Non-freeness of subgroups of SL(2,R) generated by two parabolics

Chan J.D. and Tan S. P.

Faculty of Science, National University of Singapore
21 Lower Kent Ridge Rd, Singapore 119077

ABSTRACT

For $q \in \mathbb{Q} \cap (0,4)$, we denote

$$a = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \qquad b_q = \begin{pmatrix} 1 & q \\ 0 & 1 \end{pmatrix}$$

and write G_q for the subgroup of $\mathrm{SL}_2(\mathbb{R})$ generated by a and b_q. In this project, we attempt to develop a better algorithm in order to find some non-trivial matrix $w(a, b_q)$ where

$$w(a, b_q) = b_q^{m_1} \cdot a^{m_2} \cdot b_q^{m_3} \cdot a^{m_4} \cdot \ldots \cdot a^{m_{k-1}} \cdot b_q^{m_k}$$

where $m_1, \ldots, m_k \in \mathbb{Z} \setminus \{0\}$ and $k \in \mathbb{Z} \wedge k \geq 3$, such that w is lower triangular. From there, according to Kim and Koberda, we can then find a non-trivial series of multiplications of a and b_q matrices which form the identity in $\mathrm{SL}_2(\mathbb{R})$, thus showing that the G_q is not free.

Kim and Koberda developed an algorithm used to find such a matrix. By starting with the 1×2 matrix $(1, 0)$, they post-multiply it with alternating b_q and a matrices raised to some non-zero power while trying to keep the values in the 1×2 matrix small, until it eventually arrives back at $(1, 0)$. Then, all the matrices that we have post-multiplied with $(1, 0)$ will form a lower-triangular matrix. However, among the tests that they ran, it fails for the following four numbers,

$$\frac{35}{9}, \frac{39}{10}, \frac{28}{17}, \frac{29}{17}.$$

They then developed a second, usually slower algorithm which found such a matrix for the latter two numbers but not the former two. Our goal is to develop a better algorithm that solves for the first two numbers and can be applied to a wider range of numbers.

We introduce the idea of killer intervals by Long and Reid into the first algorithm mentioned which helps to keep the numbers in the 1×2 matrix small. In this project, we consider how to generate and incorporate these killer intervals to improve the algorithm. With this, we create a faster algorithm that usually finds shorter words and we show that $39/10$ does have such a lower triangular matrix and $G_{39/10}$ is not free.

1. Introduction

For $q \in \mathbb{Q} \cap (0, 4)$, we denote

$$a = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \qquad b_q = \begin{pmatrix} 1 & q \\ 0 & 1 \end{pmatrix}$$

and write G_q for the subgroup of $\mathrm{SL}_2(\mathbb{R})$ generated by a and b_q. In this project, we attempt to develop a better algorithm in order to find some non-trivial matrix $w(a, b_q)$ where

$$w(a, b_q) = b_q^{m_1} \cdot a^{m_2} \cdot b_q^{m_3} \cdot a^{m_4} \cdot \ldots \cdot a^{m_{k-1}} \cdot b_q^{m_k}$$

where $m_1, \ldots, m_k \in \mathbb{Z} \setminus \{0\}$ and $k \in \mathbb{Z} \wedge k \geq 3$, such that w is lower triangular. From there, according to Kim and Koberda (Lemma 2.2 in [1]), we can then find a non-trivial series of multiplications of a and b_q matrices which form the identity in $\mathrm{SL}_2(\mathbb{R})$, thus showing that the G_q is not free.

Definition 1.1: *We call a matrix $w(a, b_q)$ a word if*

$$w(a, b_q) = b_q^{m_1} \cdot a^{m_2} \cdot b_q^{m_3} \cdot a^{m_4} \cdot \ldots \cdot a^{m_{k-1}} \cdot b_q^{m_k}$$

where $m_1, \ldots, m_k \in \mathbb{Z} \setminus \{0\}$ with length k. The trivial word will then be identified as the identity matrix of length 0.

Definition 1.2: *We call a matrix a letter $L(x, n)$ if*

$$L(x, n) = x^n$$

for some matrix x and $n \in \mathbb{Z} \setminus \{0\}$.

Thus, it can be seen that a word w of length k is made up of k alternating letters.
We will also denote the rational number q as s/r where s and r are coprime.

Program names will be written in the following font `Program_Name` for ease of reference to the Appendix and programs.

2. First Algorithm

We start with the algorithm written by Kim and Koberda (Appendix A in [1]). We start with the shifted remainder function.

Definition 2.1: *Let $x, y \in \mathbb{Z} \setminus \{0\}$. Setting $t = x - y\lfloor x/y \rfloor$, we define a shifted remainder of x by y as*

$$\mathrm{SR(x,y)} \coloneqq \begin{cases} t, & \text{if } t = x + y, \\ t, & \text{if } t \neq x \text{ and } |t| \leq |y| \,/\, 2, \\ t - y, & \text{otherwise} \end{cases}$$

For the purposes of computation, we only work with integers. Additionally, we identify a matrix to be equivalent to one where all its values are multiplied by the same number. Specifically, for numbers $x, y, a, b, c, d \in \mathbb{Z}$ and $r_1, r_2 \in \mathbb{Z} \setminus \{0\}$,

$$(x, y) = (r_1 x, r_1 y) \text{ and } \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} r_2 a & r_2 b \\ r_2 c & r_2 d \end{pmatrix}.$$

The main purpose of this is to make calculations simpler for the computer by reducing every number by their greatest common denominator and to avoid floating point errors when working with non-integers due to computing limitations.

In our algorithm, we start with a 1×2 matrix $(1, 0)$ and define some operations.

(1) *For some $x_1, y_1, x_2, y_2, x_3, y_3 \in \mathbb{Z}$, we define two types of post multiplication by the matrices a and b_q. For some $m, n \in \mathbb{Z} \setminus \{0\}$,*

$$(x_1, y_1) \cdot a^m = (x_2, y_2)$$

will be denoted with a single-headed arrow

$$(x_1, y_1) \overset{m}{\rightarrow} (x_2, y_2),$$

and

$$(x_2, y_2) \cdot b_q^n = (x_3, y_3)$$

will be denoted with a double-headed arrow

$$(x_2, y_2) \overset{n}{\twoheadrightarrow} (x_3, y_3).$$

We will often leave out the power above the arrow to denote multiplication of the corresponding matrix raised to some integer non-zero power. The result on the right will determine the power of the matrix to be multiplied.

Additionally, we define an additional type of operation that is not used in [1].

(2) *For some $x_1, y_1, x_2, y_2 \in \mathbb{Z}$, we define the post multiplication with some word, w*

$$(x_1, y_1) \cdot w = (x_2, y_2)$$

will be denoted with an arrow with a tail

$$(x_1, y_1) \overset{w}{\mapsto} (x_2, y_2).$$

Algorithm 2.2 (Naïve Algorithm): We can then describe the first variant of the algorithm used (`Naïve`). For a rational number $q = s/r$, it begins with

$$\begin{aligned}(1,0) = (r,0) \overset{1}{\twoheadrightarrow} (r,s) \to (\mathrm{SR}(r,s), s) = (r\mathrm{SR}(r,s), rs) \\ \overset{-1}{\twoheadrightarrow} (r\mathrm{SR}(r,s), s\big(r - \mathrm{SR}(r,s)\big) =: (x_0, sy_0)\end{aligned}$$

Then, each subsequent iterative step is given by

$$\begin{aligned}(x_i, sy_i) \to (\mathrm{SR}(x_i, sy_i), sy_i) = (x', sy_i) = (rx', rsy_i) \twoheadrightarrow \big(rx', s \cdot \mathrm{SR}(ry_i, x')\big) \\ = (x'', sy') = (x''\sigma/d\,, sy'\sigma/d) = (x_{i+1}, y_{i+1})\end{aligned}$$

where

$$\sigma = \begin{cases} 1, & \text{if } x'' \geq 0 \\ -1, & \text{otherwise} \end{cases}, \text{ and } d = \gcd(x'', y')$$

Note that as a result, x_i is always positive and only y_i can be negative. This will be a common feature in any variation of the algorithm. We always maintain that the negative sign will fall on y_i. After every iteration, we also divide x_i and y_i by their greatest common denominator. The algorithm terminates when $y_i = 0$ or $(x_i, y_i) = (x_j, y_j)$ for $\exists j < i$.

Case 1 ($y_i = 0$): This implies that there exists some non-trivial word $w\big(a, b_q\big)$ such that

$$(1,0) \overset{w(a,b_q)}{\mapsto} (1,0)$$

showing $w(a, b_q)$ is lower triangular and that G_q is not free.

Case 2 ($(x_i, y_i) = (x_j, y_j)$ for $\exists j < i$): This implies that there exists two words $w_1\big(a, b_q\big), w_2(a, b_q)$ such that

$$(1,0) \overset{w_1(a,b_q)}{\mapsto} (x_i, y_i) \overset{w_2(a,b_q)}{\mapsto} (x_i, y_i)$$

Thus, it is evident that

$$(1,0) \overset{w_1(a,b_q)}{\mapsto} (x_i, y_i) \overset{w_2(a,b_q)}{\mapsto} (x_i, y_i) \overset{\left(w_1(a,b_q)\right)^{-1}}{\mapsto} (1,0)$$

Then the word

$$w(a, b_q) = w_1(a, b_q) \cdot w_2(a, b_q) \cdot \left(w_1(a, b_q)\right)^{-1}$$

will be non-trivial as long as $w_2(a, b_q)$ is non-trivial and $w(a, b_q)$ will also be lower triangular.

Note: The 1×2 matrix represented by (x_i, y_i) will sometimes also be referred to as the rational number x_i/y_i. In this case, $(1, 0)$ will then refer to ∞ and we are looking for some non-trivial word that maps ∞ back to ∞.

The algorithm essentially tries to multiply the (x_0, y_0) by an alternating series of $L(a, m)$ followed by $L(b_q, n)$, for some $m, n \in \mathbb{Z} \setminus \{0\}$, while trying to keep the resulting (x_i, y_i) after every step as small as possible. We then hope that it eventually ends up in either case 1 or case 2.

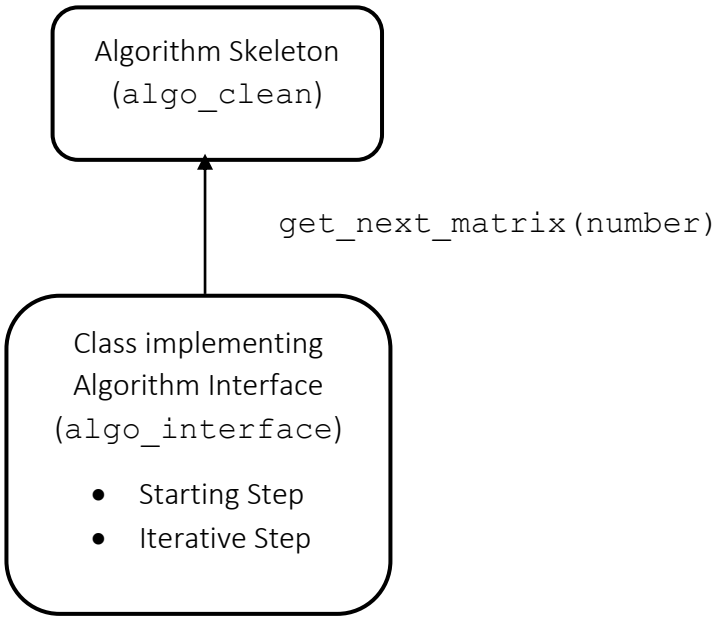
However, algorithm implemented in [1] only returns $true$ or $false$ to indicate if the algorithm terminates, showing that such a word can be found within the given number of steps. Additionally, the results in [1] when testing on rational numbers $q = s/r \in \mathbb{Q} \cap (0,4)$ and $r \leq 10$ yields that for the following two numbers

$$\frac{35}{9}, \frac{39}{10},$$

this algorithm is unable to find such a word in 5000 steps.

3. Program Structure

The original program in [1] was done in Mathematica. However, for this project, the codes were written in Python. The basic structure of the program then looks something like this:



Program 3.1 (Algorithm Skeleton): The Algorithm Skeleton determines the most basic outline for the algorithm. It requires a number $q = s/r$, the number of iterations n, and a class of algorithm. It starts by instantiating an algorithm object of the algorithm class with the numbers s and r. With each iteration, it provides the algorithm with the current number and queries for the next matrix (letter or word) to post-multiply the number with. It keeps track of the number after each iterative step and records the number and its associated word.

It will check if the algorithm has reached either case 1 or 2 of Algorithm 2.2 after every iteration. If the algorithm terminates with case 2, it has looped back to a number it has arrived at before. The program will then reference the word associated with that number the first time it is reached and construct the word according to case 2. The modified algorithms will still be using this same algorithm skeleton and only the iterative step is changed. More details can be found in Appendix A.1.

Program 3.2 (Algorithm Interface and Implementing Classes): An interface in programming is a contract that all classes implementing it must fulfil certain conditions. In this case, algorithm classes must implement the `get_next_matrix` method that takes in a number and returns the next matrix to be multiplied. The interface can be seen in Appendix A.2.

Each implementing class will have its own starting step and iterative step. The Algorithm in 2.2 is an example of such an implementing class and the starting step and iterative step are both described in 2.2. More details for the code of this algorithm can be seen in Appendix A.3. The implementing class also has a responsibility of tracking the word consisting of all the letters that the algorithm has provided to the skeleton. This way, if the algorithm terminates, we can then retrieve this word and reconstruct the desired lower triangular matrix from it to check.

4. Introducing Killer Intervals

Now we killer intervals from Section 3 of [2] into our algorithm. The idea is that for some word

$$w = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad \text{where } \gcd(a, b, c, d) = 1$$

when we post-multiply a number $m\ /\ n$ with the inverse of this word, we get the following result:

$$(m, n)\begin{pmatrix} d & -b \\ -c & a \end{pmatrix} = (dm - cn, an - bm)$$

Note that we ignore the constant factor $1/(ad - bc)$ when using the inverse.

If we compare the denominator of the number before and after the post-multiplication, we can tell that for the new denominator to be smaller than the old one,

$$|an - bm| < |n| \ \Rightarrow \left|\frac{a}{b} - \frac{m}{n}\right| < \left|\frac{1}{b}\right|.$$

As such, we note that the absolute difference between m/n and a/b has to be smaller than $|1/b|$. Thus, we can associate the word w with a *killer interval*,

$$K_w = \left(\frac{a}{b} - \left|\frac{1}{b}\right|, \frac{a}{b} + \left|\frac{1}{b}\right|\right).$$

Then the number produced from $(m, n) \cdot w^{-1}$ will have a smaller denominator if

$$\frac{m}{n} \in K_w.$$

The incorporation of killer intervals is to keep the denominator small. The overall effect on the number is such that it likely grows larger. However, we can post-multiply the appropriate $L(a, n)$ to reduce the numerator and bring the number back to within $(-1/2\,, 1/2)$. We can then describe a modified variant of the original algorithm.

Algorithm 4.1 (Naïve Killer Algorithm): We instantiate an object of this class by providing it with s and r. The algorithm will then perform some pre-processing by generating a series of killer intervals from short words of a and b_q and storing them in an interval tree for efficient queries. More details can be found in Appendix B.5.

The starting step is nearly identical to Algorithm 2.2. except that we also run the iterative step of Algorithm 2.2 two additional times. These two iterative steps are to prevent the formation of trivial words. Subsequently, every iterative step is made up of two sub-steps:

Sub-step 1: Sub-step 1 involves only the a matrix. The intention is to bring the number back to within $(-1/2\,, 1/2)$ or if it is already within $(-1/2\,, 1/2)$, to multiply it by a^{-1} instead to prevent the algorithm from continuously multiplying by $a^0 = I$. This is identical to the single arrow in the iterative step of algorithm 2.2.

$$(x_i, sy_i) \to (\mathrm{SR}(x_i, sy_i), sy_i) = \left(x_{i_1}, sy_i\right)$$

Sub-step 2: Sub-step 2 involves either a b_q matrix or a word w. First, we query the interval tree to see if there is an interval where the $\left(x_{i_1}, sy_i\right)$ falls into. If there is no such interval, we then perform the exact same step as algorithm 2.2.

No interval found:

$$\left(x_{i_1}, sy_i\right) = \left(rx_{i_1}, rsy_i\right) \twoheadrightarrow \left(rx_{i_1}, s \cdot \mathrm{SR}\left(ry_i, x_{i_1}\right)\right) = (x_{i+1}, sy_{i+1})$$

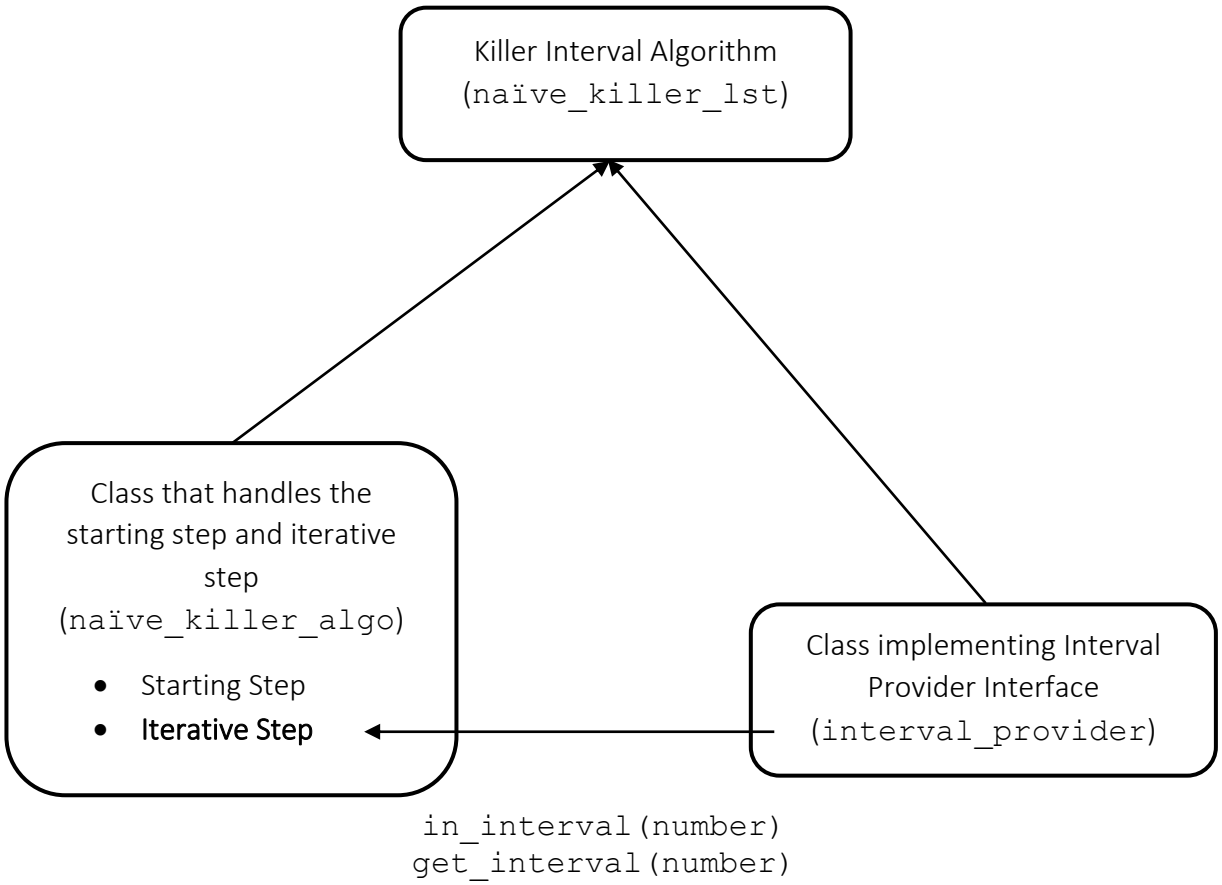
However, if there is at least one interval that this number falls into, we select one of them (details of the selection below) and its corresponding matrix, w, and multiply the inverse instead.

Interval found:

$$(x_{i_1}, sy_i) \overset{w^{-1}}{\mapsto} (x_{i+1}, sy_{i+1})$$

Since we are no longer consistently multiplying alternating letters of a and b_q, there is a possibility of cancellation and we form the trivial word by accident. However, it seems running algorithm 2.2's iterative step twice before enabling the use of killer intervals avoids this issue.

As such, the program structure will look a bit like this:



Program 4.2 (Interval Provider): The Interval Provider program is a class of programs that handles all of the following

- Generation of killer intervals
- Querying on whether a number falls within any of its killer intervals
- Querying for killer intervals that a number falls into
- Decision as to which matrix to use when the number falls into more than one killer interval

The `naive_killer_lst` contains several implementations of algorithm 4.1, each using the same `naïve_killer_algo` (implementation of sub-step 1 and 2) and differing by the `interval_provider`. As such, the difference in performance of the different killer interval algorithm variants lies solely in the Interval Provider program. We highlight only the two most promising ones.

Program 4.3 (Naïve Generator): The generator that we mainly use are words of length 2. We provide the size parameter $k \in \mathbb{N}$ and it generates the set

$$\left\{a^m \cdot b_q^n : m, n \in \mathbb{Z} \setminus \{0\}, -k \leq m \leq k, -k \leq n \leq k\right\}$$

and their corresponding intervals.

We have tried other types of generators but on testing, they do not seem to provide any significant improvement in performance of the program (more on this in Remark 6.1). Important to note that the larger n gets, the smaller the intervals become, to the point where they are rarely relevant. Additionally, the intervals are usually concentrated near 0 with many overlapping each other.

For the purposes of this paper, we have defaulted to using $k = 10$. It does not seem to make significant difference after $k > 5$ and it increases the pre-processing time if k is too large (See Appendix C).

With a generator, we can now consider how to select a matrix given that this number falls into more than one.

Program 4.4 (Deterministic Algorithm): The goal of this is to create some deterministic way to choose the best killer interval and its corresponding matrix to produce shorter non-trivial lower triangular words and/or solve for $35/9$ and $39/10$. Consider the number (m, n) before sub-step 2 of algorithm 4.1 and the set of all the matrices for which this number falls within their killer interval, S. Then we can create the set

$$R = \{(x, y) \in \mathbb{R}^2 : (x, y) = (m, n) \cdot w \text{ for some } w \in S\}$$

The hope was to find some function f such that when we look at the set

$$T = \{x \in \mathbb{R} : x = f(m, n, x, y) \text{ for some } (x, y) \in R\},$$

we can find a better algorithm if we always select the matrix $w \in S$ that gives the minimum (or maximum) value in T.

The best performing function f that we have found is

$$f(m, n, x, y) = \left| \left(\frac{m}{n}\right)^2 - \frac{\mathrm{SR}(x, y)}{y} \right|$$

and by finding the matrix which gives the minimum value with this function on each iteration. Note that when we refer to the algorithm in 4.4, it will be referring to the algorithm that uses this specific function (the program name is `Closest_Naive)`. With this algorithm, we are able to obtain significantly shorter words for some cases compared to other variants of the algorithm. However, this is still unable to solve for $35/9$ and $39/10$. Admittedly this is somewhat of a guess

but it does perform better than many of the common functions that one could think of. More on this in remark 6.4.

However, we have to consider that the deterministic methods that we have chosen is such that it will never choose the correct letters to form a non-trivial word for some of the more difficult numbers like $39/10$. As such, we introduce some randomness into the algorithm to see if this algorithm has the potential to find a non-trivial word for the likes of $35/9$ and $39/10$.

Program 4.5 (Randomised Algorithm): Using the generator in 4.3, we select the interval with the midpoint of the interval furthest away from the number (more on this in remark 6.2). After which, we randomly choose one of the matrices corresponding to this interval as there will be multiple of them (more on this in remark 6.3). Instead of just running it up to n iterations now, we have to decide the number of times we want repeat this as each run likely gives us a different result.

Most of the time, it fails to terminate under case 1 or 2. However, occasionally, it performs significantly better by either finding much shorter words compared to other algorithms or finding the non-trivial word where other algorithms have failed. It has managed to find a non-trivial word for $39/10$ (more on this in results 5.1). However, this algorithm has its limitations as it is difficult to determine why and how it performs better.

5. Results

First, we shall define how we determine whether one algorithm is performs better than another. Suppose we are comparing two algorithms A and B, there are two main criterion that we go by:

Criterion 1 (Able to solve a number): *Suppose A is able to find a non-trivial lower-triangular word for a number while B is unable to, we say that A is better than B.*

Criterion 2 (Finds shorter words): *Suppose both A and B can find a non-trivial lower-triangular word for a number. However, A finds a significantly shorter word than B (usually at least half the length compared to B) then we say that A is better than B.*

For Criterion 2 in particular, we usually test algorithms against the number $31/8$ which is the number closest to 4 that Algorithm 2.2 is able to solve, with word length of 1633. Thus, an algorithm which is able to find a word with length significantly shorter than 1633 would be considered a better algorithm.

Result 5.1 ($39/10$): Of the two numbers that remained unsolved in Appendix A of [1], the randomised algorithm in 4.4 has managed to find a trivial word of length 553 for

$$\frac{39}{10}$$

which terminates under case 2. The lower triangular matrix (where each term is already divided by the greatest common denominator of the 4 numbers in the matrix) is

[[1, 0], [-
3186547880458867218026492780753382322389084903741433248149087976158226442762102072513997365016404042448261603769333688802360414152893549425861478912377560237339793381255306060945297133906145291937131411471948157384960310, 625000000]]

The actual word can be viewed in Supplementary Materials 1 and how the results are verified can be seen in Appendix B.4. The randomised algorithm succeeds only in around 1 in 100 tries and always seems to find the exact same word.

Interesting to note that $39/10$ was solved earlier than $35/9$ as $39/10$ is the number closer to 4. But following the results in 5.2, we realise that the numbers composing the numerator and denominator also affect the difficulty of solving the number. In particular, on further testing, it appears that when running any algorithm on both $35/9$ and $39/10$, the numbers grow significantly quicker for $35/9$.

Results 5.2 ($q = s/r$, where $0 < q < 4$ and $1 < r < 17$): Using our best deterministic algorithm in 4.4, we are unable to find non-trivial words that are lower-triangular for the following rational numbers considered in this test:

[(35, 9), (39, 10), (39, 11), (40, 11), (41, 11), (42, 11), (43, 11), (47, 12), (42, 13), (43, 13), (44, 13), (46, 13), (47, 13), (48, 13), (49, 13), (50, 13), (51, 13), (51, 14), (53, 14), (55, 14), (56, 15), (58, 15), (59, 15), (57, 16), (59, 16), (61, 16), (63, 16)]

After running the randomised algorithm 4.5 on the above list of numbers for up to 5000 iterations and repeating it 500 times, the following numbers were solved:

[(39, 10), (39, 11), (41, 11), (42, 13), (43, 13), (44, 13), (51, 14), (57, 16)]

The words and matrices can be found in Supplementary Materials 1. The remaining unsolved numbers are

[(35, 9), (40, 11), (42, 11), (43, 11), (47, 12), (46, 13), (47, 13), (48, 13), (49, 13), (50, 13), (51, 13), (53, 14), (55, 14), (56, 15), (58, 15), (59, 15), (59, 16), (61, 16), (63, 16)]

For numbers with the same denominator, we observe that there is some threshold numerator where all numbers with numerator equal to or larger than that with the same denominator cannot be solved with this algorithm (Refer to Remark 6.5 for a comment on this). For numbers with different denominators, it is difficult to pinpoint any patterns other than that prime number denominators seem to be significantly more difficult to solve.

Results 5.3 ($q = s/r$, where $1 < s \leq r$ and $2 < r < 116$): Using our best deterministic algorithm in 4.4, we ran the following test with up to 10000 iterations. The full details can be viewed in Supplementary Materials 2 with the full list of numbers that were not solved at the bottom.

It appears that numbers with larger odd denominators are more difficult to solve, especially those with large prime factors. However, it also appears that numbers with larger numerators are more difficult to solve. From the list of unsolved numbers, most of the numbers are greater than $1/2$ with only a handful that are smaller than half that also have a prime number as their denominator.

However, an interesting point to note is that for the number $(109, 114)$, we actually obtain a word that is 13257 letters in length. It terminates under case 2, that is a loop is found, (the word 'repeated' being printed indicates this) which might artificially inflate the length. However, in printing out the entire series of numbers that the algorithm traverses through, we observe that the number never grows very big and as such, allows the algorithm to efficiently continue and eventually terminate under case 2. This seems to indicate that the likely issue with $35/9$ and previously, $39/10$, is indeed that the numbers that the algorithm traverse through grows too quickly and not that whatever lower-triangular word, if it exists, is too long for the algorithm to find.

Additional Comparison Tests 5.4: The deterministic algorithm in 4.4 was compared with the original algorithm 2.2 from [1].

Comparison is done based on whether a non-trivial word was found followed by the word length. A shorter word length would be more desirable for the algorithm. If the algorithm is unable to determine the word, the length will be substituted with -1. The full test and length comparison details can be found in Supplementary Materials 3. In particular, there are a few numbers we would like to highlight to compare the word length.

Number	Algorithm 2.2 (`Naive`)	Algorithm 4.4 (`Closest_Naive`)
$31/8$	1633	451
$28/17$	-1	16
$28/73$	933	40
$29/17$	-1	7
$29/73$	4753	95
$29/79$	4861	55
$29/199$	2385	15
$29/251$	1515	29
$29/263$	5721	63
$30/233$	3777	132
$30/251$	4409	35

Our algorithm seems to be able to find significantly shorter words for some while not having to sacrifice performance on each iteration as the second algorithm described in Appendix A of [1] does. The time complexity analysis can be seen in Appendix C. We observe that in particular, Algorithm 2.2 appears to struggle with some small numbers with large denominators. Our algorithm handles them easily.

Additional Comparison Tests 5.5: The randomised algorithm was used to run through test 1 used in test 5.4 to find the shorter words. We show that despite algorithm 4.4 being able to find shorter words than those found by algorithm 2.2, there are in fact shorter words to be found. The full length, word and matrices can be found in Supplementary Materials 4.

In particular, we would like to highlight the following numbers:

Number	Algorithm 2.2	Algorithm 4.4	Randomised 4.5 Shortest
23 / 6	191	107	75
26 / 7	279	359	113
27 / 7	355	223	147
31 / 8	1633	451	175

While it is likely that the randomised algorithm might not have found the shortest word possible, we show the existence of shorter words than those our deterministic algorithms have found which shows that there is significant room for improvement.

6. Remarks and Extensions:

Remark 6.1 (Regarding Generators): The generator as described in 4.3 is rather naïve and there are many ways where it could be improved.

It is evident that there are likely many other ways to generate killer intervals. One of the generators considered were the intervals generated from the matrices from this set

$$\left\{a^m \cdot c_q^n : m, n \in \mathbb{Z} \setminus \{0\}, -k \leq m \leq k, -k \leq n \leq k\right\},$$

where

$$c_q = b_q^r = \begin{pmatrix} 1 & s \\ 0 & 1 \end{pmatrix}.$$

Another set could also be obtained from

$$\left\{r_q^n : n \in \mathbb{Z} \setminus \{0\}, -k \leq n \leq k\right\},$$

where

$$r_q = a \cdot b_q$$

However, they have not shown any significant improvements. As such, most of the research was done with the generator described in 4.3 in order to simplify the algorithm.

We ran the same algorithm in 4.4 through the same test cases done in Appendix A of [1], first with the naïve generator described in 4.3 then with a combined generator that also includes the two above. There were only two cases that showed significant difference.

Number	4.4 with Naïve Generator (`Closest_Naive`)	4.4 With Combined Generator (`Closest_Combined`)
$20/7$	121	23
$30/233$	132	53

The full details of the comparison can be seen in Supplementary Materials 5.

Remark 6.2 (Choice of Interval in Algorithm 4.5): It is noted that we select the interval with midpoint furthest away from the number. For the number (m, n) and the interval with midpoint (a, b), consider the distance between the two numbers

$$\left|\frac{a}{b}-\frac{m}{n}\right| = \left|\frac{an-bm}{bn}\right| = \left|\frac{an-bm}{n}\right|\left|\frac{1}{b}\right| \Rightarrow \frac{\left|\frac{a}{b}-\frac{m}{n}\right|}{\left|\frac{1}{b}\right|} = \left|\frac{an-bm}{n}\right|.$$

$|1/b|$ is related to the size of the interval and the term

$$\left|\frac{an-bm}{n}\right|$$

is in fact the ratio of the new denominator over the old denominator. As such, if we want the greatest decrease in denominator after the multiplication of a killer interval's matrix, we would minimise the ratio of distance over size of the interval.

However, on testing, we realise that when we select intervals in this manner, it performs significantly worse than even Algorithm 2.2, failing to even solve for many of the numbers that Algorithm 2.2 was able to. On the contrary, selecting the one with the largest ratio (corresponding to the smallest decrease in denominator) consistently gives a better result compared to Algorithm 2.2 (in terms of shorter word lengths). However, curiously, it seems that if we simply select the interval with midpoint that is the greatest distance away from the number, it performs far better for $31/8$.

Remark 6.3 (Peculiar Behaviour of the Randomised Algorithm 4.5): Consider the generator in 4.3, for $m, n \in \mathbb{Z} \setminus \{0\}$,

$$a^m \cdot b_q^n = \begin{pmatrix} 1 & 0 \\ m & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & nq \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & nq \\ m & mnq+1 \end{pmatrix}.$$

As such, the killer interval is dependent entirely on b_q and n and independent of the a^m at the front. Additionally, when we apply the killer interval, we are using the inverse of the word that provided the matrix. Thus, the actual matrix that we are post-multiplying to our number is

$$\left(a^m \cdot b_q^n\right)^{-1} = b_q^{-n} \cdot a^{-m}$$

Looking at algorithm 4.4, we realise that we select the interval first which means that we choose a specific power for the b_q letter first. Then the randomised algorithm actually only chooses the power for the a letter.

Referring back to algorithm 4.1, after the application of a killer interval, if the number is outside the interval $(-1/2\,,1/2)$, we will post-multiply the appropriate a letter to bring it back within this interval. If it is already within this interval, we then post-multiply by a^{-1} instead. Since the post-multiplication of a to a number (m,n) can be seen as a translation of 1 unit in the positive x-direction of the point (m,n) on the (x,y) plane, we realise that there are only two possible outcomes. Majority of the words that cause the number to land outside of $(-1/2\,,1/2)$ will end up being translated to the same number in $(-1/2\,,1/2)$ after the next sub-step of algorithm 4.1. Let us call this number (p,q). Then, there is a possibility that the matrix we post-multiply to the number causes it to land directly on (p,q) which instead causes it to be post-multiplied by a^{-1} in the next sub-step 1 to end up as $(p-q,q)$.

As such, the only random aspect of the randomised algorithm is that it occasionally causes one of the a letters of the word to have a power one less than it would otherwise have. And it seems, this was all that was needed to solve for $39/10$ and some of the other numbers in result 5.2. This suggests that some small perturbation to the path can result in drastically different results where some numbers converge back to $(1,0)$ while others grow bigger and bigger under this algorithm. Understanding when this extra a^{-1} should be or should not be applied could allow us to develop better algorithms.

Remark 6.4 (Choice of f in Algorithm 4.4): The choice of f in 4.4 seems random and we unfortunately do not have a good reason for this specific choice of f other than that it performs better than most other functions we could think of. Some of the more intuitive ones are highlighted below.

Function 6.4.1:

$$f_1(m,n,x,y)=\left|\frac{m}{n}-\frac{\mathrm{SR}(x,y)}{y}\right|$$

This roughly translates to the distance between the new number and the old number. Both minimising and maximising this gives rather poor performance, on par with algorithm 2.2.

Function 6.4.2:

$$f_2(m,n,x,y)=|y|$$

Choosing the maximum of this function on each iteration translates to choosing the smallest decrease in denominator each time. Choosing the minimum of this function on each iteration translates to choosing the largest decrease in denominator each time. The greatest decrease in denominator performs poorly as expected since it is related to Remark 6.2. The smallest decrease does not show any significant improvement over algorithm 2.2.

Function 6.4.3:

$$f_3(m, n, x, y) = \left| \frac{\mathrm{SR}(x, y)}{y} - c \right|, \text{for some constant } c \in \mathbb{Q}$$

This translates to trying to finding some number where we can either try to get as close to as possible (via minimising) or get as far away as possible (via maximising). Some constants that we have tried are $1/2$ and 0. Both of which perform poorer than Algorithm 2.2.

The fact that the function in Algorithm 4.4 is able to give us better results than Algorithm 2.2 is kind of surprising. It shows that there is a possibility for an algorithm that uses such a function to perform well.

Remark 6.5 (Observation for Numbers Approaching 4): When running the randomised Algorithm 4.5, we notice that the numbers slightly further away from 4 (such as $39/11$ and $42/13$) actually find several non-trivial words and the one shown in Supplementary Materials 1 is actually the shortest one found. However, for the numbers closer to 4 like $39/10$, $41/11$ and $44/13$, it seems that the randomised algorithm always finds the same word whenever it terminates early.

It is likely that there exists only certain 'path's where the algorithm must land on to eventually reach a non-trivial word and these paths get fewer and fewer as the number approaches 4. The randomised algorithm occasionally helps us find this path.

APPENDIX A: Algorithm Skeleton

In this appendix, we give a detailed explanation of some of the more important programs in used in this project. For the full program, please contact Chan J.D.

A.1: Below, we have the code for the algorithm skeleton as mentioned in Program 3.1.

```
1. def run_algo(s, r, n, Algo_Class):
2.     #Checks for already known cases
3.     if abs(s / r) >= 4 or math.gcd(s, r) != 1 or s % r == 0: return True
4.     #Number tracker
5.     tracker = {}
6.     #Instantiates an Algorithm object (initiates the algorithm)
7.     algo = Algo_Class(s, r)
8.     #[1, 0] starting point representing infinity
9.     number = starting_point()
10.
11.    #Run test n times
12.    for i in range(n):
13.        #Get next matrix and multiply
14.        matrix = algo.get_next_matrix(number)
15.        number *= matrix
16.
17.        #Checks that the number has returned to infinity
18.        if number.get_q() == 0:
19.            lst = list(tracker.keys())
20.            lst.insert(0, starting_point())
21.            lst.append(number)
22.            #print(lst)
23.            print("length:", len(algo.get_word()))
24.            print("word:", algo.get_word())
25.            print("matrix:", algo.get_matrix())
26.            return True
27.
28.        #Checks that the number has looped
29.        elif number in tracker:
30.            print("repeated")
31.            lst = list(tracker.keys())
32.            lst.insert(0, starting_point())
33.            lst.append(number)
34.
35.            #Comment out the below code block for accurate word looping
36.            word = algo.get_word()
37.            prev_word = tracker[number]
38.            final_word = word + prev_word.inverse()
39.            #print(lst)
40.            print("length:", len(final_word))
41.            print("word:", final_word)
42.            print("matrix:", matrix_word(final_word, algo.get_letter_table()))
43.            return True
44.
45.        #Adds the number to the hashtable with its corresponding word
```

```
46.         else:
47.             tracker[number] = algo.get_word()
48.
49.     return False
```

As such, we observe that this program only handles the termination, either in line 18 which corresponds to case 1, line 29 which corresponds to case 2 or line 12, when the for loop expires after having ran for n iterations. The illustrates the programming idea of abstraction, which allows us to more easily change the underlying algorithm without having to reimplement everything. In this case, we only need to switch the `Algo_Class` argument in line 1 to the algorithm class that we want that implements the appropriate methods and we would be able to run it together with this function.

There is a slightly altered version (`run_algo_details`) of this function which returns the details instead of printing them so that they can be used for further analysis.

A.2: Below is the Algorithm interface mentioned in 3.2.

```
1. #Interface for algorithm implementation.
2. class Algo_Interface:
3.
4.     def __init__(self, s, r):
5.         self.s = s
6.         self.r = r
7.
8.     #Method to override to obtain the next matrix from the current number.
9.     def get_next_matrix(self, number):
10.        pass
11.
12.    def get_word(self):
13.        pass
14.
15.    def get_letter_table(self):
16.        pass
17.
18.    #Default method to get the current matrix
19.    def get_matrix(self):
20.        return matrix_word(self.get_word(), self.get_letter_table())
```

Note that this interface does not implement any concrete methods besides the `get_matrix` method. All the other methods should be implemented by the algorithm class, in particular, the `get_next_matrix` method.

A.3: An example of a class that implements the above interface is shown below. It is an implementation of Algorithm 2.2 as described in Appendix A of [1].

```
1. #Algo class that implements the algo_interface. This is the naive algorithm (f
   irst algorithm)
2. class Naive_Algo(Algo_Interface):
```

```
3.
4.      def __init__(self, s, r):
5.          super().__init__(s, r)
6.          self.start = True
7.
8.          #Keeps track of the word
9.          self.word = create_empty_word()
10.
11.         #Table of letters used
12.         naive_algo_table = {'a': a_n, 'b': lambda n : b_n(self.s, self.r, n)}
13.         self.table = Letter_Table(naive_algo_table)
14.
15.     def get_next_matrix(self, number):
16.
17.         #First iteration converts to (r, s) number
18.         if self.start:
19.             letter = Letter('b', (self.r - SR(self.r, self.s)) // self.s)
20.             self.word += letter
21.             self.start = False
22.             return self.table.get_matrix(letter)
23.
24.         #Normal naive algorithm
25.         else:
26.
27.             #Reduction of numerator by 'a' matrix
28.             a_letter = Naive_Algo.a_reduce(number)
29.             a_matrix = self.table.get_matrix(a_letter)
30.
31.             #update number
32.             number *= a_matrix
33.
34.             #Change in numerator and denominator by 'b' matrix
35.             b_letter = Naive_Algo.b_reduce(number, self.s, self.r)
36.             b_matrix = self.table.get_matrix(b_letter)
37.
38.             #combined a_b chain
39.             added_word = a_letter + b_letter
40.
41.             #update word
42.             self.word += added_word
43.
44.             return a_matrix * b_matrix
45.
46.     #Reduces the numerator of the rational_special
47.     @staticmethod
48.     def a_reduce(number):
49.         x = number.get_p()
50.         s_y = number.get_q()
51.         new_x = SR(x, s_y)
52.         added_power = -((x - new_x) // (s_y))
53.         return Letter("a", added_power)
54.
55.     @staticmethod
```

```
56.     def b_reduce(number, s, r):
57.         x = number.get_p()
58.         y = number.get_q() // s
59.         new_y = SR(r * y, x)
60.         added_power = -((r * y - new_y) // x)
61.         return Letter("b", added_power)
62.
63.
64.     def get_word(self):
65.         return self.word
66.
67.     def get_letter_table(self):
68.         return self.table
```

Note the concrete implementation of the three methods from the interface inside this class. The methods `a_reduce` and `b_reduce` in lines 48 and 56 respectively are the methods that search for the appropriate powers to carry out the iterative step in Algorithm 2.2. Observe that for an object instance of this class, upon initialisation, it keeps track of the combination of letters that it has supplied thus far in the form of a word. This can be seen in lines 9 when it creates an empty word, followed by lines 20 and 42 where it is updated whenever `get_next_matrix` is called. As such, if the algorithm succeeds in solving for the number, the word that is stored constitutes a lower triangular matrix or can be made into one.

APPENDIX B: Basic Packages and Data Structures

Here we describe all the fundamental packages that we have implemented in order to support this program and the rationale behind them.

B.1 Rational Number Package: A simple package that represents all rational numbers as two integers to prevent precision loss with the use of floating-point numbers in computation. It supports the following features:

- Rational numbers are expressed as two integers, numerator p and denominator q such that q and p are coprime
- The numbers can interact with all the common operators like '+', '*', '<' etc.
- On each operation, the greatest common denominator of both the numerator and denominator is factored out
- Supports the creation of infinity $(1, 0)$
- Supports hashing

The last point is so that it can be used within a Hash Table which allows for $O(1)$ lookup time complexity in order to check if we have looped for case 2 in Program 3.1.

On top of this, we have created a special rational number package which allows the number to be post-multiplied by a 2×2 matrix as if the number was a 1×2 matrix. This allows it to interact nicely within programs A.1, line 15 and A.3, line 32.

B.2 Matrix Package: A simple matrix package that implements matrix multiplication and power operations. We then extend this package to the AB_Matrix package, which supports only 2×2 matrices meant to be used solely for this project. It supports the following features:

- Automatic greatest common denominator reduction of matrices after construction and multiplication if all four numbers in it share a common denominator that is not 1

This is to ensure that the size of the verification matrix is not too large.

B.3 Interval Package: A package that implements intervals using the Rational Number package in B.1. Each interval consists of two rational numbers denoting its endpoints. It supports common interval queries like whether it contains a number etc.

B.4 Words Package: This package is inspired by string concatenation. It contains the implementation of letters and words as programming objects and their interactions and supports the following:

- Letters are represented by an alphabetic letter 'a', 'b', 'c' etc. and a number representing its power. The letter '`a^18`' corresponds to $L(a, 18)$
- Words are a series of letters where no two consecutive letters are of the same type. The word '`b^1, a^-1, b^2`' corresponds to the word

$$w = b^1 \cdot a^{-1} \cdot b^2 = L(b, 1) \cdot L(a, -1) \cdot L(b, 2).$$

  Note that the matrix b does not have a rational number attached to it yet. We handle them without explicitly knowing what matrix they correspond to yet
- Both support the inverse operation. For letters, the power becomes negative. For words, the powers of all the letters becomes negative and the order becomes reversed, similar to that of a matrix
- Adding a letter to a letter will create a word of length 2 if the letter types are distinct, else it creates a new letter with the power being the sum of the two letter's powers
- Adding a letter to a word is done by doing letter to letter addition for the last letter of the word to the new letter. If they are of different types, it extends the word length by 1. Else, it will change the power of the last letter. If the power of the last letter becomes 0, it is removed from the word
- Adding a word to a word is done by iterating through the second word, letter by letter and adding them one at a time. There is a possibility of complete cancellation when we add a word to its inverse
- Both words and letters do not correspond to a matrix yet. They require a Letter Table which is made up of a Python dictionary of letter types and a corresponding function that given a power, returns the corresponding 2×2 matrix raised to the appropriate power. An example can be seen in A.3, line 12 and 13
- Supports the `matrix_word` method that given a word and a letter table for some rational number q, constructs the 2×2 matrix corresponding to the word as described in definition 1.1.

Crucially, the last point is our method of verification that the algorithm does indeed produce a lower-triangular matrix. The word produced at the end of our algorithms is, together with its letter

table, used to reconstruct the matrix corresponding to it and we can easily verify that it is in fact lower-triangular.

B.5 Interval Trees: There are two main types of interval trees that we have implemented. Both take $O(n \log n)$ construction time for n intervals. They are particularly important for program 4.2.

- The first variant is used to check if there exists an interval where a number falls into. This is for the `in_interval(number)` query where we can check in $O(\log n)$ time whether the number falls into any of the n interval gaps (intervals in $(-1/2, 1/2)$ not covered by any killer intervals)
- The second variant is used to query for all intervals where a number falls into. This is used in part of the `get_interval(number)` query. It allows us to query for all such intervals in $O(\log n + m)$ time for n intervals and m intervals found.

APPENDIX C: Algorithm 4.1 and Time Complexity Analysis

Here, we describe the main workings of Program 4.1. We will only show the `get_next_matrix(number)` method of Program 4.1.

```
1. def get_next_matrix(self, number):
2.
3.     #First iteration converts to (r, s)
4.     if self.count == 0:
5.         self.count += 1
6.         #start
7.         letter = Letter('b', (self.r - SR(self.r, self.s)) // self.s)
8.         self.word += letter
9.
10.        return self.table.get_matrix(letter)
11.
12.    #a_reduce step
13.    elif self.a_step:
14.        self.count += 1
15.        #Reduction of numerator by 'a' matrix
16.        a_letter = Naive_Killer_Algo.a_reduce(number)
17.        a_matrix = self.table.get_matrix(a_letter)
18.
19.        self.word += a_letter
20.
21.        #change next_step to a_step
22.        self.a_step = False
23.
24.        return a_matrix
25.
26.    #non a_reduce step
27.    else:
28.        #change the next step back to an a_reduce
29.        self.a_step = True
30.
```

```
31.         p = number.get_p()
32.         q = number.get_q()
33.         if SR(self.r * q, p) == 0:
34.             #Change in numerator and denominator by 'b' matrix
35.             b_letter = Naive_Killer_Algo.b_reduce(number, self.s, self.r)
36.             b_matrix = self.table.get_matrix(b_letter)
37.
38.             #update word
39.             self.word += b_letter
40.
41.             return b_matrix
42.
43.         #Checks to see if the number is contained within some killer interval

44.         #Can only be run after 2 iterative steps of algorithm 2
45.         if self.count > 2 and self.interval_provider.in_interval(number) and \

46.                 self.principal_range.contains_inc(number):
47.
48.             #get interval from the Interval_Provider
49.             word, matrix = self.interval_provider.get_interval(number)
50.
51.             #update word
52.             self.word += word
53.
54.             #return matrix
55.             return matrix
56.
57.         else:
58.             #Change in numerator and denominator by 'b' matrix
59.             b_letter = Naive_Killer_Algo.b_reduce(number, self.s, self.r)
60.             b_matrix = self.table.get_matrix(b_letter)
61.
62.             #update word
63.             self.word += b_letter
64.
65.             return b_matrix
```

Note that lines 33 to 41 are to check if there is a possibility of mapping the number straight back to infinity if we multiply it by some letter of b_q which reduces the denominator to 0. Besides that, we can see that the algorithm is largely similar to the one in A.3. Here, we have split it into the two sub-steps with `self.a_step` being a Boolean flag to indicate if we are at sub-step 1 or 2.

Lines 45-49 is where the difference lies. We check whether the number lies within an interval and if it does, we query the Interval Provider for the matrix to be used.

We can now consider the time complexity analysis for program 4.4. We ignore the cost of mathematical operations such as multiplication between the number and the matrix for this analysis. However, we do acknowledge that when the number's numerator and denominator are both large, multiplication and finding the greatest common denominator can be very costly.

Suppose we use a generator described in Program 4.3 of size k, the time it takes to generate all the matrices and their intervals is $O(k^2)$ and it produces $2k$ different intervals (as per remark 6.3). Building the interval trees thus only take $O(k \log k)$ time. As such, the dominant constant cost of the algorithm is $O(k^2)$. Thus, it is often not practical to use generators with large k as it severely slows down the testing when done on many numbers with short words.

Now we consider the cost per iteration. We observe that we always need to query at least the interval gaps tree. If the number does not fall inside any of the intervals, the rest of the cost can be considered constant. However, the number of interval gaps rarely changes on testing. Once we vary the size $k \geq 5$, the gaps in the killer intervals generated are usually the same. As such, we can also treat this as a constant cost.

If the number is not inside an interval gap and thus inside one of the intervals, we then need to query the second interval tree and make a decision. The query itself takes $O(\log k + m)$ time complexity where m is the number of intervals found. Now, we consider all m intervals at once, each of which have $2k$ different matrices which gives us $2mk$ different matrices in total. We then sort all of them based on their value produced with the function f which takes up $O(mk \log mk)$ time and select the value at the first index.

Thus, overall, if we consider that the algorithm runs for n iterations, the worst-case total time complexity would be $O(k^2 + nmk \log mk)$ with the constant cost dominating for low n while the latter term dominating when n gets large. For the same number of iterations, this algorithm actually performs worse than Algorithm 2.2. However, for the more difficult numbers, it usually finds a significantly shorter word compared to Algorithm 2.2 which allows it to run for far fewer iterations and thus, be more efficient. The other crucial point to note is that none of the above steps are dependent on the size of the numerator and denominator of the number. This is in contrast to the second algorithm proposed in Appendix A of [1] where the cost of each iterative step is dependent on the size of the numerator and denominator which can grow very quickly making the algorithm extremely slow.

We do not consider the complexity of the algorithm that uses Program 4.5 as that would have to include the number of repeats that we have to run in order to obtain the word. Thus, even if the algorithm has a better time complexity than those considered above, it does not make it more efficient as we still need to run it many more times in order to solve for the number.

References

[1] **S H Kim, T Koberda**, *Non-freeness of groups generated by two parabolic elements*
[2] **B Lou, S P Tan, A D Vo**, *Hyperbolic jigsaws and families of pseudomodular groups, I,* Geom. Topol. 22 (2018) 2339-2366

Department of Mathematics, National University of Singapore, Singapore
e0014929@u.nus.edu, mattansp@nus.edu.sg