Topic: Code complexity and debugging methods

Research Question: **"To what extent does code complexity affect the time taken to debug using the debugging techniques: induction and deduction?"**

IB DP Extended Essay

Subject: Computer Science

Word count: 3,908

Session: 2018

Table of Contents

1. Introduction

1.1 The Research Question

The use of digital devices is ubiquitous in modern society and is being used for a variety of domains including communication, entertainment, education, and shopping. Technology that is very much a part of everyday life would not be possible without the field of Computer Science. Many of these technological devices are based on a system that requires user inputs, programming, and outputs. There are now, quite literally, thousands of programming languages *(Kaplan)* and a multitude of programming editors or integrated development environments (IDE). Thus, computer programmer not only have to learn multiple programming languages, they have to determine what editor features will augment this process, and understand the most effective way to debug their programs. The more complex the programs become the more difficult it is to find and fix errors. Thus, the purpose of this essay is to shed light on **"To what extent does code complexity affect the time taken to debug using the debugging techniques: induction and deduction?"**

Based on the research question and background information on debugging, this essay will explore different debugging methods and code complexity, analyze and evaluate the relationship between them.

1.2 What is Debugging?

Programming can be challenging discipline because of the need to write syntactically correct statements *(Kelleher)*. Many IDEs provide support for the programmer when writing code and include a variety of debugging features. Debugging is a process of testing software or other computer systems, identifying the errors, isolating the source of error, and correcting

or determining the way in order to get rid of the errors or 'bugs' *(Crelin)*. The term, bug, in computer technology, is the error in a computer program or software *(Rouse)*. The ultimate goal of debugging is to correct errors and make the program operate in the way that it was designed *(Dale).* It is an integral part of the software testing process, which is an integral part software development lifecycle *(Rouse)*.

Debugging is an important aspect of programming as the error for both levels of developing programs and running the software. Even the most experienced programmers, who know how to debug an application, make mistakes on the program *(Know Your Bugs: Three Kinds of Programming Errors)*. There are three different types of errors that could have occurred when writing a program: syntax errors, run-time errors, and logic errors *(Know Your Bugs: Three Kinds of Programming Errors)*. Syntax errors are caused by mistakes on typing codes, such as misspelling a keyword or using a keyword from the different program (e.g. use a keyword from Python in Java) *(Liang)*. When this occurs, the program cannot run. Run-time errors occur when the program tries to operate impossible codes such as division by zero. Lastly, logic errors are shown when the operation produces a result that a programmer did not expect even though it compiles and runs without error.

1.3 Difference between Testing and Debugging

Testing and Debugging seem to be similar terms as both detect the bugs or errors. However, they are different regarding definition and procedure. Testing is the process of detecting an error by the tester *(Dale)*. An outsider could be the tester, so some developers recruit testers to test their program to identify the bugs. On the other hands, debugging is not just detecting, but finding the cause of the bug and removing it *(Dale)*. Different from testing, debugging cannot be done by an outsider, but must be done by an insider. However, modern

debugging tools or methods work on both testing and debugging in order to shorten the process.

1.4 Determination of code complexity

The complexity of code can be determined through the cyclomatic complexity, which is the software metric. The metric measures the amount of decision logic in a single algorithm based on the structure of control flow graph *(Watson)*. The control flow graph is the nodes of the graph correspond to a single function or subroutine, a single entry and exit point, and call / recall mechanism in a typical language *(Cyclomatic Complexity)*. Simply, for structured programming, the cyclomatic complexity roughly measures the number of loops and if statements to indicate the complexity of the code.

2. Introduction to Debugging Methods

2.1 What is Debugging Method?

Debugging Method is static testing without any software which software executes the code and finds out the error *(Westman)*. It allows programmers to manually review and walk through the code to find out bugs and errors. Since it requires manual execution, it is recommended for the programmer than the developer.

2.2 Introducing several types of Debugging Method

There are four commonly used debugging methods: brute force, induction, deduction, and backtracking. The following section outlines each of these techniques and explains how they can be carried out.

2.2.1 Brute Force

Brute Force is the most common method for programmers. It can work out by the methods starting with going through the code from start to finish until the programmer finds error or oddity. During this process, Memory Dumps, which is a process in which contents of memory and storage locations are displayed and stored in hexadecimal or octal format, and Run-Time traces are examined to detect the source of error causes. If the error is found, the programmer should go over and step through the code again for the source of the oddity. However, in case the programmer could not find the error, he or she needs to step through the code again. Brute Force requires the programmer to repeat whole procedure until the programmer fixes bug or error *(Paul)*.

2.2.2 Induction

Induction is finding out the relationship between the symptom and hypotheses to discover the errors inside the code. Firstly, the programmer locates the pertinent data, enumerating all correct parts and incorrect parts of the program. Incorrect parts are the parts that seem to be major causes of the error. Then, the data should be organized in the structure to let the programmer observe the patterns. It should include general symptoms, where the errors were observed, times that the symptoms occur, and scope and magnitude of the errors. Based on the organized data, the programmer will study the relationships among the errors and devise the hypothesis that provides the hints about the possible causes of errors within patterns of clues. As the last step, the data in the hypothesis with the original data should be compared so that the hypothesis completely explains the existence of the errors. If the hypothesis is invalid or incomplete, the hypothesis should be re-devised.

2.2.3 Deduction

The deduction is using a process of elimination and refinement to proceed from some general theories and to arrive at a conclusion *(Myers).* As the first step of deduction, the programmer will enumerate the possible causes or hypothesis by developing a list of conceivable causes of error. Then, the general symptoms will be analyzed by indicating where the errors were observed, times that the symptoms occur, and scope and magnitude of the errors. This helps the elimination of the causes using the analyzed data. When all causes are eliminated, more data should be collected for the prime hypothesis. Using the remaining causes, the programmer will refine the specific hypothesis, and it will be proved with the comparison of the data in the hypothesis with the original data. This will let the hypothesis to explain the existence of the errors completely. If the hypothesis is invalid or incomplete, hypothesis will need to be refined again.

2.2.4 Backtracking

Backtracking is going back through the code to determine the part with mental reverse execution. In order to use Backtracking method, the programmer should start at the incorrect line in the algorithm and deduce the expected output from the observed output by mentally executing the program in reverse order. During this step, source code will be examined by looking backward from symptom to potential causes of errors. Then, the program will be mentally executed until the error is localized in between the code with expected result.

3. Analyzing Debugging Methods

3.1 Methodology

In order to investigate the research question, a test will be conducted. The participants of this test will be Branksome Hall Asia students in grade 12 who are taking high-level computer science and there will be two groups of four students. To select participants who are going to be tested, random sampling using a random number generator will be conducted. Between the two groups of participants, one group will be trained for 15 minutes using the debugging method of induction and another group will learn deduction. Each group will be informed about the steps of the method, without any practice example, in both English and Korean. After training debugging methods, they will debug three given codes within the taught method of debugging. In this process, the time taken to find the error will be recorded as a dependent variable and working process within flowchart will also be recorded for thorough evaluation. The rationale for only using induction and deduction only from the four debugging methods was that these methods make for a useful comparison since the approach is almost opposing. Further, the limited number of participants available meant using more than two methods would have meant that the generated results would little value since only two participants would have attempted each method.

During the test, algorithms in different complexity based on the cyclomatic complexity metrics was included *(Cyclomatic Complexity)*. The algorithm will be in three level of complexity: simple, intermediate, and complicated. Simple code will contain single loop, double loop will be the intermediate code, and complicated code will use the recursive loop. Recursion is the structure to looping that subprogram has an ability to invoke itself for determination of the repetition of code by calling the subprogram or stopping the code within stopping condition *(Dale)*. In this process, the language that does not contain translator

diagnostics is required since translator diagnostics would give the messages to assist the developer. Thus, JavaScript will be the most appropriate language in this test since it does not give translator diagnostics when a simple text editor is used.

The produced algorithm will contain three errors each. The first error will be changing the keyword in a code into keyword of different language (e.g. Change ‘else if’ in JavaScript language into ‘elif’ in Python language) to produce a syntax error. Another error is an impossible calculation in the code (e.g. create an infinite loop or miscalculation) in order to generate a run-time error. Finally, in order to make a logic error, algorithm will contain a logical fallacy (e.g. using wrong variable name or making a mistake in Boolean expression).

The collected data will be analyzed using mixed ANOVA test with SPSS Statistics v.24. ANOVA test is a statistical method that helps to compare the mean differences between two groups that have been split into two subject factors, setting one variable as ‘within-subjects’ and the other one as a ‘between-subjects’ *(Mixed ANOVA Using SPSS Statistics)*. This test is in a mixed design since the test will use a mixture of between-subjects and repeated-measures variables *(Field)*. The tool is appropriate to use for analysis of the data as it passes most of the assumptions that are required for mixed ANOVA to produce a valid result *(Mixed ANOVA Using SPSS Statistics)*. Utilizing this statistical tool, the evaluation will involve the discussion about the interaction between the measured time of induction and deduction method to debug the code with different complexity level. In this process, the mean and standard deviation for each group and test will be actively used.

3.2 Results of Using Debugging Methods

Debugging method will be used for the code of checking if array is sorted as a simple code, bubble sort as an intermediate code, and quick sort as a complicated code in JavaScript.

```
function check(a) {
    var p = true
    var k = -1
    while (k + 1 < (a.length - 1)) {
        k = k + 1
        if (a[k] > a[k + 1]) {
          p = false
        }
    }
    return p
}

a = ([1.3, 3.98, 4.6, 5.123, 6.15, 6.15]);
check(a);
console.log(check(a));
```

Fig. 1: Original Code of Checking if array is sorted in JavaScript

```
function check(-a) {
    var p = True
    var k = -1
    while (k + 1 < (a.length - 1)) {
        k = k + 1
        if (a[k] >= a[k + 1]) {
          p = False
        }
    }
    return p
}

a = ([1.3, 3.98, 4.6, 5.123, 6.15, 6.15]);
check(a);
console.log(check(a));
```

Fig. 2: Error Code of Checking if array is sorted in JavaScript

```javascript
var a = [4, 2, 9, 6, 23, 12, 34, 0, 1];

function bubbleSort(a) {
    var swap;
    do {
        swap = false;
        for (var i = 0; i < a.length - 1; i++) {
            if (a[i] > a[i + 1]) {
                var temp = a[i];
                a[i] = a[i + 1];
                a[i + 1] = temp;
                swap = true;
            }
        }
    } while (swap);
}

console.log("Original array: " + a);
bubbleSort(a);
console.log("Sorted array: " + a);

```

Fig. 3: Original Code of Bubble Sort in JavaScript

```javascript
function bubbleSort(a) {
    var swap;
    do {
        swap = False;
        for (var i = 0; i = a.length - 1; i++) {
            if (a[i] < a[i + 1]) {
                var temp = a[i];
                a[i] = a[i + 1];
                a[i + 1] = temp;
                swap = True;
            }
        }
    } while (swap);
}

var a = [4, 2, 9, 6, 23, 12, 34, 0, 1];

console.log("Original array: " + a);
bubbleSort(a);
console.log("Sorted array: " + a);

```

Fig. 4: Error Code of Bubble Sort in JavaScript

```
function quick_Sort(origArray) {
  if (origArray.length <= 1) {
    return origArray;
  }
  else {

    var left = [];
    var right = [];
    var newArray = [];
    var pivot = origArray.pop();
    var length = origArray.length;

    for (var i = 0; i < length; i++) {
      if (origArray[i] <= pivot) {
        left.push(origArray[i]);
      } else {
        right.push(origArray[i]);
      }
    }

    return newArray.concat(quick_Sort(left), pivot, quick_Sort(right));
  }
}

var myArray = [4, 2, 9, 6, 23, 12, 34, 0, 1];

console.log("Original array: " + myArray);
var sortedArray = quick_Sort(myArray);
console.log("Sorted array: " + sortedArray);
```

Fig. 5: Original Code of Quick Sort in JavaScript

```
function quick_Sort(origArray) {
  if (origArray.length => 1) {
    return origArray;
  }
  else {

    var left = [];
    var right = [];
    var newArray = [];
    var pivot = origArray.pop();
    var length = origArray.length;

    for (var i = 0; i = length; i++) {
      if (origArray[i] >= pivot) {
        left.push(origArray[i]);
      } else {
        right.push(origArray[i]);
      }
    }

    return newArray.concat(quick_Sort(left), pivot, quick_Sort(right));
  }
}

var myArray = [4, 2, 9, 6, 23, 12, 34, 0, 1];

console.log("Original array: " + myArray);
var sortedArray = quick_Sort(myArray);
console.log("Sorted array: " + sortedArray);
```

Fig. 6: Error Code of Quick Sort in JavaScript

Checking If Array is Sorted

The original algorithm in figure 1 prints out the true or false statement by checking the order of elements in an array. However, figure 2 contains three errors listed below.

- Syntax Error: Change the array syntax of JavaScript ‘var p = true’ and ‘var p = false’ into ‘var p = True’ and ‘var p = False’
- Run-Time Error: Change ‘function check(a)’ to ‘function check(-a)’
- Logic Error: Change ‘if (a[k] > a[k + 1])’ to ‘if (a[k] >= a[k + 1])

Bubble Sort

Figure 3 is an original code which arranges array elements in ascending order. Three errors used in figure 4 that avoids the algorithm to produce an expected result are listed below.

- Syntax Error: Change the array syntax of JavaScript ‘var swap === true’ into ‘var swap === True’
- Run-Time Error: Change ‘i < a.length - 1’ to ‘i = a.length – 1’ to make it run out of memory
- Logic Error: Change ‘if (a[i] < a[i + 1])’ to ‘if (a[i] > a[i + 1])’

Quick Sort

The original code of quick sort produces ascending order of array elements as shown in figure 5. On the other hands, figure 6 contains three errors, which are listed below.

- Syntax Error: Change ‘if (origArray.length <= 1)’ to ‘if (origArray.length => 1)’ so it become arrow function instead of inequality function
- Run-Time Error: Change ‘for (var i = 0; i < length; i++)’ to for (var i = 0; i = length; i++)’ to make it run out of memory
- Logic Error: Change ‘if (origArray[i] <= pivot)’ to ‘if (origArray[i] >= pivot)’

Time taken on three types of code for debugging with induction			
	Time taken to Debug (seconds)		
// Type of Code Participants //	Checking If Array is Sorted	Bubble Sort	Quick Sort
Participant 1	317	244	729
Participant 2	302	190	582
Participant 3	569	315	751
Participant 4	273	130	676

Table 1: Time taken on three types of code for debugging with induction

Time taken on three types of code for debugging with deduction			
	Time taken to Debug (seconds)		
// Type of Code Participants //	Checking If Array is Sorted	Bubble Sort	Quick Sort
Participant 1	210	172	591
Participant 2	512	483	930
Participant 3	391	466	904
Participant 4	185	149	350

Table 2: Time taken on three types of code for debugging with deduction

Average time taken on three types of code for debugging with induction and deduction			
	Time taken to Debug (seconds)		
// Type of Code Type of Method //	Checking If Array is Sorted	Bubble Sort	Quick Sort
Induction	365.25	219.75	679.5
Deduction	324.5	317.5	693.75

Table 3: Average time taken on three types of c ode for debugging with induction and deduction

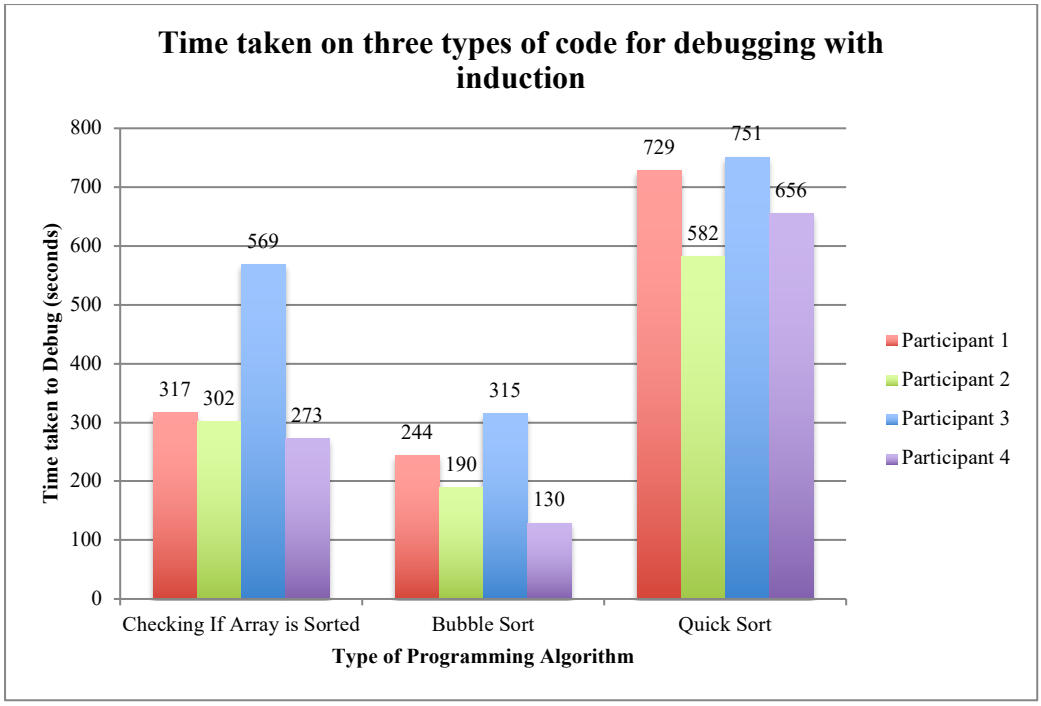

Fig 7: Time taken on three types of code for debugging with induction

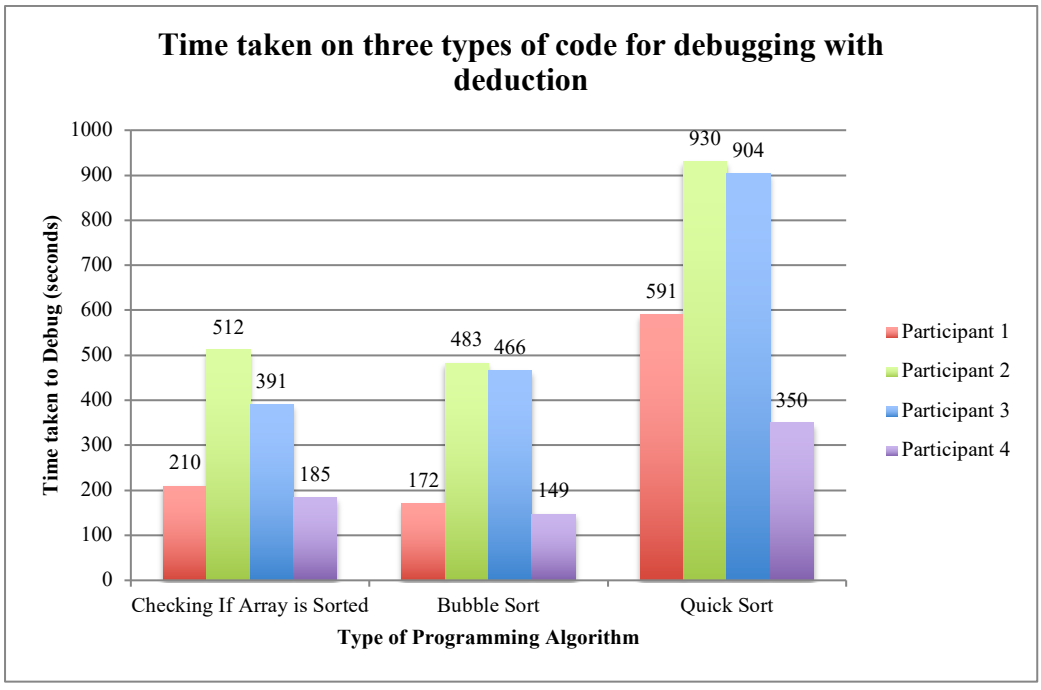

Fig. 8: Time taken on three types of code for debugging with deduction

Time taken to complete each of the debugging techniques and find the error

	Group	Mean	Std. Deviation	N
Code1	Induction	370.25	146.973	4
	Deduction	324.50	155.080	4
	Total	347.38	141.995	8
Code2	Induction	212.25	77.073	4
	Deduction	317.50	181.664	4
	Total	264.88	140.906	8
Code3	Induction	684.50	75.235	4
	Deduction	693.75	276.128	4
	Total	689.13	187.423	8

Table 4: Mean and Standard Deviation for each debugging technique by programming task

Mauchly's Test of Sphericity[a]

Measure: MEASURE_1

Within Subjects Effect	Mauchly's W	Approx. Chi-Square	df	Sig.	Epsilon[b]		
					Greenhouse-Geisser	Huynh-Feldt	Lower-bound
CodeDifficulty	.475	3.720	2	.156	.656	.906	.500

Table 5: Mauchly's Test of Sphericity for Within-Subjects

Tests of Within-Subjects Contrasts

Measure: MEASURE_1

Source	CodeDifficulty	Type III Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared
CodeDifficulty	Linear	467172.250	1	467172.250	52.550	.000	.898
	Quadratic	342394.083	1	342394.083	199.642	.000	.971
CodeDifficulty * Group	Linear	3025.000	1	3025.000	.340	.581	.054
	Quadratic	20336.333	1	20336.333	11.858	.014	.664
Error (CodeDifficulty)	Linear	53340.750	6	8890.125			
	Quadratic	10290.250	6	1715.042			

Table 6: Tests of Within-Subjects Contrasts

Tests of Between-Subjects Effects

Measure: MEASURE_1

Transformed Variable: Average

Source	Type III Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared
Intercept	4516205.042	1	4516205.042	62.168	.000	.912
Group	3151.042	1	3151.042	.043	.842	.007
Error	435868.250	6	72644.708			

Table 7: Tests of Between-Subjects Effects

1. CodeDifficulty

Measure: MEASURE_1

CodeDifficulty	Mean	Std. Error	95% Confidence Interval	
			Lower Bound	Upper Bound
1	347.375	53.415	216.673	478.077
2	264.875	49.334	144.158	385.592
3	689.125	71.549	514.052	864.198

Table 8:Mean and Standard Deviation for Within-Subjects

2. Group * CodeDifficulty

Measure: MEASURE_1

Group	CodeDifficulty	Mean	Std. Error	95% Confidence Interval	
				Lower Bound	Upper Bound
Induction	1	370.250	75.540	185.410	555.090
	2	212.250	69.769	41.531	382.969
	3	684.500	101.185	436.909	932.091
Deduction	1	324.500	75.540	139.660	509.340
	2	317.500	69.769	146.781	488.219
	3	693.750	101.185	446.159	941.341

Table 9:Mean and Standard Deviation for Between-Subjects

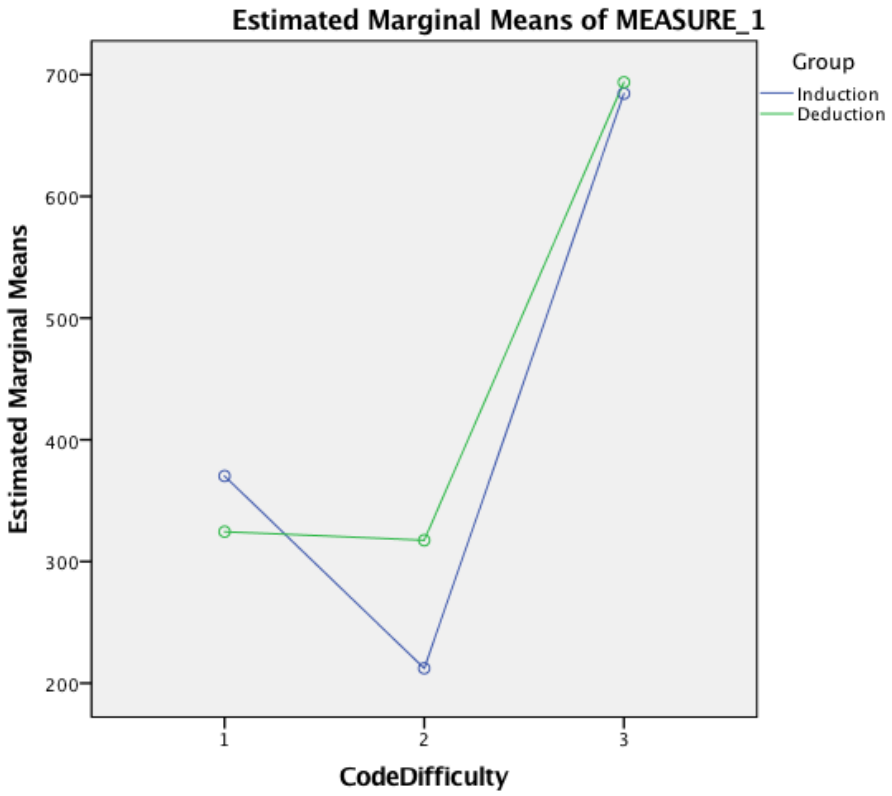

Fig. 9: Estimated Marginal Means of Measured Time on debugging the Code Difficulty with Induction and Deduction methods

3.2 Analyzing Results

Induction and deduction showed different characteristics and way of identifying the errors during the data collection. This is because of the varied approach of each method: Induction moves from the particulars of the situation to the entire during debugging process and Deduction suggests few assumptions and remove and modify it to debug *(Pal)*.

These approaches demonstrate the different level of complexity. Induction shows an intermediate level of complexity since it requires sufficient prior knowledge on both programming language and software development to contemplate the relationships between causes and effect and conclude with proved hypothesis. Deduction method also reveals intermediate complexity level as it uses a process of elimination, which requires a complete understanding on programming.

According to the figure 7, data of both period of time taken to debug the simple code with a single loop and intermediate code with a double loop were similar, which is relatively short to complicated loop. In addition, the longest period of time taken to debug the complicated code with a recursive loop. This roughly indicates that code complexity affects the time taken to debug no matter the debugging method is induction or deduction. Thus, the result supports the fundamental idea of the research question of this investigation.

The ANOVA test result from table 4 to 9 illustrates statistical significant differences between code difficulty and measured time to debug. In this process, Mauchly's test of Sphericity shown in table 5 verifies the assumption of sphericity by testing the null hypothesis that the variances of the differences are equal *(Sphericity)*. When the significance value is less than 0.05, the test is statistically significant, which means that sphericity has

been violated. According to the table 5, Mauchly's test indicated that the assumption of sphericity had not been violated, $x^2(2) = 3.720, p = 0.188$.

Results from table 9 present the similar mean values of measured time within different debugging methods as difference between induction and deduction are 45.75 (Std. = 75.540), 105.25 (Std. = 69.769), and 9.25 (Std. = 101.185) in ascending order of code complexity. The result implies that the use of induction and deduction methods does not affect the time taken to debug algorithms in different code complexity.

On the other hands, as seen in the figure 9, deduction shows lower value on debugging the algorithm with code difficulty 1 than induction and induction presents lower time taken to debug the algorithm with code difficulty 2 and 3 than deduction. This result can be inferred as the deduction is more useful in shorter and more simple code compared to induction, which is seems to be more effective on debugging the complicated code, despite the result that type of debugging method does not affect the time taken to debug.

4. Evaluating Debugging Methods

4.1 Evaluate the relationship between Debugging Methods and Code Complexity

According to the analysis of collected data, code complexity and debugging methods have the relationship by code complexity influences the debugging time using induction and deduction. As the code gets simpler by the decrease of a number of the loop, dependent variable decreases. This result has presented due to the length of the code since as a number of loop increases, the length of the code increases. The increase of the code length highly affects the time taken for debugging, due to the trace table used in the process of debugging. Trace table is a technique to test the algorithm as it stimulates the flow of the program execution *(Dale)*. Due to the fact that trace table is useful for finding out the location of the error, it is highly involved in debugging. However, as the code length increases, it requires going through the code for a longer time, which increases the time taken for debugging.

Although both debugging methods show the similar result, according to the table 9, induction is more time-efficient for complicated code and deduction is faster method for simple code. This is because of the difference of these methods, even though both methods allow them to indicate detailed information about the error to programmers. In case of induction, it requires a programmer to study the relationships of the clues and hypotheses about the causes of the error, which excessively long time for simple codes, whereas proving step helps to find out the error and fix it inside of the long and complicated code. On the contrary of this, deduction uses illumination method to detect the causes, so it can easily debug the short algorithms, but as long as it requires listing every cause and eliminate them with a reasonable reason, it takes longer time for debugging the complicated algorithm.

4.2 Limitations of the Methodology

Through the process of analyzing and evaluating the debugging methods, several limitations that needed to be explicitly discussed were shown. These are in regards to the methodology and the programming code itself.

The first major limitation was that the programming code used in the data collection, typically the bubble sort, was not enough to show the complexity of code as it has same function, which makes the debugging easier. In addition, due to the fact that computer science HL students have learnt bubble sort at the start of the course so they are more used to the code than usual, which lets decrease of the accuracy of the code.

Secondly, the methodology was deficient to reveal the limitations of each debugging method. For example to the limitation of the method, the deduction process is not self-sufficient method because they are the ways of deducing the hypothesis from the symptom and knowledge, so it is difficult to rely on the hypothesis itself. Discussion of the weaknesses is a significant procedure on evaluating the methods because it helps to compare and contrast the methods and make an objective conclusion.

Another limitation that came to light was that even though the JavaScript language does not contain translator diagnostics, the applications were still able to give hint to the programmer. In case of jsbin, it alerts the programmer about the existence of error by underlining the line in red, which let the programmer detect the cause and location of the error faster than it should be. When the atom, which is an application that lets programmer write code, is used, it colors the words that are in the same group of syntax, so the programmer can get a hint about the syntax error. This might affect the collected data, which

would have decreased the accuracy of data.

In addition, the applicants' skills were varied. Some of the applicants were more knowledgeable on programming, more familiar with the algorithm or the programming language, or more used to the debugging method, which was randomly chosen. Since getting rid of the variation of the applicants is an impossible task on sampling, so the random sampling has been used to keep the reliability of data. On the other hands, the discrepancy in ability in programming and debugging affects the collected data, so it decreases the accuracy of the result.

5. Future of Debugging

5.1 Automated Debugging Tools

Besides manual debugging methods, within the development of technology, there are several programs that contain automated debugging tools such as Visual Basic, Python idle, and NetBeans. The tools identify and correct a failure's root cause automatically *(Westman)*. Error diagnostics is the main function of it as the program detects errors at compilation time and produce the message to the user to indicate the location and the type of error. It has developed to help testing and editing process of software development cycle for software developers and programmers. The tools are useful for developers than programmers, who are trying to explore the unfamiliar code because the tool can suggest many promising starting places *(Parnin)*. For these reasons, numerous programmers and developers choose to use one of the automated debugging tools instead of the debugging method. On the other hands, it can not be defined as 'better way to debug than manual debugging methods', because debugging method is still the method that has better explanatory capability and credibility. Within the development of technology, automated debugging tools will contain fewer limitations than current tools. However, it will never be perfect, so moderate combination of manual debugging and automated debugging will cause fewer bugs in the code or software.

5.2 Manual Debugging Methods

Although debugging methods including four methods discussed in this essay are useful for detecting the errors and give suggestions to the programmers, more users are attracted to the automated debugging tools in modern days, due to its convenience. This is a great issue since many developers and programmers tend not to debug on their own but rely on the computer. Relying on debugging tools is not a desirable conclusion because various

programmers might not try to understand the code and relationship between each part fully, which will cause the slow progress on their own development.

6. Conclusion

6.1 Conclusion

Concluding the question **“To what extent does code complexity affect the time taken to debug using the debugging techniques: induction and deduction?”** depends on several factors. Programming codes vary in the type of programming language, the purpose of the code, and size of the code. Thus, it is important to understand that ‘programming code’ is vague to investigate and evaluate. Using any algorithm on the experiment, excluding the fact that there are large amounts of types in the algorithm, might cause the generalized conclusion.

Despite the limitations presented within the investigation, it was found that as code complexity increases, the time taken to debug the code also increases. In addition, induction was found to be a more useful debugging method on debugging the code with higher complexity, while deduction was found to be a more time-efficient debugging method on debugging the code with lower complexity. This conclusion was found based on the advantages and disadvantages of each method. Thus, supported by the analysis and evaluation, it can be concluded that each debugging method, typically induction and deduction, is highly related to the complexity of the code in an aspect of time measure.

Works cited

Crelin, Joy. “DEBUGGING.” Applied Science: Computer Science, EBSCOhost, May 2016, search.ebscohost.com/login.aspx?direct=true&db=sch&AN=115226571&site=scirc-live. Accessed March 2, 2017.

“Cyclomatic Complexity.” *Project Code Meter*, www.projectcodemeter.com/cost_estimation/help/GL_cyclomatic.htm. Accessed September 13, 2017.

Dale, Nell, and John Lewis. *Computer Science Illuminated*. Jones & Bartlett Learning, 2002.

Dillon, Darin. *Debugging Strategies for .Net Developers*. Apress, 2003. Accessed July 17, 2017.

Field, Andy P. *Discovering Statistics Using SPSS*. 3rd ed., SAGE, 2009. Accessed September 27, 2017.

Paul, Dr. Jody. “Testing and Debugging.” *Testing and Debugging - Dr. Jody Paul*, www.jodypaul.com/SWE/TD/TestDebug.html. Accessed May 21, 2017.

Kaplan, Randy M. "Choosing a first programming language." Proceedings of the 2010 ACM conference on Information technology education. ACM, 2010. Accessed February 9, 2017.

Kelleher, Caitlin, and Randy Pausch. "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers." ACM

Computing Surveys (CSUR) 37.2 (2005): 83-137. Accessed May 12, 2017.

"Know Your Bugs: Three Kinds of Programming Errors." Microsoft Developer Network, 2008, https://msdn.microsoft.com/en-us/library/s9ek7a19(v=vs.100).aspx. Accessed May 6, 2017.

Liang, Y. Daniel. *Introduction to Java Programming and Data Structures*. 10th ed., Pearson Education, 2018. Accessed August 11, 2017.

"Mixed ANOVA Using SPSS Statistics." *How to Perform a Mixed ANOVA in SPSS Statistics*, Laerd Statistics, statistics.laerd.com/spss-tutorials/mixed-anova-using-spss-statistics.php. Accessed October 13, 2017.

Myers, Glenford J, et al. "The Art of Software Testing." Find in a Library with WorldCat, John Wiley & Sons, 19 Jan 2017, www.worldcat.org/title/art-of-software-testing/oclc/54356468. Accessed March 2, 2017.

Pal, Kaushik. "What Is Testing and Debugging Strategy." *MrBool*, MrBool, 13 Jan. 2014, mrbool.com/what-is-testing-and-debugging-strategy/29914. Accessed May 23, 2017.

Parnin, Chris, and Alessandro Orso. *"Are Automated Debugging Techniques Actually Helping Programmers?"* New York (NY), A.C.M., 2011, www.cc.gatech.edu/~orso/papers/parnin.orso.ISSTA11.pdf. Accessed June 1, 2017.

Rouse, Margaret. “Bug.”*SearchSoftwareQuality*, SearchSoftwareQuality, Feb. 2007, searchsoftwarequality.techtarget.com/definition/bug. Accessed May 6, 2017.

“Sphericity.” *Laerd Statistics*, Laerd Statistics, statistics.laerd.com/statistical-guides/sphericity-statistical-guide.php. Accessed October 16, 2017.

Watson, Arthur Henry, et al. *Structured Testing: a Testing Methodology Using the Cyclomatic Complexity Metric*. Vol. 13, U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, 1996, books.google.co.kr/books?hl=en&lr=&id=lysRzUZhc2QC&oi=fnd&pg=PR3&dq#v=onepage&q&f=false. Accessed September 27, 2017.

Westman, Patrik. “Debugging Methods.” *Multilevel Methods in Lubrication Tribology Series*, 2000, pp. 255–258., doi:10.1016/s0167-8922(00)80013-9. Accessed May 14, 2017.