LAPIN AMK
Lapland University of Applied Sciences

INVESTIGATION OF ALMOST ABELIAN SQUARE-FREE WORDS ON THREE LETTERS

Subhash Rijal

Thesis
Lapland University of Applied Sciences
Information Technology
Bachelor of Engineering

2016

LAPIN AMK
Lapland University of Applied Sciences

Abstract of Thesis

Technology, Communication and Transport
Degree Programme in Information Technology

Author Subhash Rijal Year 2016
Supervisor Veikko Keränen
Title of Thesis Investigation of Almost Abelian Square-Free Words on Three Letters
Number of pages 30 + 7

In the study, a C program was developed for testing (almost) abelian square free words on three letters. The program was designed to read pre-generated sets of abelian square-free words, to test whether they are extendable, and finally to categorize them as favorable or unfavorable sets.

Computation of millions of words is a challenging task by computers with low processing capabilities. A conventional computer was used for programming and for testing the codes. Mostly, while testing, the boundary length for extensions was chosen small due to limited processing capacity of the computer and the subsequent enormous time consumption. The author also tested the program with higher boundary lengths and, after long computations, the results were found out to be correct.

The program was tested in many phases. The most challenging task was to process three different words at the same time with equivalent extensions and backtracking of letters.

It is anticipated that, in the near future, the program will be run in more sophisticated computers for studying a large number of words using a large boundary length.

Key words intensive computation, abelian square-free words, favourable, unfavourable

CONTENTS

LIST OF FIGURES

SYMBOLS AND TERMS

Word	A Word over a given (or fixed) alphabet ∑ is a finite string, or sequence, of letters belonging to ∑.
Anagram	A word formed by rearranging the letters of another, such as granmaa, formed from *anagram*.
Suffix	The end part of the word.
Prefix	The initial part of the word.
Catenation	Joining words *u* and *v* together to create a new word *uv*.
∑*	The set of all finite words over ∑.
As	Suffix of the word A belonging to ∑*.
pA	Prefix of the word A belonging to ∑*.
Bs	Suffix of the word B belonging to ∑*.
pB	Prefix of the word B belonging to ∑*.
Cs	Suffix of the word C belonging to ∑*.
pC	Prefix of the word C belonging to ∑*.
Ds	Suffix of the word D belonging to ∑*.
pD	Prefix of the word D belonging to ∑*.

1 INTRODUCTION

The main aim of this research is to find long, perhaps infinitely long, abelian square-free words on three letters. Words are basic objects in theoretical computer science; it is therefore natural to study about their structures. Combinatorics on words is the branch of mathematics that deals with the study of words and formal languages. The history of combinatorics is way too long. However, in the year 1906, Axel Thue initiated the field of combinatorics on words (Keränen 2009a, 3893). Construction of an unboundedly (infinitely) long abelian square-free word on four letters was challenging until 1992. The problem was then solved by Veikko Keränen through an iteration of a uniformly growing endomorphism g with $|g(a)| = 85$. Ever since, several researches are being conducted to find out long abelian square-free words on three letters - including this study. In the case of three letters, it is not possible to avoid all abelian squares for words of length greater than 7. However, the avoidance may become possible for unboundedly long words, if the shortest possible repetitions are allowed. Adopting this approach, the setting in this study is that the square xx and the repetition xxx are allowed for any letter x in the alphabet {a, b, c}. Nevertheless, all the other longer abelian squares should still be avoided.

Initially, this report discusses the mechanisms responsible through a diagrammatic representation, and later, explains theoretically how the algorithm works step by step in an understandable way. In the later part, an alternative way of finding an abelian square, i.e., the usage of a cumulative sum is described including brief description of parallel processing, Integrated Development Environment (IDE) and text editor. And finally, an appendix itself contains the program, which is authorized for further researches in case that would become useful.

2 INTRODUCTION TO ABELIAN SQUARE-FREE WORDS

The word is called an abelian square if it contains *uv* as a permutation (anagram) of each other where *uv* are equal and non-empty words whereas the word is called an abelian square-free word if it does not contain any abelian squares as a factor. Let ∑* = {a, b, c}. In the sequel, repetitions (as factors) aa, bb, cc, aaa, bbb and ccc are allowed but all longer abelian squares should be avoided. The word *w* = abacaba in ∑* = {a, b, c} is abelian square-free while the word *w* = abcabcbcac in ∑* = {a, b, c} is not. It contains the factor cabcbcac, which is an anagram repetition and needs to be avoided. However, it is not possible to avoid all abelian square-free words over thee letters of the length greater than 7. (Keränen 2009b, 3894.)

The abelian square-free words can be classified into three different categories, by increasing the length of word into fixed possible ways alternatively, in both right and left side, where backtracks of word is possible when necessary. If the upper bounds are reached, then the original word is so-far-so-good (which may also turn into unfavorable after times of experiments). On the other hand, if there is no any way to reach upper bounds then the word is regarded as an unfavorable word (Keränen 2009a, 360). The difference between a so-far-so-good word and a good word is that, the former is just a finite word, while the latter can be extended to an infinite word (Lin 2011, 6).

3 PARALLEL COMPUTING

Parallel computing is the process of using multiple processors in computation concurrently instead of using one processor exclusively. Traditional software development process wore time consuming because they used singular processors. Despite their speed limitations, singular processor generates excessive heat due to the higher flow of electrons. (Barney 2016.)

In modern computer system, works are carried into multi processors. The work done by computers containing multiple processors is less time consuming with higher efficiency. Software developers have a huge advantage of implementing data structures and algorithms developed by researchers into specialized processors for rapid increased performances that lacks in its counterpart serial processor. (Hughes & Hughes 2014.)

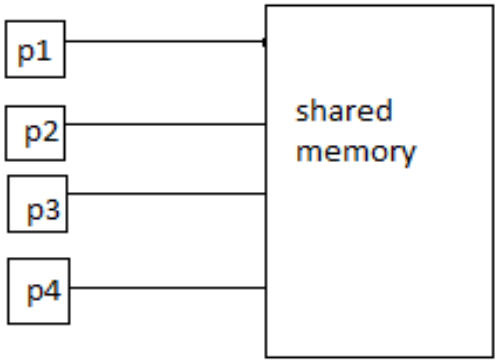

Figure 1. Parallel, Random Access Machine (PRAM) (Hughes & Hughes 2014).

Mathematically, as part of parallel computing the speedup ratio is presented as,

$$s(\rho) = \frac{T(n,1)}{T(n,P)}$$

Where, $T(n,1)$ be the fastest known sequential algorithm and $T(n,P)$ be the running time for parallel algorithm. Let '*n'* be the size of the input and '*p*' be processors. The ratio of execution time for sequential to that of execution time for parallel would result *S(p)* = *p* also said as perfect speedup which is very rare. (Wolfram World 2016.)

4 ABELIAN SQUARE-FREE WORD GENERATION

4.1 Input Words (input.txt)

The computation in square-free words on three letters with an extension to the left and to the right simultaneously once at a time was carried out until it reached a boundary. Program reads an input with the help of input.txt file whereas; results are stored in sfsg.txt (so far so good) and bad.txt files per the output generated.

Input.txt file is the collection of input, which contains at least one single block of words. A block of words means the collection of 8 different suffixes and prefixes of the words A, B, C and D. The file should be non-empty. However, this is also a collection of already generated so-far-so-good words to be tested further.

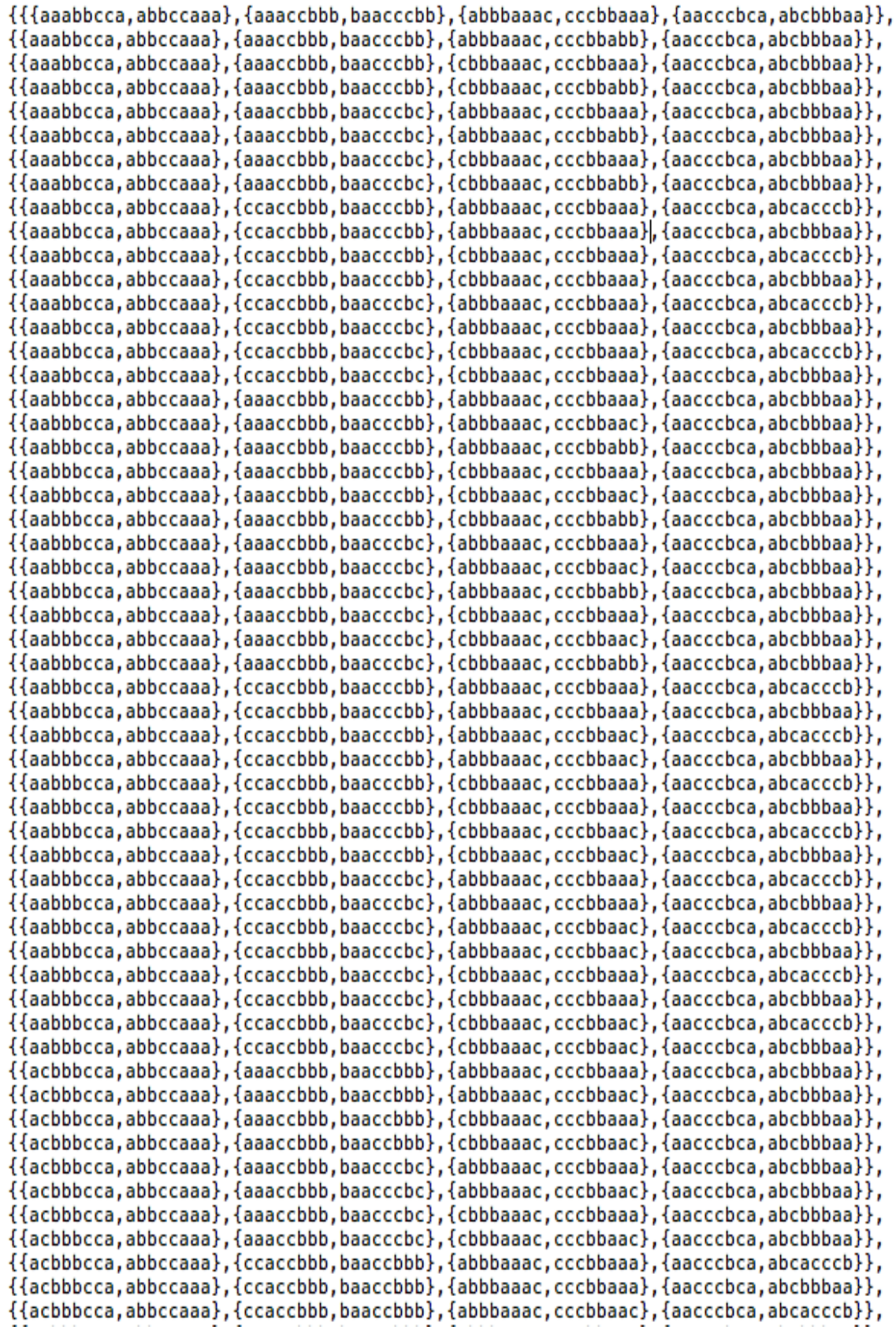

Figure 2. input.txt File

Figure 2 above has a block in the form of

{{“Asffix”,”Aprefix”},{“Bsuffix”,”Bprefix”},

{“Csuffix”,”Cprefix”},{”Dsuffix”,”Dprefix”}},

While reading the file, the first item must always be ‘suffix’ and the later one will always be ‘prefix’.

4.2 So-far-so-good Words (sfsg.txt)

So-far-so-good words are classified/saved if and only if the given words reach the boundary and still remains as a square free. Here, ‘boundary’ means the input length given by a user. The saving format for so-far-so-good word is also different than that of an input.

prefixA-Asuffix.prefixB-Bsuffix.prefixC-Csuffix.prefixD-Dsuffix i.e.,

“pA-As.pB-Bs.pC-Cs.pD-Ds“

The reason for different format in saving with comparison to *input.txt* is to make result look simpler and easier to understand. The main reason indeed is to code easily while programming in near future.

If suffix/prefix words reach the boundary length even after backtracking when necessary, then original word is saved in so-far-so-good file. Here, the original word means the single block of input from input.txt file saving in newer format as maintained.

The following output is a result of input from input.txt file above (see Figure 2) after an extension length of 2. However, all the inputs are not included above due to space limitation.

after an extension of length 2.

abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaaa-aabbbaac.abbccaaa-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaaa-aabbbaac.abbccacc-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaaa-aabbbaac.abbcccaa-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaaa-aabbbaac.abbcccab-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaaa-aabbbaac.abbcccac-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaaa-aabbbaac.abbcccba-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaaa-aabbbaac.abbcccbb-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaac-aabbbaac.abbccaaa-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaac-aabbbaac.abbccacc-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaac-aabbbaac.abbcccaa-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaac-aabbbaac.abbcccab-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaac-aabbbaac.abbcccac-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaac-aabbbaac.abbcccba-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaac-aabbbaac.abbcccbb-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbabb-aabbbaac.abbccaaa-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbabb-aabbbaac.abbccacc-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbabb-aabbbaac.abbcccaa-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbabb-aabbbaac.abbcccab-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbabb-aabbbaac.abbcccac-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbabb-aabbbaac.abbcccba-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbabb-aabbbaac.abbcccbb-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaca-aabbbaac.abbccaaa-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaca-aabbbaac.abbccacc-bbaaabca
abbccaaa-bbaaabca.bbbccaaa-aaaccbbb.cccbbaca-aabbbaac.abbcccaa-bbaaabca

Figure 3. sfsg.txt File

4.3 Bad Words (bad.txt)

Among 24 words even a single word, which after extension to the right or to the left does not reach the boundary length or contains an abelian square even if reached the boundary length, the whole block of input word is saved into bad.txt file in the same format as so-far-so-good words, i.e., “prefixA-Asuffix.prefixB-Bsuffix.prefixC-Csuffix.prefixD-Dsuffix” in short, its represented as “pA-As.pB-Bs.pC-Cs.pD-Ds“.

The following figure shows the contents of bad.txt file generated by the program after executing input file shown in Figure 2.

```
after an extension of length 2

abbccaaa-aaabbcca.baacccbb-aaaccbbb.cccbbaaa-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aaabbcca.baacccbb-aaaccbbb.cccbbabb-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aaabbcca.baacccbb-aaaccbbb.cccbbaaa-cbbbaaac.abcbbbaa-aacccbca
abbccaaa-aaabbcca.baacccbb-aaaccbbb.cccbbabb-cbbbaaac.abcbbbaa-aacccbca
abbccaaa-aaabbcca.baacccbc-aaaccbbb.cccbbaaa-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aaabbcca.baacccbc-aaaccbbb.cccbbabb-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aaabbcca.baacccbc-aaaccbbb.cccbbaaa-cbbbaaac.abcbbbaa-aacccbca
abbccaaa-aaabbcca.baacccbc-aaaccbbb.cccbbabb-cbbbaaac.abcbbbaa-aacccbca
abbccaaa-aaabbcca.baacccbb-ccaccbbb.cccbbaaa-abbbaaac.abcacccb-aacccbca
abbccaaa-aaabbcca.baacccbb-ccaccbbb.cccbbaaa-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aaabbcca.baacccbb-ccaccbbb.cccbbaaa-cbbbaaac.abcacccb-aacccbca
abbccaaa-aaabbcca.baacccbb-ccaccbbb.cccbbaaa-cbbbaaac.abcbbbaa-aacccbca
abbccaaa-aaabbcca.baacccbc-ccaccbbb.cccbbaaa-abbbaaac.abcacccb-aacccbca
abbccaaa-aaabbcca.baacccbc-ccaccbbb.cccbbaaa-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aaabbcca.baacccbc-ccaccbbb.cccbbaaa-cbbbaaac.abcacccb-aacccbca
abbccaaa-aaabbcca.baacccbc-ccaccbbb.cccbbaaa-cbbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbb-aaaccbbb.cccbbaaa-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbb-aaaccbbb.cccbbaac-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbb-aaaccbbb.cccbbabb-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbb-aaaccbbb.cccbbaaa-cbbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbb-aaaccbbb.cccbbaac-cbbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbb-aaaccbbb.cccbbabb-cbbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbc-aaaccbbb.cccbbaaa-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbc-aaaccbbb.cccbbaac-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbc-aaaccbbb.cccbbabb-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbc-aaaccbbb.cccbbaaa-cbbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbc-aaaccbbb.cccbbaac-cbbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbc-aaaccbbb.cccbbabb-cbbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbb-ccaccbbb.cccbbaaa-abbbaaac.abcacccb-aacccbca
abbccaaa-aabbbcca.baacccbb-ccaccbbb.cccbbaaa-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbb-ccaccbbb.cccbbaac-abbbaaac.abcacccb-aacccbca
abbccaaa-aabbbcca.baacccbb-ccaccbbb.cccbbaac-abbbaaac.abcbbbaa-aacccbca
abbccaaa-aabbbcca.baacccbb-ccaccbbb.cccbbaaa-cbbbaaac.abcacccb-aacccbca
```

Figure 4. bad.txt File

5 DIAGRAMMATIC VIEW OF WORD EXTENSION

In a single block of input there are 12 different words formed after concatenating 8 different suffix-prefix words from original input file (see figure 2). Extension is done twice. Firstly, the extension is done to the left and then to the right. Each of the words extends a single letter at a time.

Thus, we allow repetitions *uv* of length 1 (|u| = |v| = 1), i.e., we allow factors aa, bb, cc, aaa, bbb, ccc but no any longer abelian squares.

We denote these general productions as follows.
a -> g(a) = A = Apref …Asuffix
b -> g(b) = B = Bpref … Bsuff
c -> g(c) = C = Cpref … Csuff
d -> g(d) = D = Dpref … Dsuff. (Keränen, 2014.)

A pictorial representation of this setting is given below. The extension will continue until given (preselected) boundaries, marked by '|', are reached, or found out to be impossible to reach in which case, the whole block of word is unfavorable or bad text and its saved in bad.txt file.

The following two figures represent the extension of the word to the left and later, to the right with single character at a time.

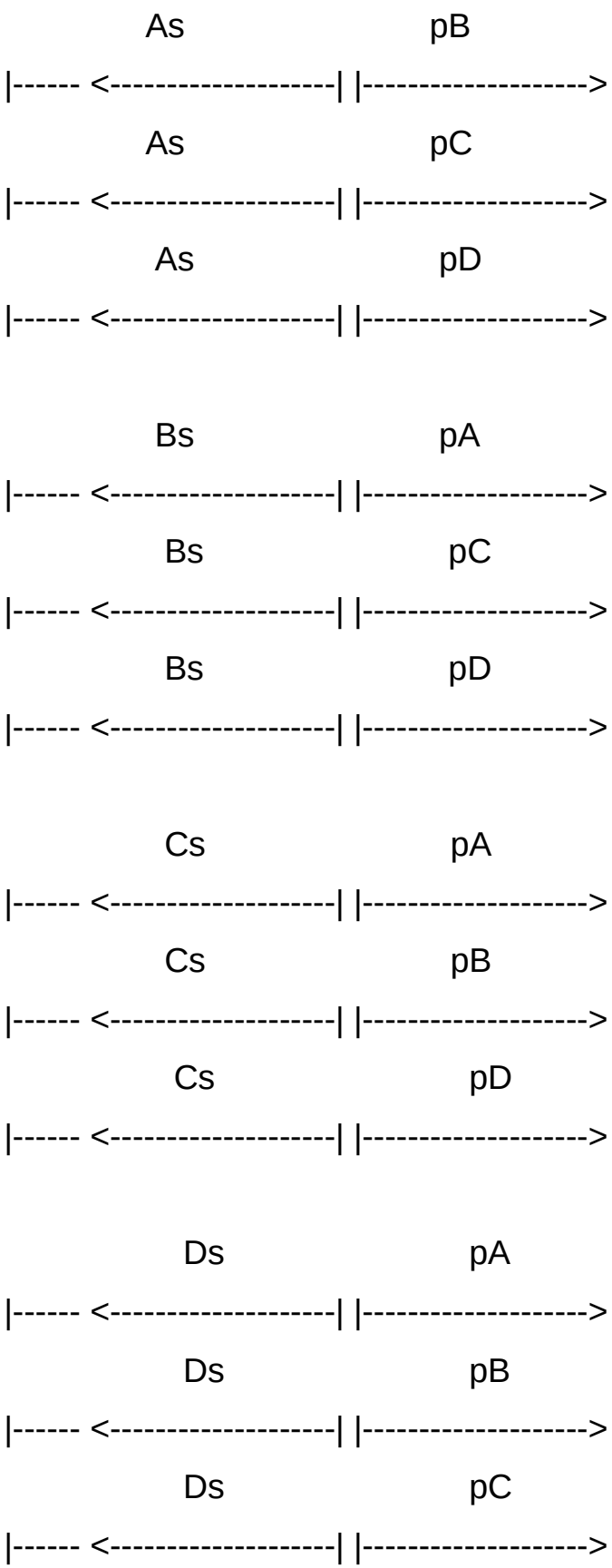

Figure 5. Extension to the Left.

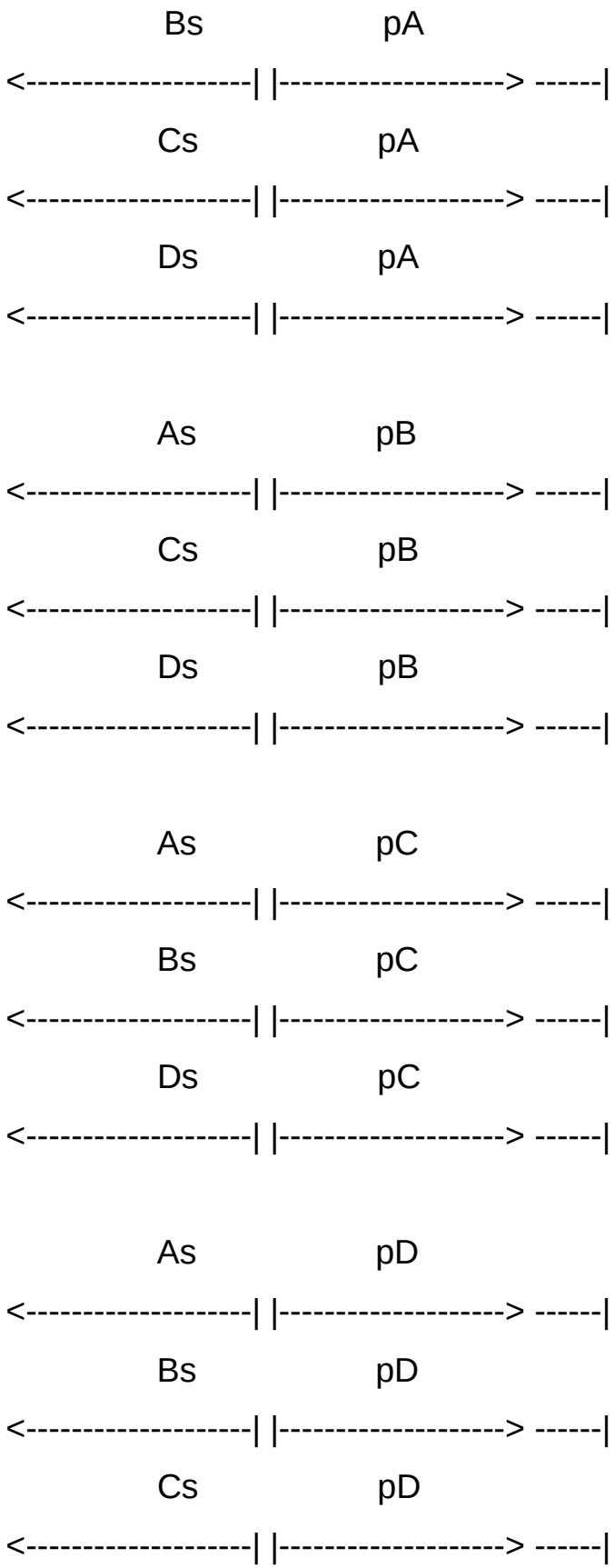

Figure 6. Extension to the Right

6 EXTENSION OF THREE WORDS AT A TIME

6.1 Extension to the Left

From an input file, within a single block, from 24 different concatenated words, three words are computed in parallel in each attempt. The character extension and a backtracking in all three words is the same.

The following diagrammatic representation shows how three words are extended to the left first and to the right with single character at a time. The process continues, and words are added and backtracked as necessary. Words are counted, as favorable if all three are anagram free.

```
              As                  pB
|-----pA1- <-------------------| |------------------->
              As                  pC
|-----pA1- <-------------------| |------------------->
              As                  pD
|-----pA1- <-------------------| |------------------->
```

Figure 7. Extension of the Words As.pB, As.pC and As.pD

The extended letter pA1 above is the same letter from (a, b or c) in all the occurrences.

```
              Bs                  pA
|-----pB1- <-------------------| |------------------->
              Bs                  pC
|-----pB1- <-------------------| |------------------->
              Bs                  pD
|-----pB1- <-------------------| |------------------->
```

Figure 8. Extension of the Words Bs.pA, Bs.pC and Bs.pD

The letter extended letter pB1 above is the same letter from (a, b or c) in all the occurrences.

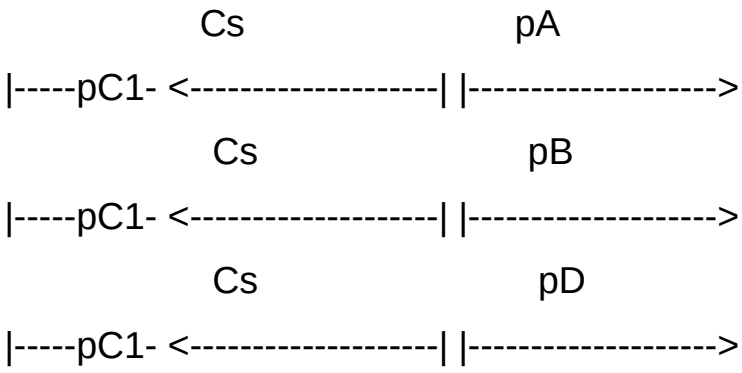

Figure 9. Extension of the Words Cs.pA, Cs.pB and Cs.pD

The letter extended letter pC1 above is the same letter from (a, b or c) in all the occurrences.

Figure 10. Extension of the Words Ds.pA, Ds.pB and Ds.pC

The letter extended letter pD1 above is the same letter from (a, b or c) in all the occurrences.

6.2 Extension to the Right

Words are extended towards right only after the execution for left extension is over. Extending to the right contains same image words like left extension but the direction of extension is different.

Figure 11. Extension of the Words Bs.pA, Cs.pA and Ds.pA

The extended letter pA1 above is the same letter from (a, b or c) in all of the occurrences.

Figure 12. Extension of the Words As.pB, Cs.pB and Ds.pB

The extended letter pB1 above is the same letter from (a, b or c) in all of the occurrences.

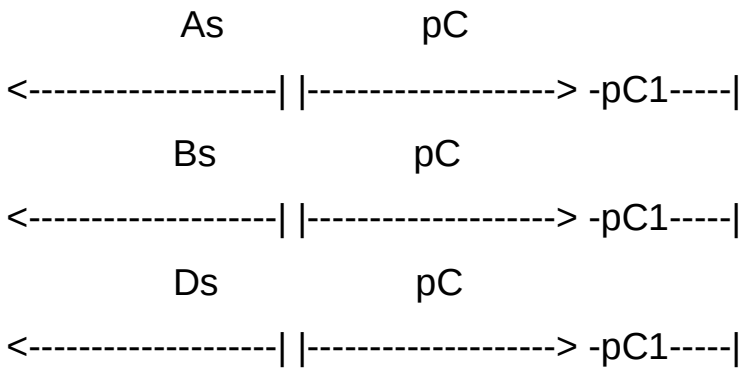

Figure 13. Extension of the Words As.pC, Bs.pC and Ds.pC

The extended letter pC1 above is the same letter from (a, b or c) in all of the occurrences.

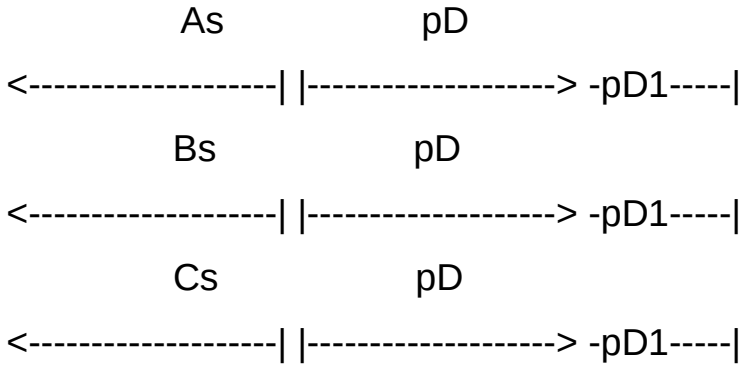

Figure 14. Extension of the Words As.pD, Bs.pD and Cs.pD

The extended letter pD1 above is the same letter from (a, b or c) in all of the occurrences.

7 WORD EXTENSION METHODOLOGY

For the word to become bad or good, it needs to go through series of testing when the program runs. The word extension methodology is presented below with an illustration. The direction of an arrow shows the side of extension (i.e., left or right) and letters after it shows the detected anagram. Some words contain neither arrow nor letters because they are already non-anagrams after extension of first letter and does not need to be tested further for the given boundary length of '1', but this does not apply to boundary length greater than 1.

Let's take the following block of input, which is in the form of {{As,Ap},{Bs,Bp},{Cs,Cp},{Ds,Dp}}.
{{{aaabbcca,abbccaaa},{aaaccbbb,baacccbb},{abbbaaac,cccbbaaa},{aacccbca,abcbbbaa}}

The letter 'a' extends to the left with the length 1.
AspB: a-aaabbcca.baacccbb ← aa aa
AspC: a-aaabbcca.cccbbaaa ← aa aa
AspD: a-aaabbcca.abcbbbaa ← aa aa

BspA: a-aaaccbbb.abbccaaa ← aa aa
BspC: a-aaaccbbb.cccbbaaa ← aa aa
BspD: a-aaaccbbb.abcbbbaa ← aa aa

CspA: a-abbbaaac.abbccaaa
CspB: a-abbbaaac.baacccbb
CspD: a-abbbaaac.abcbbbaa

DspA: a-aacccbca.abbccaaa
DspB: a-aacccbca.baacccbb ← aaacccb cabaacc
DspC: a-aacccbca.cccbbaa
Since Extension with A, B and D is detected an anagram so after modification (a->b) i.e., 'a' is modified to 'b' it would look,

AspB: b-aaabbcca.baacccbb
AspC: b-aaabbcca.cccbbaaa
AspD: b-aaabbcca.abcbbbaa

BspA: b-aaaccbbb.abbccaaa
BspC: b-aaaccbbb.cccbbaaa
BspD: b-aaaccbbb.abcbbbaa

DspA: b-aacccbca.abbccaaa ← baacc cbcaa
DspB: b-aacccbca.baacccbb
DspC: b-aacccbca.cccbbaaa
Since, DspA is still anagram so (b->c) i.e., b is modified to ‘c’.

DspA: c-aacccbca.abbccaaa ← ca ac
DspB: c-aacccbca.baacccbb ← ca ac
DspC: c-aacccbca.cccbbaaa ← ca ac

While extending to the left with single letter, it seems that suffix of D cannot be extended at all with all the possible letters from (a, b and c) so, ultimately word is unfavorable. It is not necessary to represent the extension to the right. It already proves that the word is unfavorable. The word is saved into bad.txt file as, *“abbccaaa-aaabbcca.baacccbb-aaaccbbb.cccbbaaa-abbbaaac.aacccbca-abcbbbaa”.*

Similarly, extension of word with higher boundary length can be carried out. For the word to become so-far-so-good, all 12 different words should be square free.

8 CUMULATIVE SUM

The alternative way of finding existence of abelian square is cumulative sum. Cumulative sum is a sequence of partial sums of given sequence.

For example, the word “aaabbbccca” can be presented with its position, is shown in the following table.

	a	a	a	b	b	b	c	c	c	a
$\begin{pmatrix}0\\0\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\\0\end{pmatrix}$	$\begin{pmatrix}2\\0\\0\end{pmatrix}$	$\begin{pmatrix}3\\0\\0\end{pmatrix}$	$\begin{pmatrix}3\\1\\0\end{pmatrix}$	$\begin{pmatrix}3\\2\\0\end{pmatrix}$	$\begin{pmatrix}3\\3\\0\end{pmatrix}$	$\begin{pmatrix}3\\3\\1\end{pmatrix}$	$\begin{pmatrix}3\\3\\2\end{pmatrix}$	$\begin{pmatrix}3\\3\\3\end{pmatrix}$	$\begin{pmatrix}4\\3\\3\end{pmatrix}$
P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}

Where, P_0, P_1, …, $P_{n/2}$, $P_{n/2+1}$…..P_{n-1}, P_n. is an expression for the position of a word to its infinite length.

Calculating above equation from right, checking whether ‘*ca’* and ‘*cc’* are anagrams of each other.

We have letters named ‘*ca*’= $\vec{v}$ and ‘cc’ = $\vec{u}$ which are equal in length so,

$$\vec{v} = \vec{u}.$$

$$\text{Or, } \vec{v} - \vec{u} = \vec{P}_{10} - \vec{P}_8 - (\vec{P}_8 - \vec{P}_6)$$

$$= \vec{P}_{10} - 2\vec{P}_8 + \vec{P}_6$$

$$= \begin{pmatrix}4\\3\\3\end{pmatrix} - 2X\begin{pmatrix}4\\3\\2\end{pmatrix} + \begin{pmatrix}3\\3\\0\end{pmatrix}$$

$$\therefore \vec{v} - \vec{u} \neq 0$$

Since, the result of an above calculation is non-zero, which means image word ‘cc’ and ‘ca’ isn’t anagram of each other, which is true.

Similarly, calculation by taking three image words from right respectively as ‘acc’ and ‘cbb’.

Since, the length of image word equals each other i.e., 3. So,

$$\vec{v} = \overrightarrow{u}.$$

Or, $\vec{v}$ - $\vec{u}$ = $\vec{P}_{10} - \vec{P}_7 - (\vec{P}_7 - \overrightarrow{P}_5)$

$$= \vec{P}_{10} - 2\vec{P}_7 + \vec{P}_5$$

$$= \begin{pmatrix} 4 \\ 3 \\ 3 \end{pmatrix} - 2X \begin{pmatrix} 3 \\ 3 \\ 1 \end{pmatrix} + \begin{pmatrix} 3 \\ 2 \\ 0 \end{pmatrix}$$

$\therefore$ $\vec{v}$ - $\vec{u}$ $\neq$ 0 which also means those selected letters aren't abelian square yet. With further checking by taking the length and incrementing with single character in both $\vec{v}$ and $\vec{u}$ and checking as above examples will certainly give a perfect result.

9 TECHNOLOGIES USED

9.1 Code::Blocks

According to the Code::Blocks, “it is a *free C, C++ and Fortran IDE* built to meet the most demanding needs of its users. It is designed to be very extensible and fully configurable.” Code::Blocks also supports D and Fortran in addition to C and C++.

Code::blocks was released in the year 2005. Making all the way up to version 15.12 and more, it supports multiple compilers, including MingGW, Gcc, Microsoft Visual C++ and so on. (Code::Blocks 2016.)

9.2 Scintilla and MinGW

Scintilla is a free source code editing component. It is a cross platform originally written in C++. Scintilla comes with complete source code. Scintilla provides complete feature for debugging and editing codes. Scintilla comes integrated with Code::blocks. (Scintilla 2016.)

MinGW (*Minimalist GNU for Windows*) is a native microsoft windows application development environment. MinGW contains C program libraries for implementation of -gcc. In order to compile the program, especially in windows, it requires a different approach, where MinGW is one of many. For Code::blocks, in order to use MinGW, at first it must be installed in a computer separately besides installing Code::blocks itself. (MinGW 2016.)

10 CONCLUSION

Finding more favorable words by testing pre-generated so-far-so-good words is only half of the work done in this research. The bigger challenge for now is to find good words that can be extended infinitely. In order to keep going with this experimentation, it requires much work to be done with an assistance of powerful computers and well written algorithms for reduction of computation duration as well as for proper memory management.

Computation duration may seem shorter with smaller boundary length but the workload for computers and execution time taken are directly proportional to each other with elevated boundary lengths. Thus, in the near future, researchers can test so-far-so-good words generated by the author’s program with more advanced algorithm to narrow down the results in finding good words.

BIBLIOGRAPHY

Barney, B. 2016. Introduction to Parallel Computing. Address: https://computing.llnl.gov/tutorials/parallel_comp/#Whatis. Accessed 7 April 2016.

Code::Blocks 2016. The open source, cross platform, free C, C++ and Fortran IDE. Address: http://www.codeblocks.org/. Accessed 21 March 2016.

Hughes, C.& Hughes, T. 2004. The Joys of Concurrent Programming. Address:http://www.informit.com/articles/article.aspx?p=30413&seqNum=2. Address 29 March 2016.

Keränen, V. 2009a. A powerful abelian square-free substitution over 4 letters. Theoretical computer science -TCS. vol. 410, no. 38-40, pp. 3893-3900.

Keränen, V. 2009b. Combinatorics on Words. The Mathematica Journal. no, 358-375.

Lin, J. 2011. Classification of abelian square-free words over three letters. Address:http://www.theseus.fi/bitstream/handle/10024/38608/Lin_Jing.pdf?sequence=1. Accessed 14 March 2016.

MinGW 2012. Welcome to MinGW.org. Address: http://www.mingw.org/. Accessed 23 March 2016.

Scintilla 2016. A free source code editor for Win32 and X. Address: http://www.scintilla.org/SciTE.html. Address 1 April 2016.

Wolfram MathWorld 2016a. Cumulative Sum. Address: http://mathworld.wolfram.com/CumulativeSum.html. Accessed 9 April 2016.

Worlfram MathWorld 2016b. Parallel Computing. Address: http://mathworld.wolfram.com/ParallelComputing.html. Accessed 7 April 2016.

APPENDICES

Appendix 1

header.c

```
#ifndef HEADER_H_INCLUDED
#define HEADER_H_INCLUDED

bool is_anagram (char w1[], char w2[]);
void filter_left_right(char word[],char word1[],char word2[],int maxlength,int
function_number);
void extracting_64_lettors(char word[],int maxlength,int wordlength);
void suffix_prefix_divison(char word[],const int maxlength);
void word_concatinations(char AS[], char AP[], char BS[],char BP[], char
CS[], char CP[], char DS[],char DP[],const int maxlength);
void word_formation(char word[],char word1[],char word2[],int maxlength);
void saveValueToFile(FILE *f, char s[]);
bool boolean_anagram_return(char side_a[],char side_b[],char side_c[],char
side_d[],char side_e[],char side_f[]);
int word_boundry_count = 0;
int count_processed_words=0;

#endif // HEADER_H_INCLUDED
```

main.c

```
/*****************************************************************************
Author: Subhash Rijal

 *****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "time.h"
#include <stdbool.h>
#include "header.h"
#include <string.h>
#define length_of_single_block 64
#define origional_length_of_word 16
#define extended_word_number 24
#define _size_index 500

FILE *sfsg;
FILE *fbad;
char origional_word_save[length_of_single_block+6];
int main()
{
    int left;
    int maxlength;
    int wordlength;
    FILE *f;
    long size;
    sfsg = fopen("sfsg.txt","w+");
    fbad = fopen("bad.txt","w+");

    if ( (f = fopen("input.txt", "r")) == NULL )
    {
        printf("input.txt open error!");
        exit(0);
    }
    else
    {
        fseek(f,0,SEEK_END);
        size=  ftell(f);
        rewind(f);

        char wordExtractFromFile[size];
        memset(wordExtractFromFile,'\0',sizeof(size));
        int i = 0,j=0;
        char fileExtract[size];

        while((fileExtract[i]=getc(f))!=EOF)
        {
            i++;
        }
```

```
        for(i=0; i<=size-1; i++)
        {
            if(isalnum(fileExtract[i]))
            {
                wordExtractFromFile[j]=fileExtract[i];
                j++;
            }
        }
        wordlength=strlen(wordExtractFromFile);
        printf("Enter the maxLength: ");
        scanf("%d",&left);
        maxlength = origional_length_of_word/* 16 is the  length of word initially  */+
            left;//at the moment only left extension

        extracting_64_lettors(strrev(wordExtractFromFile),maxlength,wordlength);
        fclose(f);
        fclose(sfsg);
        fclose(fbad);
    }
}

void extracting_64_lettors(char word[],int maxlength,int wordlength)
{
    int j;
    int count;
    char dummy_word[length_of_single_block+2];
            //filtering/counting 64 words per iteration.
    for(count=0; count<wordlength/length_of_single_block; count++)
    {
        memset(dummy_word,'\0',sizeof(dummy_word));
        //printf("\n%d",count);
        for(j=0; j<length_of_single_block; j++)
        {
            dummy_word[j]=word[strlen(word)-1];
            word[strlen(word)-1]='\0';
        }
        suffix_prefix_divison(dummy_word,maxlength);
    }
}

void suffix_prefix_divison(char word[],const int maxlength)
{
    int i = 0;
    char aSuffix[(strlen(word)/8)+2];
    char aPrefix[(strlen(word)/8)+2];
    char bSuffix[(strlen(word)/8)+2];
    char bPrefix[(strlen(word)/8)+2];
    char cSuffix[(strlen(word)/8)+2];
    char cPrefix[(strlen(word)/8)+2];
    char dSuffix[(strlen(word)/8)+2];
    char dPrefix[(strlen(word)/8)+2];
```

```
    for(i=0; i<=strlen(word); i++)
    {
       /*making memory free before storing the values in an  array.*/
       memset(aSuffix,'\0',sizeof(aSuffix));
       memset(aPrefix,'\0',sizeof(aPrefix));
       memset(bSuffix,'\0',sizeof(bSuffix));
       memset(bPrefix,'\0',sizeof(bPrefix));
       memset(cSuffix,'\0',sizeof(cSuffix));
       memset(cPrefix,'\0',sizeof(cPrefix));
       memset(dSuffix,'\0',sizeof(dSuffix));
       memset(dPrefix,'\0',sizeof(dPrefix));
       /*the fix number of words designed to be in suffixes and prefixes are
             extracted through these methods.*/
       strncpy(aSuffix,&word[(strlen(word))-i],sizeof(aSuffix)-2);
       strncpy(aPrefix,&word[strlen(word)-(strlen(word)-8)],sizeof(aPrefix)-2);
       strncpy(bSuffix, &word[(strlen(word))-(strlen(word)-16)], sizeof(bSuffix)-2);
       strncpy(bPrefix, &word[(strlen(word))-(strlen(word)-24)], sizeof(bPrefix)-2);
       strncpy(cSuffix, &word[(strlen(word))-(strlen(word)-32)], sizeof(bPrefix)-2);
       strncpy(cPrefix, &word[(strlen(word))-(strlen(word)-40)], sizeof(bPrefix)-2);
       strncpy(dSuffix, &word[(strlen(word))-(strlen(word)-48)], sizeof(bPrefix)-2);
       strncpy(dPrefix, &word[(strlen(word))-(strlen(word)-56)], sizeof(bPrefix)-2);
    }

    word_concatinations(aSuffix, aPrefix, bSuffix,bPrefix, cSuffix, cPrefix,
      dSuffix,dPrefix,maxlength);
}
/*joining all the words*/
void word_concatinations(char AS[], char AP[], char BS[],char BP[], char
CS[], char CP[], char DS[],char DP[],const int maxlength)
{
    char ASBP[_size_index]= {'\0'};
    char ASCP[_size_index]= {'\0'};
    char ASDP[_size_index]= {'\0'};
    char BSAP[_size_index]= {'\0'};
    char BSCP[_size_index]= {'\0'};
    char BSDP[_size_index]= {'\0'};
    char CSAP[_size_index]= {'\0'};
    char CSBP[_size_index]= {'\0'};
    char CSDP[_size_index]= {'\0'};
    char DSAP[_size_index]= {'\0'};
    char DSBP[_size_index]= {'\0'};
    char DSCP[_size_index]= {'\0'};
    int i = 0;

    //file saving format made easier, saved in this way:: pA-As.pB-Bs.pC-Cs.pD-Ds

      sprintf(origional_word_save,"%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s"
      ,AP,"-",AS,".",BP,"-",BS,".",CP,"-",CS,".",DP,"-",DS);

    //Extension to the left
```

```
    snprintf(ASBP, sizeof(ASBP), "%s%s",  AS, BP);
    snprintf(ASCP, sizeof(ASCP), "%s%s",  AS, CP);
    snprintf(ASDP, sizeof(ASDP), "%s%s",  AS, DP);
    filter_left_right(ASBP,ASCP,ASDP,maxlength,i=0);

    snprintf(BSAP, sizeof(BSAP), "%s%s",  BS, AP);
    snprintf(BSCP, sizeof(BSCP), "%s%s",  BS, CP);
    snprintf(BSDP, sizeof(BSDP), "%s%s",  BS, DP);
    filter_left_right(BSAP,BSCP,BSDP,maxlength,i=0);

    snprintf(CSAP, sizeof(CSAP), "%s%s",  CS, AP);
    snprintf(CSBP, sizeof(CSBP), "%s%s",  CS, BP);
    snprintf(CSDP, sizeof(CSDP), "%s%s",  CS, DP);
    filter_left_right(CSAP,CSBP,CSDP,maxlength,i=0);

    snprintf(DSAP, sizeof(DSAP), "%s%s",  DS, AP);
    snprintf(DSBP, sizeof(DSBP), "%s%s",  DS, BP);
    snprintf(DSCP, sizeof(DSCP), "%s%s",  DS, CP);
    filter_left_right(DSAP,DSBP,DSCP,maxlength,i=0);

    //Extending to the right side. making i =1 just to filter if   condition

    snprintf(BSAP, sizeof(BSAP), "%s%s",  BS, AP);
    snprintf(CSAP, sizeof(CSAP), "%s%s",  CS, AP);
    snprintf(DSAP, sizeof(DSAP), "%s%s",  DS, AP);
    filter_left_right(BSAP,CSAP,DSAP,maxlength,i=1);

    snprintf(ASBP, sizeof(ASBP), "%s%s",  AS, BP);
    snprintf(CSBP, sizeof(CSBP), "%s%s",  CS, BP);
    snprintf(DSBP, sizeof(DSBP), "%s%s",  DS, BP);
    filter_left_right(ASBP,CSBP,DSBP,maxlength,i=1);

    snprintf(ASCP, sizeof(ASCP), "%s%s",  AS, CP);
    snprintf(BSCP, sizeof(BSCP), "%s%s",  BS, CP);
    snprintf(DSCP, sizeof(DSCP), "%s%s",  DS, CP);
    filter_left_right(ASCP,BSCP,DSCP,maxlength,i=1);

    snprintf(ASDP, sizeof(ASDP), "%s%s",  AS, DP);
    snprintf(BSDP, sizeof(BSDP), "%s%s",  BS, DP);
    snprintf(CSDP, sizeof(CSDP), "%s%s",  CS, DP);
    filter_left_right(ASDP,BSDP,CSDP,maxlength,i=1);
}



/*filter right and left (is this method needed?? No!!)*/
void filter_left_right(char word[],char word1[],char word2[],int maxlength,int
function_number)

{
```

```
    if(function_number==0)
    {
        word_formation(strrev(word),strrev(word1),strrev(word2),maxlength);
    }
    if(function_number==1)
    {
        word_formation(word,word1,word2,maxlength);
    }
}

//main word distibution
void word_formation(char word[],char word1[],char word2[],int maxlength)
{
    //counting three words per time.
    count_processed_words=count_processed_words+3;
    int initialWordIsGood = 0;
    char
      side_A[(maxlength/2)+2],side_B[(maxlength/2)+2],side_A1[(maxlength/2)+2],
      side_B1[(maxlength/2)+2],side_A2[(maxlength/2)+2],side_B2[(maxlength/2)+
      2];
    int initialLength = strlen(word)-1;
    strcat(word1,"a"),strcat(word2,"a"),strcat(word,"a");
    int i =0;
    char state = 't';
    while(strlen(word) - 1 > initialLength)
    {
        switch(state)
        {
        case 't'://state 't' for testing
            for(i = 1; state=='t'&&i <= ((strlen(word)-1)/ 2);  i++)
            {
                memset(side_A, '\0', sizeof(side_A));
                memset(side_B, '\0', sizeof(side_A));
                memset(side_A1, '\0', sizeof(side_A));
                memset(side_B1, '\0', sizeof(side_A));
                memset(side_A2, '\0', sizeof(side_A));
                memset(side_B2, '\0', sizeof(side_B));

                strncpy(side_A, &word2[(strlen(word2)-1)-i], i+1);
                strncpy(side_B, &word2[(strlen(word2)-2)-i*2], i+1);
                strncpy(side_A1, &word1[(strlen(word1)-1)-i], i+1);
                strncpy(side_B1, &word1[(strlen(word1)-2)-i*2], i+1);
                strncpy(side_A2, &word[(strlen(word)-1)-i], i+1);
                strncpy(side_B2, &word[(strlen(word)-2)-i*2], i+1);

                /*state is turned to 'm' if any any of words are anagrams */
                      if(boolean_anagram_return(side_A,side_B,side_A1,side_B1,side
                      _A2,side_B2))
                {
                   state='m';//state 'm' for modification
                }
```

```
        }
        if(strlen(word)== maxlength)
        {
            if(state == 't')
            {

                initialWordIsGood = 3;/*initial word is 3, to notify its processing 3
words */

            }
            state = 'm';
        }
        else if(state == 't')
        {
            strcat(word1, "a");
            strcat(word2, "a");
            strcat(word, "a");
        }
        break;
    case 'm':
        switch(word[strlen(word)-1]) /*last letters of word, word1 and word2 are
                                        same */
        {
        case 'a':
            word1[strlen(word1)-1] = 'b';
            word2[strlen(word2)-1] = 'b';
            word[strlen(word)-1] = 'b';
            state = 't';
            break;
        case 'b':
            word1[strlen(word)-1] = 'c';
            word2[strlen(word)-1] = 'c';
            word[strlen(word)-1] = 'c';

            state = 't';
            break;
        case 'c':
            word1[strlen(word)-1] = '\0';
            word2[strlen(word)-1] = '\0';
            word[strlen(word)-1] = '\0';
            break;
        }
        break;
    }
}

word_boundry_count = word_boundry_count+initialWordIsGood;//counting and
    adding    the word in boundary.

if(count_processed_words==extended_word_number)
{
```

```
            if(initialWordIsGood==0||word_boundry_count<extended_word_number)
            {
                saveValueToFile(fbad, origional_word_save);
            }
else if(initialWordIsGood==3&&word_boundry_count%extended_word_number==0)
            {
                saveValueToFile(sfsg, origional_word_save);
            }
            word_boundry_count=0;
            count_processed_words=0;
        }

    }

    /*Anagram detection*/
    bool is_anagram (char w1[], char w2[])
    {
        unsigned int i, sz; /* The histogram */ int freqtbl[26]; /* Sanity check */

        if ((sz = strlen(w1)) != strlen(w2)) return false; /* Initialize the histogram */
          memset(freqtbl, 0, 26*sizeof(int));

        /* Read the first string, incrementing the corresponding histogram entry */ for (i
    = 0; i < sz; i++)
        {
            if (w1[i] >= 'A' && w1[i] <= 'Z') freqtbl[w1[i]-'A']++;
            else if (w1[i] >= 'a' && w1[i] <= 'z') freqtbl[w1[i]-'a']++;
            else

                return false;

        }
        /* Read the second string, decrementing the corresponding histogram entry */
    for (i = 0; i < sz; i++)
        {

            if (w2[i] >= 'A' && w2[i] <= 'Z')
            {
                if (freqtbl[w2[i]-'A'] == 0) return false;
                freqtbl[w2[i]-'A']--;

            }
            else if (w2[i] >= 'a' && w2[i] <= 'z')
            {
                if (freqtbl[w2[i]-'a'] == 0) return false;
                freqtbl[w2[i]-'a']--;
            }
            else
            {
                return false;
            }
```

```
    }
    return true;
}

bool boolean_anagram_return(char side_a[],char side_b[],char side_c[],char
side_d[],char side_e[],char side_f[])
{
/*Testing all three words whether they are anagram or not.*/
    if(is_anagram(side_a,side_b))
    {
        return true;
    }
    if(is_anagram(side_c,side_d))
    {
        return true;
    }
    if(is_anagram(side_e,side_f))
    {
        return true;
    }
    return false;
}

void saveValueToFile(FILE *f, char s[]) //Note: s[] and start_time should be a
pointers instead for better memory management
{
    if(strlen(s) > 1)
    {
        fprintf(f, "%s\n",s);
    }
}
```