Universal Measure of Similarity for Student Programming Assignments

Carolín Laoide-Kemp

MAI Dissertation

Presented to the University of Dublin, Trinity College in partial fulfilment of the requirements for an

MAI in Computer Engineering

Supervisor: Dr. Jonathan Dukes

Acknowledgments

I would like to thank my supervisor, Dr. Jonathan Dukes, for his support and guidance during this project. Thank you also to my parents, my brother, and sister-in-law for all their support and proof-reading. Finally, thank you to my wonderful cat Misophonia "Missy" Moo, for providing much needed stress-relief in the form of purrs and cuddles.

Carolín Laoide-Kemp

University of Dublin, Trinity College
August 2022

Contents

Table of Figures

Table of Tables

Chapter 1: Introduction

Context

This dissertation describes the creation and implementation of a tool that, given a collection of computer programs written to produce the same overall functionality, provides a measure of similarity for each program compared to every other program in the collection. Such a tool has the potential to be useful in many situations, however this project is motivated by the need to identify plagiarism in programming assignments in an educational setting.

Current approaches to measuring similarity between computer programs analyse at a source code level, which is inherently programming language specific. Some common tools that use this approach are MOSS (Measure of Software Similarity) and JPlag. MOSS uses fingerprinting (Schleimer et al, 2003) and JPlag uses pairwise comparison (Prechelt et al, 2002). This project aimed to develop a tool which would measure similarity between two programs at a machine code level. It is expected that this method will be able to flag potential instances of plagiarism using similarity scores in the same way existing tools do, but also be language independent.

Machine language (also referred to as machine code) is "the numeric codes for the operations that a particular computer can execute directly" (Hemminger, 2016). Machine code is a series of 0s and 1s. These binary digits are often converted to hexadecimal for human readability. Source code (i.e. computer programs that humans write) is compiled into machine code so that the computer can execute the instructions. There is a one-to-one mapping between machine code and its assembly language representation. "Assembly language is one level above machine language. It uses short mnemonic codes for instructions" (Hemminger, 2016). Assembly language, although rarely used by programmers, is a human readable analogue of machine code.

Table 1: Breakdown of machine code instructions

Machine Code	**ARM Assembly Instruction**	**Explanation**
`0000A0E3`	`MOV R0, #0`	Places the value 0 in the memory register R0
`010080E0`	`ADD R0, R0, R1`	Adds the values in memory registers R1 and R0 and places the answer in R0

This dissertation examines the ARM microprocessor instruction set as a case study, but the same principles could be applied to other machine architectures, e.g. Intel x86. Although assembly language will be used in this dissertation to assist with readability, the measures of similarity will be based upon the machine code rather than the human-readable representation of it.

Advantages and Disadvantages

A significant advantage of this approach is that similarity can be measured regardless of the language in which the program is written. Instead of comparing the original solutions written in a high-level language (as is the case for current programming plagiarism detection software), it makes comparisons at a machine code level. Any compiled programming language can be reduced to machine code and is therefore a candidate for analysation. Other plagiarism detection software for programming is limited in which programming languages can be analysed, as the analysis is language specific.

Another notable advantage is that this reduction to machine code neutralises certain attempts at obfuscation. Any attempt to hide plagiarism by adding comments or whitespace or by changing variable names will not be detected as meaningful changes by this approach. Only the instructions themselves are compared. This thwarts the most common techniques that students use in an attempt to hide plagiarism, e.g. changing the variable name "Number" to "num", or rewording comments so that "if x <= y; x++" becomes "if x is less than or equal to y, increase x by the value of 1".

A disadvantage of this approach is that it doesn't currently work for interpreted languages, such as Python or JavaScript, as this method analyses the native machine code directly produced by compiling. This limits the approach somewhat. However, most student programming assignments, particularly in Computer Engineering, are written in compiled languages like C, C++, and ARM Assembly.

Another disadvantage is that by removing comments, whitespace and variable names, genuine differences between programs, and not just attempts at plagiarism detection evasion, may be inadvertently removed. For example, the comments may describe why a certain method was chosen, and another student might have a similar method but a completely different reason for choosing it. This is an example of why human investigation is also needed for plagiarism detection.

Motivation

Plagiarism is "the process or practice of using another person's ideas or work and pretending that it is your own" (Cambridge University Press, 2022). This project focuses on plagiarism within a university setting, particularly within the fields of Computer Science and Computer Engineering. Plagiarism that goes undetected results in unearned qualifications and undermines the value of them. Plagiarism

poses a challenge particularly in the field of Computer Science, accounting for 37% of academic dishonesty over a decade at Stanford University (Roberts, 2002). Barrett and Cox (2005) surmised that there were higher rates in this field because personal reflection is expected of students in arts and humanities, whereas in Computer Science and Mathematics, an ideal solution is being aimed for.

Plagiarism in programming is not always copy & paste. Students may change variable names, reformat code or add comments in an attempt to pass work off as their own. Because of this, computer programming modules require a different mode of detection than essays etc, hence different tools being available specifically for this purpose. Granzer et al (2013) identify the following methods of concealing plagiarism: text replacement (e.g., renaming of variables), code reordering, code rewriting, insertion/deletion of superfluous code and mixing of code from different sources.

Cahill (2022) reports that plagiarism in Trinity College Dublin has become more problematic due to the greater use of online learning during the COVID-19 pandemic. This topic is therefore not only relevant but presents a pressing need for a solution.

Challenges

One challenge faced by the method proposed in this dissertation is its efficiency. It uses pairwise comparison which is computationally quite expensive. The proposed decoding of the machine code to perform hierarchical analysis is also computationally expensive.

Another challenge is that there is a limited set of instructions and registers used for encoding at machine code level. This means there will inevitably be some natural repetition. This is something that other tools must also deal with, given that high level programming instructions often have common and necessary elements. For example, 'for' and 'while' loops will have a consistent syntax. However, given the greater selection of instructions, the percentage of natural repetition is lower and does not pose as much of a problem.

Another challenge that must be considered is the non-binary measure of similarity between any two instructions that appear in two different programs. For example, `ADD R0, R1, R0` (or more accurately, its machine code equivalent of `0x000081E0`) and `ADD R0, R1, R2` (or its machine code equivalent of `0x020081E0`) could be interpreted as completely different instructions. This dissertation however will investigate non-binary measures of similarity between two instructions. For example, the similarity of the two instructions above was given the value 0.75 as they only differ in one operand (R0 becomes R2) and are more similar than dissimilar. Similarly, `ADD R0, R0, R1` (machine code `010080E0`) and `ADD R0,`

`R1, R0` (machine code `000081E0`) would have a similarity measurement of 1 as the order of operands in an add instruction is inconsequential.

In order to evaluate the effectiveness of any possible solution as a result of this project, it was necessary to obtain an appropriate corpus of representative programs. The programs within the corpus had to come from student assignments, as this dissertation is specifically looking at plagiarism in Computer Science student programming assignments. In order to obtain the corpus, it was necessary to apply for ethical approval from the Trinity Research Ethics Committee. The application was approved. It was however, challenging to find a way to anonymously collect files from students. Free survey products that are often used in collection of data require payment for the file upload feature. Products that allow free file upload, were unable to do so anonymously. How this was overcome will be discussed in detail in Chapter 4.

Approach

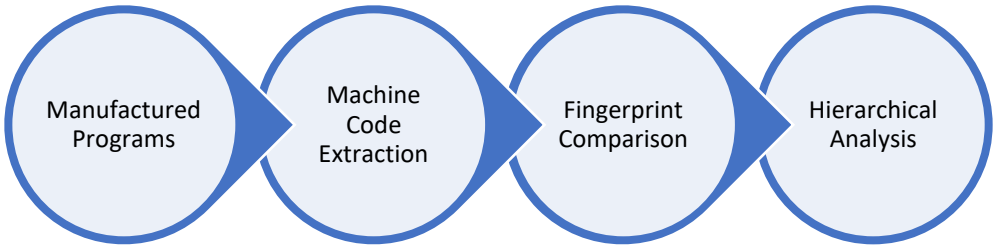

Figure 1: Simplified Summary of Approach

In order to measure similarity in the way this dissertation proposes, the machine code needs to be extracted from each solution for analysis. This was done by compiling each solution using the GNU Arm Embedded Toolchain and creating MASM listing files. The listing file shows line numbers, relative addresses, and the machine code for each instruction. A custom Python script was written to extract the machine code from each listing file. This was the method chosen for this project, but there are many ways the machine code can be obtained e.g., extracting it from the object file produced during compilation, or, if it is known where the program is stored in memory, extracting it from there. The method chosen does not have any impact on the analysis.

To perform some basic analysis, some manufactured programs were written, each changed slightly from the previous one to mimic plagiarism. The scripts range from very similar to the original to not very similar. The scripts were compiled, listing files produced and machine code extracted. Some off-the-shelf techniques were used for initial analysis of the manufactured programs. The analysis was firstly done on

the machine code itself as a whole, and then hierarchical analysis was added incrementally. This hierarchical analysis looked at a non-binary measure of similarity between instructions, as discussed above.

The hierarchical analysis was done by decoding the machine code using a tool called Capstone Engine. This produced the ARM Assembly equivalent of the instruction and its registers (where applicable), allowing the comparison of separate elements to produce a non-binary similarity score.

These manufactured programs were then passed through an already on-the-market programming plagiarism detector. The results of this were then compared with the technique proposed here. To compare them on a larger and more realistic scale, the corpus of student assignments was run through the published plagiarism detector and analysed using the technique proposed here.

Outcome

Experiments were run on both manufactured programs and the corpus of student assignments with artificially introduced instances of plagiarism to assess the effectiveness of this approach. The experiments showed that it is possible to implement measures of similarity using machine code, with promising results.

Structure of this Dissertation

Background

This section will inform on the background information required, as well as the state of the art.

Design and Implementation

This section will discuss in detail the design of the final similarity detection approach, and how it was implemented.

Evaluation

This section will discuss the experiments performed, the comparison of this approach with the state of the art, and the evaluation of experimental results

Conclusion

This section will summarise the findings of this dissertation, as well as suggestions for future work.

Chapter 2: Background

Introduction

This section of the dissertation will discuss the topic of plagiarism, its prevalence, and the motivation involved. It will also consider plagiarism particularly in a Computer Science setting, the impact it has on learning, and what forms it takes. Automated tools for measuring similarity will then be introduced, both general use and those specifically designed for programming. Finally, various algorithms for calculating similarity will be discussed.

Plagiarism

Plagiarism is "the process or practice of using another person's ideas or work and pretending that it is your own" (Cambridge University Press, 2022). Plagiarism is theft of intellectual property and undetected plagiarism devalues qualifications. According to Simatupang et al (2021), during the Covid-19 Pandemic there was a tendency for university students to plagiarise via "copy-paste" due to easy internet access in the switch to online learning. Cahill (2022) reported that there has been an increase in plagiarism in Trinity College Dublin, which coincided with the global pandemic and the move to online learning. There were 47 cases reported in the 2018/19 academic year, 79 in 2019/20 and 124 cases in 2020/21, according to the Senior Lecturer's annual report (Cahill, 2022). Bishop (2021) concluded that although technology makes it easier to detect plagiarism, it also makes it easier to plagiarise.

Dey and Sobhan (2006) report a number of reasons for plagiarism by students:

a) they may not understand the concept of plagiarism itself
b) they may not think their work is good enough
c) they may be bad at writing, researching, and referencing
d) they may wish to save time and effort
e) they may want better grades
f) they may think everyone else is doing it
g) it may also be a result of cultural factors.

Naude and Hörne (2006) add that some students may feel a certain entitlement to their grades, while others simply consider the assignment a waste of their time (Howard, 2001). Dey and Sobhan (2006) observe that plagiarism "degrades the morale of the students and reduces their productivity and creativity", but it also impacts honest students, as well as affecting academic and professional standards.

Source Code Plagiarism

With regard to plagiarism in Computer Science, Fraser (2013) notes the difference between collaboration, collusion, and plagiarism; collusion being viewed more generously than plagiarism, but still regarded as unpermitted collaboration. Computer Science is subject to a lot of cheating as there is usually an ideal solution and the best answers will therefore be similar (Fraser, 2013). Fraser comments that dishonest students often take advantage of this fact. Sabin and Sabin (1994) state that collaboration can aid student learning. Collaboration allows student discussions and active learning (Fraser, 2013). Given that collusion is unauthorised collaboration, it follows that collusion can also aid learning. As such many lecturers are willing to forgive collusion as it indicates students are thinking and learning throughout the process (Fraser, 2013). Fraser also notes that excessive collaboration leads to weaker students being unable to learn.

Granzer et al (2013) identify different types of source code plagiarism. Source code can be taken from open libraries, which although generally legal, is still plagiarism. Source code can also be obtained from solutions or examples from previous years, written by past students or by lecturers themselves. Finally, Granzer et al (2013) report that source code can be copied from other students, with or without their consent. To conceal the fact that the work is not their own, students will use methods like text replacement (e.g. renaming of variables), code reordering, code rewriting, insertion/deletion of superfluous code and mixing of code from different sources (Granzer et al, 2013). It is these methods of obfuscation that this project aims to detect using machine code analysis and comparison.

Automated Measures of Similarity

Plagiarism could theoretically be detected by interviewing students post-submission to verify their own work. However, this would be very time-consuming, hence the need for detection software. It should be noted that no matter how good the detection tools are, they need to be used in conjunction with human review. These types of software can produce a measure of similarity and flag instances of potential plagiarism but cannot categorically decide what is plagiarism and what is not.

The main objectives of algorithms that measure similarity are accuracy and efficiency. Granzer et al (2013) describe automation of plagiarism detection in the following steps:

i. the used data and functionality is analysed
ii. the analysed data and instructions are stored in an abstract way
iii. the stored information is filtered
iv. the filtered information is compared.

TurnItIn

The most well-known plagiarism detection tool is iParadigms' TurnItIn, which is most commonly used for essay-based assignments. TurnItIn scans submitted electronic work against a database of web pages and other documents (Lancaster and Culwin, 2004), and produces an originality report. This comes in the form of a HTML formatted page with sections of the document text replaced with hyperlinks to sources found in their database (Lancaster and Culwin, 2004). It is based on text matching. It was not possible to find the exact algorithm used, as it is a commercial product. Other similar products such as Grammarly (Grammarly, 2022) seem to use a text matching algorithm as well, to find duplicate content.

A text matching algorithm would not be suitable in detecting programming plagiarism. Programming languages have predetermined syntax, for example in how "while loops" are formatted. These standard blocks of syntax would be flagged as duplicate content, leading to a higher similarity score than appropriate. A text matching algorithm would not be able to detect some of the most common ways that source code is plagiarised, e.g., changing variable names and rewriting comments. For this reason, other algorithms and software are used.

MOSS

MOSS (Measure of Software Similarity) is a widely used similarity detection tool, that is used to help detect plagiarism, primarily in programming assignments (Schleimer et al, 2003), It can analyse code written in 24 different coding languages (Aiken, 2021). It accepts batches of documents and returns a set of webpages highlighting similarities (Schleimer et al, 2003). Yang (2019) states that MOSS is a "copy-detection algorithm", whose properties include whitespace insensitivity, noise suppression and position independence. To avoid comparing substrings (which is computationally expensive), MOSS implements the technique of document fingerprinting (Schleimer et al, 2003). A set of hashes (to act as the fingerprint) is computed for each document, which reduces the number of comparisons (Yang, 2019).

According to Yang (2019), applying document fingerprinting for plagiarism detection purposes takes the following approach:

1. Irrelevant features are removed during pre-processing.
2. A sequence of hashes is generated for the k-grams of the pre-processed document. K-grams are k-length sub sequences of a string (Uberoi, 2019).
3. A subset of these hashes is selected to be the document's fingerprints.
4. Pairs of documents that have similar fingerprints are flagged.

Winnowing

In order to select the fingerprints from the set of hashes, MOSS uses an algorithm called "winnowing" (Yang, 2019). A simplified description of winnowing is as

follows: the sequence of hashes generated during pre-processing is divided into w-length sub-sequences, and for each sub-sequence, the right-most minimum hash value is recorded as a fingerprint (provided it has not been recorded before) (Yang, 2019). The following example of document fingerprinting has been adapted from Yang (2019).

Document text: "Hello what to do, Missy-Moo"

1. The document is pre-processed, and irrelevant features are removed to produce "hellowhattodomissymoo"
2. The k-grams for the pre-processed document, where k=5, are:

Table 2: k-grams where k=5

hello	ellow	llowh	lowha	owhat	whatt
hatto	attod	ttodo	todom	odomi	domis
omiss	missy	issym	ssymo	symoo	

3. The hashes for each k-gram in this case were produced using the CRC-1 hash method in GeneratePlus (2022)

Table 3: Hashes for k-grams

14	23	26	1b	23	28
20	1c	2a	23	18	1c
2b	35	35	3b	37	

4. Using windows of w=4, fingerprints are selected via winnowing. The recorded fingerprints are in bold

Table 4: Windows of hashes to select fingerprints

(**14**, 23, 26, 1b)	(23, 26, **1b**, 23)	(26, 1b, 23, 28)	(1b, 23, 28, 20)	(23, 28, 20, **1c**)
(28, 20, 1c, 2a)	(20, 1c, 2a, 23)	(1c, 2a, 23, **18**)	(2a, 23, 18, 1c)	(23,18, 1c, 2b)
(18, 1c, 2b, 35)	(**1c**, 2b, 35, 35)	(**2b**, 35, 35, 3b)	(35, **35**, 3b, 37)	

5. The final fingerprints are **14 1b 1c 18 1c 2b 35**

This method leads to the following properties: no matches shorter than k are detected, there is an even distribution of hashes, and matches of at least length w+k-1 are always detected (Yang, 2019)

In reality, MOSS selects fewer fingerprints using a sturdier winnowing algorithm (Yang, 2019) which can be read about further in the paper by Schleimer et al (2003). MOSS is not affected by whitespace, noise, or order (Herold, 2021). As previously stated, MOSS is a language-specific tool, and this plays a big role in how the document is pre-processed (Yang, 2019).

JPlag

JPlag is another plagiarism detection tool, that analyses program source text written in Java, Scheme, C and C++ (Prechelt et al, 2002). It compares the programs submitted pairwise and produces a set of webpages to display the similarities found. It works by "converting each program into a stream of canonical tokens and then trying to cover one such token string by substrings taken from the other (string tiling)" (Prechelt et al, 2002). Prechelt et al state that similarity levels above 40% should be investigated.

The fact that it uses pairwise comparison makes it computationally expensive. However, the run time complexity is reduced by comparing substrings by their hash value, and the use of a hash table (Prechelt et al, 2002). Prechelt et al also state that the program easily scales to hundreds of submissions. Despite the pairwise comparison, it is stated that the run time for JPlag to read, parse and compare 99 programs (~250 lines each) is under 12 seconds on their workstation.

Although unable to provide a "direct quantitative comparison" to Moss, Prechelt et al state that JPlag's performance should be at least as good as MOSS if not better. This is based on an evaluation of the tool on "four real sets of student programs" and instances of plagiarism created by an external source. The main drawback of this tool is that it is limited in the languages it can parse.

Compare50

Compare50 (n.d.) is a code similarity detection tool that supports over 300 programming and templating languages. It is open source and can be run locally. Compare50 was used to help evaluate the approach of this dissertation. Compare50 has a number of different comparison methods:

1. Structure: "Compares code structure by removing whitespace and comments; normalising variable names, string literals, and numeric literals; and then running the winnowing algorithm"
2. Text: "Removes whitespace, then uses the winnowing algorithm to compare submissions"
3. Exact: "Removes nothing … then uses the winnowing algorithm to compare submissions"
4. Nocomments: "Removes comments, but keeps whitespace, then uses the winnowing algorithm to compare submissions"
5. Misspellings: "Compares comments for identically misspelled English words"

The structure comparison method (no.1 above) is the one that will be used in the evaluation section of this dissertation.

Compare50 produces a ranked list of the top N matches (50 by default). It also creates a folder of HTML files for each match. These HTML files show the matching

pieces of code between two submissions. Each piece is given a weight depending on how rare it is within all the other submissions, and this is factored in when producing the structure similarity score. Like MOSS, this tool uses winnowing, but it also combines the algorithm with a lexer (also called a lexical analyser) from Pygments, which takes the programming language into account. By taking the programming language into account, certain pieces of syntax, for example "if (condition){…}", are not analysed as being unique parts of code, as they are standard and will appear in every program written in this language.

Other tools

Another plagiarism detection tool available is Lichen. Like MOSS, it uses tokenising, fingerprinting and comparison of fingerprints to generate a similarity score (Peveler et al, 2019). Codequiry, also a plagiarism detection tool, uses MOSS (Issuu, 2022). There are many other tools like Unicheck and CopyLeaks which do not describe how they detect plagiarism, presumably to discourage students trying to use that information to fool them. Students, upon discovering how plagiarism is detected for a particular software, may purposefully use other methods of obfuscation that the software doesn't recognise, in order to evade detection.

Similarity Measurement Algorithms

Levenshtein

Levenshtein is an edit-based algorithm that calculates how many edits it takes to get from one document to another and was used as a foundation for analysis in this project. The edit operations that the algorithm considers are insertion, deletion, and substitution (Python Software Foundation, 2022-a). Given two documents a distance matrix is computed. This is done by comparing prefixes and filling it using the following steps (Fawzy Gad, 2020-a):

1. The minimum of the three existing elements if both prefixes are identical
2. The minimum of the three existing elements + 1 if the prefixes are different

The "three existing elements" are the values in the cells above, to the left of and to the top left of the working cell. This can be seen in the example in Table 5, adapted from Fawzy Gad (2020-a), comparing "number" and "uimhir".

Table 5: Levenshtein example of 'number' and 'uimhir'

		N	**U**	**M**	**B**	**E**	**R**
	0	1	2	3	4	5	6
U	1	min(0,1,1) + 1 = **1**	min(1,1,2) = **1**	min(2,1,3) + 1 = **2**	min(3,2,4) + 1 = **3**	min(4,3,5) + 1 = **4**	min(5,4,6) + 1 = **5**

I	2	min(1,2,1) + 1 = **2**	min(1,2,1) + 1 = **2**	min(1,2,2) + 1 = **2**	min(2,2,3) + 1 = **3**	min(3,3,4) + 1 = **4**	min(4,4,5) + 1 = **5**
M	3	min(2,3,2) + 1 = **3**	min(2,3,2) + 1 = **3**	min(2,3,2) = **2**	min(2,2,3) + 1 = **3**	min(3,3,4) + 1 = **4**	min(4,4,5) + 1 = **5**
H	4	min(3,4,3) + 1 = **4**	min(3,4,3) + 1 = **4**	min(3,4,2) + 1 = **3**	min(2,3,3) + 1 = **3**	min(3,3,4) + 1 = **4**	min(4,4,5) + 1 = **5**
I	5	min(4,5,4) + 1 = **5**	min(4,5,4) + 1 = **5**	min(4,5,3) + 1 = **4**	min(3,4,3) + 1 = **4**	min(3,4,4) + 1 = **4**	min(4,4,5) + 1 = **5**
R	6	min(5,6,5) + 1 = **6**	min(5,6,5) + 1 = **6**	min(5,6,4) + 1 = **5**	min(4,5,4) + 1 = **5**	min(4,5,4) + 1 = **5**	min(4,5,5) = **4**

The final edit distance is the highlighted cell in the bottom right corner i.e., 4.

The basic algorithm can be used in Python using the TextDistance library - a "python library for comparing distance between two or more sequences by many algorithms" (Python Software Foundation, 2022-a). The algorithms available in this library are sorted into multiple categories including edit-based, token-based, and sequence-based (Python Software Foundation, 2022-a). As stated above, edit-based algorithms compute the number of edits needed to make one string equivalent to another (Mayank, 2019). In a token-based algorithm, a set of tokens is taken in instead of strings, and the number of common tokens between sets is calculated (Mayank, 2019). Sequence-based algorithms will be discussed further below.

Sequence Matching

A sequence matching algorithm aims to find "the longest contiguous matching subsequence that contains no "junk" elements", e.g., blank lines and whitespace (Python Software Foundation, 2022-b). It "sums the sizes of all matched sequences" and calculates the similarity as: "2.0*M/T, where M=matches, T=total number of elements in both sequences" (Jaiswal, 2019). It is noted that this is expensive to compute, and the result may depend on the order of the arguments (Python Software Foundation, 2022-b).

The sequence matching algorithm was deemed inappropriate for this approach, due to its computational expense, and the fact that it would place too much emphasis on the order of instructions and matching blocks of instructions, while not detecting more subtle changes. It would also be more difficult to implement hierarchical analysis using this approach.

This algorithm can be applied using the Python library difflib and the class SequenceMatcher. It compares pairs of sequences, and the sequence elements must be hashable (Python Software Foundation, 2022-b). Jaiswal (2019) states that

SequenceMatcher finds a more “human-friendly” output than the Longest Common Subsequence algorithm.

Set Comparison

Another method converts the lists into sets (Brito, 2021). This method ignores the order of the elements. The Jaccard similarity coefficient is used (Statistics How To, 2022). It computes the intersection of the sets and determines the number of elements in the intersection. It then computes the union of the sets and determines the number of elements in the union. The similarity is then calculated by dividing the size of the intersection by the size of the union.

Set comparison was not used in this approach, as the order of the instructions is disregarded when the sequence of instructions is converted to a set. This limits the ability to detect any similarities in order. It is also more difficult to implement hierarchical analysis using this approach.

ARM Assembly Language

To understand aspects of the analysis, it is necessary to have a rudimentary understanding of ARM Assembly language. In ARM processors there are 13 “general-purpose registers” for storing information (R0-R12), a stack pointer (SP), a link register (LR) and a program counter (PC) (Arm Limited, 2022). In this dissertation, R0-R12 are generally the only registers being considered.

- The instruction MOV copies a value into a specified register. For example, `MOV R0, #1` places the value 1 in register 0. MOVS does the same thing, except the S indicates that certain condition flags are updated.
- The instruction LDR loads a value into a specified register, this value is typically a value stored in another register. For example, `LDR R0, [R1]` loads the value stored in register 1 into register 0.
- The instruction STR stores a register value at a specified address, which is typically stored in another register. For example, `STR R0, [R1]` stores the value in register 0, at the address in register 1.
- The instruction ADD adds two values and places the result in a specified register. The first operand is always a register value and the second can either be a register value or an immediate value. For example, `ADD R0, R1, #1` adds the value in register 1 with the value 1, and stores the result in register 0. Sometimes `ADD R0, R1` is used which add the value in register 0 and register 1 and places the result in register 0. ADDS does the same thing, except the S indicates that certain condition flags are updated.
- The instruction SUB subtracts one value from another and places the result in a specified register. The first operand is always a register value and the second can either be a register value or an immediate value. For example, `SUB R0, R1, #1` subtracts the value 1 from the value in register 1 and stores the

result in register 0. Sometimes `SUB R0, R1` is used which subtracts the value in register 1 from the value in register 1 and places the result in register 0. SUBS does the same thing, except the S indicates that certain condition flags are updated.

- "A label in ARM assembly is simply a name given to the address of an instruction" (University of Wisconsin-Madison, 2022).

Conclusion

It is clear that plagiarism poses a problem in academia, particularly in the area of Computer Science. There are many different forms source code plagiarism can take. These different forms make it so generic plagiarism detection tools may not be able to identify certain similarities. There are numerous plagiarism detection tools available. The most notable for essay-based work is TurnItIn, and for source code is MOSS. The tool that will be used for comparison in this dissertation is Compare50. Both MOSS and Compare50 use an algorithm called winnowing to counter the computational cost of pairwise comparison. The list of languages that MOSS can analyse is very limited, so Compare50 was primarily used for comparison in this project. Different similarity measurement algorithms include edit-based, token-based and sequence-based algorithms. An edit-based algorithm formed the foundation of analysis for this project. Software that compares machine-level code, as this project plans to, could not be found.

Chapter 3: Design and Implementation

Manufactured Programs

ARM Assembly

The first step that needed to be taken was to create some "manufactured" programs to do some initial testing and analysis on. The scripts were written in ARM assembly language to observe what effect low level changes would have on the similarity scores.

The program chosen to use as the foundation for manufactured programs was a "ValToHex" program, that converts a 32-bit unsigned value to its hexadecimal ASCII string representation. This is an assignment from CSU33D01 (Waldron, 2020). This programme was chosen as it uses a variety of instructions, and it is not too complicated. It was also long enough to provide enough material to analyse, but not so long that it would take a large amount of processing power to analyse. The original solution was then artificially plagiarised several times, each with an additional method of plagiarism obfuscation to the previous one. This created a scale of similarity.

1. The first progression was produced by changing the labels within the script. A student might make these changes to make the scripts appear different while maintaining the same function and meaning. Some examples of label changes are "loop" to "MainLoop", and "num" to "Number". This change does not require any skill on the student's part.
2. The second progression had an additional change to order and structure. Instead of the various loops being listed between the start and stop labels, they are listed at the end. They are also listed in a different order. This change requires minimal skill.
3. The third progression had an additional change of LDR instructions to MOV instructions. The immediate value was also represented in integers instead of hexadecimal for the MOV instruction. Some examples of this are `LDR R7, =0xF` became `MOV R7, #15`, and `LDR R3, =0x45` became `MOV R3, #69`. Only a basic understanding of ARM assembly is required for this change.
4. The fourth progression had an additional change to the technique. Instead of scanning the 32-bit unsigned value from left to right, it scanned it from right to left. This change does require understanding of how the program functions, which indicates it would be less likely to be considered plagiarism
5. The fifth and final progression had an additional change in optimisation. The initial program was handling each letter in the hexadecimal format separately, with a different label for A, B, C, D etc. This final change created one label for all the letters and added a constant to obtain the ascii code, similar to the number label. This change does require some understanding of

how the program functions and a familiarity with ARM assembly which indicates it would be less likely to be considered plagiarism.

The result was an original solution and 5 “plagiarised” versions of it, each becoming increasingly dissimilar. A decreasing trend of similarity between these programs was expected

C

To allow testing on scripts in a different language, 3 manufactured programs were written in C.

1. The original script calculated the value of a cubic function given x (sample.c). The method was broken into individual steps in a very simple way.
2. The first progression of this script changed only the variable names and the value of x (sample1.c)
3. The second progression did all the calculation in one step (sample2.c). Because of this, it was significantly shorter than the other scripts.

Machine Code

The next step was to find a way to access the machine code of the solutions, as this formed the core body of work for analysis. There are multiple methods for extracting the machine code from compiled code. One tool is ‘objdump’, which is commonly available with compiler toolchains. ‘Objdump’ gives access to the information within object files (produced during compilation), which includes machine code.

The method chosen for this project was extraction of the machine code from the listing file. This was achieved by compiling the solutions with the GNU Arm Embedded Toolchain to obtain MASM listing files. The command used to achieve this was `arm-none-eabi-gcc -Wa,-adhln -g -O0 -c main.s > main.lst`. ‘`-Wa`’ allows options to be passed to the assembler (Free Software Foundation, 2022-b). The ‘`-adhln`’ options enable assembly code listings, omit debugging directives, include source code, include assembly, and omit anything else (Nanjappa, 2022). ‘`-g`’ enables debug information generation. ‘`-O0`’ ensures that optimisation is turned off. ‘`-c`’ ensures that the source files will be assembled/compiled but not linked (Free Software Foundation, 2022-c).

Listing files contain the line numbers, relative addresses, and machine code for each instruction, as can be seen in this excerpt from the original ValToHex listing file.

```
1               start:
2
3 0000 44119FE5           LDR r1, =0xFEED1234
4 0004 0F22A0E3           LDR r2, = 0xF0000000
5 0008 0F70A0E3           LDR r7, = 0xF
```

```
6 000c 1050A0E3            LDR r5, = 0x10
7 0010 38C19FE5            LDR r12, =0xA1000400
8 0014 0060A0E3            LDR r6, =0x0
9 0018 1C80A0E3            MOV r8, #28
10
```

A Python function was written to extract the machine code from each listing file and place it into a list so it could be analysed. Each line was split using spaces as a delimiter. The line number, relative address and instruction were discarded and only the machine code was stored.

Sequence Matcher

The first level of off-the-shelf method of analysis was sequence matching. This was implemented using the Python difflib library. The SequenceMatcher class has options for setting junk elements and allowing "autojunk". This autojunk option marks certain items as "popular" and regards them as junk, if they appear a certain number of times (Python Software Foundation, 2022-b). Neither of these options were used, and the only junk element considered was the default whitespace. The function ratio() within this class was used to obtain a measure of the sequences' similarity, with 1 corresponding to identical and 0 corresponding to completely different. Python Software Foundation cautions that it is an expensive function to compute, and the result may depend on the order of the arguments. It was implemented using the lists of machine code values for two different files as arguments.

Set Comparison

The second level of off-the-shelf analysis method was set comparison. The Python set() function was used to convert the list of machine codes for two different files into sets (W3Schools, 2022). Due to the nature of sets, the order of the elements was now irrelevant. To compute a measure of similarity, the size of the intersection of the two sets was divided by the size of the union of the two sets. Set union was computed using the Python '|' operator and set intersection was computed using the Python '&' operator. The similarity measurement ranged from 0 to 1, with 1 being identical.

Text Distance

The third level of off-the-shelf analysis method was text distance. This was implemented using the Python textdistance library. The Levenshtein class was chosen, and the distance method was used to calculate the similarity. Since this method is based on edit distance, certain calculations had to be carried out to produce a similarity measurement that ranges from 0 to 1. The source file with the greatest number of lines between the two files being compared was identified, giving a maximum file length. A ratio was calculated by dividing the edit distance by

the maximum file length. Since edit distance measures differences rather than similarities, this ratio was subtracted from 1 to get the final similarity measurement. Additionally, for convenience, it was also rounded to 2 decimal places.

Capstone

A disassembly framework called Capstone was used to perform the necessary hierarchical analysis (Capstone, 2020). Capstone takes machine code and provides information on it in a human-readable way, while supporting multiple architectures, including ARM. It "breaks down instruction information, making it straightforward to access instruction operand & other internal instruction data" (Capstone, 2020). For example, if the machine code instruction "373083E2" is passed to Capstone, Capstone can recognise that there are two instructions within this machine code. The first is an ADDS instruction, whose operands are R0 and #0x37. The second is a branch instruction, with the destination address #0x50c. The addresses of these instructions can also be extracted.

The machine code for each instruction was placed in a byte array. The hardware architecture was set to ARM, and the hardware mode was set to Thumb. The binary code was decoded using Capstone's Python class method disasm(). From this the mnemonic of the instruction, its operands and address could be extracted. The address of the first instruction was set to 0x0000. This information was stored for each decoded instruction. This decoding was used in the hierarchical analysis of the machine code, in conjunction with an adaption of the Levenshtein algorithm.

Hierarchical Analysis

This section will explain how decoding was used to perform hierarchical analysis, using an adaptation of the Levenshtein algorithm. The basic Levenshtein distance algorithm was implemented using the method outlined by Fawzy Gad (2020-b).

Table 6: Distance matrix template for demonstration

	Instruction1a	Instruction1b
Instruction2a		
Instruction2b		

A distance matrix was created (See Table 6) and initially filled with 0's. Consider the distance matrix above and the comparison of Instruction 1b and Instruction 2b. The value of the distance between them will be placed in the yellow cell. If the two instructions are identical, the edit distance is the same as the element to its top-left (green cell) and no additional distance is added. If the instructions are different, the distances for insertion, deletion and edit are calculated. The final distance to be placed in the yellow cell is the minimum of these three values.

The distance for an insertion is the element directly above (blue cell), plus 1. The distance for a deletion is the element directly to the left (red cell), plus 1. The edit distance is the element to its top left (green cell) plus the additional distance calculated. In the basic algorithm, the additional distance is considered 1, similar to insertion and deletion. However, since this project is considering non-binary comparison, the additional distance will range from 0 to 1.

To calculate the additional distance for edit distance, Capstone was used to decode the instructions and calculate their similarity based on the mnemonics, operands, and addresses. This was done in 3 stages.

Level 1

The first level of hierarchical analysis considered was of the most basic instructions: MOVS, SUBS and ADDS, i.e, moving values into registers, subtraction, and addition. The similarities considered for these instructions can be seen in Tables 7, 8 and 9.

Table 7: Similarities for individual changes within a MOVS instruction

	MOVS[R1,R2]	**Change**
MOVS[R2,R1]	0.60	Value in R1 placed in R2, instead of value in R2 placed in R1. Similar in that both are MOVS instructions, and the same registers are used.
MOVS[R1,R3]	0.75	Value is being placed in R1 in both instructions, but source register is different. Quite similar.
MOVS[R3,R2]	0.75	The same value is being placed in both instructions, but destination register is different. Quite similar.
MOVS[R2,R3]	0.55	Both use register R2 but one as a source register and the other as a destination register. Slightly similar.
MOVS[R3,R1]	0.55	Both use register R1 but one as a destination register and the other as a source register. Slightly similar.
MOVS[R3,R4]	0.50	Different source and destination registers used. Only similarity is that the same mnemonic is used.

Table 8: Similarities for individual changes within a SUBS instruction

	SUBS[R1,R2]	**Change**
SUBS[R2,R1]	0.60	Value in R1 is subtracted from that in R2, instead of value in R2 subtracted by that in

		R1. Similar in that both are SUBS instructions, and the same registers are used.
SUBS[R1,R3]	0.75	Value in R1 is being subtracted from in both instructions, but value being subtracted is different. Quite similar.
SUBS[R3,R2]	0.65	The same value is being subtracted in both instructions, but value being subtracted from, as well as the destination register, is different. Less similar than case above.
SUBS[R2,R3]	0.55	Value in register R2 is being used in both instructions, but one as the value being subtracted from and the other as the value being subtracted. Slightly similar.
SUBS[R3,R1]	0.55	Value in register R1 is being used in both instructions, but one as the value being subtracted and the other as the value being subtracted from. Slightly similar.
SUBS[R3,R4]	0.50	Different value being subtracted and subtracted from. Only similarity is that the same mnemonic is used.

Table 9: Similarities for individual changes within an ADDS instruction

	ADDS[R1,R2]	**Change**
ADDS[R2,R1]	0.90	Values from the same registers are being added, producing the same result as addition is commutative; but a different destination register is being used, hence the slight difference.
ADDS[R1,R3]	0.75	A different register value is being added but the other operand remains the same, as well as the destination register. Quite similar.
ADDS[R3,R2]	0.70	The same register value is being added but the other operand and the destination register differs. Similar.
ADDS[R2,R3]	0.65	The same register value is being added but the other operand and the destination register differs. Less similar as R2 has changed from the source register to the destination register.
ADDS[R3,R1]	0.65	The same register value is being added but the other operand and the destination register differs. Less similar as R1 has

		changed from the destination register to the source register.
ADDS[R3,R4]	0.50	Different register values being used. Only similarity is that the same mnemonic is used.

The values in Tables 7, 8 and 9 represent the similarity measurement between two instructions. These values are then subtracted from 1 to calculate the additional distance to be added to the edit distance for the Levenshtein algorithm.

Level 2

The second level of hierarchical analysis considered was that of Branch instructions. In ARM, branch instructions break the "sequential flow of instructions" and fall into two categories: relative and absolute (Bramley, 2013). They essentially redirect the processor to another location in the program. With a relative branch, the target address is calculated based on the value of the current program counter, and the target address of an absolute branch remains the same regardless of the current program counter (Bramley, 2013). Relative branches are the more commonly used, as they are position independent (Bramley, 2013). This is ideal for this project, as we are comparing relative branch distance, instead of absolute target addresses.

It was discovered that when Capstone decoded certain machine code instructions, two mnemonics were produced, the second of which was a 'B' for branch. Capstone also produced the address of the instruction itself as well the address of where it was branching to. From this the branch distance for each instruction was calculated, by getting the absolute value of the current address minus the destination address.

It was decided that branch instructions would have a default additional distance of 0.75 (i.e., a similarity of 0.25), just for having the same mnemonic. The absolute value of the difference between the two instructions' branch distances was then calculated. If this was less than 100, the difference was then considered as a percentage. This percentage was then converted to a ratio of 0.75 and subtracted from the default value to get the additional distance. If the difference between the branch distances was zero, the instructions would be considered identical, and no additional distance would be added.

This can be summarised in the pseudo-code below:

```
If both instructions are branch instructions:

      AdditionalDistance = 0.75 <- the default

      BranchDistance1 = absolute value of (address of 1st branch
      instruction – destination address of 1st branch
      instruction)
```

```
BranchDistance2 = absolute value of (address of 2nd branch
instruction – destination address of 2nd branch
instruction)

BranchDistanceDifference = absolute value of
(BranchDistance1 – BranchDistance2)

If BranchDistanceDifference is between 0 and 100:

     PercentDistance = BranchDistanceDifference/100

     AdditionalDistance = 0.75 – (75 x
     (PercentDistance)/100)

If BranchDistance is exactly 0:

     AdditionalDistance = 0
```

Given that the additional distance for a branch instruction was always the second half of a machine code instruction, the overall distance for the machine code instruction had to calculated. This was done by halving the additional distance of the first instruction and the additional distance of the branch instruction and adding them together. This final value was then used as the additional distance for edit distance in the Levenshtein matrix.

Level 3

The third level of hierarchical analysis considered was that of Store instructions, i.e. instructions with the prefix "STR". The calculation of additional distance for these instructions is different from the methods described above. In this case a running total was kept. The default additional distance for instructions with this prefix was set at 0.75, and when a further similarity was found, a value was then subtracted.

Within the STR prefix group instruction pairs where one is "STRH" and the other is "STRB" were considered. These instructions are very similar, the only difference is that STRH stores halfwords and STRB stores bytes. Because of this, before looking any further into the operands, 0.1 is subtracted from the running total additional distance.

Within the group of pairs of STRH and STRB the operands are then considered.

Table 10: Corresponding change in edit distance for changes in instruction - STRH and STRB instructions

Type	**Subtract from running total additional distance**

The destination registers are the same and source registers are the same.	-0.3
The destination register of one is the source register of the other, and vice versa.	-0.2
The destinations registers are the same, but the source registers are different.	-0.1
The source registers are the same, but the destination registers are different.	-0.1

The next group with the STR prefix group to be considered were ones where the instruction group is exactly the same, i.e. both STR, both STRH etc. The running total additional distance is reset to 0.5. Within this group, pairs of instruction either both with STRH or both with STRB as mnemonics were considered.

Table 11: Corresponding change in edit distance for changes in instruction-identical mnemonic, STRH and STRB

Type	**Subtract from running total additional distance**
The destination register of one is the source register of the other, and vice versa.	-0.2
The destinations registers are the same, but the source registers are different.	-0.1
The source registers are the same, but the destination registers are different.	-0.1

Also within the group with identical mnemonics, pure STR instructions were considered. STR instructions have 3 operands and sometimes the last operand is an immediate value instead of a register.

Table 12: Corresponding change in edit distance for changes in instruction-identical mnemonic, STR

Type	**Subtract from running total additional distance**
The first operands are the same but the second operand of one is the third of the other and vice versa.	-0.2
The first operands are different but the second operand of one is the third of the other and vice versa.	-0.1
The second operands are the same and the first operand of one is the third of the other and vice versa.	-0.2
The second operands are different but the first operand of one is the third of the other and vice versa.	-0.1

The third operands are the same and the first operand of one is the second of the other and vice versa.	-0.2
The third operands are different but the first operand of one is the second of the other and vice versa.	-0.1
The first operand of one is the second of the other, the second operand of one is the third of the other and the third operand of one is the first of the other.	-0.1
The third operands of both are immediate values.	-0.1
The third operands of both are immediate values and the first and second operands are the same.	-0.3

Combined Levenshtein

The final approach for plagiarism detection will henceforth be referred to as "Combined Levenshtein". This approach uses the machine code for each solution as a fingerprint. The machine code for each assignment is compared pairwise using Python's textdistance library, the Levenshtein class, and the distance method. The distance produced is then converted to a similarity measurement by dividing it by the length of the longer solution in the pair. If the similarity between a pair of solutions is greater than or equal to 30%, hierarchical analysis is implemented. If the similarity measurement of the hierarchical analysis is greater than 50%, the pair is then flagged as having a high similarity. Those flagged should be further investigated by a human to identify if plagiarism has occurred or not.

Chapter 4: Evaluation

Methods Used

The machine code of files was compared using the following methods:

1. The SequenceMatcher class of the Python difflib library – referred to as Sequence Matcher
2. Set comparison – referred to as Sets
3. The Levenshtein class of the Python textdistance library – referred to as TextDistance Basic
4. Custom Levenshtein algorithm to perform hierarchical analysis using Capstone – referred to as Levenshtein Hierarchy
5. Hierarchical analysis combined with the Levenshtein class of the Python textdistance library to compare fingerprints – referred to as Combined Levenshtein.

The original programs (not the machine code as was the case with the methods above) were also compared via Compare50. It produces a similarity measure for each file in a pair being compared. To obtain one number for comparison, the similarity of the longer file was taken, like how the similarity in Levenshtein was calculated using the longer file to form a ratio. If the two files were of the same length the higher similarity was taken. This will be referred to as Compare50

ValToHex files

The first set of experiments was run on the ValToHex files, i.e. the original solution and 5 solutions with increasing numbers of changes from the original, summarised in Table 13.

Table 13: Summary of differences between ValToHex files

File 1	File 2	Shorthand	Differences between File 1 and File 2
ValToHex	ValToHex_Progression1	Origin+Prog1	Change in labels
ValToHex	ValToHex_Progression2	Origin+Prog2	Change in labels, order & structure
ValToHex	ValToHex_Progression3	Origin+Prog3	Change in labels, order & structure, LDR to MOV
ValToHex	ValToHex_Progression4	Origin+Prog4	Change in labels, order & structure, LDR to MOV, technique
ValToHex	ValToHex_Progression5	Origin+Prog5	Change in labels, order & structure, LDR to MOV, technique, optimisation

This produced the results in Table 14:

Table 14: Results of automated measures of similarity on ValToHex files

	Sequence Matcher	Sets	TextDistance Basic	Levenshtein Hierarchy	Combined Levenshtein	Compare 50
Origin+ Prog1	1.00	1.00	1.00	1.00	1.00	1.00
Origin+ Prog2	0.19	0.42	0.49	0.61	0.61	1.00
Origin+ Prog3	0.19	0.42	0.49	0.61	0.61	0.91
Origin+ Prog4	0.13	0.33	0.31	0.55	0.55	0.92
Origin+ Prog5	0.16	0.17	0.13	0.23	0.13	0.78
Prog1+ Prog2	0.19	0.42	0.49	0.61	0.61	1.00
Prog1+ Prog3	0.19	0.42	0.49	0.61	0.61	0.91
Prog1+ Prog4	0.13	0.33	0.31	0.55	0.55	0.92
Prog1+ Prog5	0.16	0.17	0.13	0.23	0.13	0.77
Prog2+ Prog3	1.00	1.00	1.00	1.00	1.00	0.91
Prog2+ Prog4	0.80	0.82	0.80	0.93	0.93	0.92
Prog2+ Prog5	0.27	0.18	0.18	0.26	0.18	0.77
Prog3+ Prog4	0.80	0.82	0.80	0.93	0.93	1.00
Prog3+ Prog5	0.27	0.18	0.18	0.26	0.18	0.91
Prog4+ Prog5	0.39	0.28	0.27	0.29	0.27	0.91

At first glance it is obvious that Compare50 has, overall, a higher similarity rating when compared to the other methods. The Levenshtein Hierarchy and Combined Levenshtein methods have higher similarities than Sequence Matcher, Sets and TextDistance Basic but not as high as Compare50.

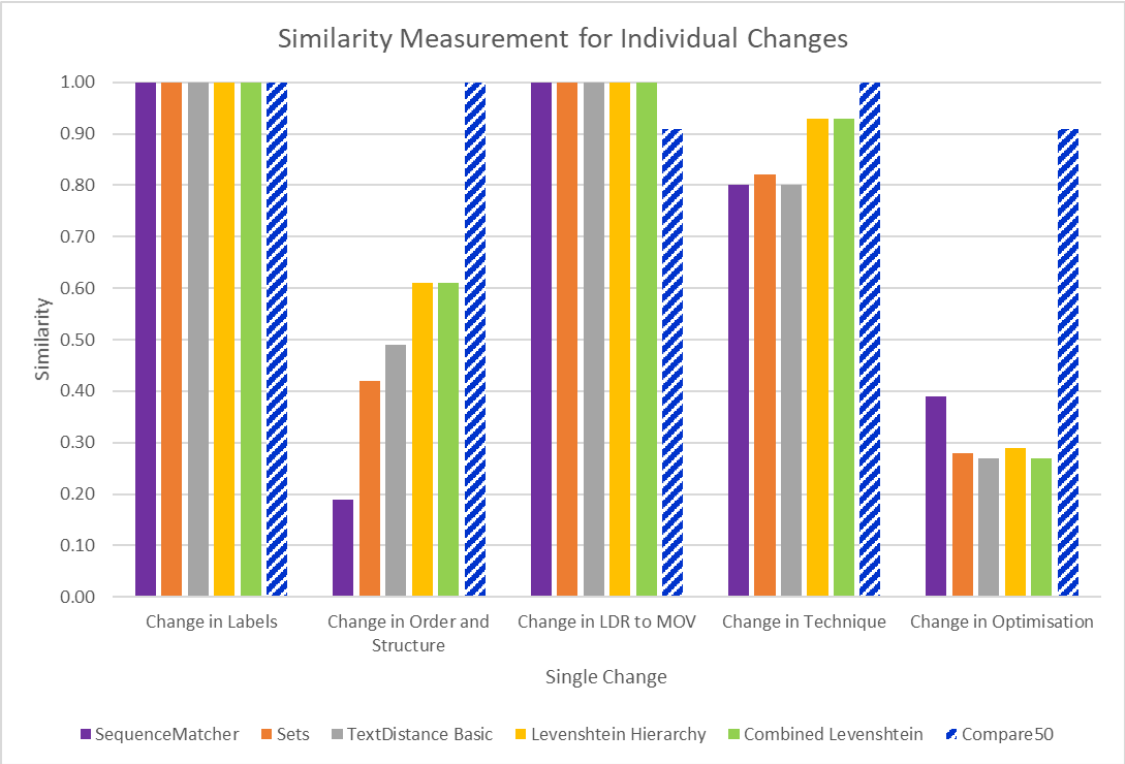

Figure 2: Similarity Measurements for Individual Changes to ValToHex Files

From this data one can extract what effect individual changes had on the similarity scores, as seen in Figure 2. None of the similarity measurements viewed the change in labels as a real change. If a student used this method as a way to plagiarise, it would be immediately detected by all tools.

The change to order and structure caused a varied change in similarity. Sequence Matcher produced quite a low similarity score, as one would expect from its algorithm. It detects long matching sequences and if the order and structure is changed, so do the sequences available for detection. Sets produced a slightly higher measurement of similarity, presumably because sets do not place any value on the order of elements. The TextDistance Basic method produced a slightly higher similarity.

Levenshtein Hierarchy and Combined Levenshtein produced even higher measurements for change to order and structure. But standing apart from the others was Compare50 which gave a 100% similarity measurement, regardless of any change to the order and structure. While this change would be detected by Combined Levenshtein, it would not be detected as clearly as by Compare50.

In contrast, none of the tools except Compare50 detected the change from LDR to MOV as a real change. This can be accounted for by the fact that the other tools compared files at a machine code level which, from the evidence, did not differentiate between LDR and MOV. Although it did not find 100% similarity with

this change, Compare50 found 91% similarity which is high enough to alert for potential plagiarism.

The change in technique produced a range of similarity measurements. Compare50 regarded this as no real change, with Levenshtein Hierarchy and Combined Levenshtein regarding it as 93% similar. Sequence Matcher, Sets and TextDistance Basic were all approximately 10% lower in their similarity scores. For such a large change in how the problem was approached (i.e. scanning the input value from right to left instead of from left to right), it was surprising how high the similarity scores were for all methods of comparison. This is an example of when human inspection is necessary for plagiarism detection tools.

The final change in optimisation produced another range of similarities. Again, Compare50 stood apart, giving a similarity measurement of 91%. Sequence Matcher unexpectedly produced the second highest measurement, albeit only giving 39% similarity. All the other tools were in the 20-30% range. Like with the change in technique, the Compare50 result for such a large change seems high, as the change produces quite a different looking program.

These results indicate that Combined Levenshtein follows the same trend as Compare50 for most of the individual changes, although Compare50 generally produced higher similarity scores than the other tools.

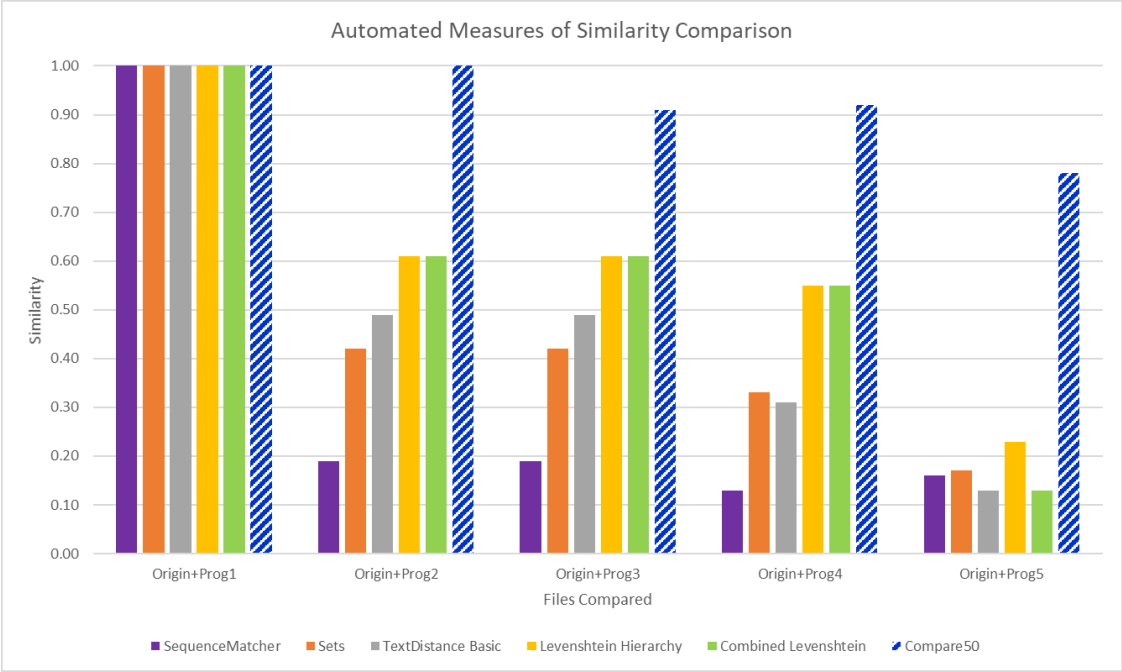

Figure 3: Bar Chart of Measures of Similarity for ValToHex Files

Figure 3 shows how all the artificially plagiarised solutions compared with the original. As expected, the more changes applied to the original solution, the more the similarity decreases. It is observed once more how Compare50 produces an overall higher measure of similarity than the other tools. This may indicate that its

threshold for indicating plagiarism is also higher. Combined Levenshtein seems to mimic the trend of identical to quite dissimilar a bit more clearly than Compare50. As expected, Combined Levenshtein and Levenshtein Hierarchy have overall higher measures of similarity than the more basic methods of TextDistance Basic, Sets and Sequence Matcher.

C Files

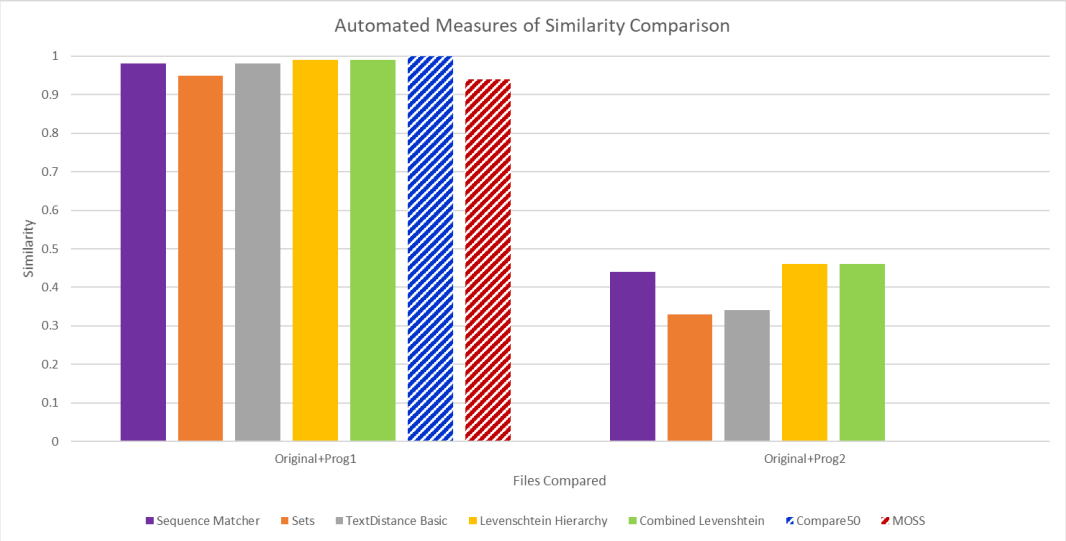

Figure 4: Bar Chart of Measures of Similarity for C Files

A second, similar but smaller, set of experiments was run on the C files. This was to ensure that the tool worked regardless of the language, and that it enabled a comparison with MOSS which works with C but not with Arm Assembly Language. Again, all the tools except MOSS and Compare50 were run on the machine code of the source files. MOSS and Compare50 were run on the C files themselves.

All tools found the original script to have upwards of 90% similarity with the first progression. The fact that the variable names had been changed and one of the variable values changed also, made little to no difference.

However, Neither MOSS nor Compare50 found any matches between the original C file and the second progression, which had the same function except carried out in one step instead of being broken down into multiple steps. In this case Combined Levenshtein performed better in identifying similarity between the files, however as the similarity was under 50%, it did not seem high enough to be flagged as potential plagiarism.

Corpus

Development of Corpus

The initial goal was to obtain a corpus with varying programming languages with solutions to an assignment in three different Computer Science modules. This was unfeasible due to circumstances beyond our control, therefore 3 assignments from Introduction to Computing 1 (CSU11021) were chosen instead. These assignments were called Prime Numbers, Set Union and lowerCamelCase. The solutions to these assignments were written in ARM Assembly language.

To ensure anonymity, a Qualtrics survey was created with a link to a Dropbox. The Qualtrics survey verified the students' consent and produced a random number for each submission that they could use in place of their names when submitting the files to Dropbox. An email address was also provided for them to use, instead of their personal one. A separate Qualtrics survey was created for each assignment. The students who were enrolled in this module were e-mailed with the aid of the Supervisor for this project. This allowed the creation of a realistic but anonymous corpus for testing.

Thirteen of the solutions were for an assignment called Prime Numbers, where the task was to write a program that counts the number of prime numbers that are less than a specified integer in register R1 and places the result in register R0. Some pseudocode and hints were provided. This may have resulted in a slightly higher similarity.

Ten of the solutions were to an assignment called Set Union. The task here was to write a program that would take the sets at the addresses in R1 and R2 and store their union at the address in R0. No pseudocode or hints were provided. The solutions are expected to have a lower similarity than the prime number assignments due to this.

Eight of the solutions were to an assignment called lowerCamelCase. Students were asked to write a program to take an ASCII string stored at the address in R1, convert it to lowerCamelCase and store in at the address in R0. No pseudocode or hints were provided so a lower similarity than Prime Numbers is also expected.

The Lecturer's solutions to the three assignments were also added to the corpus.

In order to test the tool, several artificial instances of plagiarism obfuscation were inserted into the corpus. Solution "580 – lowerCamelCase.s" was chosen. This program was artificially plagiarised in multiple ways:

1. Every instance of R2 was changed to R3
2. ASCII characters were changed to their ASCII code e.g. `CMP R2, #'A'` changed to `CMP R2, #0x41`

3. Comments were changed

The first artificially plagiarised solution ("580P1 – lowerCamelCase") had change no. 1 implemented.

The second artificially plagiarised solution ("580P2 – lowerCamelCase") had change no. 2 implemented.

The third artificially plagiarised solution ("580P3 – lowerCamelCase") had change no. 3 implemented.

The fourth artificially plagiarised solution ("580P4 – lowerCamelCase") had change no. 1 and 2 implemented.

The fifth artificially plagiarised solution ("580P5 – lowerCamelCase") had change no. 1 and 3 implemented.

The sixth artificially plagiarised solution ("580P6 – lowerCamelCase") had change no. 2 and 3 implemented.

The seventh artificially plagiarised solution ("580P7 – lowerCamelCase") had change no. 1, 2 and 3 implemented.

The final corpus with submitted solutions, lecturer's solutions and artificially plagiarised solutions contained 41 files.

Experimentation

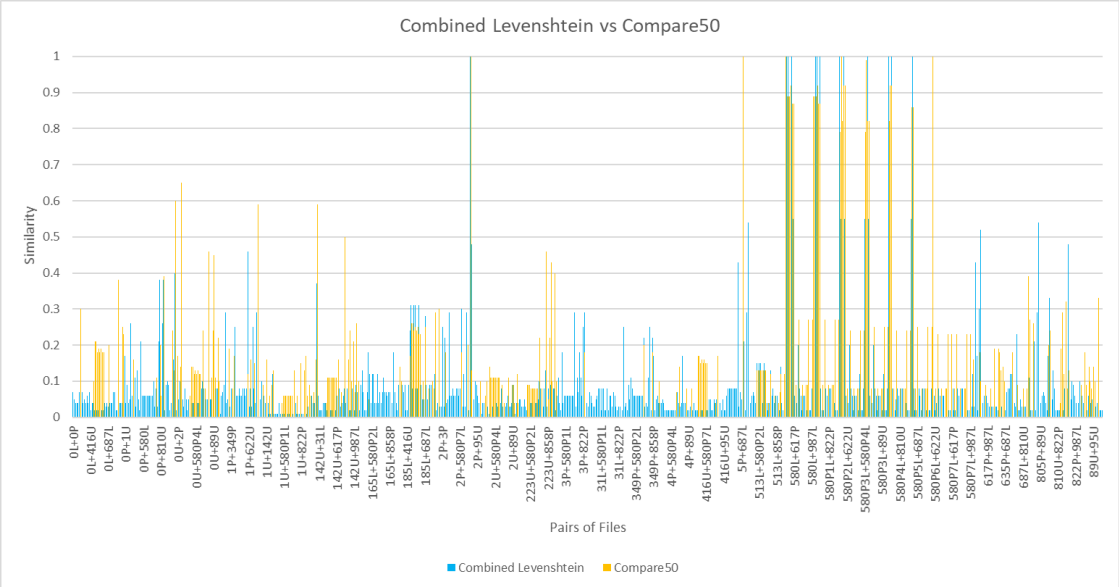

Figure 5: Bar Chart of Measures of Similarity for Corpus

Combined Levenshtein and Compare50 were applied to the corpus of solutions, with Combined Levenshtein being applied to the machine code of the files and

Compare50 being applied to the files themselves. A similarity score for every file pair was produced by Combined Levenshtein, and Compare50 produced similarity scores for 310 matches. This is represented in Figure 5. Of note is that both tools follow the same overall trend. There are a small number of instances that Combined Levenshtein spikes and Compare50 doesn't spike and vice versa.

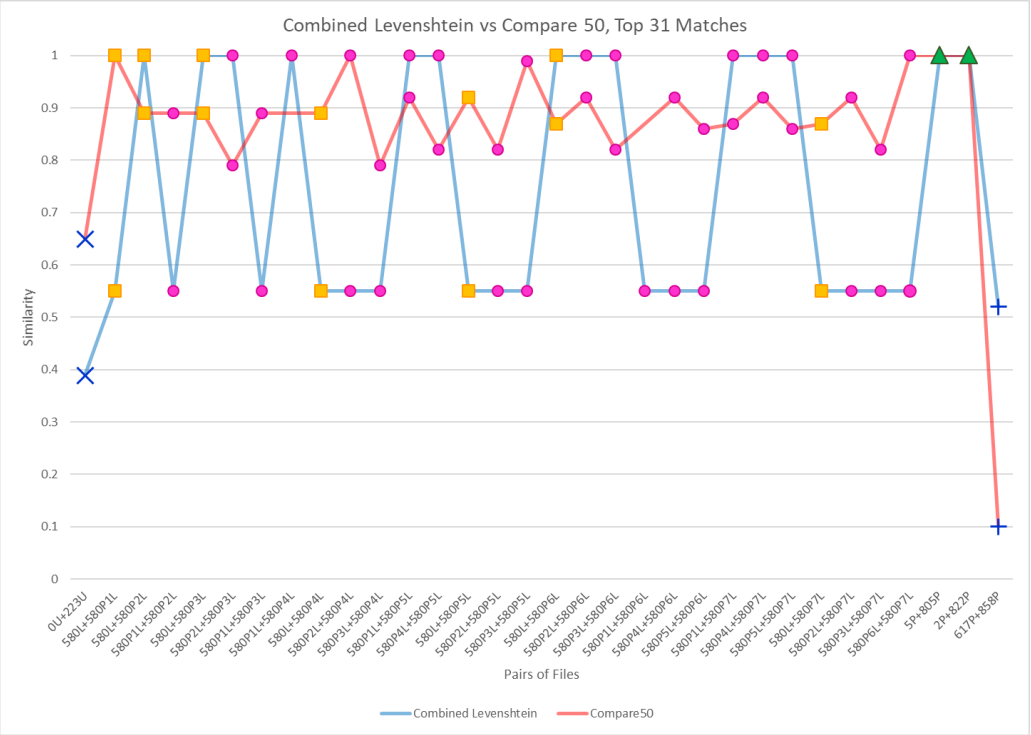

Figure 6: Graph of Top 31 Measures of Similarity for Combined Levenshtein and Compare50 in Corpus

Combined Levenshtein identified 31 pairs of files with 50% or over similarity. The top 31 Combined Levenshtein similarities and the top 31 Compare50 matches are graphed in Figure 6. Both tools picked up on the artificially plagiarised files, as can be seen in Table 15. The squares on the graph indicate the pairs of the original file and the plagiarised files. Other resemblances that were picked up on are similarities between the plagiarised files themselves. This is to be expected and they are marked as circles. These are all true positives.

Table 15: Measure of similarity for artificially plagiarised files in corpus

File 1	File 2	Combined Levenshtein	Compare 50	No. of changes from original
580-lowerCamelCase	580P1-lowerCamelCase	0.55	1.00	1
580-lowerCamelCase	580P2-lowerCamelCase	1.00	0.89	1

580-lowerCamelCase	580P3-lowerCamelCase	1.00	0.89	1
580-lowerCamelCase	580P4-lowerCamelCase	0.55	0.89	2
580-lowerCamelCase	580P5-lowerCamelCase	0.55	0.92	2
580-lowerCamelCase	580P6-lowerCamelCase	1.00	0.87	2
580-lowerCamelCase	580P7-lowerCamelCase	0.55	0.87	3

Two other matches identified by both Combined Levenshtein and Compare50 are flagged with 100% similarity. These are marked as triangles in Figure 5. The pairs of files are 5-primes and 805-primes, and 2-primes and 822-primes. Upon inspection these pairs do appear to be identical and as such are considered spontaneously occurring instances of copy-paste plagiarism.

The final 2 pairs in this set are 0-union and 233-union (marked as 'X'), and 617-primes and 858-primes (marked as '+'). The first pair (referred to as 0U+233U) is not flagged as potential plagiarism by Combined Levenshtein, as it only had a similarity of 39%. It was however the 10th match for Compare50, with a similarity of 65%. It should be noted that 0-union is the lecturer's solution to the assignment.

Upon inspection, a block of 21 lines of code seems to have been copied and pasted. Some of the other similarities Compare50 detected were blocks of comment, which upon inspection of other solutions to the same assignment, seem to be a generic start and end to the assignment. Despite this, the blatant copy and paste of 21 lines of code from a lecturer's solution indicates plagiarism and this should be considered a false negative for Combined Levenshtein. This is perhaps a result of 50% being too high a lower limit for plagiarism flagging.

The second pair (referred to as 617P+858P) is flagged as potential plagiarism by Combined Levenshtein, with a similarity of 52%. It was however Compare50's 145th match, with a similarity of 10%. Upon inspection, these two files do appear quite similar. The comments have changed, but the same instructions are used, in a slightly different order, but enough that the function remains the same. A few instructions have been changed inconsequentially e.g. `ADD R2, #1` changed to `ADD R2, R2, #1`.

This assignment was however one of the assignments for which pseudocode was provided, indicating that there might be a higher level of similarity. The other two pairs flagged for this assignment were identical copies of each other. The average similarity of pairs of solutions to the prime number assignment was ~20% for Combined Levenshtein and ~8% for Compare50. The similarity detected is above

average for both Combined Levenshtein and Compare50, therefore this can also be considered a true positive.

There appear to be no false positives for Combined Levenshtein as there weren't any pairs of solutions to different assignments which were flagged as high similarity e.g. a solution to the Set Union assignment paired with a solution to the lowerCamelCase assignment. In fact for Combined Levenshtein, the highest similarity measurement for a pair of solutions to different assignments was 13%. In contrast, the equivalent percentage for Compare50 was 39%.

Chapter 5: Conclusion

This project achieved its goal. A method was developed for detecting plagiarism at a machine code level. The machine code was used as a fingerprint for the original program and was compared against other programs using Python's textdistance library and Levenshtein class. If the similarity was found to be greater than or equal to 30%, a hierarchical analysis was performed on the machine code. An adapted Levenshtein algorithm was used to calculate a non-binary similarity measurement between instructions. This approach was compared with a contemporary plagiarism detection tool called Compare50. Both the developed method (referred to as Combined Levenshtein) and Compare50 were run on a corpus of student programming assignments, with several instances of artificial plagiarism inserted. The threshold for Combined Levenshtein was set to 50%. In this experiment Combined Levenshtein had 29 true positives, 1 false negative and 0 false positives.

Future Work

Further analysis could be done on the comparison between Combined Levenshtein and Compare50 using a significantly larger corpus. The corpus in this project included 41 files (including artificial instances of plagiarism). A larger corpus would be able to confirm the results of this dissertation and allow for further in-depth comparisons. It would also allow for a more educated decision on what the threshold should be when flagging potential instances of plagiarism

Ideally a more expansive corpus with programs written in multiple languages would be used to develop this research further. The corpus in this dissertation was limited to student programming assignments written in ARM Assembly Language. A more expansive corpus would confirm the suitability of this approach for other languages.

The non-binary measures of similarity between instructions could also be revisited. The values assigned to these similarity measurements were not arbitrary, but there is scope to investigate this further, which would provide an opportunity for more precise definitions. It could also be investigated what effect these values had on the overall results.

Algorithms other than Levenshtein could be used in future work to perform a non-binary measure of similarity when performing hierarchical analysis. Levenshtein was chosen for this project; however there are many other edit-based algorithms that could be implemented in its place, for example Hamming, Jaro-Winkler and Smith-Waterman (Python Software Foundation, 2022-a). Token based algorithms could also be implemented and investigated.

There are also more efficient ways to access the machine code of compiled programs. The method of compiling, accessing the listing file and extracting the machine code via a Python script was not the most efficient, especially if larger groups of files are to be analysed. The tool 'objdump' could be used to extract the machine code from the object file, as object files are automatically created during compilation. The machine code could also be accessed directly within memory, although this could prove more difficult.

A method for optimising fingerprinting with less computational cost could be investigated. A second pass of fingerprinting using winnowing could also prove to be an efficient method.

Finally, this method could be broadened to include an approach for interpreted languages. Both JavaScript (Adhikary, 2020) and Python (Bagheri, 2020) produce bytecode when run. This could perhaps be extracted and analysed in a similar way.

Closing Remark

In conclusion, it is possible to implement measures of similarity using machine code. It is a viable method of plagiarism detection that is also language independent. The results of this project are promising and there are many areas for future work.

Bibliography

Adhikary, T. (2020). *JavaScript Interpreted or Compiled? The Debate is Over*. Available at: https://blog.greenroots.info/javascript-interpreted-or-compiled-the-debate-is-over [Accessed 30 July 2022]

Aiken, A. (2021) *Moss: A System for Detecting Software Similarity*. Available at: https://theory.stanford.edu/~aiken/moss/ [Accessed 28 January 2022]

Arm Limited (2022). *ARM registers*. Available at: https://developer.arm.com/documentation/dui0473/m/overview-of-the-arm-architecture/arm-registers [Accessed 30 July 2022]

Bagheri, R. (2020). *Understanding Python Bytecode*. Available at: https://towardsdatascience.com/understanding-python-bytecode-e7edaae8734d [Accessed 30 July 2022]

Barrett, R., Cox, A.L. (2005). 'At least they are learning something: the hazy line between collaboration and collusion'. *Assessment & Evaluation in Higher Education*, 30(2), pp. 107–122.

Bishop, H. (2021). *Plagiarism in Online Classes*. Available at: https://clas.ucdenver.edu/harvey-bishop/2021/01/19/plagiarism-online-classes [Accessed 28 July 2022]

Bramley, J. (2013). *Branch and Call Sequences Explained*. Available at: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/branch-and-call-sequences-explained [Accessed 28 July 2022]

Brito, M. (2021). *The Best Ways to Compare Two Lists in Python*. Available at: https://miguendes.me/python-compare-lists [Accessed 28 July 2022]

Cahill, S. (2022). 'Reported Plagiarism Nearly Triples Pre-Pandemic Levels', *University Times*, 10 May. Available at: https://universitytimes.ie/2022/05/reported-plagiarism-nearly-triples-pre-pandemic-levels/ [Accessed 28 July 2022]

Cambridge University Press (2022). *Cambridge Dictionary.* Available at: https://dictionary.cambridge.org/dictionary/english/plagiarism [Accessed 28 January 2022]

Capstone (2020). *Capstone The Ultimate Disassembler*. Available at: https://www.capstone-engine.org/ [Accessed 28 July 2022]

Compare50 (n.d.). *compare50*. Available at: https://cs50.readthedocs.io/projects/compare50/en/latest/ [Accessed 28 July 2022]

Dey, S. K., Sobhan, M. A. (2006) 'Impact of Unethical Practices of Plagiarism on Learning, Teaching and Research in Higher Education: Some Combating Strategies', in *2006 7th International Conference on Information Technology Based Higher Education and Training*. Ultimo, Australia, 10-13 July 2006, pp. 388-393. Available at: https://doi.org/10.1109/ITHET.2006.339791 [Accessed 28 July 2022]

Fawzy Gad, A. (2020-a). *Measuring Text Similarity Using the Levenshtein Distance*. Available at https://blog.paperspace.com/measuring-text-similarity-using-levenshtein-distance/ [Accessed 28 July 2022]

Fawzy Gad, A. (2020-b). *Implementing The Levenshtein Distance for Word Autocompletion and Autocorrection*. Available at https://blog.paperspace.com/implementing-levenshtein-distance-word-autocomplete-autocorrect/ [Accessed 28 July 2022]

Fraser R. (2014) 'Collaboration, Collusion and Plagiarism in Computer Science Coursework', *Informatics in Education,* 13(2), pp. 179-195

Free Software Foundation (2022-a). *3.11 Options That Control Optimization*. Available at: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options [Accessed 28 July 2022]

Free Software Foundation (2022-b). *3.14 Passing Options to the Assembler*. Available at: https://gcc.gnu.org/onlinedocs/gcc/Assembler-Options.html#Assembler-Options [Accessed 28 July 2022]

Free Software Foundation (2022-c). *3.2 Options Controlling the Kind of Output*. Available at: https://gcc.gnu.org/onlinedocs/gcc/Overall-Options.html#Overall-Options [Accessed 28 July 2022]

GeneratePlus. (2022). *Hash String*. Available at: https://generate.plus/en/hash [Accessed 28 July 2022]

Grammarly. (2022). *Plagiarism Checker by Grammarly*. Available at: https://www.grammarly.com/plagiarism-checker [Accessed 28 July 2022]

Granzer, W., Praus, F., Balog, P. (2013). 'Source Code Plagiarism in Computer Engineering Courses'. *Journal on Systemics Cybernetics and Informatics*, 11(6), pp. 22-26

Hemminger, D. (2016). *Machine Language*. Available at: https://www.britannica.com/technology/machine-language [Accessed 28 July 2022]

Herold, B. (2021) *How to Detect Code Plagiarism*. Available at: https://medium.com/codex/how-to-detect-code-plagiarism-de147cb4f421 [Accessed 28 January 2022]

Howard, R. (2001). 'Forget About Policing Plagiarism. Just Teach', *The Chronicle of Higher Education*, 16 November. Available at: https://www.chronicle.com/article/forget-about-policing-plagiarism-just-teach/ [Accessed 28 July 2022]

Issuu, *codequirychecker.* Available at: https://issuu.com/codequirychecker [Accessed 28 January 2022]

Jaiswal, N. (2019). *SequenceMatcher in Python*. Available at: https://towardsdatascience.com/sequencematcher-in-python-6b1e6f3915fc [Accessed 28 July 2022]

Jereb, E., Urh, M., Jerebic, J., Šprajc, P. (2018) 'Gender differences and the awareness of plagiarism in higher education'. *Social Psychology of Education: An International Journal*, 21(2), pp. 409-426.

Lancaster, T. Culwin, F. (2004). 'Using freely available tools to produce a partially automated plagiarism detection process', in *Beyond the comfort zone: Proceedings of the 21st ASCILITE Conference*. Perth, Australia, 5-8 December 2004, pp. 520-529, Available at: https://www.ascilite.org/conferences/perth04/procs/lancaster.html [Accessed 28 January 2022]

Mayank, M. (2019). *String similarity – the basic know your algorithms guide!*. Available at: https://itnext.io/string-similarity-the-basic-know-your-algorithms-guide-3de3d7346227 [Accessed 28 July 2022]

Nanjapp, A. (2019). *How to generate assembly code using gcc*. Available at: https://codeyarns.com/tech/2019-12-30-how-to-generate-assembly-code-using-gcc.html#gsc.tab=0 [Accessed 28 July 2022]

Naude, E. J., Hörne, T. (2006). 'Cheating or 'Collaborative Work': Does it Pay?'. *Issues in Informing Science and Information Technology*, 3(2006), pp. 459-466.

Peveler, M., Gurjar, T., Maicus, E., Aikens, A., Christoforides, A., Cutler, B. (2019). 'Lichen: Customizable, Open Source Plagiarism Detection in Submitty', in *SIGCSE '19: Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. Online, 22 February 2019, pp 1270, Available at: https://doi.org/10.1145/3287324.3293867 [Accessed 28 January 2022]

Prechelt, L., Malpohl, G., Philippsen, M. (2002). 'Finding Plagiarisms among a Set of Programs with JPlag'. *Journal of Universal Computer Science*, 8(11), pp 1016-1038.

Python Software Foundation. (2022-a). *textdistance 4.3.0*. Available at: https://pypi.org/project/textdistance/ [Accessed 28 July 2022]

Python Software Foundation. (2022-b). *difflib*. Available at: https://docs.python.org/3/library/difflib.html [Accessed 28 July 2022]

Roberts, E. (2002). 'Strategies for promoting academic integrity in CS courses'. *Frontiers in Education*, 3, F3G14–19

Sabin, R. E., Sabin, E.P. (1994). 'Collaborative learning in an introductory computer science course'. *ACM SIGCSE Bulletin*, 26(1), pp. 304-308.

Schleimer, S., Shawcross Wilkerson, D., Aiken, A. (2003). 'Winnowing: Local algorithms for document fingerprinting', in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data.* San Diego, California, USA, 9-12 June 2003 pp. 76-85, Available at: https://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf [Accessed 28 January 2022]

Simatupang et al. (2021). 'The Plagiarism Tendency During Covid-19 Pandemic'. *Turkish Journal of Computer and Mathematics Education,* 12(14), pp 4600-4607.

Statistics How To. (2022). *Jaccard Index/Similarity Coefficient*. Available at: https://www.statisticshowto.com/jaccard-index/ [Accessed 28 July 2022]

Uberoi, A. (2019). *Spelling Correction using K-Gram Overlap*. Available at: https://www.geeksforgeeks.org/spelling-correction-using-k-gram-overlap/ [Accessed 28 July 2022]

University of Wisconsin-Madison (2022). *Branches*. Available at: https://ece353.engr.wisc.edu/arm-assembly/branches/ [Accessed 28 July 2022]

W3Schools (2022). *Python Sets*. Available at: https://www.w3schools.com/python/python_sets.asp [Accessed 28 July 2022]

Waldron, J. (2020). *ValToHex.s*. Available at: https://www.scss.tcd.ie/John.Waldron/CSU33D01/Assignments/ValToHex.s [Accessed 28 July 2022]

Yang, D. (2019) *How MOSS Works*. Available at: https://yangdanny97.github.io/blog/2019/05/03/MOSS [Accessed 28 January 2022]

Appendices

Appendix A: ValToHex.s

```
start:

      LDR r1, =0xFEED1234
      LDR r2, = 0xF0000000
      LDR r7, = 0xF
      LDR r5, = 0x10
      LDR r12, =0xA1000400
      LDR r6, =0x0
      MOV r8, #28


loop:
      AND r3, r1, r2
      MOV r3, r3, LSR r8
      CMP r3, #10
      BCC num
      B letter

      B stop

num:
      ADD r3, #48
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE loop
      B stop

letter:
      CMP r3, #10
      BEQ Alet
      CMP r3, #11
      BEQ Blet
      CMP r3, #12
      BEQ Clet
      CMP r3, #13
      BEQ Dlet
      CMP r3, #14
      BEQ Elet
      CMP r3, #15
      BEQ Flet
```

```
      B loop

Alet:
      LDR r3, =0x41
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE loop
      B stop

Blet:
      LDR r3, =0x42
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE loop
      B stop

Clet:
      LDR r3, =0x43
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE loop
      B stop

Dlet:
      LDR r3, =0x44
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE loop
      B stop

Elet:
      LDR r3, =0x45
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
```

```
      BNE loop
      B stop

Flet:
      LDR r3, =0x46
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE loop
      B stop

stop:
      B     stop
```

Appendix B: ValToHex_Progression1.s

```
start:

      LDR r1, =0xFEED1234
      LDR r2, = 0xF0000000
      LDR r7, = 0xF
      LDR r5, = 0x10
      LDR r12, =0xA1000400
      LDR r6, =0x0
      MOV r8, #28


MainLoop:
      AND r3, r1, r2
      MOV r3, r3, LSR r8
      CMP r3, #10
      BCC Number
      B Letter

      B stop

Number:
      ADD r3, #48
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE MainLoop
      B stop

Letter:
      CMP r3, #10
      BEQ LetterA
      CMP r3, #11
      BEQ LetterB
      CMP r3, #12
      BEQ LetterC
      CMP r3, #13
      BEQ LetterD
      CMP r3, #14
      BEQ LetterE
      CMP r3, #15
      BEQ LetterF
      B MainLoop

LetterA:
      LDR r3, =0x41
```

```
        STR r3, [r12]
        ADD r12, r12, #1
        SUB r8, #4
        MOV r2, r2, LSR #4
        CMP r2, r6
        BNE MainLoop
        B stop

LetterB:
        LDR r3, =0x42
        STR r3, [r12]
        ADD r12, r12, #1
        SUB r8, #4
        MOV r2, r2, LSR #4
        CMP r2, r6
        BNE MainLoop
        B stop

LetterC:
        LDR r3, =0x43
        STR r3, [r12]
        ADD r12, r12, #1
        SUB r8, #4
        MOV r2, r2, LSR #4
        CMP r2, r6
        BNE MainLoop
        B stop

LetterD:
        LDR r3, =0x44
        STR r3, [r12]
        ADD r12, r12, #1
        SUB r8, #4
        MOV r2, r2, LSR #4
        CMP r2, r6
        BNE MainLoop
        B stop

LetterE:
        LDR r3, =0x45
        STR r3, [r12]
        ADD r12, r12, #1
        SUB r8, #4
        MOV r2, r2, LSR #4
        CMP r2, r6
        BNE MainLoop
        B stop

LetterF:
```

```
        LDR r3, =0x46
        STR r3, [r12]
        ADD r12, r12, #1
        SUB r8, #4
        MOV r2, r2, LSR #4
        CMP r2, r6
        BNE MainLoop
        B stop

stop:
        B     stop
```

Appendix C: ValToHex_Progression2.s

```
start:

      LDR r1, =0xFEED1234
      LDR r2, = 0xF0000000
      LDR r7, = 0xF
      LDR r5, = 0x10
      LDR r12, =0xA1000400
      LDR r6, =0x0
      MOV r8, #28


MainLoop:
      AND r3, r1, r2
      MOV r3, r3, LSR r8
      CMP r3, #10
      BCC Number
      B Letter

      B stop

stop:
      B     stop


Letter:
      CMP r3, #10
      BEQ LetterA
      CMP r3, #11
      BEQ LetterB
      CMP r3, #12
      BEQ LetterC
      CMP r3, #13
      BEQ LetterD
      CMP r3, #14
      BEQ LetterE
      CMP r3, #15
      BEQ LetterF
      B MainLoop

LetterF:
      LDR r3, =0x46
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE MainLoop
```

```
      B stop

LetterE:
      LDR r3, =0x45
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE MainLoop
      B stop

LetterD:
      LDR r3, =0x44
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE MainLoop
      B stop

LetterC:
      LDR r3, =0x43
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE MainLoop
      B stop

LetterB:
      LDR r3, =0x42
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE MainLoop
      B stop

LetterA:
      LDR r3, =0x41
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
```

```
        BNE MainLoop
        B stop

Number:
        ADD r3, #48
        STR r3, [r12]
        ADD r12, r12, #1
        SUB r8, #4
        MOV r2, r2, LSR #4
        CMP r2, r6
        BNE MainLoop
        B stop
```

Appendix D: ValToHex_Progression3.s

```
start:

      LDR r1, =0xFEED1234
      MOV r2, #4026531840
      MOV r7, #15
      MOV r5, #16
      LDR r12, =0xA1000400
      MOV r6, #0
      MOV r8, #28


MainLoop:
      AND r3, r1, r2
      MOV r3, r3, LSR r8
      CMP r3, #10
      BCC Number
      B Letter

      B stop

stop:
      B     stop


Letter:
      CMP r3, #10
      BEQ LetterA
      CMP r3, #11
      BEQ LetterB
      CMP r3, #12
      BEQ LetterC
      CMP r3, #13
      BEQ LetterD
      CMP r3, #14
      BEQ LetterE
      CMP r3, #15
      BEQ LetterF
      B MainLoop

LetterF:
      MOV r3, #70
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE MainLoop
```

```
      B stop

LetterE:
      MOV r3, #69
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE MainLoop
      B stop

LetterD:
      MOV r3, #68
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE MainLoop
      B stop

LetterC:
      MOV r3, #67
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE MainLoop
      B stop

LetterB:
      MOV r3, #66
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
      BNE MainLoop
      B stop

LetterA:
      MOV r3, #65
      STR r3, [r12]
      ADD r12, r12, #1
      SUB r8, #4
      MOV r2, r2, LSR #4
      CMP r2, r6
```

```
        BNE MainLoop
        B stop

Number:
        ADD r3, #48
        STR r3, [r12]
        ADD r12, r12, #1
        SUB r8, #4
        MOV r2, r2, LSR #4
        CMP r2, r6
        BNE MainLoop
        B stop
```

Appendix E: ValToHex_Progression4.s

```
start:

	LDR r1, =0xFEED1234
	MOV r2, #15
	MOV r7, #15
	MOV r5, #16
	LDR r12, =0xA1000400
	MOV r6, #4026531840
	MOV r8, #0


MainLoop:
	AND r3, r1, r2
	MOV r3, r3, LSR r8
	CMP r3, #10
	BCC Number
	B Letter

	B stop

stop:
	B    stop


Letter:
	CMP r3, #10
	BEQ LetterA
	CMP r3, #11
	BEQ LetterB
	CMP r3, #12
	BEQ LetterC
	CMP r3, #13
	BEQ LetterD
	CMP r3, #14
	BEQ LetterE
	CMP r3, #15
	BEQ LetterF
	B MainLoop

LetterF:
	MOV r3, #70
	STR r3, [r12]
	ADD r12, r12, #1
	ADD r8, #4
	MOV r2, r2, LSL #4
	CMP r2, r6
	BNE MainLoop
```

```
      B stop

LetterE:
      MOV r3, #69
      STR r3, [r12]
      ADD r12, r12, #1
      ADD r8, #4
      MOV r2, r2, LSL #4
      CMP r2, r6
      BNE MainLoop
      B stop

LetterD:
      MOV r3, #68
      STR r3, [r12]
      ADD r12, r12, #1
      ADD r8, #4
      MOV r2, r2, LSL #4
      CMP r2, r6
      BNE MainLoop
      B stop

LetterC:
      MOV r3, #67
      STR r3, [r12]
      ADD r12, r12, #1
      ADD r8, #4
      MOV r2, r2, LSL #4
      CMP r2, r6
      BNE MainLoop
      B stop

LetterB:
      MOV r3, #66
      STR r3, [r12]
      ADD r12, r12, #1
      ADD r8, #4
      MOV r2, r2, LSL #4
      CMP r2, r6
      BNE MainLoop
      B stop

LetterA:
      MOV r3, #65
      STR r3, [r12]
      ADD r12, r12, #1
      ADD r8, #4
      MOV r2, r2, LSL #4
      CMP r2, r6
```

```
        BNE MainLoop
        B stop

Number:
        ADD r3, #48
        STR r3, [r12]
        ADD r12, r12, #1
        ADD r8, #4
        MOV r2, r2, LSL #4
        CMP r2, r6
        BNE MainLoop
        B stop
```

Appendix F: ValToHex_Progression5.s

```
start:

      LDR r1, =0xFEED1234
      MOV r2, #15
      MOV r7, #15
      MOV r5, #16
      LDR r12, =0xA1000400
      MOV r6, #4026531840
      MOV r8, #0


MainLoop:
      AND r3, r1, r2
      MOV r3, r3, LSR r8
      CMP r3, #10
      BCC Number
      B Letter

      B stop

stop:
      B     stop


Letter:
      ADD r3, #55
      STR r3, [r12]
      ADD r12, r12, #1
      ADD r8, #4
      MOV r2, r2, LSL #4
      CMP r2, r6
      BNE MainLoop
      B stop


Number:
      ADD r3, #48
      STR r3, [r12]
      ADD r12, r12, #1
      ADD r8, #4
      MOV r2, r2, LSL #4
      CMP r2, r6
      BNE MainLoop
      B stop
```

Appendix G: 580 - lowerCamelCase.s

```
  .syntax unified
  .cpu cortex-m4
  .thumb
  .global  Main

Main:

  @ Follow the instructions in the handout for Assignment 7

  @
  @ TIP: To view memory when debugging your program you can ...
  @
  @   Add the watch expression to see individual characters:
(char[64])newString
  @
  @   OR
  @
  @   Add the watch expression to see the string:
(char*)&newString
  @
  @   OR
  @
  @   Open a 'Memory View' specifying the address &newString and
length at
  @   least 128. You can open a Memory View with ctrl-shift-p
type
  @   'View Memory' (cmd-shift-p on a Mac).
  @

  @ Let R0 = newString
  @ Let R1 = originalString
  @ Let R2 = char

  LDRB  R2, [R1]        @ byte[originalString] = char;

WhNotCharStart:         @ while
  CMP   R2, #'A'        @  ((char < 'A'
  BLT   ToFirstChar     @     &&
  CMP   R2, #'Z'        @     char > 'Z')
  BLT   WhileUpper      @     &&
  CMP   R2, #'a'        @   (char < 'a'
  BLT   ToFirstChar     @     &&
  CMP   R2, #'z'        @     char > 'z'))
  BLT   WhileUpper      @ {

ToFirstChar:            @
  ADD   R1, R1, #1      @     originalString++;
```

```
  LDRB  R2, [R1]        @     byte[originalString] = char;

  B     WhNotCharStart  @ }

WhileUpper:             @ while
  CMP   R2, #'A'        @  (char >= 'A'
  BLT   IfNotChar       @   &&
  CMP   R2, #'Z'        @   char <= 'Z')
  BGT   WhileLower      @ {

  ADD   R2, R2, #0x20   @     char += 0x20;
  STRB  R2, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R2, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

WhileLower:             @ while
  CMP   R2, #'a'        @  (char >= 'a'
  BLT   IfNotChar       @   &&
  CMP   R2, #'z'        @   char <= 'z')
  BGT   IfNotChar       @ {

  STRB  R2, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R2, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

IfNotChar:              @ if
  CMP   R2, #0          @  (char != 0)
  BEQ   EndIf           @ {
  ADD   R1, R1, #1      @     originalString++;
  LDRB  R2, [R1]        @     byte[originalString] = char;

IfToUpper:              @ If
  CMP   R2, #'A'        @  (char > 'A'
  BLT   IfNotChar       @   &&
  CMP   R2, #'Z'        @   char < 'Z')
  BLT   StayUpper       @ {

ToUpper:                @ If
  CMP   R2, #'a'        @  (char > 'a'
  BLT   IfNotChar       @   &&
  CMP   R2, #'z'        @   char < 'z')
  BGT   IfNotChar       @ {
```

```
  SUB   R2, R2, #0x20   @         char -= 0x20;
StayUpper:
  STRB  R2, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R2, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }
EndIf:                  @
  LDR   R2, =0          @ char = 0;
  STRB  R2, [R0]        @ byte[newString] = char;

  @ End of program ... check your result


End_Main:
  BX    lr
```

Appendix H: 580P1 - lowerCamelCase.s

```
  .syntax unified
  .cpu cortex-m4
  .thumb
  .global  Main

Main:

  @ Follow the instructions in the handout for Assignment 7

  @
  @ TIP: To view memory when debugging your program you can ...
  @
  @   Add the watch expression to see individual characters:
(char[64])newString
  @
  @   OR
  @
  @   Add the watch expression to see the string:
(char*)&newString
  @
  @   OR
  @
  @   Open a 'Memory View' specifying the address &newString and
length at
  @   least 128. You can open a Memory View with ctrl-shift-p
type
  @   'View Memory' (cmd-shift-p on a Mac).
  @

  @ Let R0 = newString
  @ Let R1 = originalString
  @ Let R3 = char

  LDRB  R3, [R1]        @ byte[originalString] = char;

WhNotCharStart:         @ while
  CMP   R3, #'A'        @  ((char < 'A'
  BLT   ToFirstChar     @     &&
  CMP   R3, #'Z'        @     char > 'Z')
  BLT   WhileUpper      @     &&
  CMP   R3, #'a'        @   (char < 'a'
  BLT   ToFirstChar     @     &&
  CMP   R3, #'z'        @     char > 'z'))
  BLT   WhileUpper      @ {

ToFirstChar:            @
  ADD   R1, R1, #1      @     originalString++;
```

```
  LDRB  R3, [R1]        @     byte[originalString] = char;

  B     WhNotCharStart  @ }

WhileUpper:             @ while
  CMP   R3, #'A'        @  (char >= 'A'
  BLT   IfNotChar       @   &&
  CMP   R3, #'Z'        @   char <= 'Z')
  BGT   WhileLower      @ {

  ADD   R3, R3, #0x20   @     char += 0x20;
  STRB  R3, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R3, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

WhileLower:             @ while
  CMP   R3, #'a'        @  (char >= 'a'
  BLT   IfNotChar       @   &&
  CMP   R3, #'z'        @   char <= 'z')
  BGT   IfNotChar       @ {

  STRB  R3, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R3, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

IfNotChar:              @ if
  CMP   R3, #0          @  (char != 0)
  BEQ   EndIf           @ {
  ADD   R1, R1, #1      @     originalString++;
  LDRB  R3, [R1]        @     byte[originalString] = char;

IfToUpper:              @ If
  CMP   R3, #'A'        @  (char > 'A'
  BLT   IfNotChar       @   &&
  CMP   R3, #'Z'        @   char < 'Z')
  BLT   StayUpper       @ {

ToUpper:                @ If
  CMP   R3, #'a'        @  (char > 'a'
  BLT   IfNotChar       @   &&
  CMP   R3, #'z'        @   char < 'z')
  BGT   IfNotChar       @ {
```

```
  SUB   R3, R3, #0x20   @         char -= 0x20;
StayUpper:
  STRB  R3, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R3, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }
EndIf:                  @
  LDR   R3, =0          @ char = 0;
  STRB  R3, [R0]        @ byte[newString] = char;

  @ End of program ... check your result


End_Main:
  BX    lr
```

Appendix I: 580P2 - lowerCamelCase.s

```
  .syntax unified
  .cpu cortex-m4
  .thumb
  .global  Main

Main:

  @ Follow the instructions in the handout for Assignment 7

  @
  @ TIP: To view memory when debugging your program you can ...
  @
  @   Add the watch expression to see individual characters:
(char[64])newString
  @
  @   OR
  @
  @   Add the watch expression to see the string:
(char*)&newString
  @
  @   OR
  @
  @   Open a 'Memory View' specifying the address &newString and
length at
  @   least 128. You can open a Memory View with ctrl-shift-p
type
  @   'View Memory' (cmd-shift-p on a Mac).
  @

  @ Let R0 = newString
  @ Let R1 = originalString
  @ Let R2 = char

  LDRB  R2, [R1]        @ byte[originalString] = char;

WhNotCharStart:         @ while
  CMP   R2, #0x41        @  ((char < 'A'
  BLT   ToFirstChar     @     &&
  CMP   R2, #0x5A        @     char > 'Z')
  BLT   WhileUpper      @     &&
  CMP   R2, #0x61        @   (char < 'a'
  BLT   ToFirstChar     @     &&
  CMP   R2, #0x7A        @     char > 'z'))
  BLT   WhileUpper      @ {

ToFirstChar:            @
  ADD   R1, R1, #1      @     originalString++;
```

```
  LDRB  R2, [R1]        @     byte[originalString] = char;

  B     WhNotCharStart  @ }

WhileUpper:             @ while
  CMP   R2, #0x41        @  (char >= 'A'
  BLT   IfNotChar       @   &&
  CMP   R2, #0x5A        @   char <= 'Z')
  BGT   WhileLower      @ {

  ADD   R2, R2, #0x20   @     char += 0x20;
  STRB  R2, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R2, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

WhileLower:             @ while
  CMP   R2, #0x61        @  (char >= 'a'
  BLT   IfNotChar       @   &&
  CMP   R2, #0x7A        @   char <= 'z')
  BGT   IfNotChar       @ {

  STRB  R2, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R2, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

IfNotChar:              @ if
  CMP   R2, #0          @  (char != 0)
  BEQ   EndIf           @ {
  ADD   R1, R1, #1      @     originalString++;
  LDRB  R2, [R1]        @     byte[originalString] = char;

IfToUpper:              @ If
  CMP   R2, #0x41        @  (char > 'A'
  BLT   IfNotChar       @   &&
  CMP   R2, #0x5A        @   char < 'Z')
  BLT   StayUpper       @ {

ToUpper:                @ If
  CMP   R2, #0x61        @  (char > 'a'
  BLT   IfNotChar       @   &&
  CMP   R2, #0x7A        @   char < 'z')
  BGT   IfNotChar       @ {
```

```
  SUB   R2, R2, #0x20   @         char -= 0x20;
StayUpper:
  STRB  R2, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R2, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }
EndIf:                  @
  LDR   R2, =0          @ char = 0;
  STRB  R2, [R0]        @ byte[newString] = char;

  @ End of program ... check your result


End_Main:
  BX    lr
```

Appendix J: 580P3 - lowerCamelCase.s

```
  .syntax unified
  .cpu cortex-m4
  .thumb
  .global  Main

Main:

  @ Follow the instructions in the handout for Assignment 7

  @
  @ TIP: To view memory when debugging your program you can ...
  @
  @   Add the watch expression to see individual characters:
(char[64])newString
  @
  @   OR
  @
  @   Add the watch expression to see the string:
(char*)&newString
  @
  @   OR
  @
  @   Open a 'Memory View' specifying the address &newString and
length at
  @   least 128. You can open a Memory View with ctrl-shift-p
type
  @   'View Memory' (cmd-shift-p on a Mac).
  @

  @ Let R0 = newString
  @ Let R1 = originalString
  @ Let R2 = char

  LDRB  R2, [R1]        @ byte of originalString is a character;

WhNotCharStart:         @ while loop
  CMP   R2, #'A'        @
  BLT   ToFirstChar     @ branch if R3 less than 65
  CMP   R2, #'Z'        @
  BLT   WhileUpper      @ branch if R3 less than 90
  CMP   R2, #'a'        @
  BLT   ToFirstChar     @ branch if R3 less than 97
  CMP   R2, #'z'        @
  BLT   WhileUpper      @ branch if R3 less than 122

ToFirstChar:            @
  ADD   R1, R1, #1      @     originalString++;
```

```
  LDRB  R2, [R1]        @     byte[originalString] = char;

  B     WhNotCharStart  @ }

WhileUpper:             @ while loop
  CMP   R2, #'A'        @  (char >= 'A'
  BLT   IfNotChar       @   &&
  CMP   R2, #'Z'        @   char <= 'Z')
  BGT   WhileLower      @ {

  ADD   R2, R2, #0x20   @     char += 0x20;
  STRB  R2, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R2, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

WhileLower:             @ while loop
  CMP   R2, #'a'        @  (char >= 'a'
  BLT   IfNotChar       @   &&
  CMP   R2, #'z'        @   char <= 'z')
  BGT   IfNotChar       @ {

  STRB  R2, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R2, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

IfNotChar:              @ if loop
  CMP   R2, #0          @  (char != 0)
  BEQ   EndIf           @ {
  ADD   R1, R1, #1      @     originalString++;
  LDRB  R2, [R1]        @     byte[originalString] = char;

IfToUpper:              @ If loop
  CMP   R2, #'A'        @  (char > 'A'
  BLT   IfNotChar       @   &&
  CMP   R2, #'Z'        @   char < 'Z')
  BLT   StayUpper       @ {

ToUpper:                @ If loop
  CMP   R2, #'a'        @  (char > 'a'
  BLT   IfNotChar       @   &&
  CMP   R2, #'z'        @   char < 'z')
  BGT   IfNotChar       @ {
```

```
  SUB   R2, R2, #0x20   @         char -= 0x20;
StayUpper:
  STRB  R2, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R2, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }
EndIf:                  @
  LDR   R2, =0          @ char = 0;
  STRB  R2, [R0]        @ byte[newString] = char;

  @ End of program ... check your result


End_Main:
  BX    lr
```

Appendix K: 580P4 - lowerCamelCase.s

```
  .syntax unified
  .cpu cortex-m4
  .thumb
  .global  Main

Main:

  @ Follow the instructions in the handout for Assignment 7

  @
  @ TIP: To view memory when debugging your program you can ...
  @
  @   Add the watch expression to see individual characters:
(char[64])newString
  @
  @   OR
  @
  @   Add the watch expression to see the string:
(char*)&newString
  @
  @   OR
  @
  @   Open a 'Memory View' specifying the address &newString and
length at
  @   least 128. You can open a Memory View with ctrl-shift-p
type
  @   'View Memory' (cmd-shift-p on a Mac).
  @

  @ Let R0 = newString
  @ Let R1 = originalString
  @ Let R3 = char

  LDRB  R3, [R1]        @ byte[originalString] = char;

WhNotCharStart:         @ while
  CMP   R3, #0x41        @  ((char < 'A'
  BLT   ToFirstChar     @     &&
  CMP   R3, #0x5A        @     char > 'Z')
  BLT   WhileUpper      @     &&
  CMP   R3, #0x61        @   (char < 'a'
  BLT   ToFirstChar     @     &&
  CMP   R3, #0x7A        @     char > 'z'))
  BLT   WhileUpper      @ {

ToFirstChar:            @
  ADD   R1, R1, #1      @     originalString++;
```

```
  LDRB  R3, [R1]        @     byte[originalString] = char;

  B     WhNotCharStart  @ }

WhileUpper:             @ while
  CMP   R3, #0x41        @  (char >= 'A'
  BLT   IfNotChar       @   &&
  CMP   R3, #0x5A        @   char <= 'Z')
  BGT   WhileLower      @ {

  ADD   R3, R3, #0x20   @     char += 0x20;
  STRB  R3, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R3, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

WhileLower:             @ while
  CMP   R3, #0x61        @  (char >= 'a'
  BLT   IfNotChar       @   &&
  CMP   R3, #0x7A        @   char <= 'z')
  BGT   IfNotChar       @ {

  STRB  R3, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R3, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

IfNotChar:              @ if
  CMP   R3, #0          @  (char != 0)
  BEQ   EndIf           @ {
  ADD   R1, R1, #1      @     originalString++;
  LDRB  R3, [R1]        @     byte[originalString] = char;

IfToUpper:              @ If
  CMP   R3, #0x41        @  (char > 'A'
  BLT   IfNotChar       @   &&
  CMP   R3, #0x5A        @   char < 'Z')
  BLT   StayUpper       @ {

ToUpper:                @ If
  CMP   R3, #0x61        @  (char > 'a'
  BLT   IfNotChar       @   &&
  CMP   R3, #0x7A        @   char < 'z')
  BGT   IfNotChar       @ {
```

```
  SUB   R3, R3, #0x20   @         char -= 0x20;
StayUpper:
  STRB  R3, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R3, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }
EndIf:                  @
  LDR   R3, =0          @ char = 0;
  STRB  R3, [R0]        @ byte[newString] = char;

  @ End of program ... check your result


End_Main:
  BX    lr
```

Appendix L: 580P5 - lowerCamelCase.s

```
  .syntax unified
  .cpu cortex-m4
  .thumb
  .global  Main

Main:

  @ Follow the instructions in the handout for Assignment 7

  @
  @ TIP: To view memory when debugging your program you can ...
  @
  @   Add the watch expression to see individual characters:
(char[64])newString
  @
  @   OR
  @
  @   Add the watch expression to see the string:
(char*)&newString
  @
  @   OR
  @
  @   Open a 'Memory View' specifying the address &newString and
length at
  @   least 128. You can open a Memory View with ctrl-shift-p
type
  @   'View Memory' (cmd-shift-p on a Mac).
  @

  @ Let R0 = newString
  @ Let R1 = originalString
  @ Let R3 = char

  LDRB  R3, [R1]        @ byte[originalString] = char;

WhNotCharStart:         @ while loop
  CMP   R3, #'A'        @
  BLT   ToFirstChar     @ branch if R3 less than 65
  CMP   R3, #'Z'        @
  BLT   WhileUpper      @ branch if R3 less than 90
  CMP   R3, #'a'        @
  BLT   ToFirstChar     @ branch if R3 less than 97
  CMP   R3, #'z'        @
  BLT   WhileUpper      @ branch if R3 less than 122

ToFirstChar:            @
  ADD   R1, R1, #1      @     originalString++;
```

```
  LDRB  R3, [R1]        @     byte[originalString] = char;

  B     WhNotCharStart  @ }

WhileUpper:             @ while loop
  CMP   R3, #'A'        @  (char >= 'A'
  BLT   IfNotChar       @   &&
  CMP   R3, #'Z'        @   char <= 'Z')
  BGT   WhileLower      @ {

  ADD   R3, R3, #0x20   @     char += 0x20;
  STRB  R3, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R3, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

WhileLower:             @ while loop
  CMP   R3, #'a'        @  (char >= 'a'
  BLT   IfNotChar       @   &&
  CMP   R3, #'z'        @   char <= 'z')
  BGT   IfNotChar       @ {

  STRB  R3, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R3, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

IfNotChar:              @ if loop
  CMP   R3, #0          @  (char != 0)
  BEQ   EndIf           @ {
  ADD   R1, R1, #1      @     originalString++;
  LDRB  R3, [R1]        @     byte[originalString] = char;

IfToUpper:              @ If loop
  CMP   R3, #'A'        @  (char > 'A'
  BLT   IfNotChar       @   &&
  CMP   R3, #'Z'        @   char < 'Z')
  BLT   StayUpper       @ {

ToUpper:                @ If loop
  CMP   R3, #'a'        @  (char > 'a'
  BLT   IfNotChar       @   &&
  CMP   R3, #'z'        @   char < 'z')
  BGT   IfNotChar       @ {
```

```
  SUB   R3, R3, #0x20   @         char -= 0x20;
StayUpper:
  STRB  R3, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R3, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }
EndIf:                  @
  LDR   R3, =0          @ char = 0;
  STRB  R3, [R0]        @ byte[newString] = char;

  @ End of program ... check your result


End_Main:
  BX    lr
```

Appendix M: 580P6 - lowerCamelCase.s

```
  .syntax unified
  .cpu cortex-m4
  .thumb
  .global  Main

Main:

  @ Follow the instructions in the handout for Assignment 7

  @
  @ TIP: To view memory when debugging your program you can ...
  @
  @   Add the watch expression to see individual characters:
(char[64])newString
  @
  @   OR
  @
  @   Add the watch expression to see the string:
(char*)&newString
  @
  @   OR
  @
  @   Open a 'Memory View' specifying the address &newString and
length at
  @   least 128. You can open a Memory View with ctrl-shift-p
type
  @   'View Memory' (cmd-shift-p on a Mac).
  @

  @ Let R0 = newString
  @ Let R1 = originalString
  @ Let R2 = char

  LDRB  R2, [R1]        @ byte[originalString] = char;

WhNotCharStart:         @ while loop
  CMP   R2, #0x41       @
  BLT   ToFirstChar     @ branch if R3 less than 65
  CMP   R2, #0x5A       @
  BLT   WhileUpper      @ branch if R3 less than 90
  CMP   R2, #0x61       @
  BLT   ToFirstChar     @ branch if R3 less than 97
  CMP   R2, #0x7A       @
  BLT   WhileUpper      @ branch if R3 less than 122

ToFirstChar:            @
  ADD   R1, R1, #1      @     originalString++;
```

```
  LDRB  R2, [R1]        @     byte[originalString] = char;

  B     WhNotCharStart  @ }

WhileUpper:             @ while loop
  CMP   R2, #0x41        @  (char >= 'A'
  BLT   IfNotChar       @   &&
  CMP   R2, #0x5A        @   char <= 'Z')
  BGT   WhileLower      @ {

  ADD   R2, R2, #0x20   @     char += 0x20;
  STRB  R2, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R2, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

WhileLower:             @ while loop
  CMP   R2, #0x61        @  (char >= 'a'
  BLT   IfNotChar       @   &&
  CMP   R2, #0x7A        @   char <= 'z')
  BGT   IfNotChar       @ {

  STRB  R2, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R2, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

IfNotChar:              @ if loop
  CMP   R2, #0          @  (char != 0)
  BEQ   EndIf           @ {
  ADD   R1, R1, #1      @     originalString++;
  LDRB  R2, [R1]        @     byte[originalString] = char;

IfToUpper:              @ If loop
  CMP   R2, #0x41        @  (char > 'A'
  BLT   IfNotChar       @   &&
  CMP   R2, #0x5A        @   char < 'Z')
  BLT   StayUpper       @ {

ToUpper:                @ If loop
  CMP   R2, #0x61        @  (char > 'a'
  BLT   IfNotChar       @   &&
  CMP   R2, #0x7A        @   char < 'z')
  BGT   IfNotChar       @ {
```

```
  SUB   R2, R2, #0x20   @         char -= 0x20;
StayUpper:
  STRB  R2, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R2, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }
EndIf:                  @
  LDR   R2, =0          @ char = 0;
  STRB  R2, [R0]        @ byte[newString] = char;

  @ End of program ... check your result


End_Main:
  BX    lr
```

Appendix N: 580P7 - lowerCamelCase.s

```
  .syntax unified
  .cpu cortex-m4
  .thumb
  .global  Main

Main:

  @ Follow the instructions in the handout for Assignment 7

  @
  @ TIP: To view memory when debugging your program you can ...
  @
  @   Add the watch expression to see individual characters:
(char[64])newString
  @
  @   OR
  @
  @   Add the watch expression to see the string:
(char*)&newString
  @
  @   OR
  @
  @   Open a 'Memory View' specifying the address &newString and
length at
  @   least 128. You can open a Memory View with ctrl-shift-p
type
  @   'View Memory' (cmd-shift-p on a Mac).
  @

  @ Let R0 = newString
  @ Let R1 = originalString
  @ Let R3 = char

  LDRB  R3, [R1]        @ byte[originalString] = char;

WhNotCharStart:         @ while loop
  CMP   R3, #0x41       @
  BLT   ToFirstChar     @ branch if R3 less than 65
  CMP   R3, #0x5A       @
  BLT   WhileUpper      @ branch if R3 less than 90
  CMP   R3, #0x61       @
  BLT   ToFirstChar     @ branch if R3 less than 97
  CMP   R3, #0x7A       @
  BLT   WhileUpper      @ branch if R3 less than 122

ToFirstChar:            @
  ADD   R1, R1, #1      @     originalString++;
```

```
  LDRB  R3, [R1]        @     byte[originalString] = char;

  B     WhNotCharStart  @ }

WhileUpper:             @ while loop
  CMP   R3, #0x41        @  (char >= 'A'
  BLT   IfNotChar       @   &&
  CMP   R3, #0x5A        @   char <= 'Z')
  BGT   WhileLower      @ {

  ADD   R3, R3, #0x20   @     char += 0x20;
  STRB  R3, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R3, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

WhileLower:             @ while loop
  CMP   R3, #0x61        @  (char >= 'a'
  BLT   IfNotChar       @   &&
  CMP   R3, #0x7A        @   char <= 'z')
  BGT   IfNotChar       @ {

  STRB  R3, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R3, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }

IfNotChar:              @ if loop
  CMP   R3, #0          @  (char != 0)
  BEQ   EndIf           @ {
  ADD   R1, R1, #1      @     originalString++;
  LDRB  R3, [R1]        @     byte[originalString] = char;

IfToUpper:              @ If loop
  CMP   R3, #0x41        @  (char > 'A'
  BLT   IfNotChar       @   &&
  CMP   R3, #0x5A        @   char < 'Z')
  BLT   StayUpper       @ {

ToUpper:                @ If loop
  CMP   R3, #0x61        @  (char > 'a'
  BLT   IfNotChar       @   &&
  CMP   R3, #0x7A        @   char < 'z')
  BGT   IfNotChar       @ {
```

```
  SUB   R3, R3, #0x20   @         char -= 0x20;
StayUpper:
  STRB  R3, [R0]        @     word[newString] = char;
  ADD   R1, R1, #1      @     originalString++;
  ADD   R0, R0, #1      @     newString++;

  LDRB  R3, [R1]        @     byte[originalString] = char;
  B     WhileUpper      @ }
EndIf:                  @
  LDR   R3, =0          @ char = 0;
  STRB  R3, [R0]        @ byte[newString] = char;

  @ End of program ... check your result


End_Main:
  BX    lr
```