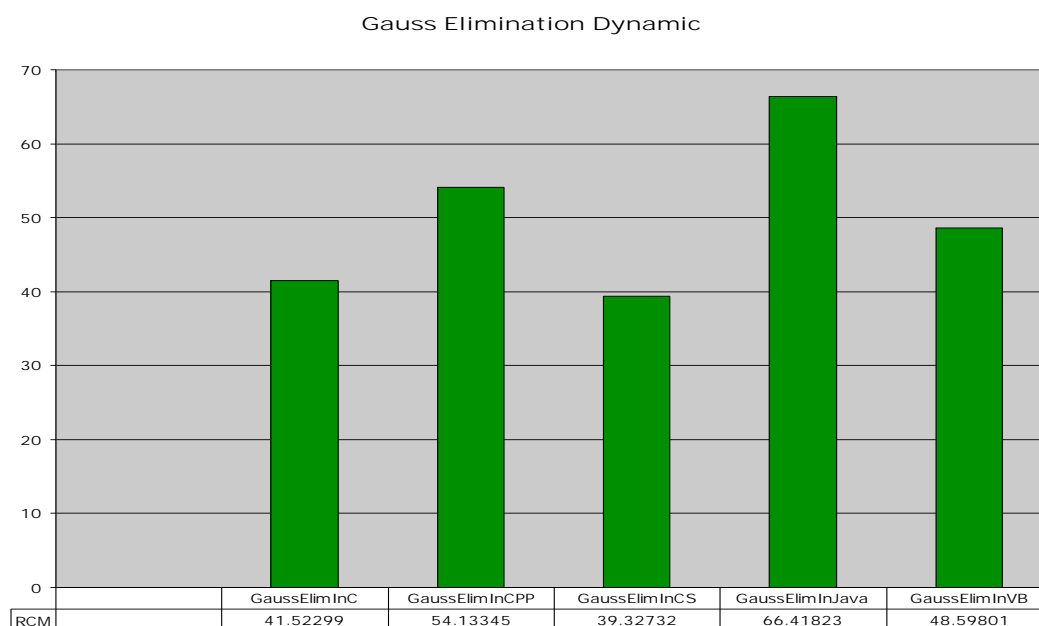


Figure 9.7 Gaussian Elimination Dynamic Measurement RCM Results.

9.2.8 Minimum and Maximum

The results were also a little surprising for this algorithm as well. C# was once again the best performer. While weak in memory size, the execution time was the best among the others. C was again second with strong areas in memory size and total objects created. C again was one of the best in execution times. Visual BASIC performed well considering that this algorithm is intended to be in worst-case time (Cormen et al., 2001). The strengths for Visual BASIC lie in execution time and in the number of routines executed while running the program. Java was fourth for this algorithm with weakness again in memory size, objects created, and execution times. The most surprising of all of the results for this algorithm was that C++ was the worst performer. With its diverse application, C++ was thought to have performed better given that this algorithm is intended for worst-case time analysis (Cormen et al., 2001; Sebesta, 1999). The major weakness in this implementation was in the area of total routine calls. C++ was the worst in this area. Also, execution time was a factor.

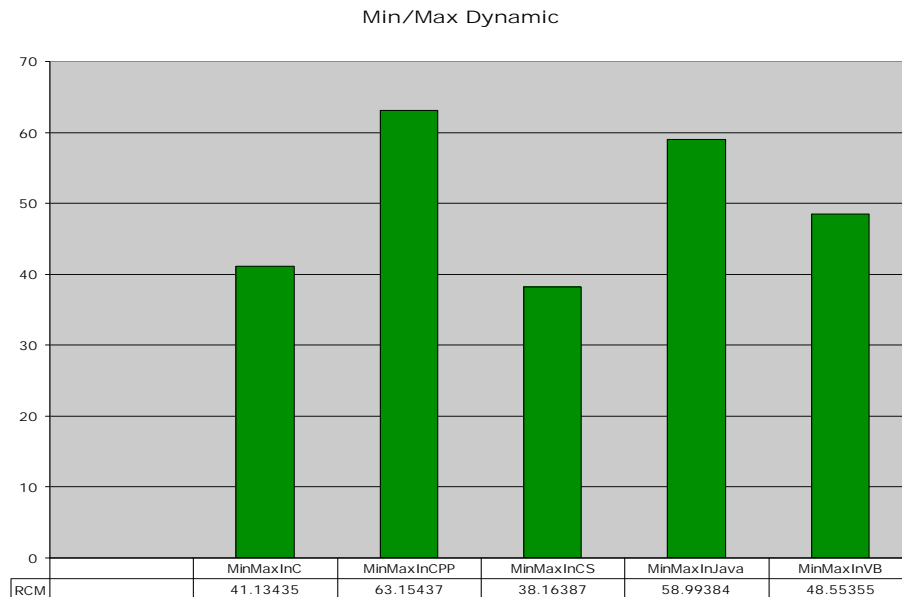


Figure 9.8 Minimum and Maximum Dynamic Measurement RCM Results.

9.2.9 Random Selection

The results for this algorithm were as expected. Random Selection is complex, in worst-case time, and involves recursion (Cormen et al., 2001). As a result, C# was the best performer overall but by just less than one point over C. C# once again excelled in execution time, total routine calls, and memory size. C was second with memory size its greatest strength. C was a little weaker in this algorithm for execution times, however. C++ was third with strengths in memory size, and execution time, but weak in the areas of total routine calls, routines executed, and total routines. Java was fourth with weakness in memory size and objects created. Java performed better in this algorithm for the routines executed metric. The worst performer was Visual BASIC. While Visual BASIC produced the smallest memory size, it was the weakest in almost every area. Its execution times were the worst of any language across all algorithms.

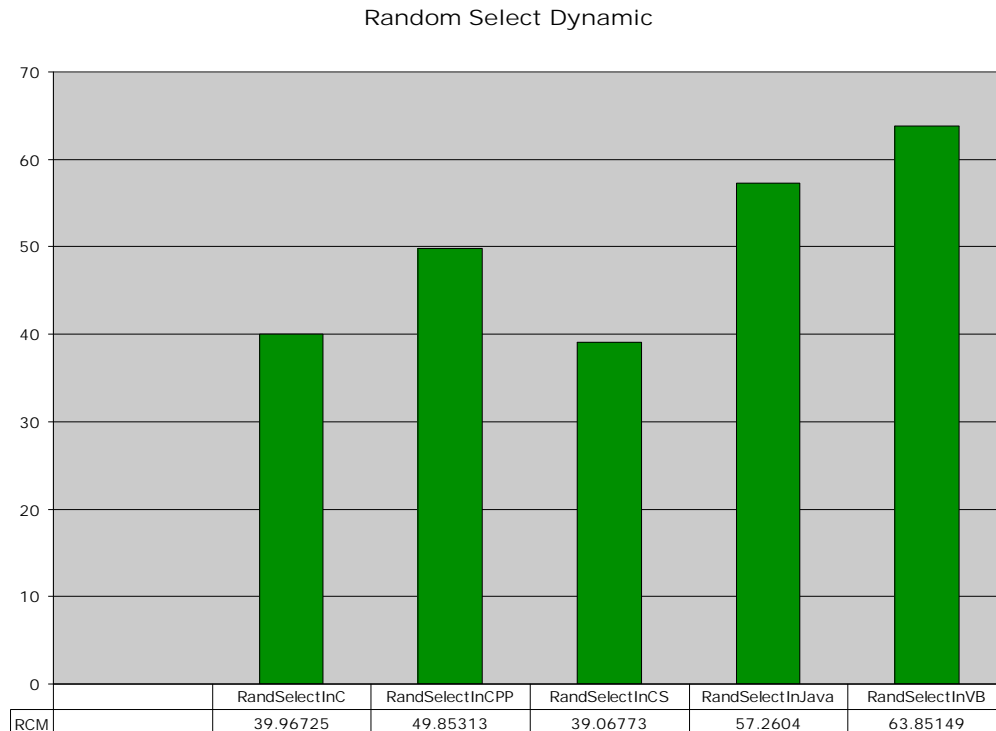


Figure 9.9 Random Selection Dynamic Measurement RCM Results.

9.3 Evaluation of Results

Since the dynamic metrics are a measure of the performance of each program, as with the static metrics, it is important to discuss a few trends. The C and C# implementations were always the best performers. This makes sense since C was designed for systems programming, which tends to take many operations that must be done in short amounts of time. In programming operating systems, resource management and efficiency were key areas in the C language design (Pratt & Zelkowitz, 2001: Sebesta, 1999). C consistently had the best results for memory size and was always strong in execution times. It seems that C performs under its design considerations. C# performed well since it is very closely tied with the Microsoft Windows operating system

(Petzold, 2001). C# might not perform as well when built to compile under other environments.

The three remaining languages were somewhat more variable. C++, while efficient and fast, seemed to always produce high values for the total number of routines defined. This is not surprising since the code written for each algorithm was intended to take advantage of the object-oriented features of C++. The libraries needed to run these programs all included data and operations that were not always necessary for each program but are available to the programmer. This is why smaller numbers were found in the measurements for routines executed.

Java, whose developers normally boast of the language's memory management capability, always seemed to fall short in this area (Sebesta, 1999). The largest values for memory size were found using Java. This may be because garbage collection is found to occur after the program terminates and since the profiler takes its memory snapshot at peak memory usage levels. If garbage collection were to be handled more frequently, the language may have performed better in this area.

Visual BASIC had the most variable results for its memory usage. In many cases it was the worst performer, but there were instances where memory usages were small. Another highly variable area was in the number of objects created. The Random Selection algorithm produced an odd result in that the total objects created was very large while the memory usage was small. Each object created for this algorithm may not have been very large but many still needed to be processed causing execution times to suffer as a result. While Visual BASIC generally did well in the static measurement portion of this analysis, clearly the variability found in this language's implementations were as a result

of the poor structure mentioned in Chapter 4 (Sebesta, 1999). Visual BASIC is not strongly typed and therefore memory is created dynamically. The programmer does not have full control over this memory allocation and so unpredictable results tend to occur (Sebesta, 1999).

9.4 Conclusions

Each language seemed to perform as the designers intended. C and C# were the most efficient languages while C++, Java, and Visual BASIC were not. The designs of the latter three development environments centered on code writing ease and data structuring rather than on performance (Pratt & Zelkowitz, 2001; Sebesta, 1999). Again the important thing about studying metrics is what can be learned (Munson, 2003). As with the static metrics, with the trends presented here, developers will have a more educated outlook on how well languages perform for given programming problems allowing for better decisions throughout the software life cycle.

CHAPTER X

METADATA MEASUREMENT ANALYSIS

10.1 Introduction

The third and final part of the measurement analysis is on the .NET metadata. As was talked about earlier, the metadata is the .NET Framework's way of passing information from one assembly (executable) to another, even if the different assemblies are written in different languages (Petzold, 2001). The benefit of the .NET environment is that pre-written modules may be used as long as they are compatible with other .NET programs. Each program has a set of tables contained within the executable code that work like a database. Each table describes some information that other programs can read from and use by making calls to routines, reading and using publicly stored data, or declaring instances of objects contained in each program (Petzold, 2001).

As will be seen in the following sections, each program produced different data even though each program was written using the same format. This is a direct effect of the different language constructs that are used (Petzold, 2001; Pratt & Zelkowitz, 2001; Sebesta, 1999). This is very much like using the English language in human speech as a communication system. If the .NET Framework is thought of as its own communication system, then it is agreed that there are possibly many ways to say the same thing. So in a way, the different languages that run on the .NET Framework are like the different

expressions that people can use in the English communication system. Therefore, all of these different uses of the Framework must be converted to a common format that all assemblies using the Framework are able to understand. It is because of this reason that different data resulted for each program written for each algorithm.

10.2 Individual Algorithm Results

10.2.1 Linear Search

The leader in this algorithm was the implementation written in C#. This is not surprising as the language was designed to work closely with the Windows operating system through the use of the .NET Framework (Petzold, 2001). Second for this algorithm was the C implementation. Despite the object oriented nature of C#, C performed well since it is *not* object oriented. There is no reason then to provide data contained in objects that are never used. The third performer was Visual BASIC. With this language there are less constructs as, again, it was not designed for complex processing (Sebesta, 1999). Fewer symbols are then needed for the language to do its work, and for it to be understood by other languages. Next was C++, which was slightly higher than the other implementations as a result of the way it performs object oriented operations. With the *#include* notation to import libraries, there are many objects that are compiled into the program that are not used. C does this as well, but looking back at the static measurement results, there were always less imported libraries in the C programs than used by C++. Finally, Java performed with the highest RCM value. This likely is caused by the Java virtual machine being compiled into the program (Pratt & Zelkowitz,

2001). Not only is the source code for the program included in the program, but also so is the Java virtual machine instruction set, and this must be passed to other programs.

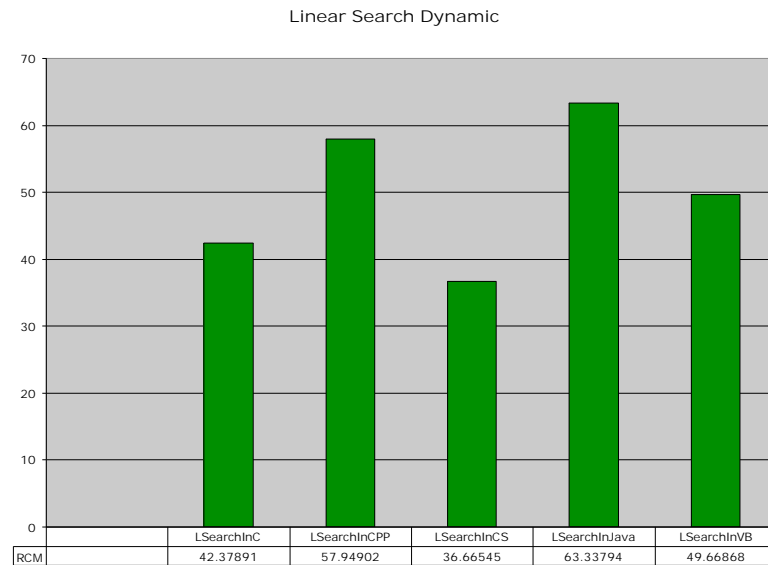


Figure 10.1 Linear Search Metadata Measurement RCM Results.

10.2.2 Bubblesort

For this algorithm, C# was once again the leader, as again this language is closely related to the Windows operating system. Second again was the C program as it does not use object-oriented features and therefore produces less metadata to be passed from one program to another. Third and fourth respectively were C++ and Java. Higher RCM values can be found here because again, C++ uses many more libraries than the other languages and these data have to be made available. Also, more objects tend to be created for C++ programs than the others, as can be seen in the dynamic measurement data in Chapter 9. Java has its virtual machine compiled into the program. Again, this causes a need for additional metadata to be created so that other programs can import what it needs of the Java virtual machine to execute methods in Java objects. Finally,

Visual BASIC produced the highest RCM value. Again, Visual BASIC is not meant for complex processing, and a Bubblesort has many instructions when compared to other faster sorting algorithms (Cormen et al., 2001: Pratt & Zelkowitz, 2001: Sebesta, 1999).

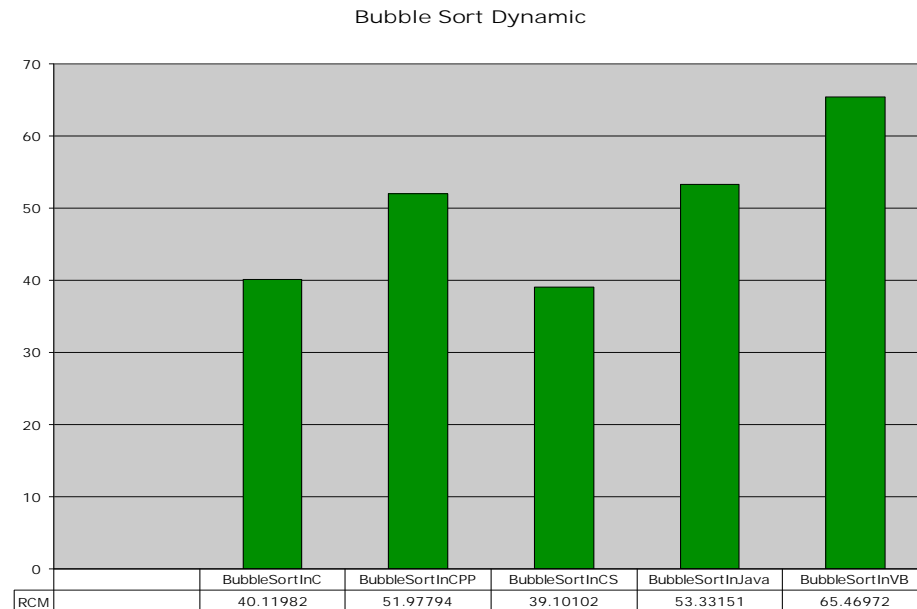


Figure 10.2 Bubblesort Metadata Measurement RCM Results.

10.2.3 Quicksort

For this algorithm, C was the least complex solution. With the efficient way in which C programs are structured, C seems ideal for Quicksort as it handles recursion well. Second, and for the same reasons as before, C# performed well. C and C# were close in this algorithm separated by less than half an RCM point. Third for this algorithm was C++, performing better as it gains some of the benefits of C, but at a disadvantage with the way it handles its object oriented structure. Fourth for this algorithm was Visual BASIC. This was slightly surprising based on Visual BASIC's simple design (Pratt & Zelkowitz, 2001: Sebesta, 1999). Recursion is considered a complex process with an

implied loop, yet it did better than Java, a language designed for higher-level applications (Cormen et al., 2001: Pratt & Zelkowitz, 2001: Sebesta, 1999). And finally, Java is fifth with the highest RCM value for this algorithm with a result of over sixty.

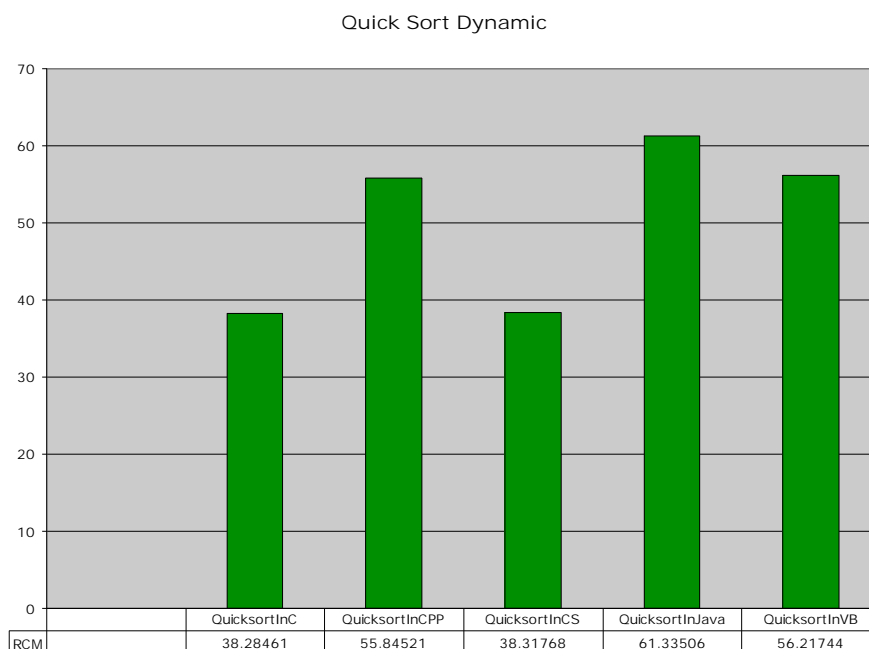


Figure 10.3 Quicksort Metadata Measurement RCM Results.

10.2.4 Naïve String Matching

A slightly surprising result in this algorithm is that C# was the leader. By looking at the static measurements, it was one of the more complex algorithms with the way it handles its string processing. Second on this list was C, and this does make sense as it handles its strings as though they are arrays of characters, not with a separate object with perhaps unused properties as in the many of the other languages (Pratt & Zelkowitz, 2001: Sebesta, 1999). Third for this algorithm was Visual BASIC, which is not unexpected. Since the algorithm was probably the simplest in the study, producing some of the lowest measurements in almost every category, it seems to follow along with the

design of Visual BASIC discussed in Chapter 4 (Pratt & Zelkowitz, 2001: Sebesta, 1999). Fourth was Java, still hurt by the fact that the Java virtual machine instruction set must be compiled into the assembly. Finally, last was C++. Even though its string processing is like C with its array of character array notation, it produced high measurement as a result of the way it handles its class implementation.

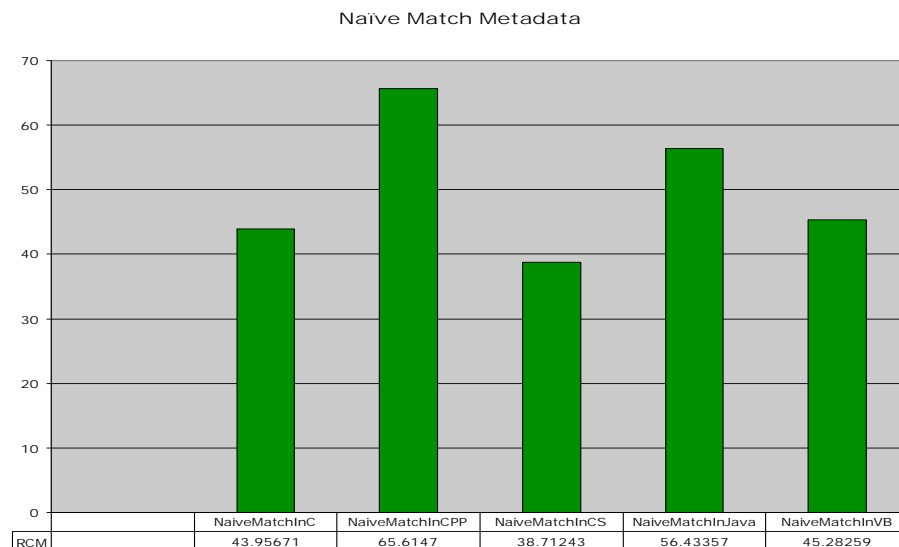


Figure 10.4 Naïve String Matching Metadata Measurement RCM Results.

10.2.5 KMP String Matching

First for this algorithm, with the least complex metadata, was C#. Again, this is slightly surprising since it had complex measurements in the other categories. Because C# is so tightly bound to the Windows operating system, it produced less metadata (Petzold, 2001). Second again was C, which again was the result of its simple notation for handling strings (Pratt & Zelkowitz, 2001: Sebesta, 1999). Third again was Visual BASIC. The RCM value for this algorithm was, however, higher than the Naïve String Matching value since this is a more complex algorithm. Visual BASIC again does not

always do well when given a complex problem since it is not designed for this (Pratt & Zelkowitz, 2001: Sebesta, 1999). Fourth on the list was again Java, with a much higher RCM value, due once again to the virtual machine having to be added to the assembly. Finally, C++ was found to be the most complex, as its class implementation continues to add to the complexity of code written in that language.

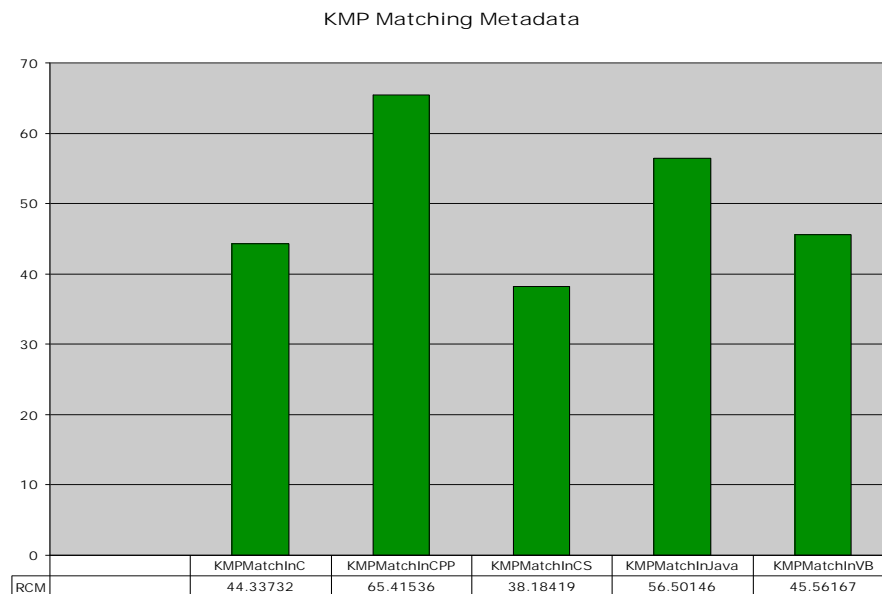


Figure 10.5 KMP String Matching Metadata Measurement RCM Results.

10.2.6 Polynomial Addition

For this algorithm, the leader with the lowest RCM value was C#. The algorithm is not complex, and C# gains much from being tightly bound to the Windows operating system (Cormen et al., 2001: Petzold, 2001). It is this combination of factors that gave C# the edge. Second was Visual BASIC, which makes sense since this algorithm is simple, and since this was the intent of Visual BASIC as a language (Pratt & Zelkowitz, 2001: Sebesta, 1999). Third was the C language, although not that distant in results from Visual BASIC. Fourth for Polynomial Addition was Java, which continues to show

complexity in the fact that it has to compile the virtual machine as part of its programs.

Last again was C++, which has struggled all throughout this category of measurements.

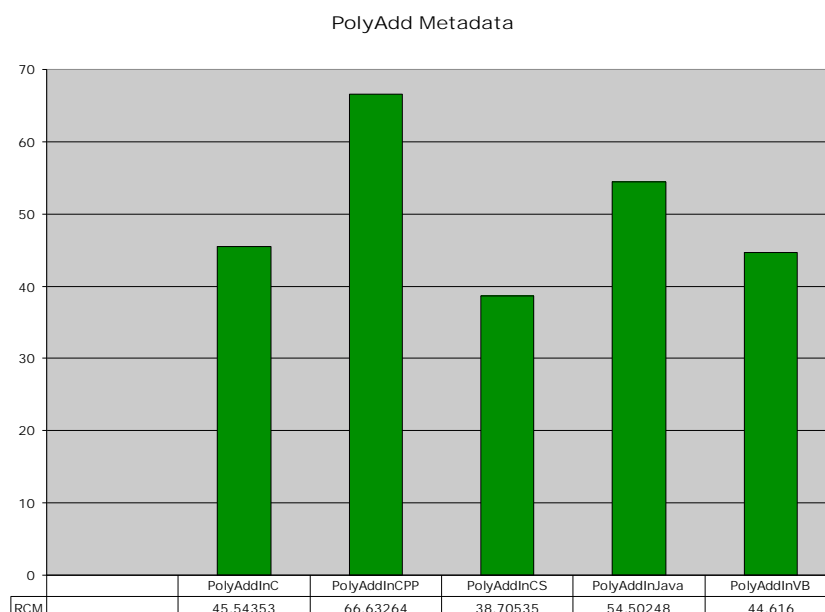


Figure 10.6 Polynomial Addition Metadata Measurement RCM Results.

10.2.7 Gaussian Elimination

The results for this algorithm were about as expected. The leader was once again C#, gaining its strength from the fact that it is tightly bound to the Windows operating system (Petzold, 2001). Second was the C implementation, which makes sense, as this algorithm is the most complex in this study (Cormen et al., 2001). While not tested in this study, it is clear that the systems programming design of C pays off well for a complex program with many calculations. Visual BASIC was third, with its simple code structure. Fourth was Java, and this makes sense since it is a complex algorithm with a virtual machine compiled into the program. Last again was C++, which, as in many of the algorithms before, continues to be hurt by the way its class definitions are designed.

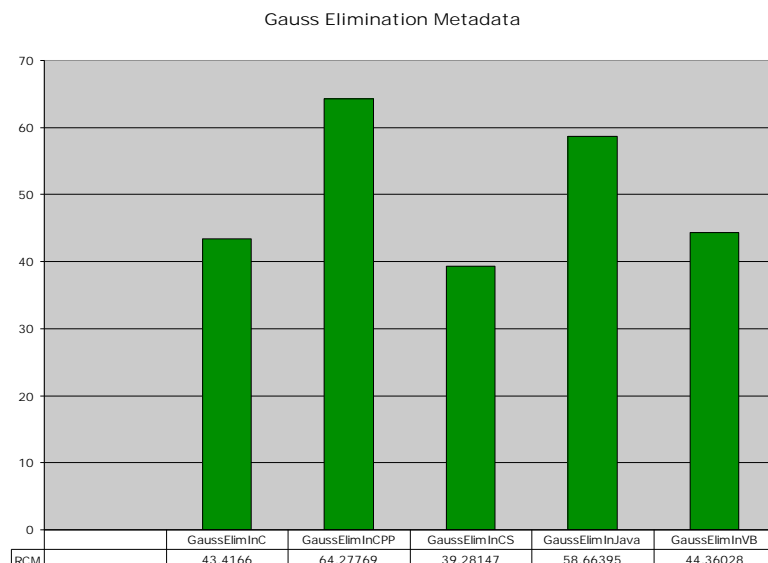


Figure 10.7 Gaussian Elimination Metadata Measurement RCM Results.

10.2.8 Minimum and Maximum

This algorithm produced some different results. First was C#, which again uses the Windows operating system closely (Petzold, 2001). Second this time was Visual BASIC. This is due to its simple structure on a simple algorithm (Cormen et al., 2001: Pratt & Zelkowitz, 2001: Sebesta, 1999). Third this time was C, which is a little surprising given that this algorithm is simple and requires nothing complex. The C implementation did well statically and dynamically, however more transferable metadata was created for this implementation. Fourth was Java, which again, as before, is continuously hurt by its virtual machine implementation. Lastly again was C++, which needs additional metadata to pass all of the libraries that can be used by programmers. This has been a major problem for C++ over the course of this part of the study.

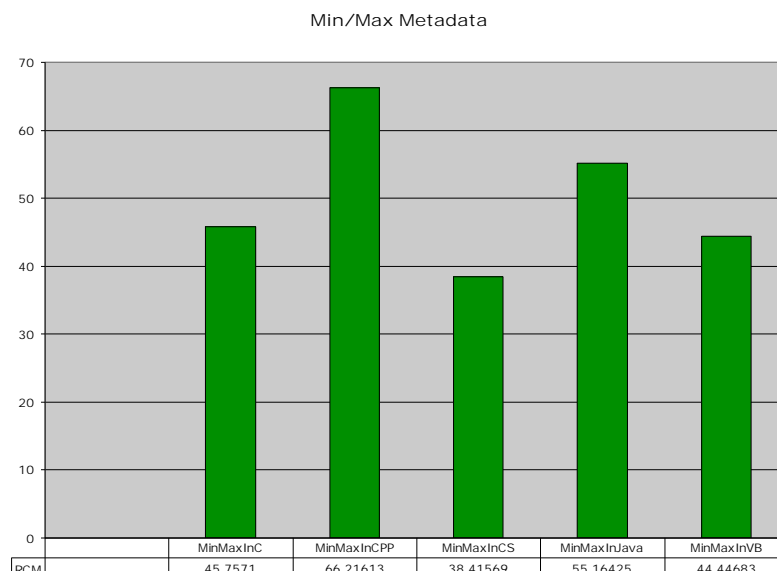


Figure 10.8 Minimum and Maximum Metadata Measurement RCM Results.

10.2.9 Random Selection

The lowest RCM value for this algorithm was posted once again by C#. Second was the C implementation, which makes sense as it did well with the Quicksort algorithm, and since both Quicksort and Random Selection are both recursive. Third was Visual BASIC, posting a much higher RCM value for this algorithm than previously. It also posted a slightly higher RCM value for Quicksort. Recursion can be complex when it uses an implied loop rather than an actual loop and because of the way Visual BASIC is structured, with its simple notation, recursion caused some complexities to appear in the implementation (Cormen et al., 2001: Pratt & Zelkowitz, 2001: Sebesta, 1999). Java was once again fourth, still plagued by its issues involving the virtual machine compiled into the program. Last again was C++, which has been consistent for almost every algorithm.

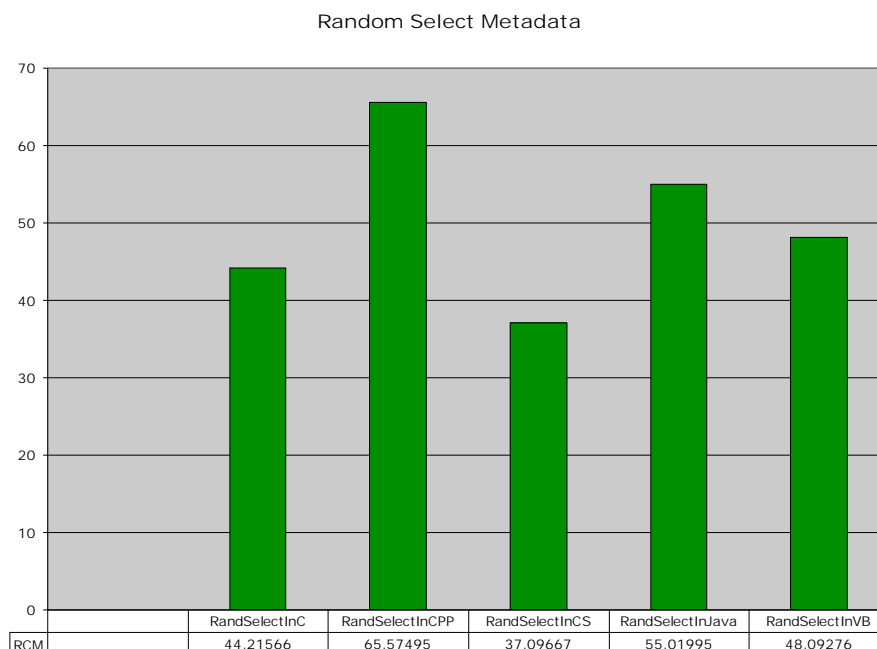


Figure 10.9 Random Selection Metadata Measurement RCM Results.

10.3 Evaluation of Results

Again, as with the static and dynamic measurements, a few trends are important to discuss. The first is that C++ continues to be hurt by all of the libraries it imports and by the way it handles its class definitions. The other languages that use classes of objects all use them as part of the main object, where C++ must define one externally. It is this notation, and the fact that it imports libraries that are not always used by programmers, which inflates its measurement numbers. The other language that did not perform well was Java, which as was mentioned in the previous sections, gains complexity in the fact that its virtual machine must be included into the programs. This complexity is seen despite the fact that C# and Java have similar constructs.

The language that continued to show the least complexity was C#. This completely makes sense since C# was designed to work closely with the Windows operating system. Since the operating system does not need much information from this language, the only metadata that is created is simply used for passing to other languages (Petzold, 2001). The other languages in this study must create additional objects as a result of the fact that they must lower their notations into the .NET Framework format. C# does not have to do this as its design already includes .NET features (Petzold, 2001). C also performed well as it was typically second or third in almost every algorithm. It gains ground on its competitors because it is not object oriented and it does not import or export many unused objects, like C++, a super set of C (Pratt & Zelkowitz, 2001: Sebesta, 1999).

The language that was variable, once again, was Visual BASIC. In simpler algorithms, Visual BASIC did well, keeping true to its designers' intent (Sebesta, 1999). It did not perform as well when compared to the other languages when presented with a complex algorithm since Visual BASIC was not designed for serious programming (Pratt & Zelkowitz, 2001). It uses simple structures, causing the need for less metadata to be passed from one program to another, but uses large amounts of metadata in complex algorithms since it must create many more of its simple structures to complete the same tasks as the other languages.

10.4 Conclusions

After seeing the "inside" of a .NET assembly, one can now see exactly how the executable programs themselves are created. Each program produced what it needed in

order for other programs written on top of the .NET Framework to understand and gain access to its data structures and function prototypes. Some languages needed more of this data, others needed less, but this information is valuable in understanding how .NET programs function against different types of algorithms.

CHAPTER XI

OVERALL MEASUREMENT ANALYSIS

11.1 Introduction

In order for this study to be complete, a look at the overall analysis of each algorithm must be present. Each language has its positives and negatives, and these were seen in the previous three chapters. A language might perform better in one category of measurement data than in another. When all of these positives and negatives are placed into a single process, a clear picture of which languages perform better can be seen. Using this overall view, a developer can see how well languages perform given a specific programming problem.

All of the measurement data taken previously was used in this portion of the study. The RCM values calculated take into account the static, dynamic, and metadata measurement values. Each RCM value represents a whole view of each language's performance with respect to each algorithm. All of the differences, no matter how small, that are found in the measurements will all be a factor in this part of the study. Now that all of the individual pieces have been seen in detail, an overall look will prove useful in the definition of how languages will perform given different problems and conditions.

11.2 Individual Algorithm Results

11.2.1 Linear Search

For this algorithm, C# was the leader with an RCM value of nearly 37. Second for Linear Search is the C implementation. This is a simple algorithm and C did well statically and dynamically. Visual BASIC came in third, which makes sense, since while this algorithm may be simple, dynamically Visual BASIC did not perform as well as some of the other languages (Pratt & Zelkowitz, 2001; Sebesta, 1999). Fourth was C++, which has an interesting result because both the static and dynamic measurements were much higher than the other algorithms, but it performed well enough dynamically that it did not fall behind completely. Java was last for this algorithm, having the only RCM value over 60. Java was also interesting in that it performed well in the static category, but the dynamic and metadata measurements were not as good as the other languages, causing Java to fall behind to fifth.

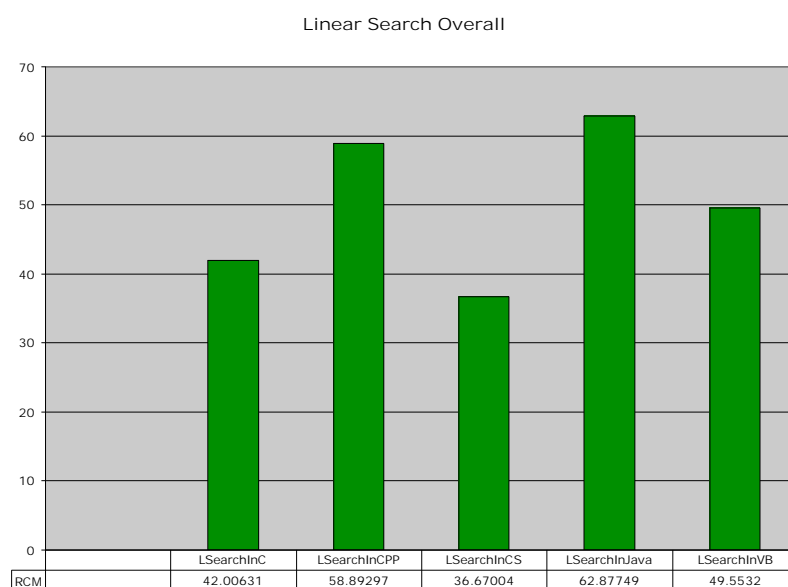


Figure 11.1 Linear Search Overall Measurement RCM Results.

11.2.2 Bubblesort

The implementation that performed with the lowest RCM value was C.

Bubblesort is considered a simple sorting algorithm and as a result, C had lower measurement values for each of the three categories (Cormen et al., 2001). Second was C#, showing once again that the language takes many benefits from being tightly bound to the Windows operating system (Petzold, 2001). Third for this algorithm was the Java implementation. Java performed well statically and dynamically but fell behind in the metadata category, causing the language to fall slightly behind. Fourth was Visual BASIC, which performed well in the static category but was hurt in the dynamic and metadata measurements. Finally, C++ follows the end of this list giving in to its static performance. While C++ performed well in the dynamic measurements, the metadata and static measurements proved to be its main areas of complexity.

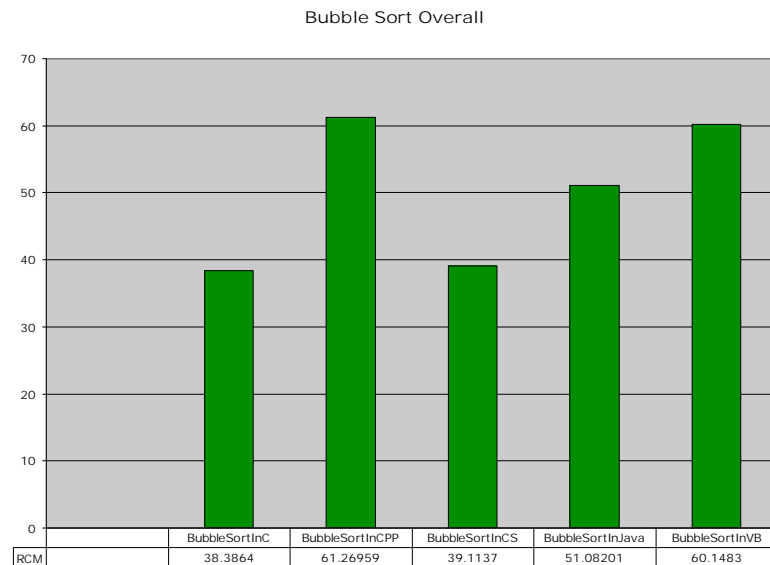


Figure 11.2 Bubblesort Overall Measurement RCM Results.

11.2.3 Quicksort

The least complex solution for this algorithm was written in C#. In this algorithm, C# produced small numbers for all of the categories in this study. Second was C, which did well in all of the categories for both recursive algorithms. The remaining three languages all produced much higher numbers for Quicksort and were close to each other by comparison to C and C#. Third was Java, which produced higher measurements in the metadata and dynamic categories. Visual BASIC scored fourth, as Quicksort is a more complex algorithm (Cormen et al., 2001). Visual BASIC was not designed for this type of processing and, while it performed well in the static measurement category, it did not, however, perform well in the others (Pratt & Zelkowitz, 2001; Sebesta, 1999). C++ was once again fifth, showing that there was great complexity in its static and metadata measurements. It did perform well dynamically, but the other categories had measurements too high for it to compete with the other languages.

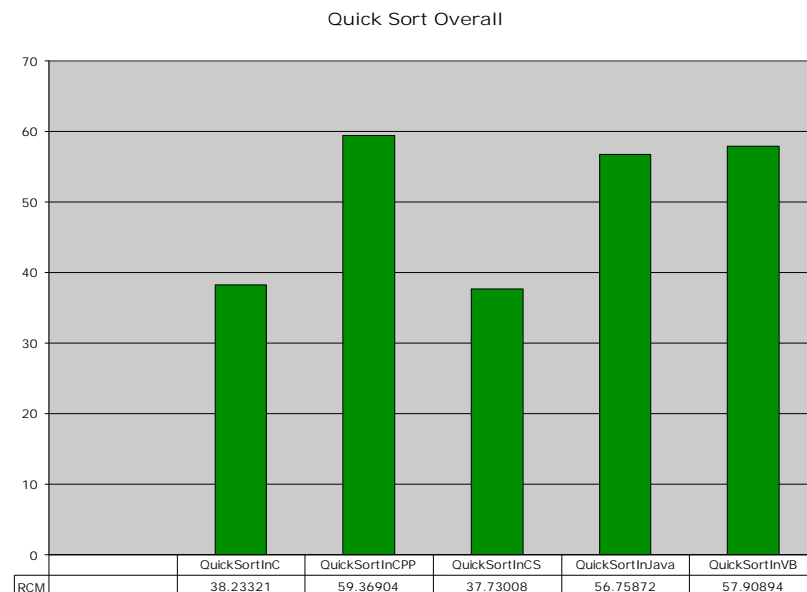


Figure 11.3 Quicksort Overall Measurement RCM Results.

11.2.4 Naïve String Matching

This algorithm produced interesting results. C was the leader for this algorithm as it gains benefit by the way it handles strings as an array of characters rather than as a separate class of objects. It is quick to parse those arrays and it can handle string processing quickly (Pratt & Zelkowitz, 2001; Sebesta, 1999). Second was C#, which is interesting because it uses a separate string object with its own set of functions and data members, but the language fared so well in the dynamic and metadata categories that it came out ahead of the others. Third for this algorithm was Visual BASIC, which scored much higher than the first two languages as a result of its high dynamic measurements. The language produced sound static numbers, but did not fare well in the other categories. Fourth was Java, also posting a high RCM number. This was due to a combination of things. These include the virtual machine structure of Java, its higher dynamic measurements, and its lack of performance statically. Last was C++, which, while exactly like C in string processing, its static and metadata numbers brought it to fifth for the Naïve String Matching algorithm.

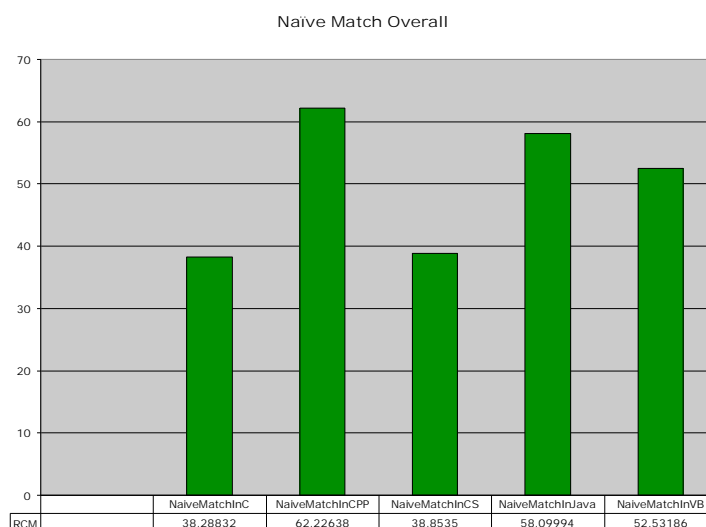


Figure 11.4 Naïve String Matching Overall Measurement RCM Results.

11.2.5 KMP String Matching

The KMP String Matching algorithm had almost the same results as its less complex counterpart. C# was the leader for this algorithm as it out performed C dynamically due to its closeness with the Windows operating system. C was second on the list again, taking strength from its array of characters notation for strings. Third was Visual BASIC, actually performing better for this algorithm, which is a little strange given Visual BASIC's design for simple programming (Pratt & Zelkowitz, 2001: Sebesta, 1999). Fourth again was Java, taking on the same disadvantages as it did previously, creating complexities from its virtual machine implementation and its metadata measurements being higher than most of the other languages. Dynamically Java also did not perform well posting higher run-times than many of the other languages. Finally C++ comes in fifth for this algorithm, taking again disadvantages in the metadata and static measurements. The language performed well dynamically but not enough to out weigh the other categories, forcing it to fifth for this algorithm.

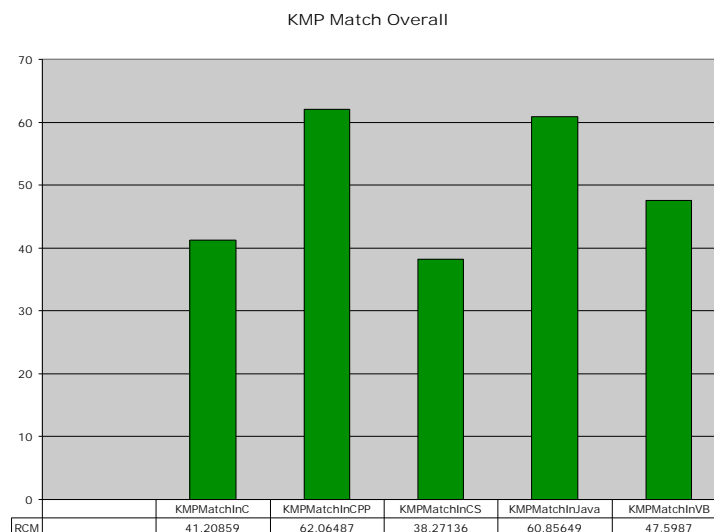


Figure 11.5 KMP String Matching Overall Measurement RCM Results.

11.2.6 Polynomial Addition

This algorithm is one of the simplest in the study, and the languages seemed to perform to their design considerations (Petzold, 2001: Pratt & Zelkowitz, 2001: Sebesta, 1999). The leader for this algorithm was once again C#, gaining its strengths once again from its closeness with the Windows operating system. Also, it performed well in each of the three categories and better than its competitors. C was second, using its design for systems programming to perform the calculations quickly when run. Also, C performed well in the metadata and static categories of measurement. Third, and not far off from the first two, was Visual BASIC, using its simple design to perform well statically and dynamically. It was hurt slightly in the metadata category forcing it behind the first two languages. Fourth was C++, gaining some of its strength in its dynamic measurements, but it found disadvantages in the metadata and static categories of measurement. Java falls fifth for this algorithm, plagued by its lesser performance in the dynamic category. The slow run-times and high amount of memory used caused Java to fall behind the others.

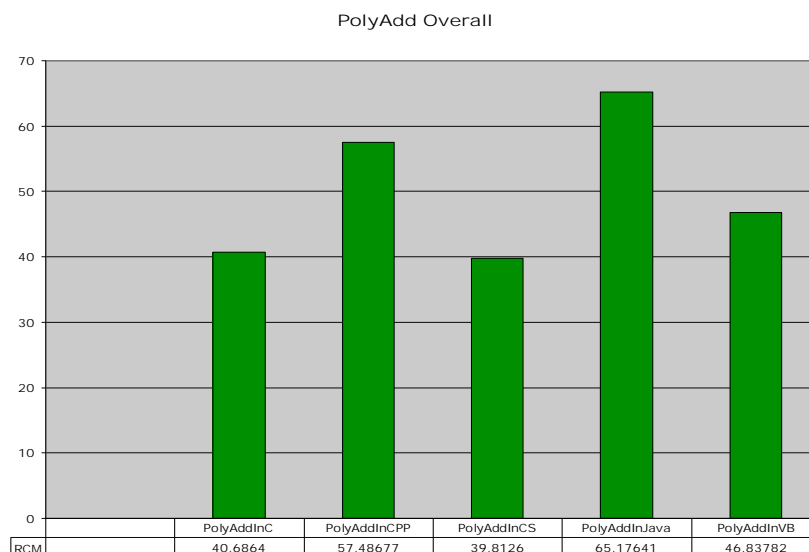


Figure 11.6 Polynomial Addition Overall Measurement RCM Results.

11.2.7 Gaussian Elimination

This algorithm is probably the most complex in this study (Cormen et al., 2001). As a result of this complexity, it seems that the languages performed as expected based on the design considerations discussed in Chapter 4. C# was again the leader for this algorithm, once again using its features that are tightly bound to the Windows operating system. C proved that its system programming design could calculate numbers quickly and with small memory usage, giving strength to its dynamic measurements. Third was Visual BASIC, which falls behind the others due to its lack of design for such a highly complex algorithm (Pratt & Zelkowitz, 2001; Sebesta, 1999). Fourth for this algorithm was C++, which struggled mainly in the static and metadata areas. It did, however, perform well dynamically but not enough to carry the rest of the weight. Last again was Java, falling behind in all three categories, but mainly in the dynamic measurements. Its run-times were slower than the rest and it used larger amounts of memory.

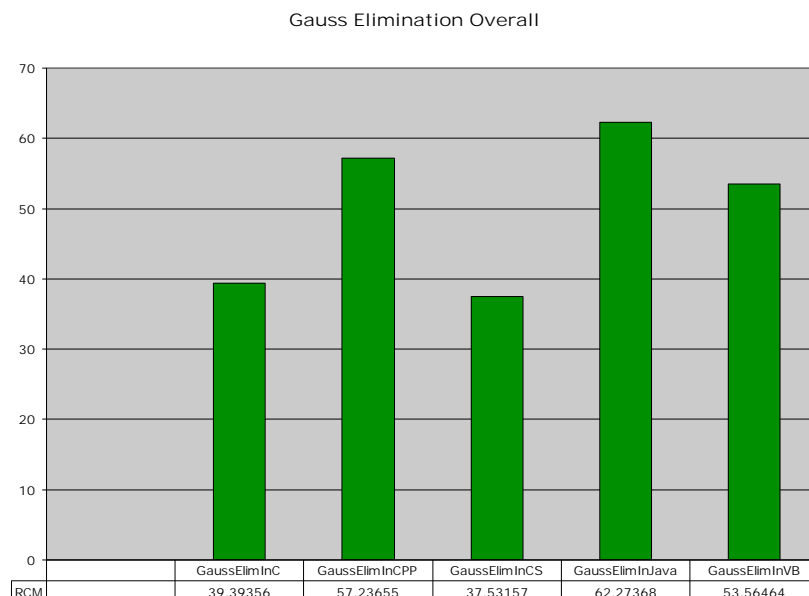


Figure 11.7 Gaussian Elimination Overall Measurement RCM Results.

11.2.8 Minimum and Maximum

This algorithm also produced the expected results. C# was once again first and continues to draw strengths in the metadata and dynamic categories. C was second, performing well dynamically and statically, but the metadata measurements proved complex enough to bring it to a higher RCM value than C#. Third for this algorithm was Visual BASIC. The algorithm is not all that complex but Visual BASIC fell behind in the static and dynamic categories forcing it slightly back (Cormen et al., 2001). Fourth was Java, which again struggles in the same areas: the metadata and the dynamic measurements. Last for this algorithm was C++, which struggled in all three areas when compared to the other languages. It had the highest LOC metrics, memory usage, and metadata measurements.

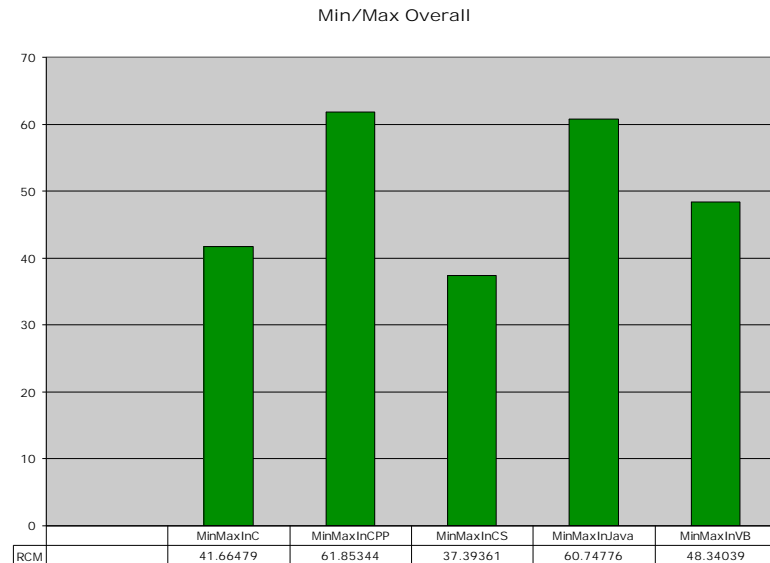


Figure 11.8 Minimum and Maximum Overall Measurement RCM Results.

11.2.9 Random Selection

For this algorithm, C# was the leader with the lowest RCM value. It once again out performed the other languages in all three categories. C was second, as it also did well with Quicksort. Both Random Selection and Quicksort are recursive and this can tend to be a complex process, but C did better in almost all areas than most of the other languages (Cormen et al., 2001). Third for this algorithm was actually Java, which is a little surprising, given that for simpler algorithms it did not perform as well. Java still struggled however in the dynamic and metadata categories. The language that came in fourth for Random Selection was Visual BASIC, which struggled in all three areas. This language also was behind some of the others in the Quicksort algorithm. Last again was C++, which found its main complexity factor in the metadata measurement section of this study.

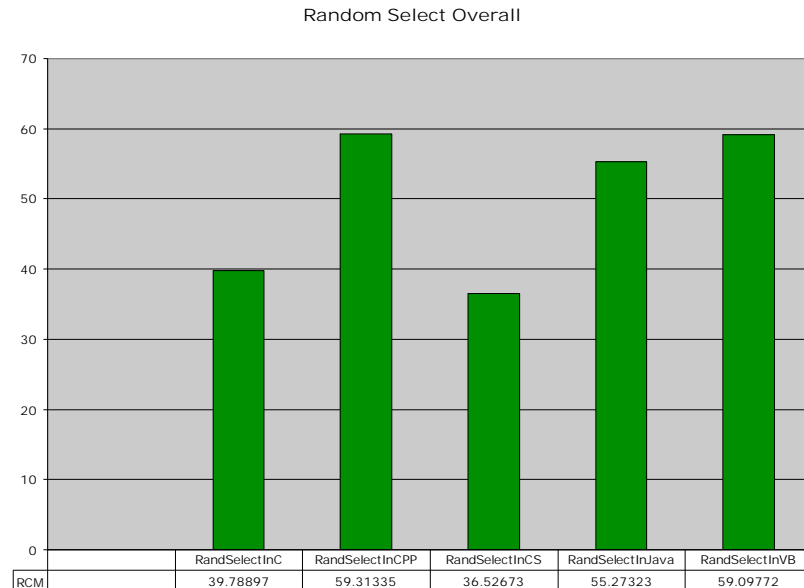


Figure 11.9 Random Selection Overall Measurement RCM Results.

11.3 Evaluation of Results

As with the other measurement chapters, a few trends are important to highlight. For every algorithm either C or C# were the least complex solutions. These languages have drawn strength from very different language features. C was designed for systems programming and for scientific applications (Pratt & Zelkowitz, 2001; Sebesta, 1999). It is important to highlight because these two important programming areas use languages that must perform quickly and with low memory usages (Pratt & Zelkowitz, 2001; Sebesta, 1999). C# on the other hand was not designed for such purposes but instead gains much strength from the Windows operating system (Petzold, 2001). It was designed to be an integral part of the .NET Framework and many of the function calls are native to the environment that is used in this study. As a result, the language uses less memory and produces less metadata. The C# language did not perform as well statically but it gained enough ground in the other categories that it produced lower RCM values.

The languages that were somewhat more variable and complex were Visual BASIC, Java, and C++. Visual BASIC was probably the most variable of all of the languages. There were cases where it had lower RCM values, and cases where it was fifth for a given algorithm. The variability can be attributed to the fact that Visual BASIC was intended to be a teaching language (Sebesta, 1999). It was not designed for large data processing or fast run-times. In simpler algorithms, however, it works as a better language because the simple structure of the language when combined with a simple algorithm will produce smaller measurement values for all three categories. Thus not much can be learned from the language, as it does not have predictable behavior.

The two most complex languages in the study were C++ and Java. Java struggled again and again due to its virtual machine structure. The language performed well statically, but when the metadata and the dynamic factors are considered, the language becomes more complex as a result of the virtual machine being compiled into each program. Java programs produced higher memory values, metadata, and slower run-times, causing a level of complexity that might be quickly seen by a developer writing code under hardware constraints (Pratt & Zelkowitz, 2001; Sebesta, 1999). The C++ language, on the other hand, found problems and complexities in other areas. While Java performed well when considered statically, C++ did not. In fact, C++ was most complex using static measurements than all of the other languages for every algorithm. Also, C++ had much larger metadata measurements than the other languages. The strength of C++ was found in its dynamic measurements as it produced faster run-times and lower memory amounts. Unfortunately for C++, however, the other two categories were so complex that it forced C++ in the last position for most of the algorithms in the study.

11.4 Conclusions

In this part of the study, all of the results were based on all measurement values taken from the other three categories. It is now possible to see how each language compares when all data is taken into account. Some languages performed as expected given the descriptions in Chapter 4, and others performed with some unexpected results. Regardless of the results, a clear view of how each language performs can now be seen. The RCM values can be used to describe the fault-prone nature of each program. This does not mean, however, that the programs measured actually have faults. RCM values instead measure the possibility that faults might exist. The higher the RCM value for a given program as compared to the same program written in the other languages, the more likely bugs might be found (Munson, 2003; Munson & Khoshgoftaar, 1990). While this might not mean much for small programs such as the ones in this study, it can be a useful tool when used on a piece of software with thousands of lines of code. The following table shows a summary of the overall RCM results taken on each language during this project.

	C	C++	C#	Java	Visual BASIC
Linear Search	42.00631	58.89297	36.67004	62.87749	49.5532
Bubble Sort	38.3864	61.26959	39.1137	51.08201	60.1483
Quick Sort	38.23321	59.36904	37.73008	56.75872	57.90894
Naïve String Match	38.28832	62.22638	38.8535	58.09994	52.53186
KMP String Match	41.20859	62.06487	38.27136	60.85649	47.5987
Polynomial Addition	40.6864	57.48677	39.8126	65.17641	46.83782
Gaussian Elimination	39.39356	57.23655	37.53157	62.27368	53.56464
Minimum / Maximum	41.66479	61.85344	37.39361	60.74776	48.34039
Polynomial Addition	39.78897	59.31335	36.52673	55.27323	59.09772

Table 11.1 Summary of Overall RCM Values.

CHAPTER XII

RESEARCH PRODUCTS AND CONCLUSIONS

12.1 Lessons Learned

As was stated in Chapter 1, the motivation for this project was to provide a useful tool that developers can use in choosing an implementation language. By coding several different algorithms, it is possible to see how programs written in different languages behave and compare to each other. Now that all of the data has been collected, analyzed, and discussed, it is possible to understand the lessons learned from this project. All of the lessons come from the experience of writing code in several different languages, maintaining coding style, and taking applicable measurements on each program to see exactly how the program behaves.

The first and most important lesson to be learned is a classical one. Simply put, there is no perfect method for solving software engineering problems and there is no perfect development tool for any situation (Munson, 2003). At the time this experiment was designed, it was originally thought that given certain data constructs or processing methods (i.e. recursion), certain languages would perform with better measurement data. This hypothesis was not studied, however, because there is not a large enough scope for such an undertaking. In order to gain a full understanding of each language and how it performs, more than once compiler set must be used in order for this hypothesis to have

any validity. While this was not exactly the case, certain languages did have their strengths and weaknesses in certain situations. For complex algorithms, defined for purposes of this project as those more difficult to write, the C, Java, and Visual BASIC languages performed well when static measurement was applied. This makes sense since each language was designed to support the production of simple source code (Pratt & Zelkowitz, 2001; Sebesta, 1999). Simple source code, as in Munson's work and for the purposes of this study, produces lower RCM values (Munson & Khoshgoftaar, 1990). The C# and C++ implementations, on the other hand, were constructed of source code that was more complex when static measurements were applied as in section 5.3 above. Dynamically, C#, C, and C++ were the better performers as can be seen from their RCM values. This is mainly because while C has a wide variety of applications, its main design focus was for systems programming which requires efficiency of time and memory (Pratt & Zelkowitz, 2001). C# is tightly coupled with the Microsoft Windows XP operating system, as has been mentioned previously, and so code was also more efficient in terms of run-times and memory (Petzold, 2001). While it may seem like C++ programs are highly complicated when compared to programs written in the other languages, its design focused on information hiding and code structure (Sebesta, 1999). C++ header files are available for programmers to "take for granted" the workings of class objects. Another main benefit of using C++ is that the language is actually a superset of C, and performs well when dynamic measurements are applied.

In metadata analysis, the clearly least complex implementations, those with the smallest RCM values, were written in C# and again this is a result of its design. As was mentioned earlier, C# was designed to work closely with the operating system and

therefore many of its basic function calls are native to Windows (Petzold, 2001). One thing that is gained from less metadata is a smaller executable file, which in turn produces smaller memory usage. Of course in today's computers, memory and space are much "cheaper" in both cost and size than in the earlier days of computing, but this is significant in that the programs written in C# produce less complex code. The other languages did not fare as well when metadata analysis was applied, C++ being the worst. In each C++ program, metadata measurements were much higher when compared to the other languages. C++ uses external libraries for its function calls and must communicate with the operating system more indirectly than C#, and therefore, in order for other .NET programs to understand its structure, more transferable data is required. C, Java, and Visual BASIC all performed between these two extremes.

Another important lesson learned in conducting this research is that no one raw measurement can fully describe how large or complex a program may be (Munson, 2003). Groups of metrics that measure different aspects of software must be combined in order to gain a full understanding of how a program performs (Wohlin, 1996). After the measurement data were collected, it was believed that certain programs would have higher or lower RCM values than they actually did. This is why it is important to produce as many valid and reproducible measurements as possible in which meaning and understanding of the program can be gained.

12.2 Software Development Questions and Answer Guidelines

Listed here are common questions developers may ask when choosing a language to use for a given programming problem. Answers to these questions may vary based on the developer's priorities and so certain programming languages may be more logical choices than others. The question is listed first followed by suggested answers. Based on these answers, specific programming languages are recommended for the specific problem based on the measurement analysis conducted in this project.

12.2.1 Questions Regarding Static Software Attributes

In simple, slower algorithms, which factor is the most important?

If the answer to this question is readability, then there are two languages that would best fit this requirement. The first was C with its low static RCM values, found in Chapter 8. C had the lowest RCM value for almost every algorithm as applied to static measurements. The second language, Visual BASIC, was designed to be easily used by non-technical students and with its English-like structure it can be easily read and understood (Petzold, 2001; Sebesta, 1999). Visual BASIC also did well in RCM values and raw measurements, but this might not always be the deciding factor for a readability requirement. From the static RCM values, it can be concluded that these programs had fewer lines of code and cyclomatic complexity, therefore allowing for easier understanding of the contents within each C and Visual BASIC program.

For simple algorithms, if the language that produces the simplest program is desired, languages that are likely choices tend to be C, C#, and Java. C, meant for more complex algorithms and systems programming, also does well in simple algorithms with

its linear structure (Pratt & Zelkowitz, 2001: Sebesta, 1999). In these simple algorithms, C produced smaller lines of code, fewer compiler directives, and had smaller RCM values. C# and Java, while not the simplest overall, use a code structure that includes data, routines, and the entry point all coupled in the same object (Petzold, 2001: Pratt & Zelkowitz, 2001: Sebesta, 1999). This allows for fewer lines of code and fewer compiler directives. Visual BASIC can also be included in this list, as its RCM values were typically very low for static attributes. Overall Visual BASIC performed well in all areas. It is both useful for code readability and code simplicity given a simpler programming task.

In more complex algorithms, which factor is the most important?

If readability is the answer to this question, C++, C#, and Java stand out as the leading choices. C++ again uses information hiding and class definitions to reduce the amount of code to be read and understood (Pratt & Zelkowitz, 2001: Sebesta, 1999). In the case of both C# and Java, since routines, data, and program entry point can all be contained within a single class, it brings down the complexity measurements of more complex algorithms. Developers do not have to traverse large quantities of code to understand code flow and organization (Petzold, 2001: Sebesta, 1999). With this type of language structure, fewer lines of code and compiler directives are needed to conduct complex programming tasks. Visual BASIC would most likely not be the best candidate if code readability were the primary factor in choosing a language for a more complex algorithm. While the language uses simple constructs, these constructs can tend to be difficult to put together for larger programs. The Visual BASIC Gaussian Elimination

program is a good example. Since this was a more complex algorithm, Visual BASIC struggled in compiler directives, lines of code, and the total RCM value gained from Principal Components Analysis (Cormen et al., 2001: Sebesta, 1999).

If overall code simplicity is the factor desired by developers for more complex algorithms, then C is the perfect choice. C was designed for algorithms such as these, and with its linear structure, programs in this language tend to have fewer lines of code and compiler directives, bringing down the RCM values (Pratt & Zelkowitz, 2001). C, in almost every case for static software attributes, had the smallest RCM value for almost every algorithm. Its versatile and adaptable design makes it very capable of being used for the most complex of algorithms.

12.2.2 Questions Regarding Dynamic Software Attributes

In simple, slower algorithms, which factor is the most important?

If run-times efficiency is the main requirement for choosing a language, then the clear leader is the C# programming language. Since C# was designed to be tightly coupled with the Microsoft Windows operating system, its function calls interface with the operating system itself producing much faster run-times (Petzold, 2001). In most cases, the run-times for C# were faster than the other programs in each algorithm. A close second would be C, whose very design was intended to produce programs with faster run-times for use in systems programming (Pratt & Zelkowitz, 2001: Sebesta, 1999). For slower algorithms, a third choice would be Visual BASIC. Since this language was designed to take on smaller projects, Visual BASIC does well in this

category. Its run-times were among the fastest. The language, however, did not perform as well for more complex algorithms as this goes against its design (Sebesta, 1999).

If developers feel that memory management is a key requirement over run-times, then C would be the best choice. C had for almost every algorithm the smallest memory usage. Second on this list would again be C# which interfaces well with the Microsoft Windows operating system (Petzold, 2001). Visual BASIC was the most variable in this area. While sometimes the leader, most often Visual BASIC was last in memory usage and tended to yield unreliable results. The most surprising of all was how Java performed in this area. While Sun Microsystems, the creators of the Java platform, boasts of how well programs in their language perform in terms of memory management, Java did the worst in this category. While Java does handle all creation and deletion of pointers for the programmer, the garbage collection happens at the end of the run, causing the snapshot of memory to yield a high value (Pratt & Zelkowitz, 2001; Sebesta, 1999).

Finally, if the answer to this question is in regards to the number of routines that the program must define and execute, the most consistent for simpler algorithms was the C language. Its total routines, routines executed, and total routine calls were most often among the smallest. C++ actually tends to do well in this category. While the total routines defined are slightly high, its routines executed measurement was lower. C# also does well in this category, producing strong measurements in all three routine counting metrics. Visual BASIC, however, as with its memory management, was erratic and could not yield reliable results. While the language sometimes produces smaller numbers for the routine measurements, often times it was the other extreme, making Visual BASIC difficult to predict.

In more complex algorithms, which factor is the most important?

If run-time efficiency is again the main requirement for choosing an implementation language, then for complex algorithms the best choice is C#. C#, for both the simple and the complex programming tasks, showed the strongest measurement values for run-time speed in seconds. C was a close competitor in most cases and so this language would also be a good candidate for an implementation language based on a run-time efficiency requirement. Lastly, C++ has decent measurements in this area. Since C++ is compiled in a similar manner to C (since C++ is a super set of C it tends to use the same compiler), it is understandable why this is true (Pratt & Zelkowitz, 2001; Sebesta, 1999). If run-time efficiency is a major factor in choosing a language, the Java language would most likely not be a candidate worth considering. This language for almost every complex algorithm performed the worst in this area.

If memory management is the key factor for a complex algorithm, C and C# would be the likely candidates. C had the most consistent use of memory in the complex algorithms used in this study. Each program run in C produced memory snapshots less than 10 Kilobytes. C#, while not nearly the performer C was in this regard, had the second strongest measurements in this area. While Visual BASIC often performed well in the memory management category, the results were unreliable. Visual BASIC would be a better choice for simpler algorithms. Once again, the biggest surprise is how Java performed. Java, boasting of clean memory usage and automatic garbage collection, produced memory snapshots much larger than its competitors.

Lastly, if routine definition and execution were the major factor in choosing a language, the clear choice for complex algorithms would be C#. While the total routine calls was on occasion high, its coupling with the Windows operating system allowed for fewer routines to be defined and used in the various runs of the C# programs (Petzold, 2001). C would be a good second choice. While its total routine calls is lower than C#, more routines are defined than in the C# programs. This leads to declarations in memory that are not always necessary. A surprising result in this category is how well Visual BASIC performed. For simpler algorithms, Visual BASIC did not perform well, but for the more complex, results were relatively consistent.

12.2.3 Questions Regarding Metadata Complexity

In simpler, slower algorithms, what is the most important factor?

If the program space is the most important of design factors, then C# is the clear choice. The language had the smallest RCM value for every algorithm in the study. It is clear that C#'s design improves the overall performance of the way .NET programs can perform. A second choice would be C as it also had simple data that was needed in order for another program to understand its structure.

If space is not the issue, and again factors like readability are the issue at hand, then languages like C++, Java, and Visual BASIC are the answers to this question. These three languages had issues in the metadata category but again, efficiency of time and space were not the design considerations for these languages. Of course, this is a matter of programmer taste and the development model.

If space and time were not the issues when developing software, what languages would be the preferred choice?

As we have seen, it is clear that C and C# are the leaders in this study with respect to the metadata measurements. But if space and time are not the issues, and factors like reliability and readability are more important for the project at hand, then C++ is the key answer. Again, it uses the concept of information hiding, allowing developers to let the objects do the work for them. Visual BASIC would again be near the top of the list as it has its simple, English-like structure that has given non-technical students the chance to learn programming (Pratt & Zelkowitz, 2001; Sebesta, 1999). The language today is fully functional, but it retains its traditional approach of being simple to read and write.

12.2.4 Language Recommendations Based On Overall Performance

Based on the static, dynamic, and .NET metadata measurements, and the PCA that was performed, the overall best performers were C# and C. In almost every case, either one or the other had the lowest RCM values for each algorithm in the study. In many cases, the two languages performed almost equally. The third best performer among the five languages was Java. While Java performed very well statically, it did not perform as well dynamically or in the metadata measurements. Fourth was Visual BASIC, which proved that simple language structure could improve complexity scores in some areas. Finally C++ would be last for a language recommendation but it must be pointed out that this is simply based on the measurements analyzed. C++ has language features that allow developers to have easier organization of source code combined with its object-oriented features. It has been made clear that language does have a factor in

algorithm performance, and that even with a common platform and coding style, these difference make themselves visible when measurements are applied. Illustrating these differences was the final goal of this research.

12.3 Further Research

Further writings on this study could include comparing languages across platforms using this study's same approach. Perhaps a developer could compare the RCM scores obtained in a Unix environment to those obtained here in the Microsoft environment and see how each language behaves differently. Another direction that this study could take is to include scripting languages such as Perl, Tcl, etc. Since these languages are interpreted and not compiled, it might introduce some interesting results when measurements are applied dynamically. Some studies have already been done comparing scripting languages, but as was mentioned earlier, this was not done on a common platform (Prechelt, 2005).

12.4 General Conclusions

The goal of all the work performed in this study is to give software engineers an understanding of how languages perform when applied to classical algorithms and measurement techniques. The results in this study are based only on the measurements defined in Chapter 6. Also, the results of this study are only valid when used in the Microsoft .NET environment and might not have the same behaviors when used elsewhere. If programmers feel that a language that is easier to read and write would be a better fit for the problem at hand, then another approach to deciding the language would

be necessary. This research only looks at where complexities in programming languages exist and what developers need to think about in that regard. Sadly, measurement on source code tends to happen very late in the software life cycle (Munson, 2003). It is hoped that from the results presented in this study, developers will have an understanding of how C, C++, C#, Java, and Visual BASIC will perform before used.

APPENDIX A

SOURCE CODE

A.1 Linear Search

A.1.1 LinearSearchInC.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int LSearch (int L[], int n, int x, int index)
{
    int Index = index;

    while (Index <= n && L[Index] != x)
        Index = Index + 1;

    if (Index > n)
        Index = 0;

    return Index;
}

int main (void)
{
    int x = 0;
    int n = 100;
    int index = 0;
    int L[100];

    srand((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < 100; i++)
        L[i] = (rand() % 100);

    x = (rand() % 100);
```

```

    index = LSearch(L, n, x, index);

    if (index == 0)
        printf("%d was not found in L.", x);
    else
        printf("%d was found at index %d.", x, index);

    for (int i = 0; i < 100; i++)
        L[i] = 0;

    index = 0;
    x = 0;

    return 0;
}

```

A.1.2 LinearSearchInCPP.cpp

```

#include <iostream>
#include "LinearSearchClass.h"

using namespace std;

int main (void)
{
    LinearSearch *linearsearch = new LinearSearch;
    int index = 0;

    index = linearsearch -> DoLinearSearch();

    if (index == 0)
        cout << linearsearch -> GetX() << " not found in L." << endl;
    else
        cout << linearsearch -> GetX() << " found at index " << index << "." <<
endl;

    delete linearsearch;
    index = 0;

    return 0;
}

```

A.1.3 LinearSearchClass.h

```

class LinearSearch
{

```

```

    public:
        LinearSearch (void);
        ~LinearSearch (void);

        int DoLinearSearch (void);
        int GetX (void);

    private:
        int x;
        int n;
        int index;
        int L[100];
};

```

A.1.4 LinearSearchClass.cpp

```

#include <stdlib.h>
#include <time.h>

#include "LinearSearchClass.h"

LinearSearch :: LinearSearch (void)
{
    srand ((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < 100; i++)
        L[i] = (rand() % 100);

    x = (rand() % 100);

    index = 0;
    n = 100;
}

LinearSearch :: ~LinearSearch (void)
{
    for (int i = 0; i < 100; i++)
        L[i] = 0;

    index = 0;
    x = 0;
    n = 0;
}

int LinearSearch :: DoLinearSearch (void)
{

```

```

        while (index <= n && L[index] != x)
            index = index + 1;

        if (index > n)
            index = 0;

        return index;
    }

    int LinearSearch :: GetX (void)
    {
        return x;
    }

```

A.1.5 LinearSearchInCS.cs

```

using System;

namespace LinearSearchInCS
{
    public class LinearSearchInCS
    {
        private int x;
        private int n;
        private int index;
        private int[] L = new int [100];

        public LinearSearchInCS()
        {
            x = 0;
            n = 100;
            index = 0;

            Random r = new Random();

            for (int i = 0; i < 100; i++)
                L[i] = r.Next(0, 100);

            x = r.Next(0, 100);
        }

        private void DoLinearSearch ()
        {
            while (index < n && L[index] != x)
                index = index + 1;

```

```

        if (index > n)
            index = 0;
    }

    public static void Main ()
    {
        LinearSearchInCS lsearch = new LinearSearchInCS();

        lsearch.DoLinearSearch();

        if (lsearch.index == 0)
            Console.WriteLine(lsearch.x + " not found in L.");
        else
            Console.WriteLine(lsearch.x + " found at index " +
lsearch.index + ".");
    }
}

```

A.1.6 LinearSearchInJava.jsl

```

import java.util.*;

public class LinearSearchInJava
{
    private int x;
    private int n;
    private int index;
    private int[] L = new int [100];

    public LinearSearchInJava()
    {
        x = 0;
        n = 100;
        index = 0;

        Random r = new Random();

        for (int i = 0; i < 100; i++)
            L[i] = r.nextInt() % n;

        x = r.nextInt() % n;
    }

    private void DoLinearSearch ()
    {

```

```

        while (index < n && L[index] != x)
            index = index + 1;

        if (index > n)
            index = 0;
    }

    public static void main (String args[])
    {
        LinearSearchInJava lsearch = new LinearSearchInJava();

        lsearch.DoLinearSearch();

        if (lsearch.index == 0)
            System.out.println(lsearch.x + " not found in L.");
        else
            System.out.println(lsearch.x + " found at index " + lsearch.index +
".");
    }
}

```

A.1.7 LinearSearchInVB.vb

```

Imports System
Imports Microsoft.VisualBasic

Public Class LinearSearchInVB
    Shared Sub Main()
        Dim x = 0
        Dim n = 100
        Dim index = 0
        Dim i = 0
        Dim L(100) As Integer

        Randomize()

        For i = 0 To 99 Step 1
            L(i) = Int((100 - 0 + 1) * Rnd()) + 0
        Next

        x = Int((100 - 0 + 1) * Rnd()) + 0

        While index < n And L(index) <> x
            index = index + 1
        End While
    End Sub
End Class

```



```

    If index > n Then
        index = 0
    End If

    If index = 0 Then
        Console.WriteLine("{0} was not found in L.", x)
    Else
        Console.WriteLine("{0} was found at index {1}.", x, index)
    End If
End Sub
End Class

```

A.2 Bubblesort

A.2.1 BubbleSortInC.cpp

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void BubbleSort (int *A, int n)
{
    int temp = 0;
    bool sorted = false;

    for (int i = 1; i < n; i++)
    {
        sorted = true;

        for (int j = 0; j <= n-1-i; j++)
        {
            if (A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
                sorted = false;
            }
        }

        if (sorted == true)
            break;
    }
}

```

```

int main (void)
{
    int A[100];
    int n = 100;

    srand((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < 100; i++)
        A[i] = (rand() % 100);

    BubbleSort(A, n);

    printf("Sorted Array is as follows:\n");

    for (int i = 0; i < 100; i++)
    {
        printf("%d\n", A[i]);
        A[i] = 0;
    }

    printf("End of Array.\n");

    return 0;
}

```

A.2.2 BubbleSortInCPP.cpp

```

#include <iostream>
#include <stdlib.h>
#include <time.h>
#include "BubbleSortClass.h"

using namespace std;

int main (void)
{
    BubbleSort *bubblesort = new BubbleSort;
    int A[100];
    int n = 100;

    srand((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < 100; i++)
        A[i] = (rand() % 100);

    bubblesort -> DoBubbleSort(A, n);
}

```

```

    cout << "Sorted Array:" << endl;

    for (int i = 0; i < 100; i++)
    {
        cout << A[i] << endl;
        A[i] = 0;
    }

    cout << "End of Array." << endl;

    delete bubblesort;

    return 0;
}

```

A.2.3 BubbleSortClass.h

```

class BubbleSort
{
    public:
        BubbleSort (void);
        ~BubbleSort (void);

        void DoBubbleSort (int *A, int n);

    private:
        bool sorted;
        int temp;
};

```

A.2.4 BubbleSortClass.cpp

```

#include "BubbleSortClass.h"

BubbleSort :: BubbleSort (void)
{
    sorted = false;
    temp = 0;
}

BubbleSort :: ~BubbleSort (void)
{
    sorted = false;
    temp = 0;
}

```

```

void BubbleSort :: DoBubbleSort (int *A, int n)
{
    for (int i = 1; i < n; i++)
    {
        sorted = true;

        for (int j = 0; j <= n-1-i; j++)
        {
            if (A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
                sorted = false;
            }
        }

        if (sorted == true)
            break;
    }
}

```

A.2.5 BubbleSortInCS.cs

```

using System;

namespace BubbleSortInCS
{
    public class BubbleSortInCS
    {
        private int n;
        private int temp;
        private int[] A = new int [100];
        private bool sorted;

        public BubbleSortInCS()
        {
            n = 100;
            temp = 0;
            sorted = false;

            Random r = new Random();

            for (int i = 0; i < 100; i++)
                A[i] = r.Next() % 100;
        }
    }
}

```

```

    }

    private void DoBubbeSort()
    {
        for (int i = 1; i < n; i++)
        {
            sorted = true;

            for (int j = 0; j <= n-1-i; j++)
            {
                if (A[j] > A[j+1])
                {
                    temp = A[j];
                    A[j] = A[j+1];
                    A[j+1] = temp;
                    sorted = false;
                }
            }

            if (sorted == true)
                break;
        }
    }

    public static void Main()
    {
        BubbleSortInCS bsort = new BubbleSortInCS();

        bsort.DoBubbeSort();

        Console.WriteLine("Sorted Array as follows:");

        for (int i = 0; i < 100; i++)
            Console.WriteLine(bsort.A[i]);

        Console.WriteLine("End of Array.");
    }
}

```

A.2.6 BubbleSortInJava.jsl

```

import java.util.*;

public class BubbleSortInJava
{

```

```

private int n;
private int temp;
private int[] A = new int [100];
private boolean sorted;

public BubbleSortInJava()
{
    n = 100;
    temp = 0;
    sorted = false;

    Random r = new Random();

    for (int i = 0; i < 100; i++)
        A[i] = r.nextInt() % 100;
}

private void DoBubbleSort()
{
    for (int i = 1; i < n; i++)
    {
        sorted = true;

        for (int j = 0; j <= n-1-i; j++)
        {
            if (A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
                sorted = false;
            }
        }

        if (sorted == true)
            break;
    }
}

public static void main (String args[])
{
    BubbleSortInJava bsort = new BubbleSortInJava();

    bsort.DoBubbleSort();

    System.out.println("Sorted Array is as follows:");
}

```

```

        for (int i = 0; i < 100; i++)
            System.out.println(bsort.A[i]);

        System.out.println("End of Array");
    }
}

```

A.2.7 BubbleSortInVB.vb

```

Imports System
Imports Microsoft.VisualBasic

Public Class BubbleSortInVB
    Shared Sub Main()
        Dim n = 100
        Dim temp = 0
        Dim i = 0
        Dim j = 0
        Dim A(100) As Integer
        Dim sorted = False

        Randomize()

        For i = 0 To 99 Step 1
            A(i) = Int((100 - 0 + 1) * Rnd()) + 0
        Next

        For i = 0 To n - 1 Step 1
            sorted = True

            For j = 0 To n - 1 - i Step 1
                If A(j) > A(j + 1) Then
                    temp = A(j)
                    A(j) = A(j + 1)
                    A(j + 1) = temp
                    sorted = False
                End If
            Next

            If sorted = True Then
                Exit For
            End If
        Next

        System.Console.WriteLine("Sorted Array as follows:")
    End Sub
End Class

```

```

    For i = 0 To 99 Step 1
        System.Console.WriteLine("{0}", A(i))
    Next

    System.Console.WriteLine("End of Array.")
End Sub
End Class

```

A.3 Quicksort

A.3.1 QuicksortInC.cpp

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int Partition (int *A, int p, int r)
{
    int x = A[r];
    int i = p - 1;
    int temp = 0;

    for (int j = p; j <= r - 1; j++)
    {
        if (A[j] <= x)
        {
            i = i + 1;

            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }

    temp = A[i + 1];
    A[i + 1] = A[r];
    A[r] = temp;

    return (i + 1);
}

void Quicksort (int *A, int p, int r)
{
    int q = 0;

```



```

        if (p < r)
        {
            q = Partition(A, p, r);
            Quicksort(A, p, q - 1);
            Quicksort(A, q + 1, r);
        }
    }

int main (void)
{
    int A[100];
    int n = 100;

    srand((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < n; i++)
        A[i] = (rand() % 100);

    Quicksort(A, 1, n);

    printf("Sorted Array is as follows:\n");

    for (int i = 0; i < n; i++)
    {
        printf("%d\n", A[i]);
        A[i] = 0;
    }

    printf("End of Array.\n");

    return 0;
}

```

A.3.2 QuicksortInCPP.cpp

```

#include <iostream>
#include <time.h>
#include <stdlib.h>
#include "QuicksortClass.h"

using namespace std;

int main (void)
{

```

```

int A[100];
int n = 100;

QuicksortClass *QuicksortClass = new QuicksortClass;

srand((unsigned int) time((time_t *) NULL));

for (int i = 0; i < n; i++)
    A[i] = (rand() % 100);

QuicksortClass -> DoQuicksort(A, 1, n);

cout << "Sorted Array is as follows: " << endl;

for (int i = 0; i < n; i++)
{
    cout << A[i] << endl;
    A[i] = 0;
}

cout << "End of Array." << endl;

delete QuicksortClass;

return 0;
}

```

A.3.3 QuicksortClass.h

```

class QuicksortClass
{
    public:
        QuicksortClass (void);
        ~QuicksortClass (void);

        void DoQuicksort (int *A, int p, int r);

    private:
        int Partition (int *A, int p, int r);

        int temp;
};

```

A.3.4 QuicksortClass.cpp

```

#include "QuicksortClass.h"

```

```

QuicksortClass :: QuicksortClass (void)
{
    temp = 0;
}

QuicksortClass :: ~QuicksortClass (void)
{
    temp = 0;
}

int QuicksortClass :: Partition (int *A, int p, int r)
{
    int x = A[r];
    int i = p - 1;

    temp = 0;

    for (int j = p; j <= r - 1; j++)
    {
        if (A[j] <= x)
        {
            i = i + 1;

            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }

    temp = A[i + 1];
    A[i + 1] = A[r];
    A[r] = temp;

    return (i + 1);
}

void QuicksortClass :: DoQuicksort (int *A, int p, int r)
{
    int q = 0;

    if (p < r)
    {
        q = Partition(A, p, r);
        DoQuicksort(A, p, q - 1);
        DoQuicksort(A, q + 1, r);
    }
}

```

```
    }
}
```

A.3.5 QuicksortInCS.cs

```
using System;

namespace QuicksortInCS
{
    public class QuicksortInCS
    {
        private int n;
        private int temp;
        private int[] A = new int [100];

        public QuicksortInCS()
        {
            n = 100;
            temp = 0;

            Random r = new Random();

            for (int i = 0; i < n; i++)
                A[i] = r.Next(0, n);
        }

        private int Partition (int p, int r)
        {
            int x = A[r];
            int i = p - 1;

            temp = 0;

            for (int j = p; j <= r - 1; j++)
            {
                if (A[j] <= x)
                {
                    i = i + 1;

                    temp = A[i];
                    A[i] = A[j];
                    A[j] = temp;
                }
            }

            temp = A[i + 1];
```

```

        A[i + 1] = A[r];
        A[r] = temp;

        return (i + 1);
    }

    public void DoQuicksort (int p, int r)
    {
        int q = 0;

        if (p < r)
        {
            q = Partition(p, r);
            DoQuicksort(p, q - 1);
            DoQuicksort(q + 1, r);
        }
    }

    public static void Main ()
    {
        QuicksortInCS qsort = new QuicksortInCS();

        qsort.DoQuicksort(0, qsort.n - 1);

        Console.WriteLine("The Array is as follows:");

        for (int i = 0; i < qsort.n; i++)
        {
            Console.WriteLine(qsort.A[i]);
            qsort.A[i] = 0;
        }
    }
}

```

A.3.6 QuicksortInJava.jsl

```

import java.util.*;

public class QuicksortInJava
{
    private int temp;
    private int n;
    private int[] A = new int [100];

    public QuicksortInJava()

```

```

{
    temp = 0;
    n = 100;

    Random r = new Random();

    for (int i = 0; i < n; i++)
        A[i] = r.nextInt() % n;
}

private int Partition (int p, int r)
{
    int x = A[r];
    int i = p - 1;

    temp = 0;

    for (int j = p; j <= r - 1; j++)
    {
        if (A[j] <= x)
        {
            i = i + 1;

            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }

    temp = A[i + 1];
    A[i + 1] = A[r];
    A[r] = temp;

    return (i + 1);
}

public void DoQuicksort (int p, int r)
{
    int q = 0;

    if (p < r)
    {
        q = Partition(p, r);
        DoQuicksort(p, q - 1);
        DoQuicksort(q + 1, r);
    }
}

```

```

    }

    public static void main (String args[])
    {
        QuicksortInJava qsort = new QuicksortInJava();

        qsort.DoQuicksort(0, qsort.n - 1);

        System.out.println("The Array is sorted as follows:");

        for (int i = 0; i < qsort.n; i++)
            System.out.println(qsort.A[i]);

        System.out.println("End of Array.");
    }
}

```

A.3.7 QuicksortInVB.vb

Imports System

Imports Microsoft.VisualBasic

Public Class QuicksortInVB

Function Partition(ByRef A() As Integer, ByRef p As Integer, ByRef r As Integer)

Dim x = A(r)

Dim i = p - 1

Dim j As Integer

Dim temp = 0

For j = p To r - 1 Step 1

 If A(j) <= x Then

 i = i + 1

 temp = A(i)

 A(i) = A(j)

 A(j) = temp

 End If

Next

temp = A(i + 1)

A(i + 1) = A(r)

A(r) = temp

Return (i + 1)

End Function

```

Sub DoQuicksort(ByRef A() As Integer, ByRef p As Integer, ByRef r As Integer)
    Dim q = 0

    If p < r Then
        q = Partition(A, p, r)
        DoQuicksort(A, p, q - 1)
        DoQuicksort(A, q + 1, r)
    End If
End Sub

Shared Sub Main()
    Dim i
    Dim n = 100
    Dim A(100) As Integer
    Dim qsort As New QuicksortInVB

    Randomize()

    For i = 0 To 99 Step 1
        A(i) = Int((100 - 0 + 1) * Rnd()) + 0
    Next

    qsort.DoQuicksort(A, 1, (n - 1))

    Console.WriteLine("Sorted Array is as follows:")

    For i = 0 To 99 Step 1
        Console.WriteLine("{0}", A(i))
    Next

    Console.WriteLine("End of Array.")
End Sub
End Class

```

A.4 Naïve String Matching

A.4.1 NavieMatchInC.cpp

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void NaiveMatch(char T[], char P[])
{
    int n = strlen(T);

```



```

    int m = strlen(P);

    for (int s = 0; s <= (n - m); s++)
    {
        if (strncmp(P, &T[s], m) == 0)
            printf("Pattern found at shift %d\n", s);
    }
}

int main (void)
{
    char P[] = "ball";
    char T[] = "Take me out to the ball game, Take me out to the crowd, Buy me
some peanuts and Cracker Jack, I don't care if I ever get back, So lets root, root root for
the Home Team,If they don't win its a shame, For its ONE, TWO, THREE strikes your
out, At the old ball game!";

    NaiveMatch(T, P);

    return 0;
}

```

A.4.2 NaiveMatchInCPP.cpp

```

#include "NaiveMatchClass.h"

int main (void)
{
    char P[] = "ball";
    char T[] = "Take me out to the ball game, Take me out to the crowd, Buy me
some peanuts and Cracker Jack, I don't care if I ever get back, So lets root, root root for
the Home Team,If they don't win its a shame, For its ONE, TWO, THREE strikes your
out, At the old ball game!";

    NaiveMatch *N_Match = new NaiveMatch;

    N_Match -> DoNaiveMatch(T, P);

    delete N_Match;

    return 0;
}

```

A.4.3 NaiveMatchClass.h

```

class NaiveMatch

```

```

{
    public:
        NaiveMatch (void);
        ~NaiveMatch (void);

        void DoNaiveMatch (char T[], char P[]);

    private:
        int n;
        int m;
};

```

A.4.4 NaiveMatchClass.cpp

```

#include <iostream>
#include <string.h>
#include "NaiveMatchClass.h"

using namespace std;

NaiveMatch :: NaiveMatch (void)
{
    n = 0;
    m = 0;
}

NaiveMatch :: ~NaiveMatch (void)
{
    n = 0;
    m = 0;
}

void NaiveMatch :: DoNaiveMatch (char T[], char P[])
{
    n = strlen(T);
    m = strlen(P);

    for (int s = 0; s <= (n - m); s++)
    {
        if (strncmp(P, &T[s], m) == 0)
            cout << "Pattern found at shift " << s << endl;
    }
}

```

A.4.5 NaiveMatchInCS.cs

```

using System;

namespace NaiveMatchInCS
{
    public class NaiveMatchInCS
    {
        private int n;
        private int m;
        private String P;
        private String T;

        public NaiveMatchInCS()
        {
            n = 0;
            m = 0;

            P = "ball";
            T = "Take me out to the ball game, Take me out to the crowd, Buy
me some peanuts and Cracker Jack, I don't care if I ever get back, So lets root, root root
for the Home Team,If they don't win its a shame, For its ONE, TWO, THREE strikes
your out, At the old ball game!";
        }

        public void DoNaiveMatch ()
        {
            char [] subText = new char[P.Length];

            n = T.Length;
            m = P.Length;

            for (int s = 0; s <= (n - m); s++)
            {
                String compareText = T.Substring(s, m);

                if (P.CompareTo(compareText) == 0)
                    Console.WriteLine("Pattern found at shift " +
s);
            }
        }

        public static void Main ()
        {
            NaiveMatchInCS N_Match = new NaiveMatchInCS();

            N_Match.DoNaiveMatch();
        }
    }
}

```

```
    }
}
```

A.4.6 NaiveMatchInJava.jsl

```
import java.util.*;
```

```
public class NaiveMatchInJava
{
```

```
    private int n;
    private int m;
    private String P;
    private String T;
```

```
    public NaiveMatchInJava()
    {
```

```
        n = 0;
        m = 0;
```

```
        P = "ball";
```

```
        T = "Take me out to the ball game, Take me out to the crowd, Buy me
some peanuts and Cracker Jack, I don't care if I ever get back, So lets root, root root for
the Home Team,If they don't win its a shame, For its ONE, TWO, THREE strikes your
out, At the old ball game!";
```

```
    }
```

```
    public void DoNaiveMatch ()
    {
```

```
        char [] subText1 = new char[P.get_Length()];
```

```
        n = T.get_Length();
        m = P.get_Length();
```

```
        for (int s = 0; s <= (n - m); s++)
```

```
        {
            T.getChars(s, (s + m), subText1, 0);
```

```
            String subText2 = new String(subText1);
```

```
            if (P.compareTo(subText2) == 0)
```

```
                System.out.println("Pattern found at shift " + s);
```

```
        }
```

```
    }
```

```
    public static void main (String args[])
    {
```

```

        NaiveMatchInJava N_Match = new NaiveMatchInJava();

        N_Match.DoNaiveMatch();
    }
}

```

A.4.7 NaiveMatchInVB.vb

```

Imports System
Imports Microsoft.VisualBasic

Public Class NaiveMatchInVB
    Shared Sub Main()
        Dim T As String = "Take me out to the ball game, Take me out to the crowd, Buy
me some peanuts and Cracker Jack, I don't care if I ever get back, So lets root, root root
for the Home Team,If they don't win its a shame, For its ONE, TWO, THREE strikes
your out, At the old ball game!"
        Dim P As String = "ball"
        Dim n = T.Length
        Dim m = P.Length
        Dim s = 0

        For s = 0 To (n - m) Step 1
            Dim compareText As String = T.Substring(s, m)

            If (P.CompareTo(compareText) = 0) Then
                Console.WriteLine("Pattern was found at shift {0}", s)
            End If
        Next
    End Sub
End Class

```

A.5 KMP String Matching

A.5.1 KMPMatchInC.cpp

```

#include <stdio.h>
#include <string.h>

int * ComputePrefixFunction (char *P)
{
    int m = strlen(P);
    int *pi = new int[m];

```

```

int k = 0;

pi[0] = 0;

for (int q = 2; q <= m; q++)
{
    while ((k > 0) && (P[k] != P[q]))
        k = pi[k];

    if (P[k] == P[q])
        k = k + 1;

    pi[q] = k;
}

return pi;
}

void KMPMatch (char *T, char *P)
{
    int n = strlen(T);
    int m = strlen(P);
    int *pi = new int[n];

    int q = 0;

    pi = ComputePrefixFunction(P);

    for (int i = 1; i < n; i++)
    {
        while ((q > 0) && (P[q] != T[i]))
            q = pi[q];

        if (P[q] == T[i])
            q = q + 1;

        if (q == m)
        {
            printf("Pattern found at shift %d\n", ((i + 1) - m));
            q = pi[q];
        }
    }
}

int main (void)
{

```

```

        char P[] = "ball";
        char T[] = "Take me out to the ball game, Take me out to the crowd, Buy me
some peanuts and Cracker Jack, I don't care if I ever get back, So lets root, root root for
the Home Team,If they don't win its a shame, For its ONE, TWO, THREE strikes your
out, At the old ball game!";

        KMPMatch(T, P);

        return 0;
}

```

A.5.2 KMPMatchInCPP.cpp

```

#include "KMPMatchClass.h"

int main (void)
{
    char P[] = "ball";
    char T[] = "Take me out to the ball game, Take me out to the crowd, Buy me
some peanuts and Cracker Jack, I don't care if I ever get back, So lets root, root root for
the Home Team,If they don't win its a shame, For its ONE, TWO, THREE strikes your
out, At the old ball game!";

    KMPMatchClass *kmpMatchClass = new KMPMatchClass(T, P);

    kmpMatchClass -> DoKMPMatch(T, P);

    delete kmpMatchClass;

    return 0;
}

```

A.5.3 KMPMatchClass.h

```

class KMPMatchClass
{
public:
    KMPMatchClass (char *T, char *P);
    ~KMPMatchClass (void);

    void DoKMPMatch (char *T, char *P);

private:
    int * ComputePrefixFunction (char *P);

    int n;
}

```

```

        int m;
};

```

A.5.4 KMPMatchClass.cpp

```

#include "KMPMatchClass.h"
#include <string.h>
#include <iostream>

using namespace std;

KMPMatchClass :: KMPMatchClass (char *T, char *P)
{
    n = strlen(T);
    m = strlen(P);
}

KMPMatchClass :: ~KMPMatchClass (void)
{
    n = 0;
    m = 0;
}

int * KMPMatchClass :: ComputePrefixFunction (char *P)
{
    int *pi = new int[m];
    int k = 0;

    pi[0] = 0;

    for (int q = 2; q <= m; q++)
    {
        while ((k > 0) && (P[k] != P[q]))
            k = pi[k];

        if (P[k] == P[q])
            k = k + 1;

        pi[q] = k;
    }

    return pi;
}

void KMPMatchClass :: DoKMPMatch (char *T, char *P)
{

```



```

int *pi = new int[n];

int q = 0;

pi = ComputePrefixFunction(P);

for (int i = 1; i < n; i++)
{
    while ((q > 0) && (P[q] != T[i]))
        q = pi[q];

    if (P[q] == T[i])
        q = q + 1;

    if (q == m)
    {
        cout << "Pattern found at shift " << ((i + 1) - m) << endl;;
        q = pi[q];
    }
}
}

```

A.5.5 KMPMatchInCS.cs

```

using System;

namespace KMPMatchInCS
{
    public class KMPMatchInCS
    {
        private int n;
        private int m;
        private String P;
        private String T;

        public KMPMatchInCS()
        {
            P = "ball";
            T = "Take me out to the ball game, Take me out to the crowd, Buy
me some peanuts and Cracker Jack, I don't care if I ever get back, So lets root, root root
for the Home Team,If they don't win its a shame, For its ONE, TWO, THREE strikes
your out, At the old ball game!";

            n = T.Length;
            m = P.Length;
        }
    }
}

```

```

private int [] ComputePrefixFunction ()
{
    int[] pi = new int[n];
    int k = 0;

    pi[0] = 0;

    for (int q = 2; q <= m; q++)
    {
        while ((k > 0) && (P[k] != P[q]))
            k = pi[k];

        if (P[k] == P[q - 1])
            k = k + 1;

        pi[q] = k;
    }

    return pi;
}

public void DoKMPPMatch ()
{
    int[] pi = new int[n];
    int q = 0;

    pi = ComputePrefixFunction();

    for (int i = 1; i < n; i++)
    {
        while ((q > 0) && (P[q] != T[i]))
            q = pi[q];

        if (P[q] == T[i])
            q = q + 1;

        if (q == m)
        {
            Console.WriteLine("Pattern found at shift " +
                ((i + 1) - m));
            q = pi[q];
        }
    }
}

```

```

        public static void Main ()
        {
            KMPMatchInCS K_Match = new KMPMatchInCS();

            K_Match.DoKMPMatch();
        }
    }
}

```

A.5.6 KMPMatchInJava.jsl

```

import java.util.*;

public class KMPMatchInJava
{
    private int n;
    private int m;
    private String P;
    private String T;

    public KMPMatchInJava ()
    {
        P = "ball";
        T = "Take me out to the ball game, Take me out to the crowd, Buy me
some peanuts and Cracker Jack, I don't care if I ever get back, So lets root, root root for
the Home Team,If they don't win its a shame, For its ONE, TWO, THREE strikes your
out, At the old ball game!";

        n = T.get_Length();
        m = P.get_Length();
    }

    private int[] ComputePrefixFunction ()
    {
        int pi[] = new int[m];
        int k = 0;

        pi[0] = 0;

        for (int q = 2; q < m; q++)
        {
            while ((k > 0) && (P.get_Chars(k) != P.get_Chars(q)))
                k = pi[k];

            if (P.get_Chars(k) == P.get_Chars(q))
                k = k + 1;
        }
    }
}

```

```

        pi[q] = k;
    }

    return pi;
}

public void DoKMPMatch ()
{
    int pi[] = new int[n];
    int q = 0;

    pi = ComputePrefixFunction();

    for (int i = 1; i < n; i++)
    {
        while ((q > 0) && (P.get_Chars(q) != T.get_Chars(i)))
            q = pi[q];

        if (P.get_Chars(q) == T.get_Chars(i))
            q = q + 1;

        if (q == m)
        {
            System.out.println("Patter found at shift " + ((i + 1) - m));
            q = pi[q - 1];
        }
    }
}

public static void main (String args[])
{
    KMPMatchInJava K_Match = new KMPMatchInJava();

    K_Match.DoKMPMatch();
}
}

```

A.5.7 KMPMatchInVB.vb

Imports System

Imports Microsoft.VisualBasic

Public Class KMPMatchInVB

Function ComputePrefixFunction(ByVal P As String)

Dim m = P.Length

```

Dim pi(m) As Integer
Dim k = 0
Dim q As Integer

pi(0) = 0

For q = 2 To (m - 1) Step 1
    While ((k > 0) And (P.Chars(k) <> P.Chars(q)))
        k = pi(k)
    End While

    If P.Chars(k) = P.Chars(q) Then
        k = k + 1
    End If

    pi(q) = k
Next

Return pi
End Function

Sub DoKMPSMatch(ByVal T As String, ByVal P As String)
    Dim n = T.Length
    Dim m = P.Length
    Dim pi(n) As Integer
    Dim q = 0
    Dim i As Integer

    pi = ComputePrefixFunction(P)

    For i = 0 To (n - 1) Step 1
        While (q > 0) And (P.Chars(q) <> T.Chars(i))
            q = pi(q)
        End While

        If P.Chars(q) = T.Chars(i) Then
            q = q + 1
        End If

        If q = m Then
            Console.WriteLine("Pattern found at shift {0}", ((i + 1) - m))
            q = pi(q)
        End If
    Next
End Sub

```

```

Shared Sub Main()
    Dim T As String = "Take me out to the ball game, Take me out to the crowd, Buy
me some peanuts and Cracker Jack, I don't care if I ever get back, So lets root, root root
for the Home Team,If they don't win its a shame, For its ONE, TWO, THREE strikes
your out, At the old ball game!"
    Dim P As String = "ball"
    Dim K_Match As New KMPMatchInVB

    K_Match.DoKMPMatch(T, P)
End Sub
End Class

```

A.6 Polynomial Addition

A.6.1 PolyAddInC.cpp

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (void)
{
    int A[100];
    int B[100];
    int C[100];

    srand((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < 100; i++)
    {
        A[i] = (rand() % 100);
        B[i] = (rand() % 100);
    }

    for (int j = 0; j < 100; j++)
        C[j] = A[j] + B[j];

    printf("Polynomial coefficients as follows:");

    for (int k = 0; k < 100; k++)
        printf("%d\n", C[k]);

    return 0;
}

```

A.6.2 PolyAddInCPP.cpp

```
#include "PolyAddClass.h"

int main (void)
{
    PolyAddClass *polyAddClass = new PolyAddClass();

    polyAddClass -> DoPolyAdd();
    polyAddClass -> OutputCoEs();

    delete polyAddClass;

    return 0;
}
```

A.6.3 PolyAddClass.h

```
class PolyAddClass
{
    public:
        PolyAddClass (void);

        void DoPolyAdd (void);
        void OutputCoEs (void);

    private:
        int A[100];
        int B[100];
        int C[100];
};
```

A.6.4 PolyAddClass.cpp

```
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include "PolyAddClass.h"

using namespace std;

PolyAddClass :: PolyAddClass (void)
{
    srand((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < 100; i++)
```

```

        {
            A[i] = (rand() % 100);
            B[i] = (rand() % 100);
        }
    }

void PolyAddClass :: DoPolyAdd (void)
{
    for (int j = 0; j < 100; j++)
        C[j] = A[j] + B[j];
}

void PolyAddClass:: OutputCoEs (void)
{
    cout << "Polynomial coefficients as follows:" << endl;

    for (int k = 0; k < 100; k++)
        cout << C[k] << endl;
}

```

A.6.5 PolyAddInCS.cs

using System;

```

namespace PolyAddInCS
{
    class PolyAddInCS
    {
        private int[] A = new int [100];
        private int[] B = new int [100];
        private int[] C = new int [100];

        public PolyAddInCS()
        {
            Random r = new Random();

            for (int i = 0; i < 100; i++)
            {
                A[i] = r.Next() % 100;
                B[i] = r.Next() % 100;
            }
        }

        private void DoPolyAdd()
        {
            for (int j = 0; j < 100; j++)

```



```

        C[j] = A[j] + B[j];
    }

    private void OutputCoEs()
    {
        Console.WriteLine("Polynomial coefficients as follows:");

        for (int k = 0; k < 100; k++)
            Console.WriteLine(C[k]);
    }

    public static void Main()
    {
        PolyAddInCS polyAddInCS = new PolyAddInCS();

        polyAddInCS.DoPolyAdd();
        polyAddInCS.OutputCoEs();
    }
}

```

A.6.6 PolyAddInJava.jsl

```

import java.util.*;

public class PolyAddInJava
{
    private int[] A = new int[100];
    private int[] B = new int[100];
    private int[] C = new int[100];

    public PolyAddInJava()
    {
        Random r = new Random();

        for (int i = 0; i < 100; i++)
        {
            A[i] = r.nextInt() % 100;
            B[i] = r.nextInt() % 100;
        }
    }

    private void DoPolyAdd()
    {
        for (int j = 0; j < 100; j++)

```

```

        C[j] = A[j] + B[j];
    }

    private void OutputCoEs()
    {
        System.out.println("Polynomial coefficients as follows:");

        for (int k = 0; k < 100; k++)
            System.out.println(C[k]);
    }

    public static void main (String args[])
    {
        PolyAddInJava polyAddInJava = new PolyAddInJava();

        polyAddInJava.DoPolyAdd();
        polyAddInJava.OutputCoEs();
    }
}

```

A.6.7 PolyAddInVB.vb

Imports System

Imports Microsoft.VisualBasic

Public Class PolyAddInVB

Shared Sub Main()

Dim A(100) As Integer

Dim B(100) As Integer

Dim C(100) As Integer

Dim i As Integer

Randomize()

For i = 0 To 100 Step 1

A(i) = Int((100 - 0 + 1) * Rnd()) + 0

B(i) = Int((100 - 0 + 1) * Rnd()) + 0

Next

For i = 0 To 100 Step 1

C(i) = A(i) + B(i)

Next

Console.WriteLine("Polynomial coefficients as follows:")

For i = 0 To 100 Step 1

```

        Console.WriteLine("{0}", C(i))
    Next
End Sub
End Class

```

A.7 Gaussian Elimination

A.7.1 GaussElimInC.cpp

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define n 100

float A[n][n];
float b[n];
float x[n];
int Pi[n];

void LUP_Decomposition (void)
{
    int k_prime = 0;
    int temp = 0;

    for (int i = 0; i < n; i++)
        Pi[i] = i;

    for (int k = 0; k < n; k++)
    {
        double p = 0;

        for (int i = k; i < n; i++)
        {
            if (abs(A[i][k]) > p)
            {
                p = abs(A[i][k]);
                k_prime = i;
            }
        }

        if (p == 0)

```

```

        {
            printf("Error, Singular Matrix");
            return;
        }

        temp = Pi[k];
        Pi[k] = Pi[k_prime];
        Pi[k_prime] = temp;

        for (int i = 0; i < n; i++)
        {
            temp = A[k][i];
            A[k][i] = A[k_prime][i];
            A[k_prime][i] = temp;
        }

        for (int i = (k + 1); i < n; i++)
        {
            A[i][k] = (A[i][k] / A[k][k]);

            for (int j = (k + 1); j < n; j++)
                A[i][j] = A[i][j] - (A[i][k] * A[k][j]);
        }
    }
}

void LUP_Solve (void)
{
    float y[n];
    float temp = 0;

    y[1] = b[Pi[1]];

    for (int i = 0; i < n; i++)
    {
        for (int j = 1; j < (i - 1); j++)
            temp = temp + (A[i][j] * y[j]);

        y[i] = (b[Pi[i]] - temp);
        temp = 0;
    }

    x[n] = (y[n] / A[n][n]);

    for (int i = (n - 1); i >= 0; i--)
    {

```

```

        for (int j = i + 1; j < n; j++)
            temp = temp + (A[i][j] * x[j]);

        x[i] = ((y[i] - temp) / A[i][i]);
        temp = 0;
    }
}

int main (void)
{
    int Uks = n;

    srand((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A[i][j] = (rand() % 100);

    for (int i = 0; i < n; i++)
        b[i] = (rand() % 100);

    LUP_Decomposition();
    LUP_Solve();

    printf("Solutions for %d unknowns:\n", Uks);

    for (int i = 0; i < n; i++)
        printf("%f\n", x[i]);

    return 0;
}

```

A.7.2 GaussElimInCPP.cpp

```

#include "GaussClass.h"

int main (void)
{
    GaussElim *gaussElim = new GaussElim();

    gaussElim -> LUP_Decomposition();
    gaussElim -> LUP_Solve();
    gaussElim -> OutputUnknowns();

    delete gaussElim;
}

```

```

        return 0;
    }

```

A.7.3 GaussClass.h

```

#define n 100

class GaussElim
{
    public:
        GaussElim (void);

        void LUP_Decomposition (void);
        void LUP_Solve (void);
        void OutputUnknowns (void);

    private:
        float A[n][n];
        float b[n];
        float x[n];
        int Pi[n];
};

```

A.7.4 GaussClass.cpp

```

#include <stdio.h>
#include <time.h>
#include <iostream>
#include "GaussClass.h"

using namespace std;

GaussElim :: GaussElim (void)
{
    srand((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A[i][j] = (rand() % 100);

    for (int i = 0; i < n; i++)
        b[i] = (rand() % 100);
}

void GaussElim :: LUP_Decomposition (void)
{

```

```

int k_prime = 0;
int temp = 0;

for (int i = 0; i < n; i++)
    Pi[i] = i;

for (int k = 0; k < n; k++)
{
    double p = 0;

    for (int i = k; i < n; i++)
    {
        if (abs(A[i][k]) > p)
        {
            p = abs(A[i][k]);
            k_prime = i;
        }
    }

    if (p == 0)
    {
        cout << "Error, Singular Matrix" << endl;
        return;
    }

    temp = Pi[k];
    Pi[k] = Pi[k_prime];
    Pi[k_prime] = temp;

    for (int i = 0; i < n; i++)
    {
        temp = A[k][i];
        A[k][i] = A[k_prime][i];
        A[k_prime][i] = temp;
    }

    for (int i = (k + 1); i < n; i++)
    {
        A[i][k] = (A[i][k] / A[k][k]);

        for (int j = (k + 1); j < n; j++)
            A[i][j] = A[i][j] - (A[i][k] * A[k][j]);
    }
}
}

```

```

void GaussElim :: LUP_Solve (void)
{
    float y[n];
    float temp = 0;

    y[1] = b[Pi[1]];

    for (int i = 0; i < n; i++)
    {
        for (int j = 1; j < (i - 1); j++)
            temp = temp + (A[i][j] * y[j]);

        y[i] = (b[Pi[i]] - temp);
        temp = 0;
    }

    x[n] = (y[n] / A[n][n]);

    for (int i = (n - 1); i >= 0; i--)
    {
        for (int j = i + 1; j < n; j++)
            temp = temp + (A[i][j] * x[j]);

        x[i] = ((y[i] - temp) / A[i][i]);
        temp = 0;
    }
}

void GaussElim :: OutputUnknowns (void)
{
    cout << "Solutions for " << n << " unknowns:" << endl;

    for (int i = 0; i < n; i++)
        cout << x[i] << endl;
}

```

A.7.5 GaussElimInCS.cs

```

using System;

namespace GaussElimInCS
{
    public class GaussElimInCS
    {
        private const int n = 100;
        private float [ , ] A = new float[n, n];
    }
}

```



```

private float [] b = new float[n];
private float [] x = new float [n];
private int [] Pi = new int[n];

public GaussElimInCS()
{
    Random r = new Random();

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A[i, j] = r.Next() % n;

    for (int i = 0; i < n; i++)
        b[i] = r.Next() % n;
}

private void LUP_Decomposition()
{
    int k_prime = 0;
    int temp1 = 0;

    for (int i = 0; i < n; i++)
        Pi[i] = i;

    for (int k = 0; k < n; k++)
    {
        double p = 0;

        for (int i = k; i < n; i++)
        {
            if (Math.Abs(A[i, k]) > p)
            {
                p = Math.Abs(A[i, k]);
                k_prime = i;
            }
        }

        if (p == 0)
        {
            Console.WriteLine("Error, Singular Matrix");
            return;
        }

        temp1 = Pi[k];
        Pi[k] = Pi[k_prime];
        Pi[k_prime] = temp1;
    }
}

```

```

        for (int i = 0; i < n; i++)
        {
            float temp2 = A[k, i];
            A[k, i] = A[k_prime, i];
            A[k_prime, i] = temp2;
        }

        for (int i = (k + 1); i < n; i++)
        {
            A[i, k] = (A[i, k] / A[k, k]);

            for (int j = (k + 1); j < n; j++)
                A[i, j] = A[i, j] - (A[i, k] * A[k, j]);
        }
    }
}

private void LUP_Solve()
{
    float []y = new float[n];
    float temp = 0;

    y[1] = b[Pi[1]];

    for (int i = 0; i < n; i++)
    {
        for (int j = 1; j < (i - 1); j++)
            temp = temp + (A[i, j] * y[j]);

        y[i] = (b[Pi[i]] - temp);
        temp = 0;
    }

    x[n - 1] = (y[n - 1] / A[n - 1, n - 1]);

    for (int i = (n - 1); i >= 0; i--)
    {
        for (int j = i + 1; j < n; j++)
            temp = temp + (A[i, j] * x[j]);

        x[i] = ((y[i] - temp) / A[i, i]);
        temp = 0;
    }
}

```

```

private void OutputUnknwons()
{
    Console.WriteLine("Solutions for " + n + " unknowns:");

    for (int i = 0; i < n; i++)
        Console.WriteLine(x[i]);
}

public static void Main()
{
    GaussElimInCS gaussElim = new GaussElimInCS();

    gaussElim.LUP_Decomposition();
    gaussElim.LUP_Solve();
    gaussElim.OutputUnknwons();
}
}
}

```

A.7.6 GaussElimInJava.jsl

```

import java.util.*;

public class GaussElimInJava
{
    private static int n = 100;
    private float [ , ] A = new float[n, n];
    private float [] b = new float[n];
    private float [] x = new float [n];
    private int [] Pi = new int[n];

    public GaussElimInJava()
    {
        Random r = new Random();

        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                A[i, j] = r.nextInt() % n;

        for (int i = 0; i < n; i++)
            b[i] = r.nextInt() % n;
    }

    private void LUP_Decomposition()
    {
        int k_prime = 0;

```

```

int temp1 = 0;

for (int i = 0; i < n; i++)
    Pi[i] = i;

for (int k = 0; k < n; k++)
{
    double p = 0;

    for (int i = k; i < n; i++)
    {
        if (Math.abs(A[i, k]) > p)
        {
            p = Math.abs(A[i, k]);
            k_prime = i;
        }
    }

    if (p == 0)
    {
        System.out.println("Error, Singular Matrix");
        return;
    }

    temp1 = Pi[k];
    Pi[k] = Pi[k_prime];
    Pi[k_prime] = temp1;

    for (int i = 0; i < n; i++)
    {
        float temp2 = A[k, i];
        A[k, i] = A[k_prime, i];
        A[k_prime, i] = temp2;
    }

    for (int i = (k + 1); i < n; i++)
    {
        A[i, k] = (A[i, k] / A[k, k]);

        for (int j = (k + 1); j < n; j++)
            A[i, j] = A[i, j] - (A[i, k] * A[k, j]);
    }
}

private void LUP_Solve()

```

```

{
    float []y = new float[n];
    float temp = 0;

    y[1] = b[Pi[1]];

    for (int i = 0; i < n; i++)
    {
        for (int j = 1; j < (i - 1); j++)
            temp = temp + (A[i, j] * y[j]);

        y[i] = (b[Pi[i]] - temp);
        temp = 0;
    }

    x[n - 1] = (y[n - 1] / A[n - 1, n - 1]);

    for (int i = (n - 1); i >= 0; i--)
    {
        for (int j = i + 1; j < n; j++)
            temp = temp + (A[i, j] * x[j]);

        x[i] = ((y[i] - temp) / A[i, i]);
        temp = 0;
    }
}

private void OutputUnknwons()
{
    System.out.println("Solutions for " + n + " unknowns:");

    for (int i = 0; i < n; i++)
        System.out.println(x[i]);
}

public static void main (String args[])
{
    GaussElimInJava gaussElim = new GaussElimInJava();

    gaussElim.LUP_Decomposition();
    gaussElim.LUP_Solve();
    gaussElim.OutputUnknwons();
}
}

```

A.7.7 GaussElimInVB.vb

Imports System

Imports Microsoft.VisualBasic

Public Class GaussElimInVB

Const n = 100

Dim A(n, n) As Single

Dim b(n) As Single

Dim x(n) As Single

Dim Pi(n) As Integer

Sub LUP_Decomposition()

Dim k_prime = 0

Dim temp = 0

Dim i As Integer

Dim k As Integer

Dim j As Integer

Dim temp1 As Integer

Dim temp2 As Single

For i = 0 To n Step 1

Pi(i) = i

Next

For k = 0 To n Step 1

Dim p As Single

p = 0

For i = k To n Step 1

If Math.Abs(A(i, k)) > p Then

p = Math.Abs(A(i, k))

k_prime = i

End If

Next

If p = 0 And i <> (n + 1) Then

Console.WriteLine("Error, Singular Matrix")

End

End If

temp1 = Pi(k)

Pi(k) = Pi(k_prime)

Pi(k_prime) = temp1

For i = 0 To n Step 1

```

        temp2 = A(k, i)
        A(k, i) = A(k_prime, i)
        A(k_prime, i) = temp2
    Next

    For i = (k + 1) To n Step 1
        A(i, k) = (A(i, k) / A(k, k))

        For j = (k + 1) To n Step 1
            A(i, j) = A(i, j) - (A(i, k) * A(k, j))
        Next
    Next
Next
End Sub

Sub LUP_Solve()
    Dim y(n) As Single
    Dim temp As Single
    Dim i As Integer
    Dim j As Integer

    y(0) = b(Pi(0))

    For i = 0 To n Step 1
        For j = 1 To (i - 1) Step 1
            temp = temp + (A(i, j) * y(j))
        Next

        y(i) = (b(Pi(i)) - temp)
        temp = 0
    Next

    x(n) = (y(n) / A(n, n))

    For i = n To 0 Step -1
        For j = i + 1 To n Step 1
            temp = temp + (A(i, j) * x(j))
        Next

        x(i) = ((y(i) - temp) / A(i, i))
        temp = 0
    Next
End Sub

Shared Sub Main()
    Dim gaussElim As New GaussElimInVB

```

```

Dim i As Integer
Dim j As Integer

Randomize()

For i = 0 To n Step 1
    For j = 0 To n Step 1
        gaussElim.A(i, j) = Int((n - 0 + 1) * Rnd()) + 0
    Next
Next

For i = 0 To n Step 1
    gaussElim.b(i) = Int((n - 0 + 1) * Rnd()) + 0
Next

gaussElim.LUP_Decomposition()
gaussElim.LUP_Solve()

Console.WriteLine("Solutions for {0} unknowns:", gaussElim.n)

For i = 0 To n Step 1
    Console.WriteLine(gaussElim.x(i))
Next
End Sub
End Class

```

A.8 Minimum and Maximum

A.8.1 MinMaxInC.cpp

```

#include <stdio.h>
#include <time.h>

int Minimum (int *A)
{
    int min = A[0];

    for (int i = 1; i < 100; i++)
        if (min > A[i])
            min = A[i];

    return min;
}

int Maximum (int *A)

```



```

{
    int max = A[0];

    for (int i = 1; i < 100; i++)
        if (max < A[i])
            max = A[i];

    return max;
}

int main (void)
{
    int A[100];
    int min = 0;
    int max = 0;

    srand((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < 100; i++)
        A[i] = (rand() % 100);

    min = Minimum(A);
    max = Maximum(A);

    printf("The minimum value in the array is %d\n", min);
    printf("The maximum value in the array is %d\n", max);

    return 0;
}

```

A.8.2 MinMaxInCPP.cpp

```

#include <iostream>
#include "MinMaxClass.h"

using namespace std;

int main (void)
{
    int min = 0;
    int max = 0;

    MinMaxClass *mmc = new MinMaxClass();

    min = mmc -> DoMinimum();
    max = mmc -> DoMaximum();
}

```

```

    cout << "The minimum value in the array is " << min << endl;
    cout << "The maximum value in the array is " << max << endl;

    delete mmc;

    return 0;
}

```

A.8.3 MinMaxClass.h

```

class MinMaxClass
{
    public:
        MinMaxClass (void);

        int DoMinimum (void);
        int DoMaximum (void);

    private:
        int A[100];
};

```

A.8.4 MinMaxClass.cpp

```

#include <stdlib.h>
#include <time.h>
#include "MinMaxClass.h"

MinMaxClass :: MinMaxClass (void)
{
    srand((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < 100; i++)
        A[i] = (rand() % 100);
}

int MinMaxClass :: DoMinimum (void)
{
    int min = A[0];

    for (int i = 1; i < 100; i++)
        if (min > A[i])
            min = A[i];

    return min;
}

```

```

}

int MinMaxClass :: DoMaximum (void)
{
    int max = A[0];

    for (int i = 1; i < 100; i++)
        if (max < A[i])
            max = A[i];

    return max;
}

```

A.8.5 MinMaxInCS.cs

```

using System;

namespace MinMaxClass
{
    public class MinMaxClass
    {
        private int[] A = new int[100];
        private int min = 0;
        private int max = 0;

        public MinMaxClass()
        {
            Random r = new Random();

            for (int i = 0; i < 100; i++)
                A[i] = r.Next() % 100;
        }

        private void DoMinimum()
        {
            min = A[0];

            for (int i = 1; i < 100; i++)
                if (min > A[i])
                    min = A[i];
        }

        private void DoMaximum()
        {
            max = A[0];

```

```

        for (int i = 1; i < 100; i++)
            if (max < A[i])
                max = A[i];
    }

    public static void Main()
    {
        MinMaxClass mmc = new MinMaxClass();

        mmc.DoMinimum();
        mmc.DoMaximum();

        Console.WriteLine("The minimum value in the array is {0}",
mmc.min);
        Console.WriteLine("The maximum value in the array is {0}",
mmc.max);
    }
}

```

A.8.6 MinMaxInJAVA.jsl

```

import java.util.*;

public class MinMaxClass
{
    private int[] A = new int[100];
    private int min = 0;
    private int max = 0;

    public MinMaxClass()
    {
        Random r = new Random();

        for (int i = 0; i < 100; i++)
            A[i] = r.nextInt() % 100;
    }

    private void DoMinimum()
    {
        min = A[0];

        for (int i = 1; i < 100; i++)
            if (min > A[i])
                min = A[i];
    }
}

```

```

private void DoMaximum()
{
    max = A[0];

    for (int i = 1; i < 100; i++)
        if (max < A[i])
            max = A[i];
}

public static void main (String args[])
{
    MinMaxClass mmc = new MinMaxClass();

    mmc.DoMinimum();
    mmc.DoMaximum();

    System.Console.WriteLine("The minimum value in the array is " +
mmc.min);
    System.Console.WriteLine("The maximum value in the array is " +
mmc.max);
}
}

```

A.8.7 MinMaxInVB.vb

```

Imports System
Imports Microsoft.VisualBasic

```

```

Public Class MinMaxInVB
    Dim A(100) As Integer
    Dim min As Integer
    Dim max As Integer

```

```

    Sub DoMinimum()
        Dim i As Integer

```

```

        min = A(0)

```

```

        For i = 1 To 100 Step 1
            If min > A(i) Then
                min = A(i)
            End If

```

```

        Next
    End Sub

```

```

Sub DoMaximum()
    Dim i As Integer

    max = A(0)

    For i = 1 To 100 Step 1
        If max < A(i) Then
            max = A(i)
        End If
    Next
End Sub

Shared Sub Main()
    Dim mmc As New MinMaxInVB
    Dim i As Integer

    Randomize()

    For i = 0 To 100 Step 1
        mmc.A(i) = Int((100 - 0 + 1) * Rnd()) + 0
    Next

    mmc.DoMinimum()
    mmc.DoMaximum()

    Console.WriteLine("The minimum value in the array is {0}", mmc.min)
    Console.WriteLine("The maximum value in the array is {0}", mmc.max)
End Sub
End Class

```

A.9 Random Selection

A.9.1 RandomSelectInC.cpp

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int Partition (int *A, int p, int r)
{
    int x = A[r];
    int i = p - 1;
    int temp = 0;

    for (int j = p; j <= r - 1; j++)

```

```

    {
        if (A[j] <= x)
        {
            i = i + 1;

            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }

    temp = A[i + 1];
    A[i + 1] = A[r];
    A[r] = temp;

    return (i + 1);
}

int RandomizedPartition (int *A, int p, int r)
{
    int temp = 0;
    int i = (rand() % r) + p;

    temp = A[r];
    A[r] = A[i];
    A[i] = temp;

    return Partition(A, p, r);
}

int RandomizedSelect (int *A, int p, int r, int i)
{
    int q = 0;
    int k = 0;

    if (p == r)
        return A[p];

    q = RandomizedPartition(A, p, r);
    k = q - p + 1;

    if (i == k)
        return A[q];
    else if (i < k)
        return RandomizedSelect(A, p, q - 1, i);
    else

```

```

        return RandomizedSelect(A, q + 1, r, i - k);
    }

int main (void)
{
    int A[100];
    int n = 100;
    int x = 0;

    srand((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < n; i++)
        A[i] = (rand() % 100);

    x = RandomizedSelect(A, 0, 100, 1);

    printf("The ith smallest element in the array where i = 1 is %d", x);

    return 0;
}

```

A.9.2 RandomSelectInCPP.cpp

```

#include <iostream>
#include "RandomSelectClass.h"

using namespace std;

int main (void)
{
    int x = 0;

    RandomSelect *rc = new RandomSelect();

    x = rc -> RandomizedSelect(0, 99, 1);

    cout << "The ith smallest element in the array where i = 1 is " << x << endl;

    delete rc;

    return 0;
}

```

A.9.3 RandomSelectClass.h

```

class RandomSelect

```



```

{
    public:
        RandomSelect (void);

        int RandomizedSelect (int p, int r, int i);

    private:
        int RandomizedPartition (int p, int r);
        int Partition (int p, int r);

        int A[100];
};

```

A.9.4 RandomSelectClass.cpp

```

#include <stdlib.h>
#include <time.h>
#include "RandomSelectClass.h"

RandomSelect :: RandomSelect (void)
{
    srand((unsigned int) time((time_t *) NULL));

    for (int i = 0; i < 100; i++)
        A[i] = (rand() % 100);
}

int RandomSelect :: RandomizedSelect (int p, int r, int i)
{
    int q = 0;
    int k = 0;

    if (p == r)
        return A[p];

    q = RandomizedPartition(p, r);
    k = q - p + 1;

    if (i == k)
        return A[q];
    else if (i < k)
        return RandomizedSelect(p, q - 1, i);
    else
        return RandomizedSelect(q + 1, r, i - k);
}

```

```

int RandomSelect :: RandomizedPartition (int p, int r)
{
    int temp = 0;
    int i = (rand() % r) + p;

    temp = A[r];
    A[r] = A[i];
    A[i] = temp;

    return Partition(p, r);
}

```

```

int RandomSelect :: Partition (int p, int r)
{
    int x = A[r];
    int i = p - 1;
    int temp = 0;

    for (int j = p; j <= r - 1; j++)
    {
        if (A[j] <= x)
        {
            i = i + 1;

            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }

    temp = A[i + 1];
    A[i + 1] = A[r];
    A[r] = temp;

    return (i + 1);
}

```

A.9.5 RandomSelectInCS.cs

```

using System;

namespace RandomSelect
{
    public class RandomSelect
    {
        private int[] A = new int [100];
    }
}

```

```

public RandomSelect()
{
    Random r = new Random();

    for (int i = 0; i < 100; i++)
        A[i] = r.Next(0, 100);
}

private int RandomizedSelect (int p, int r, int i)
{
    int q = 0;
    int k = 0;

    if (p == r)
        return A[p];

    q = RandomizedPartition(p, r);
    k = q - p + 1;

    if (i == k)
        return A[q];
    else if (i < k)
        return RandomizedSelect(p, q - 1, i);
    else
        return RandomizedSelect(q + 1, r, i - k);
}

private int RandomizedPartition (int p, int r)
{
    Random x = new Random();

    int temp = 0;
    int i = (x.Next(0, p)) + r;

    temp = A[r];
    A[r] = A[i];
    A[i] = temp;

    return Partition(p, r);
}

private int Partition (int p, int r)
{
    int x = A[r];
    int i = p - 1;

```

```

        int temp = 0;

        for (int j = p; j <= r - 1; j++)
        {
            if (A[j] <= x)
            {
                i = i + 1;

                temp = A[i];
                A[i] = A[j];
                A[j] = temp;
            }
        }

        temp = A[i + 1];
        A[i + 1] = A[r];
        A[r] = temp;

        return (i + 1);
    }

    public static void Main ()
    {
        int x = 0;

        RandomSelect rm = new RandomSelect();

        x = rm.RandomizedSelect(0, 99, 1);

        Console.WriteLine("The ith smallest element in the array where i
= 1 is " + x);
    }
}

```

A.9.6 RandomSelectInJava.jsl

```

import java.util.*;

public class RandomSelect
{
    private int[] A = new int [100];

    public RandomSelect()
    {
        Random r = new Random();
    }
}

```

```

        for (int i = 0; i < 100; i++)
            A[i] = r.nextInt() % 100;
    }

    private int RandomizedSelect (int p, int r, int i)
    {
        int q = 0;
        int k = 0;

        if (p == r)
            return A[p];

        q = RandomizedPartition(p, r);
        k = q - p + 1;

        if (i == k)
            return A[q];
        else if (i < k)
            return RandomizedSelect(p, q - 1, i);
        else
            return RandomizedSelect(q + 1, r, i - k);
    }

    private int RandomizedPartition (int p, int r)
    {
        Random x = new Random();

        int temp = 0;
        int i = x.nextInt() % r;

        temp = A[r];
        A[r] = A[Math.abs(i)];
        A[Math.abs(i)] = temp;

        return Partition(p, r);
    }

    private int Partition (int p, int r)
    {
        int x = A[r];
        int i = p - 1;
        int temp = 0;

        for (int j = p; j <= r - 1; j++)
        {

```

```

        if (A[j] <= x)
        {
            i = i + 1;

            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }

    temp = A[i + 1];
    A[i + 1] = A[r];
    A[r] = temp;

    return (i + 1);
}

public static void main (String args[])
{
    int x = 0;

    RandomSelect rm = new RandomSelect();

    x = rm.RandomizedSelect(0, 99, 1);

    System.Console.WriteLine("The ith smallest element in the array where i
= 1 is " + x);
}
}

```

A.9.7 RandomSelectInVB.vb

Imports System

Imports Microsoft.VisualBasic

Public Class RandomSelectInVB

Function Partition(ByRef A() As Integer, ByRef p As Integer, ByRef r As Integer)

Dim x = A(r)

Dim i = p - 1

Dim j As Integer

Dim temp = 0

For j = p To r - 1 Step 1

 If A(j) <= x Then

 i = i + 1

```

        temp = A(i)
        A(i) = A(j)
        A(j) = temp
    End If
Next

```

```

    temp = A(i + 1)
    A(i + 1) = A(r)
    A(r) = temp

```

```

    Return (i + 1)
End Function

```

```

Function RandomizedPartition(ByRef A() As Integer, ByRef p As Integer, ByRef r As
Integer)

```

```

    Dim i = 0
    Dim temp = 0

```

```

    Randomize()

```

```

    i = Int((r - p + 1) * Rnd()) + 0

```

```

    temp = A(r)
    A(r) = A(i)
    A(i) = temp

```

```

    Return Partition(A, p, r)
End Function

```

```

Function RandomizedSelect(ByRef A() As Integer, ByRef p As Integer, ByRef r As
Integer, ByRef i As Integer)

```

```

    Dim q = 0
    Dim k = 0

```

```

    If p = r Then
        Return A(p)
    End If

```

```

    q = RandomizedPartition(A, p, r)
    k = q - p + 1

```

```

    If i = k Then
        Return A(q)
    ElseIf i < k Then
        Return RandomizedSelect(A, p, q - 1, i)
    Else

```

```

        Return RandomizedSelect(A, q + 1, r, i - k)
    End If
End Function

Shared Sub Main()
    Dim i
    Dim x
    Dim n = 100
    Dim A(100) As Integer
    Dim rc As New RandomSelectInVB

    Randomize()

    For i = 0 To 99 Step 1
        A(i) = Int((100 - 0 + 1) * Rnd()) + 0
    Next

    x = rc.RandomizedSelect(A, 0, 100, 1)

    Console.WriteLine("The ith smallest element in the array where i = 1 is {0}", x)
End Sub
End Class

```


APPENDIX B

RAW MEASUREMENT DATA

B.1 Static Measurements

Linear Search	C	C++	C#	Java	Visual BASIC
Physical Lines of Code	46	80	48	44	34
Effective Lines of Code	29	29	26	26	27
Code Statements	20	33	18	18	25
McCabe's V(g)	8	11	9	7	10
Compiler Directives	3	5	1	1	2
Bubblesort	C	C++	C#	Java	Visual BASIC
Physical Lines of Code	53	81	60	53	44
Effective Lines of Code	29	36	32	29	35
Code Statements	22	33	25	24	26
McCabe's V(g)	8	10	9	9	10
Compiler Directives	3	4	1	1	2
Quicksort	C	C++	C#	Java	Visual BASIC
Physical Lines of Code	66	97	60	67	60
Effective Lines of Code	36	53	36	36	46
Code Statements	28	39	30	30	44
McCabe's V(g)	8	10	9	9	11
Compiler Directives	3	5	1	1	2
Naïve String Matching	C	C++	C#	Java	Visual BASIC
Physical Lines of Code	25	56	44	43	20
Effective Lines of Code	14	32	23	23	17
Code Statements	8	21	17	18	15
McCabe's V(g)	4	5	4	5	7
Compiler Directives	3	4	1	1	2
KMP String Matching	C	C++	C#	Java	Visual BASIC
Physical Lines of Code	60	91	72	69	60
Effective Lines of Code	33	50	38	37	47

Code Statements	23	35	27	27	35
McCabe's V(g)	12	14	11	11	17
Compiler Directives	2	4	1	1	2
Polynomial Addition	C	C++	C#	Java	Visual BASIC
Physical Lines of Code	28	57	44	42	28
Effective Lines of Code	17	33	22	21	22
Code Statements	13	22	16	16	20
McCabe's V(g)	4	6	7	7	8
Compiler Directives	3	5	1	1	2
Gaussian Elimination	C	C++	C#	Java	Visual BASIC
Physical Lines of Code	110	133	118	115	115
Effective Lines of Code	62	76	64	63	91
Code Statements	53	61	55	55	80
McCabe's V(g)	19	21	21	21	24
Compiler Directives	4	6	1	1	2
Minimum and Maximum	C	C++	C#	Java	Visual BASIC
Physical Lines of Code	44	66	48	45	49
Effective Lines of Code	26	39	26	25	37
Code Statements	19	26	18	18	26
McCabe's V(g)	8	9	9	11	13
Compiler Directives	2	5	1	1	2
Random Selection	C	C++	C#	Java	Visual BASIC
Physical Lines of Code	77	99	86	83	80
Effective Lines of Code	47	59	48	47	61
Code Statements	35	42	36	36	55
McCabe's V(g)	10	11	11	13	15
Compiler Directives	3	5	1	1	2

Table B.1 Static Raw Measurements.

B.2 Dynamic Measurements

Linear Search	C	C++	C#	Java	Visual BASIC
Size in Memory (Bytes)	8638	24314	21022	181358	34513
Total Objects Created	40	274	185	3114	585
Execution Time Max (Seconds)	2.734	3.001	2.281	3.016	2.828
Execution Time Min (Seconds)	1.813	2.172	1.765	2.626	1.968
Execution Time Average (Seconds)	2.0563	2.4457	1.9376	2.825	2.5294

Total Routines	433	1110	198	633	430
Routines Executed	108	372	108	383	248
Total Routine Calls	1034	7674	394	21540	7855

Bubblesort	C	C++	C#	Java	Visual BASIC
Size in Memory (Bytes)	8642	24298	22902	202586	33494
Total Objects Created	41	274	275	4176	438423
Execution Time Max (Seconds)	2.531	3.875	2.296	3.469	3.578
Execution Time Min (Seconds)	1.765	2.375	1.719	2.656	2.782
Execution Time Average (Seconds)	1.9703	2.8734	1.8858	3.0439	3.1626
Total Routines	368	1109	201	665	435
Routines Executed	96	370	111	405	250
Total Routine Calls	3208	23275	3495	28762	176229

Quicksort	C	C++	C#	Java	Visual BASIC
Size in Memory (Bytes)	8684	24290	22908	202408	3821
Total Objects Created	42	274	276	4174	80725
Execution Time Max (Seconds)	2.406	3.172	2.484	3.453	3.297
Execution Time Min (Seconds)	1.844	2.421	1.782	2.625	1.937
Execution Time Average (Seconds)	1.9671	2.6125	1.9521	3.0018	2.672
Total Routines	369	1110	202	666	434
Routines Executed	96	370	111	407	249
Total Routine Calls	3294	23603	3547	28973	20710

Naïve String Matching	C	C++	C#	Java	Visual BASIC
Size in Memory (Bytes)	8642	24298	28458	187752	31443
Total Objects Created	41	274	460	3370	735
Execution Time Max (Seconds)	1.11	1.344	1.078	1.703	3.095
Execution Time Min (Seconds)	0.813	1.031	0.844	1.344	1.781
Execution Time Average (Seconds)	0.986	1.2344	0.9703	1.3936	2.2292
Total Routines	362	1101	214	629	225
Routines Executed	91	365	123	380	125
Total Routine Calls	970	8496	1035	22199	5186

KMP String Matching	C	C++	C#	Java	Visual BASIC
Size in Memory (Bytes)	8676	24295	22954	181934	34437
Total Objects Created	42	275	181	3111	1278
Execution Time Max (Seconds)	2.36	2.25	1.125	2.578	1.172
Execution Time Min (Seconds)	1.734	1.873	0.828	1.344	0.922
Execution Time Average (Seconds)	1.9156	2.052	0.9767	1.5685	1.0532

Total Routines	364	1110	191	624	205
Routines Executed	92	360	105	379	106
Total Routine Calls	468	8421	206	21180	7975

Polynomial Addition	C	C++	C#	Java	Visual BASIC
Size in Memory (Bytes)	8668	24324	23818	203160	49330
Total Objects Created	42	275	276	4163	1207
Execution Time Max (Seconds)	1.125	1.391	1.125	1.719	1.25
Execution Time Min (Seconds)	0.859	1.078	0.844	1.437	0.984
Execution Time Average (Seconds)	1.039	1.2781	0.9564	1.5781	1.1185
Total Routines	367	1108	202	666	388
Routines Executed	95	372	112	408	226
Total Routine Calls	3622	24659	3686	29208	8027

Gaussian Elimination	C	C++	C#	Java	Visual BASIC
Size in Memory (Bytes)	8768	24374	65582	331356	81930
Total Objects Created	44	276	282	5604	934
Execution Time Max (Seconds)	1.343	1.75	1.141	1.75	1.282
Execution Time Min (Seconds)	0.984	1.36	0.922	1.422	1.75
Execution Time Average (Seconds)	1.1639	1.6002	1.07167	1.6001	1.2297
Total Routines	433	1116	209	706	405
Routines Executed	122	397	117	435	239
Total Routine Calls	50808	81214	23287	75835	58690

Minimum and Maximum	C	C++	C#	Java	Visual BASIC
Size in Memory (Bytes)	8660	24550	21636	152270	31282
Total Objects Created	42	276	193	2607	410
Execution Time Max (Seconds)	1.11	1.324	1	1.578	1.234
Execution Time Min (Seconds)	0.844	1.093	0.719	1.25	0.985
Execution Time Average (Seconds)	0.9422	1.1579	0.8999	1.3375	1.1358
Total Routines	369	1111	202	535	382
Routines Executed	95	390	111	328	223
Total Routine Calls	776	23289	429	7087	5936

Random Selection	C	C++	C#	Java	Visual BASIC
Size in Memory (Bytes)	8993	24706	22794	205664	3985
Total Objects Created	45	275	279	4283	81725
Execution Time Max (Seconds)	1.45	1.576	1.231	1.593	3.569
Execution Time Min (Seconds)	1.21	1.329	1.057	1.254	2.587
Execution Time Average (Seconds)	1.3078	1.4587	1.1125	1.3578	3.0489

Total Routines	380	1266	205	591	459
Routines Executed	100	522	119	370	291
Total Routine Calls	768	10139	487	23590	21815

Table B.2 Dynamic Raw Measurements.

B.3 Metadata Measurements

Linear Search	C	C++	C#	Java	Visual BASIC
Type References	10	20	5	15	14
Type Definitions	3	84	2	2	2
Fields	2	82	4	4	0
Methods	5	159	3	5	2
Member References	7	17	6	18	16
Assembly References	2	2	1	3	2
Stand Alone Signatures	4	129	2	2	1
Bubblesort	C	C++	C#	Java	Visual BASIC
Type References	11	20	4	14	11
Type Definitions	5	85	2	2	2
Fields	3	81	4	4	0
Methods	7	158	3	5	2
Member References	6	17	5	15	14
Assembly References	2	2	1	3	2
Stand Alone Signatures	6	131	3	3	1
Quicksort	C	C++	C#	Java	Visual BASIC
Type References	11	20	4	14	11
Type Definitions	5	86	2	2	2
Fields	3	81	3	3	0
Methods	8	159	4	6	4
Member References	6	17	5	15	15
Assembly References	2	2	1	3	2
Stand Alone Signatures	6	130	4	4	3
Naïve String Matching	C	C++	C#	Java	Visual BASIC
Type References	10	20	6	16	9
Type Definitions	6	87	2	2	2
Fields	3	82	4	4	0
Methods	6	156	3	5	2
Member References	6	17	7	20	11
Assembly References	2	2	1	3	2
Stand Alone Signatures	6	131	2	2	1

KMP String Matching	C	C++	C#	Java	Visual BASIC
Type References	10	20	5	16	9
Type Definitions	6	87	2	2	2
Fields	3	82	4	4	0
Methods	7	156	4	5	2
Member References	6	17	6	20	11
Assembly References	2	2	1	3	2
Stand Alone Signatures	8	134	2	2	1
Polynomial Addition	C	C++	C#	Java	Visual BASIC
Type References	11	20	4	14	7
Type Definitions	4	84	2	2	2
Fields	2	80	3	3	0
Methods	6	157	4	6	2
Member References	6	17	5	15	8
Assembly References	2	2	1	3	2
Stand Alone Signatures	3	130	3	3	1
Gaussian Elimination	C	C++	C#	Java	Visual BASIC
Type References	12	21	7	21	10
Type Definitions	7	87	2	2	2
Fields	7	82	5	5	5
Methods	8	160	5	8	4
Member References	8	18	10	26	14
Assembly References	2	2	1	4	2
Stand Alone Signatures	7	135	5	6	3
Minimum and Maximum	C	C++	C#	Java	Visual BASIC
Type References	11	20	4	14	7
Type Definitions	3	84	2	2	2
Fields	2	81	3	3	3
Methods	8	157	4	6	4
Member References	6	17	4	17	7
Assembly References	2	2	1	3	2
Stand Alone Signatures	5	128	3	3	3
Random Selection	C	C++	C#	Java	Visual BASIC
Type References	11	20	5	15	12
Type Definitions	3	84	2	2	2
Fields	1	80	1	1	0
Methods	9	158	5	7	5
Member References	6	17	5	18	14
Assembly References	2	2	1	3	2
Stand Alone Signatures	7	130	5	4	4

B.3 Metadata Raw Measurements.

APPENDIX C

PCA-RCM TOOL OUTPUT

C.1 Static RCM Results

C.1.1 Linear Search

Module	DOMAIN1	RCM
LSearchInC	0.02059	50.20592
LSearchInCPP	1.78576	67.85758
LSearchInCS	-0.67606	43.23943
LSearchInJava	-1.01667	39.83332
LSearchInVB	-0.11363	48.86375

C.1.2 Bubblesort

Module	DOMAIN1	DOMAIN2	RCM
BubbleSortInC	-0.89617	1.26372	45.86751
BubbleSortInCPP	1.76799	0.64755	68.82833
BubbleSortInCS	-0.31316	-0.29715	46.04476
BubbleSortInJava	-0.76727	-0.23703	41.97851
BubbleSortInVB	0.20861	-1.37709	47.28089

C.1.3 Quicksort

Module	DOMAIN1	DOMAIN2	RCM
QuickSortInC	-0.72580	0.92770	47.38275
QuickSortInCPP	1.56988	0.95232	68.25952
QuickSortInCS	-0.82095	-0.26198	41.45637
QuickSortInJava	-0.71540	-0.05921	43.27473
QuickSortInVB	0.69227	-1.55884	49.62663

C.1.4 Naïve String Matching

Module	DOMAIN1	RCM
NaiveMatchInC	-1.18986	38.10136
NaiveMatchInCPP	1.50454	65.04539
NaiveMatchInCS	0.27967	52.79666
NaiveMatchInJava	0.27592	52.75918
NaiveMatchInVB	-0.87026	41.29740

C.1.5 KMP String Matching

Module	DOMAIN1	DOMAIN2	RCM
KMPMatchInC	-0.93832	-0.08991	40.91790
KMPMatchInCPP	1.50481	1.08511	67.93742
KMPMatchInCS	-0.62999	0.40321	45.56369
KMPMatchInJava	-0.71650	0.24888	44.20303
KMPMatchInVB	0.78001	-1.64729	51.37796

C.1.6 Polynomial Addition

Module	DOMAIN1	DOMAIN2	RCM
PolyAddInC	-1.06713	-1.49415	33.52708
PolyAddInCPP	1.80359	-0.63226	62.89410
PolyAddInCS	-0.26580	0.55495	50.28573
PolyAddInJava	-0.37464	0.55885	49.34546
PolyAddInVB	-0.09603	1.01261	53.94764

C.1.7 Gaussian Elimination

Module	DOMAIN1	DOMAIN2	RCM
GaussElimInC	-1.04198	-0.03810	40.90937
GaussElimInCPP	0.22681	1.81079	61.37518
GaussElimInCS	-0.43493	-0.50142	43.67452
GaussElimInJava	-0.49140	-0.69212	42.19850
GaussElimInVB	1.74151	-0.57915	61.84244

C.1.8 Minimum and Maximum

Module	DOMAIN1	DOMAIN2	RCM
PolyAddInC	-1.06713	-1.49415	33.52708
PolyAddInCPP	1.80359	-0.63226	62.89410
PolyAddInCS	-0.26580	0.55495	50.28573
PolyAddInJava	-0.37464	0.55885	49.34546
PolyAddInVB	-0.09603	1.01261	53.94764

C.1.9 Random Selection

Module	DOMAIN1	DOMAIN2	RCM
RandomSelectInC	-0.94195	0.15369	43.54998
RandomSelectInCPP	0.75028	1.68540	66.32114
RandomSelectInCS	-0.74285	-0.12394	43.39678
RandomSelectInJava	-0.56052	-0.65792	41.52603
RandomSelectInVB	1.49504	-1.05723	55.20607

C.2 Dyanmic PCA Results

C.2.1 Linear Search

Module	DOMAIN1	DOMAIN2	RCM
LSearchInC	-0.81650	0.10585	42.37891
LSearchInCPP	0.38930	1.63342	57.94902
LSearchInCS	-1.15573	-0.84411	36.66545
LSearchInJava	1.62794	-0.93566	63.33794
LSearchInVB	-0.04501	0.04049	49.66868

C.2.2 Bubblesort

Module	DOMAIN1	DOMAIN2	RCM
BubbleSortInC	-1.11255	0.04126	40.11982
BubbleSortInCPP	0.63694	-0.88373	51.97794
BubbleSortInCS	-1.26334	0.12175	39.10102
BubbleSortInJava	0.81756	-0.94899	53.33151
BubbleSortInVB	0.92139	1.66971	65.46972

C.2.3 Quicksort

Module	DOMAIN1	DOMAIN2	RCM
QuicksortInC	-1.08609	-0.46104	38.28461
QuicksortInCPP	0.74679	-0.47627	55.84521
QuicksortInCS	-1.11409	-0.35265	38.31768
QuicksortInJava	1.34132	-0.55524	61.33506
QuicksortInVB	0.11208	1.84519	56.21744

C.2.4 Naïve String Matching

Module	DOMAIN1	DOMAIN2	DOMAIN3	RCM
NaiveMatchInC	-1.05511	-0.23480	-0.38044	39.52060
NaiveMatchInCPP	0.02189	-0.79517	1.62310	49.49306
NaiveMatchInCS	-0.90031	-0.23748	-0.78964	39.71005
NaiveMatchInJava	1.64170	-0.60304	-0.75242	57.39745
NaiveMatchInVB	0.29183	1.87049	0.29941	63.87885

C.2.5 KMP String Matching

Module	DOMAIN1	DOMAIN2	RCM
KMPMatchInC	-0.17511	-1.09910	42.66456
KMPMatchInCPP	0.88759	-1.11421	51.57864
KMPMatchInCS	-1.21317	0.26460	41.14096
KMPMatchInJava	1.32868	1.26631	67.98971
KMPMatchInVB	-0.82799	0.68239	46.62614

C.2.6 Polynomial Addition

Module	DOMAIN1	DOMAIN2	RCM
PolyAddInC	-0.88937	-0.11951	41.05787
PolyAddInCPP	0.42288	1.76815	57.94366
PolyAddInCS	-0.93629	-0.49619	39.78714
PolyAddInJava	1.69861	-0.83207	64.79111
PolyAddInVB	-0.29583	-0.32037	46.42020

C.2.7 Gaussian Elimination

Module	DOMAIN1	DOMAIN2	RCM
GausseElimInC	-0.77269	-0.37148	41.52299
GausseElimInCPP	0.89778	-1.51512	54.13345
GausseElimInCS	-1.26054	0.46903	39.32732
GausseElimInJava	1.32563	1.27590	66.41823
GausseElimInVB	-0.19018	0.14167	48.59801

C.2.8 Minimum and Maximum

Module	DOMAIN1	DOMAIN2	RCM
MinMaxInC	-0.90963	-0.11025	41.13435
MinMaxInCPP	0.74145	1.70408	63.15437
MinMaxInCS	-1.14799	-0.31541	38.16387
MinMaxInJava	1.44206	-1.20336	58.99384
MinMaxInVB	-0.12588	-0.07505	48.55355

C.2.9 Random Selection

Module	DOMAIN1	DOMAIN2	DOMAIN3	RCM
RandomSelectInC	-0.64571	-0.89580	-0.16622	39.96725
RandomSelectInCPP	-0.21674	1.16667	-1.41276	49.85313
RandomSelectInCS	-0.82612	-0.93131	0.14039	39.06773
RandomSelectInJava	-0.19579	1.12466	1.44623	57.26040
RandomSelectInVB	1.88436	-0.46421	-0.00764	63.85149

C.3 Metadata PCA Results

C.3.1 Linear Search

Module	DOMAIN1	DOMAIN2	RCM
LSearchInC	-0.60613	-0.24699	43.45476
LSearchInCPP	1.87152	-0.52873	65.30878
LSearchInCS	-0.89357	-1.26426	36.93502
LSearchInJava	-0.08477	1.46179	54.76302
LSearchInVB	-0.28705	0.57819	49.53842

C.3.2 Bubblesort

Module	DOMAIN1	DOMAIN2	RCM
BubbleSortInC	-0.52032	-0.12501	44.67509
BubbleSortInCPP	1.88772	-0.47684	66.25950
BubbleSortInCS	-0.88911	-1.31706	37.27259
BubbleSortInJava	-0.11808	1.46743	53.71004
BubbleSortInVB	-0.36021	0.45149	48.08278

C.3.3 Quicksort

Module	DOMAIN1	DOMAIN2	RCM
QuickSortInC	-0.52171	-0.14108	44.61035
QuickSortInCPP	1.88834	-0.47243	66.24503
QuickSortInCS	-0.89203	-1.32571	37.18531
QuickSortInJava	-0.13665	1.44702	53.51360
QuickSortInVB	-0.33795	0.49220	48.44570

C.3.4 Naïve String Matching

Module	DOMAIN1	DOMAIN2	RCM
NaiveMatchInC	-0.55684	-0.23541	43.95671
NaiveMatchInCPP	1.87634	-0.51242	65.61470
NaiveMatchInCS	-0.79522	-1.06681	38.71243
NaiveMatchInJava	0.02280	1.71051	56.43357
NaiveMatchInVB	-0.54708	0.10413	45.28259

C.3.5 KMP String Matching

Module	DOMAIN1	DOMAIN2	RCM
KMPMatchInC	-0.53224	-0.19415	44.33732
KMPMatchInCPP	1.87211	-0.52712	65.41536
KMPMatchInCS	-0.83021	-1.10450	38.18419
KMPMatchInJava	0.02340	1.68885	56.50146
KMPMatchInVB	-0.53306	0.13693	45.56167

C.3.6 Polynomial Addition

Module	DOMAIN1	DOMAIN2	RCM
PolyAddInC	-0.47771	0.03275	45.54353
PolyAddInCPP	1.89096	-0.46411	66.63264
PolyAddInCS	-0.80086	-1.21421	38.70535
PolyAddInJava	-0.03756	1.61332	54.50248
PolyAddInVB	-0.57483	0.03225	44.61600

C.3.7 Gaussian Elimination

Module	DOMAIN1	DOMAIN2	RCM
GausseElimInC	-0.54357	-0.37995	43.41660
GausseElimInCPP	1.87924	-0.49260	64.27769
GausseElimInCS	-0.75503	-0.86295	39.28147
GausseElimInJava	0.00492	1.83561	58.66395
GausseElimInVB	-0.58555	-0.10010	44.36028

C.3.8 Minimum and Maximum

Module	DOMAIN1	DOMAIN2	RCM
MinMaxInC	-0.46142	0.03739	45.75710
MinMaxInCPP	1.88344	-0.49220	66.21613
MinMaxInCS	-0.82373	-1.16836	38.41569
MinMaxInJava	-0.02092	1.65162	55.16425
MinMaxInVB	-0.57736	-0.02845	44.44683

C.3.9 Random Selection

Module	DOMAIN1	DOMAIN2	RCM
RandomSelectInC	-0.55178	-0.17728	44.21566
RandomSelectInCPP	1.87559	-0.51840	65.57495
RandomSelectInCS	-0.89751	-1.24688	37.09667
RandomSelectInJava	-0.06480	1.54239	55.01995
RandomSelectInVB	-0.36150	0.40017	48.09276

C.4 Overall PCA Results

C.4.1 Linear Search

Module	DOMAIN1	DOMAIN2	RCM
LSearchInC	-0.59458	-0.53907	42.00631
LSearchInCPP	1.74330	-0.83196	58.89297
LSearchInCS	-1.18779	-0.63916	36.67004
LSearchInJava	0.21542	1.85091	62.87749
LSearchInVB	-0.17636	0.15927	49.55320

C.4.2 Bubblesort

Module	DOMAIN1	DOMAIN2	DOMAIN3	RCM
BubbleSortInC	-0.87967	-0.69416	-0.37961	38.38640
BubbleSortInCPP	1.76322	-0.88300	0.02818	61.26959
BubbleSortInCS	-0.99675	-0.80783	0.47012	39.11370
BubbleSortInJava	0.02515	1.09555	-1.51729	51.08201
BubbleSortInVB	0.08805	1.28944	1.39859	60.14830

C.4.3 Quicksort

Module	DOMAIN1	DOMAIN2	DOMAIN3	RCM
QuickSortInC	-0.86671	-0.62944	-0.59915	38.23321
QuickSortInCPP	1.63222	-1.06518	-0.16741	59.36904
QuickSortInCS	-1.16335	-0.52243	-0.00474	37.73008
QuickSortInJava	0.34271	1.61022	-1.02251	56.75872
QuickSortInVB	0.05512	0.60683	1.79383	57.90894

C.4.4 Naïve String Matching

Module	DOMAIN1	DOMAIN2	DOMAIN3	RCM
NaiveMatchInC	-0.78996	-0.86739	-0.29299	38.28832
NaiveMatchInCPP	1.74898	-0.67623	0.59625	62.22638
NaiveMatchInCS	-0.59153	-0.78039	-0.76851	38.85350
NaiveMatchInJava	0.43571	1.53566	-1.09385	58.09994
NaiveMatchInVB	-0.80320	0.78835	1.55910	52.53186

C.4.5 KMP String Matching

Module	DOMAIN1	DOMAIN2	DOMAIN3	RCM
KMPMatchInC	-0.46101	-0.17270	-1.54207	41.20859
KMPMatchInCPP	1.82679	-0.70989	0.00793	62.06487
KMPMatchInCS	-0.99566	-0.66914	-0.07659	38.27136
KMPMatchInJava	0.18254	1.91586	0.12043	60.85649
KMPMatchInVB	-0.55266	-0.36413	1.49030	47.59870

C.4.6 Polynomial Addition

Module	DOMAIN1	DOMAIN2	RCM
PolyAddInC	-0.86889	-0.36440	40.68640
PolyAddInCPP	1.59069	-1.16564	57.48677
PolyAddInCS	-0.95083	-0.39790	39.81260
PolyAddInJava	0.69448	1.77302	65.17641
PolyAddInVB	-0.46546	0.15492	46.83782

C.4.7 Gaussian Elimination

Module	DOMAIN1	DOMAIN2	DOMAIN3	RCM
GaussElimInC	-0.70067	-0.40206	-0.92442	39.39356
GaussElimInCPP	1.56922	-1.16484	-0.01351	57.23655
GaussElimInCS	-1.14455	-0.38403	-0.39659	37.53157
GaussElimInJava	0.67423	1.75630	-0.51192	62.27368
GaussElimInVB	-0.39822	0.19463	1.84644	53.56464

C.4.8 Minimum and Maximum

Module	DOMAIN1	DOMAIN2	RCM
MinMaxInC	-0.71299	-0.43225	41.66479
MinMaxInCPP	1.79735	-0.75397	61.85344
MinMaxInCS	-1.04166	-0.71784	37.39361
MinMaxInJava	0.16150	1.88139	60.74776
MinMaxInVB	-0.20420	0.02267	48.34039

C.4.9 Random Selection

Module	DOMAIN1	DOMAIN2	DOMAIN3	RCM
RandomSelectInC	-0.69703	-0.66557	-0.34606	39.78897
RandomSelectInCPP	1.86088	-0.54339	-0.33777	59.31335
RandomSelectInCS	-0.94027	-0.75333	-0.65767	36.52673
RandomSelectInJava	-0.14438	0.08665	1.91475	55.27323
RandomSelectInVB	-0.07921	1.87564	-0.57326	59.09772

REFERENCES

- AutomatedQA, Corp. (1996). AQtime 4 – Automated Profiling and Debugging. <http://www.automatedqa.com/products/aqtime/index.asp> (2004, November 18).
- Ballintjn, M. R. & Ten Cate, C. (1997). Sex Differences in the Vocalizations and Syrinx of the Collared Dove (*Streptopelia Decaocto*). *The Auk* 114 (1), 22-39.
- Bates, B. (2004). C# as a First Language: A Comparison with C++. *Journal of Computing Sciences in Colleges*, 19 (3).
- Bergin, J. (1996). Java as a Better C++. *ACM SIGPLAN Notices*, 31 (11).
- Brosgol, B. M. (1998). A Comparison of Ada and Java as a Foundation Teaching Language. *ACM SIGAda Ada Letters*, 18 (5), 12-38.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to Algorithms* (2nd ed.). Massachusetts: MIT Press.
- Dewar, E. W. (2003). Multivariate Analysis of Mammalian Communities: Membership and Species Lineage Ranges in the Tertiary of North America. *Journal of Vertebrate Paleontology*, 23 (3), 45A-46A.
- Ebert, C. (1995). Tracing Complexity through the Software Process. *Proceedings of the 1st International Conference on Engineering of Complex Computer Systems*. 23-30.
- Fenton, N. (1994). Software Measurement: A Necessary Scientific Basis. *IEEE Transactions on Software Engineering*, 20 (3), 199-206.
- Fenton, N. E. & Neil, M. (1999). Software Metrics: Successes, Failures and New Directions. *The Journal of Systems and Software*, 47, 149-157.
- Feuer, A. R. & Gehani, N. H. (1982). A Comparison of the Programming Language C and Pascal. *ACM Computing Surveys*, 14 (1).
- Halstead, M. H. (1977). *Elements of Software Science*. New York: Elsevier.
- Harel, E. C. & McLean, E. R. (1985). The Effects of Using a Nonprocedural Computer Language on Programmer Productivity. *MIS Quarterly*, June, 109-120.

- Hoare, C. A. R. (1961). Algorithm 63 Partition. *Communications of the ACM*, 4 (7), 321.
- Hoare, C. A. R. (1961). Algorithm 64 Quicksort. *Communications of the ACM*, 4 (7), 321.
- Hoare, C. A. R. (1962). Quicksort. *Computer Journal*, 5 (1), 10-15.
- Jackson, J. E. (1991). *A User's Guide to Principal Components*. New Jersey: John Wiley & Sons, Inc.
- Joyce, R., Webb, R., & Peacock, J. L. (2003). Associations Between Perinatal Interventions and Hospital Stillbirth Rates and Neonatal Mortality. *Archives of Disease in Childhood Fetal and Neonatal Edition*. 89: F51-F56.
- Jung, H., Pivka, M., & Kim, J. (2000). An Empirical Study of Complexity Metrics in Cobol Programs. *The Journal of Systems and Software*, 51, 111-118.
- Knuth, D. E., Morris, J. H., & Pratt, V. R. (1977). Fast Pattern Matching in Strings. *SIAM Journal of Computing*, 6 (2), 323-350.
- M Squared Technologies (2005). RSM Downloads. *Resource Standard Software Source Code Metrics For C, C++, C#, Java and Visual BASIC*.
http://www.msquaredtechnologies.com/m2rsm_demo.php (2005, July 2).
- Martin, W. A. (1971). Sorting. *ACM Computing Surveys*, 3 (4), 147-174.
- McCabe, T. (1976). A Complexity Measure. *IEEE Transactions on Software Reliability SE-2*, 5: 308-320.
- Munson, J. C. (2003). *Software Engineering Measurement*. Florida: CRC Press LLC.
- Munson, J. C. & Khoshgoftaar, T. M. (1990). Applications of a Relative Complexity Metric for Software Project Management. *Journal of Systems and Software*, 12 (3), 283-291.
- Myers, B. A. (2002). Towards More Natural Functional Programming Languages. *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, October: 1.
- Navarro-Prieto, R. & Canas, J. J. (2001). Are Visual Programming Languages Better? The Role of Imagery in Program Comprehension. *International Journal of Human-Computer Studies*, 54 (6), 799-829.
- Perry, D. E. & Stieg, C. S. (1990). *Software Faults in Evolving a Large, Real-Time System: a Case Study*. New Jersey: AT&T Bell Laboratories.

- Petzold, C. (2002). *Programming Windows with C#*. Washington: Microsoft Press.
- Pratt, T. W. & Zelkowitz, M. V. (2001). *Programming Languages: Design and Implementation* (4th ed.). New Jersey: Prentice Hall.
- Prechelt, L. (2005). An Empirical Comparison of C, C++, Java, Perl, Python, REXX, and Tcl for a Search/String-Processing Program. *Technical Report 2000-5*.
- Sebesta, R. W. (1999). *Concepts of Programming Languages* (2nd ed.). Massachusetts: Addison Wesley Longman, Inc.
- Sedgewick, R. (1983). *Algorithms*. Massachusetts: Addison-Wesley Publishing Company, Inc.
- The Code Project (2002). A.NET Assembly Viewer.
<http://www.codeproject.com/dotnet/asmex.asp> (2005, July 2).
- Wemple, B. C. (2004). Water Quantity and Quality Dynamics in High-Elevation Watersheds: Developing a Scientific Approach to Understanding Ski Area Impacts in Vermont.
http://www.uvm.edu/envnr/vtwater/?Page=progress_reports/wemple_2004.html
 (2005, July 9).
- Weyuker, E. J. (1988). Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, 14 (9), 1357-1365.
- Wohlin, C. (1996). Revisiting Measurement of Software Complexity. *Proceeding of the 3rd Asia-Pacific Software Engineering Conference*. 35-43.
- Zeijl E., du Bois-Reymond, M., & te Poel, Y. (2001). Young Adolescents' Leisure Patterns. *Society and Leisure*, 24 (2, Fall 2001), 379-402.
- Zhao, M., Wohlin, C., Ohlsson, N., & Xie, M. (1998). A Comparison Between Software Design and Code Metrics for the Prediction of Software Fault Content. *Information and Software Technology*, 40. 801-809.

VITA

Jason Lawrence Michlowitz was born in Smithtown, New York, on December 8, 1979, the son of Ralph and Barbara Michlowitz. He completed his work as an undergraduate student at Texas State University–San Marcos in May of 2002 with a Bachelor's Degree in Computer Science. From June 2002 until February 2004, work was hard to find in the field of computer science, so Jason entered into Texas State University–San Marcos once again in the fall of 2002 to attend graduate school seeking a Master of Science in Software Engineering. Jason is currently living in Scottsdale, Arizona and is currently employed at Atronic Americas, LLC working as a game software engineer in the casino gaming industry.

Permanent Address: 16831 North 58th Street #217
Scottsdale, Arizona 85254

This thesis was typed by Jason Lawrence Michlowitz.