LAPIN AMK
Lapland University of Applied Sciences

PARALLEL GENERATION OF ABELIAN PATTERN-FREE WORDS ON THREE LETTERS

Gavrilenko Aleksandr

Thesis
Lapland University of Applied Sciences
Information Technology
Bachelor of Engineering

2016

Author	Aleksandr Gavrilenko
Supervisor	Veikko Keränen
Title of Thesis	Parallel Generation of Abelian Pattern-Free Words on Three Letters
Number of pages	32 + 24

An abelian square is a word that can be divided into two consecutive parts such that the first part is a permutation (anagram) of the second part, that is, the number of occurrences of each letter is the same for the both parts. This thesis explores the programmatic approach to generation of long words not containing almost (that is, shortest one letter repetitions are allowed) abelian squares using Rust programming language.

The thesis briefly covers the history of the related field of combinatorics on words, the benefits of using Rust and the process of implementation of two programs that can be used to generate the longest word that does not contain almost abelian squares.

As the main result of the thesis, two programs utilizing different approaches were created. The programs can be used for the later research in the field.

Key words: Rust, Abelian Square Avoidance, Parallel Computing

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

1.1. Preliminaries

The reader might ignore this subsection if they are familiar with terminology.

An *alphabet* Σ is a finite non-empty set of elements called *letters*. A *word* over Σ is a sequence of letters belonging to the Σ. The *empty word* is represented by λ. The set of all finite words [all non-empty finite words] is denoted as Σ^* [Σ^+]. On the Σ^* the binary operation of *catenation* is defined (with the neutral element being λ). The structures Σ^* and Σ^+ are called the *free monoid* and the *free semigroup* respectively.

Let $w = x_1 \cdots x_m, x_i \in \Sigma$. The *length* of the word w, denoted by $|w|$, is the number of occurrences of letters in the word, i.e., $|w| = m$. Let $\Sigma = \{a_1, \ldots, a_n\}$. The number of occurrences of the letter $x, x \in \Sigma$, in w is denoted by $|w|_x$, or by $|w|_i$ if $x = a_i$. The notation $\psi(w)$ stands for *Parikh vector* of w; $\psi(w) = (|w|_1, \ldots, |w|_n)$.

A word u is called a *factor* of word w if there exist words p and s such that $w = pus$. If $p = \lambda$ [$s = \lambda$], then u is called a *prefix* [*suffix*] of w. FACT(w), PREF(w) and SUFF(w) denote sets of all factors, prefixes and suffixes of w respectively.

For a given integer $k, k \geq 2$, a *k-repetition* is a non-empty word of the form R^k. An *abelian k-repetition* is a non-empty word of the form $P_1 \cdots P_k$, where $\psi(P_i) = \psi(P_j)$ for all $1 \leq i, j \leq k$. [Abelian] 2-repetitions and 3-repetitions are also called [abelian] squares and cubes. A word is called [abelian] *k-repetition free* if it does not contain a factor that is an [abelian] k-repetition. If it is possible to construct infinite [abelian] k-repetition free words over a given alphabet Σ, then [abelian] k-repetitions are said to be *avoidable* over Σ.

A *morphism* h is a mapping between free monoids Σ^* and Δ^* satisfying $h(uv) = h(u)h(v)$ for every u and v in Σ^*. In particular, $h(\lambda) = \lambda$. An *endomorphism* (usually denoted by g) is a morphism with $\Sigma = \Delta$.

For a given integer $k \geq 2$ a morphism $\sigma : \Sigma^* \to \Delta^*$ is called *k-free* [*a-k-free*], if all the words $\sigma(w)$ are k-free [a-k-free] for every k-free [a-k-free] word $w \in \Sigma^*$.

1.2. History

At the beginning of 20th century Axel Thue laid the foundation of the combinatorics on words in his remarkable paper 'Über unendliche Zeichenreihen' [1]. However, the

paper remained largely unnoticed at the time. The lack of interest in the area, more or less, continued until 1961 when Erdös [2] raised a question whether abelian squares can be avoided in words of infinite length over a four-letter alphabet.

In 1970, Pleasants [3] proved existence of an infinite abelian square-free word over five letters. Four years later Etringer et al. [4] demonstrated that every infinite word over two letters contains arbitrarily long abelian squares. In 1979, Dekking [5] proved existence of (1) an infinitely long abelian 4-repetition free sequence over two letters and (2) an infinitely long abelian cube-free sequence over three letters.

The next notable achievement happened in 1992 when Keränen [6] published endomorphism g_{85} which allowed to generate abelian square-free words of unlimited length over 4 letter-alphabet. In 1993 Carpi [7] formulated sufficient conditions for morphisms to be a-k-free and confirmed g_{85} to be abelian square free. Additional mappings, based on g_{85} were discovered by Carpi [8] in 1998. As a continuation of research, 12^4 morphisms was presented by Keränen [9] in 2009.

1.3. Current State

The next step after discovering a way to avoid abelian squares over 4 letters would be to do so over 3 letters. However, no such words having length greater than 7 exist. Therefore, nowadays the research of pattern avoidance in words is more focused on almost abelian square-free words (a-a-2-free or AA2F) over a 3 letter alphabet, i.e., words that allow repetitions xx and xxx for a letter x, but do not contain other abelian squares.

This project takes a programmatic approach to the problem, attempting to generate a significant amount of relevant data for further research.

The first major task is to create a program which will attempt to generate a-a-2-f words in order of increasing length, using various search optimization techniques. The second task is to create a program that will read a list of a-a-2-f words from input file, and sort the words into two groups. The first group should include words that can be extended both to the left and to the right for predetermined length, and, consequently, have a probability of being used as a building block for longer words. The second group should include words that cannot be extended.

2. Language Choice Justification

The task described in the 1.3 is computation heavy due to exponential growth (as the worst-case scenario) of the number of words required to be checked as the length increases. Because of this, the most prioritized features of languages are (1)high performance and (2) simple parallelization. Effectively, the first requirement dismisses interpreted languages. Garbage collected languages are also undesirable, because GC algorithms need to consume the computational resources themselves.

Finally, the author made decision to use Rust [10] for the following reasons:

1. Performance. Rust demonstrates speed comparable to that of C and C++ languages, as demonstrated by benchmarks [11, 12]. To the large degree this can be attributed to the fact that rustc (the compiler of Rust) is a fronted for LLVM, which is most well-known for being the backend for the Clang —the C-family language compiler used by Apple.

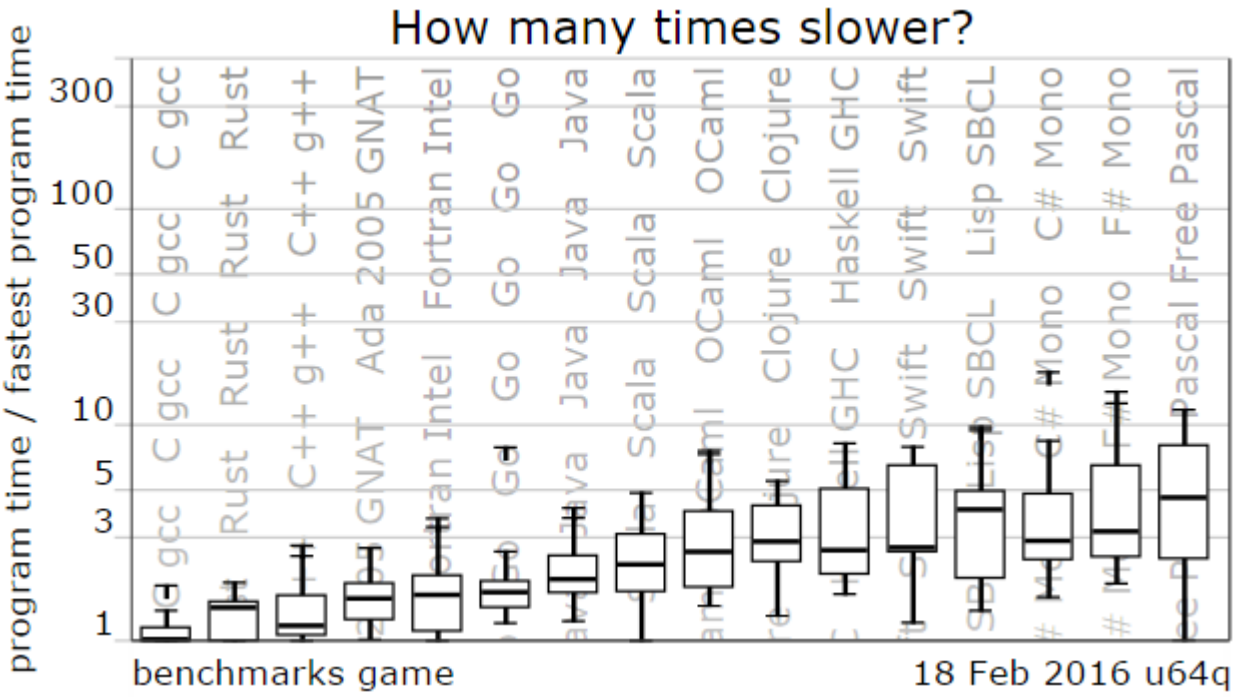

Figure 1: The Computer Language Benchmarks Game: Fastest Languages. Source:[11]

2. Parallelization. Using safe subset of Rust (henceforth written as *Safe Rust* when opposed to *Unsafe Rust*, and simply *Rust* otherwise), programmer is guaranteed to avoid data races [13, 14]. Race conditions are still possible, but guaranteed

not to cause memory violation in Safe Rust [13]. It should be noted, that while data races have a rigidly defined conditions to be considered such, race conditions lean closer to being a logical error, and therefore it is sufficiently harder (if at all possible) to prevent them programmatically [15].

3. Memory management. By utilizing novel approach to memory management (only used before in an experimental C dialect —Cyclone), Rust guarantees memory safety. That is, the language prevents dereferencing null or dangling pointers, reading uninitialized memory and violating the strictly defined (see subsection 3.2) pointer aliasing rules [13].

3. Introduction to Rust Syntax

Rust is positioned as a low-level systems languages, potentially aiming to replace C. Due to this, the syntax is largely reminiscent of that of C in order to ease the transition from C to Rust for experienced programmers. Nevertheless, Rust brings concepts from functional programming (henceforth *FP*) languages [16], which might be unfamiliar for a hypothetical C programmer. This section aims to cover some of the differences and familiarize the reader with the Rust syntax and the language idioms. To accomplish this, code listings will frequently be employed. Some material will be presented as commentaries in the listings. The reader should be aware that this is not an exhaustive description of the language; the references [17–19] are advised for deepening the understanding.

3.1. Primitive Types and Variable Declaration

Table 1 describes the majority of the primitive types that exist in Rust, as well as **Box**, **Vec** and **String** types, which are not primitive, but widely used.

Table 1: Primitive Types in Rust

Type	Size	Meaning
bool	1 byte	A boolean value: either true of false
char	4 bytes	A Unicode character
i8	1 byte	A signed integer number
i16	2 bytes	
i32	4 bytes	
i64	8 bytes	
u8	1 byte	An unsigned integer number
u16	2 bytes	
u32	4 bytes	
u64	8 bytes	
isize	Varies: 4 bytes for x86 programs and 8 bytes for x86_64 programs	A signed integer number size of which depends on size of pointer of the underlying machine
usize		An unsigned integer number size of which depends on size of pointer of the underlying machine
f32	4 bytes	A floating point number
f64	8 bytes	
Box<T>	$size(usize)$	A unique pointer to heap-allocated data of type T.
[T; N]	$N * size(T)$ bytes	An array of values of type T. Values can be accessed by index as `myArray[n]`. Arrays use zero-based numeration in Rust. Arrays can automatically coerce to slices
&[T]	$2 * size(usize)$	A slice: a dynamically-sized view into a contiguous sequence
Vec<T>	$3 * size(usize)$	Is a pointer to a heap-allocated growable list of values of type T. Can automatically coerce to slices
(type1, ..., typeN)	$size(type1) + \cdots + size(typeN)$	A tuple: ordered list of values of different types. Values can be accessed by index as `myTuple.0`. Index should be defined at compile-time. Tuples use zero-based numeration in Rust
str	unsized	A stack-allocated sequence of Unicode characters. Because it is unsized, there is no way to pass it by value
String	$3 * size(usize)$	Is a pointer to a heap-allocated sequence of Unicode characters

Variables in Rust are immutable by default. To declare a variable as mutable, programmer needs to explicitly specify mutability with the **mut** keyword. Aliasing is strictly limited in Rust, the exact rules are covered in subsection 3.2. Listing 1 demonstrates mutable and immutable variable declaration.

```
1 fn main() {
2         //Use the let keyword to declare variables
3         //Variables are immutable by default
```

```
4           let day: i64 = 15;
5           //Mutable variables are declared via let mut
6           //Types can be inferred by compiler if there is enough data
7           //Integer literals can have suffixes, e.g. u8, i8, ..., u64, i64
8           //Floating point literals have suffixes f32 and f64
9           let mut year = 2010i32;
10          year += 5;
11
12          //Immutable variables can be initialized with a delay
13          let version:f32;
14          version = 1.0f32;
15
16          //Type `str` is unsized, and as such cannot be stored directly.
17          //The reference to it is used.
18          let month: &str = "May";
19
20          //`String`s', however, can be stored in a variable directly.
21          let lang: String = "Rust".to_owned();
22
23          println!("The version {:.1} of {} was announced on {} {}, {}.",
24                  version, lang, month, day, year);
25          //Output: The version 1.0 of Rust was announced on May 15, 2015.
26
27          let my_array: [i64; 3] = [0; 3];
28          //or let my_array: [i64; 3] = [0, 0, 0];
29          let my_vector: Vec<i64> = vec![1, 2, 3, 4, 5];
30
31          //tuple declaration
32          let is_aa2f_pair: (String, bool) = ("abacaba".to_owned(), true);
33
34          //Tuples can be destructured into variables
35          let (word, is_aa2f) = is_aa2f_pair;
36          println!("Word {}'s aa2f status is: {}", word, is_aa2f);
37          //Output: Word abacaba's aa2f status is: true
38
39  }
```

Listing 1: Variable Declaration Demonstration

3.2. Memory Management

Rust uses memory management model first implemented in the Cyclone language [20, 21]. At any given point of time, each unit of data is owned by exactly one binding. When this binding goes out of scope, the data is deallocated. This concept applies to heap-allocated data as well. When, for example, a smart unique pointer (**Box**) goes out of scope, the heap-allocated data it was pointing to is deallocated. Ownership can be transferred (or the data can be moved) to another binding. This approach allows to deterministically free memory without enforcing a programmer to do it manually (which is error-prone) or using a garbage collector (which is slow).

Data can also be borrowed, both mutably or immutably. A borrow cannot outlive the owned binding. This is guaranteed by specifying lifetimes, however, more often than not, the compiler can elide them automatically. There can be only one mutable borrow (and no immutable ones) or indefinitely many immutable borrows. When the borrow goes out of scope, the borrowed data is returned. If the need arises to truly share the data (instead of borrowing), it can done by using a reference-counted pointer

(**Rc<T>**) or its thread-safe atomic version (**Arc<T>**). When doing so, one still needs to synchronize the access to the shared data, which often leads to using types similar to **Arc<Mutex<T>>**.

```
fn consume_vec(arg_vec: Vec<i64>) {
}

fn demonstrate_borrowing_and_deallocation() {
    let mut x = 0i64;
        let y = 10i64;
        //Boxing the data allocates it on the heap
        //and creates a unique pointer to it
        let mut z = Box::new(20i64);
        //Blocks are denoted by curly braces.
        //At the end of each block, all owned variables are deallocated
        //and all borrows are returned.
        {
                //Mutable variables can be borrowed either
                //by mutable reference or by immutable reference

                //x_1 is, in terms of C, a constant pointer to non-constant data
                let x_1: &mut i64 = &mut x;

                %//There can be only one mutable reference:
                %//let mut x_2: &mut i64 = &mut x; //Compilation error:
                %//???

                %//Immutable data cannot be mutably borrowed
                %//!!!!!!!!!!!!!
                //Alternative syntax, note the "ref" keyword
                //and lack of "&" on the right-hand side
                let ref y_1 = y;
                //The number of immutable references is not limited
                let ref y_2 = y;


                //Note that the z_1 is immutable, but
                //the reference to boxed data it holds is mutable
                let z_1: &mut Box<i64> = &mut z;

                //Immutable variables can be only borrowed immutably
                //let mut y_1: &mut i64 = &mut y; //Compilation error:
                //Cannot borrow immutable variable as mutable
                let y_1: &i64 = &y;

                //References need to be dereferenced by "*"
                *x_1 = *x_1 + 5;
                //Note the double dereferencing for "z_1"
                //You need to dereference the reference to access the Box
                //And you need to dereference the Box to access the data
                **z_1 += 5;

                //*y_1 += 5;//Compilation error:
                //Cannot assign to immutable borrowed content

                //Functions and macros often dereference args implicitly
                println!("x_1:{}, y_1:{}, z_1:{}", x_1, y_1, z_1);
                //Output: x_1:5, y_1:10, z_1:25

                //Data cannot be borrowed twice
                //println!("x:{}, y:{}, z:{}", x, y, z);//Compilation error:
                //Cannot borrow x as immutable because
                //it is already borrowed as mutable

                //End of block, "z_1" is dropped
                //"y_2" and "y_1" is dropped
                //"x_1" is dropped
        }
        //Since the references were used, the original
        //variables' values changed as well
        println!("x:{}, y:{}, z:{}", x, y, z);
        //"z" (including data on the heap), "y" and "x" are deallocated
}
```

```
71
72  fn demonstrate_moving() {
73          //Data can be owned only by exactly one variable.
74          //If it is being assigned to another variable, and type
75          //implements the `Copy` trait, the new value is constructed.
76          //Otherwise, data ownership is transferred.
77          //i64 is `Copy`
78          let x = 0i64;
79          let mut y = x;
80          y += 5;
81          println!("x:{}, y:{}", x, y);
82          //Output: x:0, y:5
83
84          //Vec<_> is not `Copy`
85          let vec_1 = vec![0i64, 1, 2];
86          //Note that vec_2 is mutable, but vec_1 was not.
87          //Mutability is not a property of data,
88          //but a property of binding.
89          let mut vec_2 = vec_1;
90          vec_2.push(3);
91          //println!("vec_1: {:?}", vec_1); //Compilation error:
92          //Use of moved value
93
94          println!("vec_2:{:?}", vec_2);
95          //Output: vec_2:[0, 1, 2, 3]
96
97          //Function `consume_vec` receives argument by value (as opposed
98          //to "by reference"), receiving the ownership, and consumes it.
99          consume_vec(vec_2);
100
101         //println!("vec_2: {:?}", vec_2); //Compilation error:
102         //Use of moved value
103 }
104
105 fn main() {
106         demonstrate_borrowing_and_deallocation();
107         demonstrate_moving();
108
109 }
```

Listing 2: Memory Management Demonstration

3.3. Custom Data Types

Writing a complex program using only predefined types would be inconvenient. For that reason, a programmer is allowed to introduce additional types. Custom data types in Rust are created through either of two keywords:

- **struct** – usually used to define a classic C-like struct.
- **enum** – used to define an enumerated type.

Rust's structs are generally similar to those of C. Specifically, a struct in either of languages can be considered a list of variables of different types grouped together. However, the concepts are not completely identical. That is, structs in Rust can implement methods as well, which bears similarity to C++ classes. On the other hand, unlike C++ classes, Rust structs cannot inherit from each other, due to the fact that the language itself adopts the "composition over inheritance" principle.

Enumerations are more powerful in Rust than in most other programming languages that use the **enum** keyword. In particular, Rust enumerations can have data associated with each enumeral. In other words, enumeration in Rust is a *sum type* (also known as *tagged union*). Sum types are usually present in FP languages, e.g., Haskell and OCaml [16].

Example of creating custom data types is given in Listing 3. Idiomatic way of working with enumerations is described in subsection 3.4.

```
//Defining custom data structure with 2 fields
struct Vector2 {
        x: f64,
        y: f64,
}

//Defining methods of the struct
impl Vector2 {
        //Rust does not have constructors
        //By convention *static method* "new" is used
        fn new(in_x: f64, in_y: f64) -> Vector2 {
                return Vector2 {
                        x: in_x,
                        y: in_y,
                };
        }

        //Method that does not mutate "self".
        fn len(&self) -> f64 {
                //Note the lack of semicolon
                //Identical to `return` statement
                (self.x * self.x + self.y * self.y).sqrt()
        }

        //Method that mutates "self".
        fn normalize(&mut self) {
                let len = self.len();
                self.x /= len;
                self.y /= len;
        }

        //Method that consumes and drops "self".
        fn into_tuple(self) -> (f64, f64) {
                (self.x, self.y)
        }
}

//Defining custom enumerable.
//In Rust enumerals can also store other types inside
enum Task {
        Sum(i32, i32),
        Print(String),
        DoNothing
}

fn main() {
        let mut my_vec_1 = Vector2 {
                x: 3f64,
                y: 4f64,
        };
        let my_vec_2 = Vector2::new(8f64, 6f64);
        println!("Lengths are {} and {}", my_vec_1.len(), my_vec_2.len());
        //Output: Lengths are 5 and 10

        my_vec_1.normalize();
        //Following line causes compiler error
        //"my_vec_2" is immutable, but method "normalize"
        //takes mutable reference to "self"

        //my_vec_2.normalize();
```

```
62          //structs can be destructured
63          let Vector2{x: x_1, y: y_1} = my_vec_1;
64          //Shorthand notation
65          let Vector2{x, y} = my_vec_2;
66
67          println!("Destructured: ({},{}) and ({},{})", x_1, y_1, x, y);
68          //Output: Destructured: (0.6,0.8) and (8,6)
69
70          //Initializing 3 enumerables with different values
71          let enum_1 = Task::DoNothing;
72          let enum_2 = Task::Sum(3, 5);
73          let enum_3 = Task::Print("Hello".to_owned());
74          //enums' associated data can be destructured as well
75          //Conditional destructuring
76          if let Task::Print(msg) = enum_3 {
77                  println!("{}", msg);
78                  //Output: Hello
79          }
80  }
```

Listing 3: Complex Data Types Demonstration

3.4. Control Flow

To implement any sort of logic, control flow structures need to be used. On this matter rust does not differ drastically from widely used languages: it employs the **if-then-else** expressions, **match** expressions (similar to **switch-case** in C), **for** loops and **while** loops. Additionally, the **loop** keyword indicates an infinite loop; the compiler will be able to apply better optimizations to the **loop** repetitions than to the **while (true)**. However, the **do-while** loops are not present in Rust.

The programmer should acknowledge that Rust is an expression language (as opposed to a statement language), hence (almost) everything returns a value. Albeit, oftentimes the value returned is a *unit*, i.e., an empty tuple.

Mechanism of throwing and catching exceptions is used by numerous programming languages, but is not present in Rust, for the following reason [22]:

> Exceptions complicate understanding of control-flow, they express validity/invalidity outside of the type system, and they interoperate poorly with multithreaded code (a major focus of Rust).

Instead, a function which may fail should return a sum type, which can be either the result (if the function was executed successfully) or an error (otherwise). The standard library provides two enumerables for that purpose: (1) **Option<T>**, enumerals of which are **Some(T)** and **None**, thus making it represent a nullable value, and (2) **Result<T, E>**, which can be either **Ok<T>** or **Err<E>**.

```
fn demonstrate_branching() {
        //Unlike C, Rust does not require parentheses in conditionals
        if true {
                println!("It works!");
        } else {
                println!("This should not happen");
        }
        //Output: It works!

        //Rust is an expression language (as opposed to statement
 ↪  language)
        //So everything can be evaluated
        let x = 3i64;
        let y = 5i64;
        //let max = if x >= y {x;} else {y};//Compilation error:
        //branches return values of different types

        let max = if x >= y {x} else {y};

        //Rust's analog of C switch\case statement
        match max {
                3 => println!("Three is greater than five!"),
                5 => println!("Five is greater than three"),
                //match enforces exhaustiveness checking.
                //That is, every possible option should be checked.
                //"_" means "anything else" in this context.
                _ => {
                        println!("This should not be happening");
                        panic!();
                }
        }
        //Output: Five is greater than three

        //Can also be used as expression
        let opt: Option<i64> = Some(10);
        let num = match opt {
                Some(x) => x,
                None => 0,
        };
        //Enums Option and Result also implement method "unwrap"
        //which is a shorthand for getting a value if it is Some/Ok and
 ↪  panicking otherwise
        let res: Result<String, i64> = Ok("Something".to_owned());
        println!("{}", res);
        //Output: Ok("Something")
        println!("{}", res.unwrap());
        //Output: Something
}

fn demonstrate_looping() {
        //infinite loop
        loop {
                //breaking out of innerwards-most loop
                break;
        }

        //while loop
        let mut i = 0i64;
        while i < 5 {
            i += 1;
                if i == 3 {
                        //ingnoring the rest of the loop body
                        //and going to next interation
                        continue;
                }
                print!("{} ", i);
        }
        println!("");
        //Output:1 2 4 5

        //There are no `do-while` loops, however

        //For loops can iterate over anything that
        //implements trait `Iterator`

        //Looping over range:
```

```
        for j in 5..10 {
                print!("{} ", j);
        }
        println!("");
        //Output:5 6 7 8 9

        //Looping over arrays:
        let mut nums = [-2i64, -1, 0, 1, 2];
        let mut sum = 0i64;
        let mut sum_of_squares = 0i64;

        //Looping over immutable references:
        for num in nums.iter() {
                sum += *num;
        }
        println!("{}", sum);
        //Output: 0

        //Looping over mutable references:
        for num in nums.iter_mut() {
                *num *= *num;
        }
        println!("{:?}", nums);
        //Output:[4, 1, 0, 1, 4]

        //Looping and taking ownership. Consumes vec
        for num in nums.into_iter() {
                sum_of_squares += num;
        }
        println!("{}", sum_of_squares);
        //Output: 10
        //println!("{:?}", nums);//Compilation error:
        //use of moved value
}

fn main() {
        demonstrate_branching();
        demonstrate_looping();
}
```

Listing 4: Control Flow Demonstration

3.5. Polymorphism

Polymorphism in Rust is achieved via traits and generics (functions and data types) with trait bounds. While generics may be familiar to the reader from other languages, and as such will not be covered in detail, traits deserve deeper explanation.

Schärli et al. [23] introduced traits in 2003 and described them as "Composable units of behaviour". They noticed that in Object-Oriented Programming classes play two competing roles: (1) *generator of instances*, and as such, class should be logically complete, and (2) *unit of reuse*, which should be as small as possible. Due to this observation, the team proposed the idea of traits, which are, when described in a simplified way, collections of required and provided methods. Each class (or, in case of Rust, struct) can implement any amount of traits. Each trait may depend on any amount of other traits. Traits do not specify and cannot access any state variables. Traits have a way to resolve name conflicts, therefore classes (or structs) can implement

traits which have methods with the same name. Listing 5 provides examples of working with traits in Rust.

It should also be noted, that traits often are considered similar to Haskell's type-classes [16, 22]; this can be interpreted as another sign of FP influence on Rust.

```
//Trait declaration
trait MyTrait {
        fn say_hello(&self);
}
trait MyOtherTrait {}

//Data type need to implement both MyTrait and MyOtherTrait
//to be able to implement MyDependentTrait
trait MyDependentTrait: MyTrait + MyOtherTrait {
        fn required_method(&self);

        fn say_hello(&self);

        fn provided_method(&self) {
                println!("provided_method");
                println!("It is same for every type by default, but can be redefined");
        }
}
//Some traits can be derived by compiler
#[derive(Clone,Copy,Debug)]
struct Foo {
    x: isize,
}

impl MyTrait for Foo {
        fn say_hello(&self) {
                println!("Hello from Foo");
        }
}

#[derive(Clone,Copy,Debug)]
struct Bar<T: MyTrait> {
        data: T,
}
impl<T: MyTrait> MyTrait for Bar<T> {
        fn say_hello(&self) {
                println!("Hello from Bar's implementation of MyTrait");
        }
}

impl<T: MyTrait> MyOtherTrait for Bar<T> {}

impl<T: MyTrait> MyDependentTrait for Bar<T> {
        fn required_method(&self) {
                println!("Calling hello on self and on data from Bar's required_method");
                //self.say_hello(); //Name conflict
                //Need to specify which trait's implementation exactly to call
                MyDependentTrait::say_hello(self);
                self.data.say_hello();
        }

        fn say_hello(&self) {
                println!("Hello from Bar's implementation of MyDependentTrait");
        }
}

//generic function
fn exec_say_hello<T: MyTrait>(greeter: &T) {
        greeter.say_hello();
}

fn main() {
```

```
66          let foo = Foo{x: 0};
67          //"Bar" did not specify the stored type explicitly,
68          //but rather "any type that implement MyTrait".
69          //"Foo" fits this requirement.
70          let bar = Bar{data: Foo{x: 5}};
71          exec_say_hello(&foo);
72          //Output: Hello from Foo
73
74          //exec_say_hello only has "MyTrait" as a trait bound, so no name
    ↪   conflict occurs
75          exec_say_hello(&bar);
76          //Output: Hello from Bar's implementation of MyTrait
77
78          //Calling say_hello on "bar" manually will lead to compilation
    ↪   error
79          bar.required_method();
80          //Output:
81          //Calling hello on self and on data from Bar's required_method
82          //Hello from Bar's implementation of MyDependentTrait
83          //Hello from Foo
84  }
85
```

Listing 5: Polymorphism Demonstration

4. Implementation

4.1. Task Overview

As it was outlined in the Introduction (see 1.3), two programs needed to be done in the scope of this study. The primary concern is achieving high performance. The reason for that is the exponential growth of the number of possible extensions. Provided that the modern CPUs have multiple cores, the programs should parallelize the workload for the best performance.

The first program (hereafter mentioned as "Extender") attempts to create a word of maximum possible length. It should also be able to continue working on words from input file, rather than start anew each time. The input file for this program is a list of words that will be used as prefixes, separated by line breaks.

The second program (hereafter "Classifier") marks words from input file as "satisfactory" or "unsatisfactory" based on the amount of symbols that can be appended and prepended (not simultaneously). The idea behind is to create a list of possible parts from which considerably longer words can be constructed. It might also be that additional restrictions need to be imposed in order to achieve better sorting accuracy. Therefore the program will not only prohibit factors which can be divided into two parts with equal number of occurrences of letters. Words with factors that can be divided into two parts the difference between which equals a user defined value (hereafter

referenced as "template vector") will be avoided as well. By providing only template vector $(0, 0, 0)$, the user reduces the task to avoiding abelian squares. The input file should be in the following format:

1. Number of template vectors (n).

2. Template vectors; consists of n lines. The individual elements of each template vector are separated by spaces.

3. Space separated numbers l and r representing by how many letters the words should be extended to the left and to the right.

4. Words that should be extended. Each word starts on a new line.

Despite the notable differences between these two programs, a considerable amount of code written for one of them can be reused by the other. The decision was made to implement the Extender first and later to modify it into Classifier. The full code of the programs can be found in appendices 1, 2, 3, 4, 5 and 6.

4.2. Single-Threaded Code

Instead of implementing the multithreaded code immediately, the author decided to write a single-threaded version of the Extender program and parallelize it afterward. This section covers the programming of the single-threaded version of the Extender. A word will be represented as a list of unsigned 8-bit integer values, that is, **Vec<u8>**.

One of the most important pieces of code in this program is the function that checks if a word is AA2F or not. Listing 6 demonstrates the most straightforward approach to implementing this function. For better readability, calculation of the Parikh vector of a word or a factor was moved into independent function.

```
1  //Parikh vector has 3 elements because with are working with alphabet of 3
   ↪  characters
2  fn calculate_parikh_vector(factor: &[u8]) -> [usize; 3] {
3      let mut result = [0; 3];
4      for letter in factor.iter() {
5          result[*letter as usize] += 1;
6      }
7      result
8  }
9
10 fn is_aa2f_full_check(word: &Vec<u8>) -> bool {
11     let l = word.len();
12     //words of length lesser than 4 are always AA2F
13     for part_l in 4..(l + 1) {
14         for window in word.windows(part_l) {
```

```
            //factors of odd length cannot be split into halves
            if part_l % 2 == 1 {
                continue;
            }
            let pv_1 = calculate_parikh_vector(&window[0..(part_l / 2)]);
            let pv_2 = calculate_parikh_vector(&window[(part_l /
↪  2)..part_l]);
            if  pv_1 == pv_2 {
                return false;
            }
        }
    }
    true
}
```

Listing 6: AA2F Full Check

As it was noted earlier in the thesis, the number of words grows exponentially as the length increases, therefore narrowing down the search field is advised. This can be done by checking only those words that are created by appending a single character to a word that is already known to be AA2F. To do so intermediate results need to be stored. The infinite loop that acts in this way is can be found in Listing 7. It also demonstrates how creation of words with given prefix can be done.

```
pub fn word_to_string(word: &Vec<u8>) -> String {
    let utf: Vec<u8> = word.iter().map(|&x| x + 97).collect();
    String::from_utf8(utf).expect(&*format!("Failed at line {}", line!()))
}

fn spawn_children(parent: &Vec<u8>) -> Vec<Vec<u8>> {
    let mut result = Vec::with_capacity(3);
    for i in 0..3 {
        let mut new_word: Vec<u8> = Vec::with_capacity(parent.len() + 1);
        new_word.clone_from(parent);
        new_word.push(i as u8);
        result.push(new_word);
    }
    result
}

fn generate_words(prefixes: &Vec<Vec<u8>>) {
    let mut stack: Vec<Vec<u8>> = Vec::new();
    stack.clone_from(prefixes);
    //if necessary add empty word to begin generating words from scratch
    if stack.len() == 0 {
        stack.push(Vec::new());
    }
    let mut max_len: usize = 0;
    //execute the loop until the stack is empty
    while let Some(parent) = stack.pop() {
        let children: Vec<Vec<u8>> = spawn_children(&parent);
        let aa2f_children: Vec<Vec<u8>> = children
            .into_iter()
            .filter(|wrd| is_aa2f_full_check(wrd))
            .collect();
        for child in aa2f_children.into_iter() {
            //print new word only if it is the longest known
            if child.len() > max_len {
                max_len = child.len();
                println!("{} {}", child.len(), word_to_string(&child))
            }
            stack.push(child);
        }
    }
}
```

Listing 7: Single-Threaded Extender

It should be noted, that different data structures can be used to alter the order of word processing. For example, using a stack (a Last-In-First-Out data structure) will result in a depth-first search (that is, longest words first), while using a queue (a First-In-First-Out data structure) will result in a breadth-first search (shortest words first).

When utilizing such approach, it is known that the parent word of any word is AA2F and therefore any factor that was present in the parent need not be checked. This leaves us to check only suffixes of newly created words. By reducing the number of factors needed to be processed to check a single word a performance boost can be gained. The optimized AA2F checking function that uses this method is available in Listing 8.

Another optimization worth implementing is using cumulative Parikh vectors. Essentially, cumulative Parikh vector is a list of Parikh vectors of all prefixes of a word. Generation of cumulative Parikh vector has time complexity $O(n)$, where n is the size of the entire word. After this the Parikh vector of any factor of this word can be calculated by finding a difference of only two element of cumulative Parikh vector. This operation executes in constant time ($O(1)$). Contrast with $O(m)$ (where m is the factor's length; $0 \leq m \leq n$) required to calculate a Parikh vector of each factor when using direct approach.

```
fn calculate_parikh_diff(p1: &[usize; 3], p2: &[usize; 3]) -> [usize; 3] {
    let mut result = [0; 3];
    for i in 0..p1.len() {
        result[i] = p2[i] - p1[i];
    }
    result
}
fn calculate_cumul_pv(word: &Vec<u8>) -> Vec<[usize; 3]> {
    let mut result = Vec::with_capacity(word.len() + 1);
    result.push([0; 3]);
    for i in 0..word.len() {
        let mut new_elem = result[i].clone();
        new_elem[word[i] as usize] += 1;
        result.push(new_elem);
    }
    result
}
pub fn is_aa2f_partial_check(word: &Vec<u8>) -> bool {
    if word.len() < 4 {
        return true;
    }
    let l = word.len();
    let cumul_pv = calculate_cumul_pv(word);
```

```
    for i in 2..(l / 2 + 1) {
        let diff2 = calculate_parikh_diff(&cumul_pv[l - i], &cumul_pv[l]);
        let diff1 = calculate_parikh_diff(&cumul_pv[l - 2 * i],
↪   &cumul_pv[l - i]);
        if diff1 == diff2 {
            return false;
        }
    }
    true
}
```

Listing 8: Optimized AA2F Check

The last modification to the **generate_words** function worth noting is using a closure function which would determine when to stop. This closure function evaluates each word before it is added to the queue and returns a boolean value representing whether generation should halt or continue.

```
fn generate_words<F>(prefixes: &Vec<Vec<u8>>, should_halt: F) -> String
where F: 'static + Fn(&Vec<u8>) -> bool{
    let mut stack: Vec<Vec<u8>> = Vec::new();
    stack.clone_from(prefixes);
    //add empty word if necessary to begin generating words from scratch
    if stack.len() == 0 {
        stack.push(Vec::new());
    }
    let mut max_len: usize = 0;
    //execute the loop until the stack is
    while let Some(parent) = stack.pop() {
        let children: Vec<Vec<u8>> = spawn_children(&parent);
        let aa2f_children: Vec<Vec<u8>> = children
            .into_iter()
            .filter(|wrd| is_aa2f_partial_check(wrd))
            .collect();
        for child in aa2f_children.into_iter() {
            //print new word only if it is the longest known
            if child.len() > max_len {
                max_len = child.len();
                println!("{} {}", child.len(), word_to_string(&child))
            }
            if should_halt(&child) {
                return word_to_string(&child);
            }
            stack.push(child);
        }
    }
    unreachable!()
}

fn main() {
        //Attempt to generate and print a 10-character long word starting
↪   with "abc"
    let final_word = generate_words(&vec![vec![0u8, 1, 2]], |word:
↪   &Vec<u8>| word.len() >= 10);
    //Output: "Final word: abcccbbbca"
    println!("Final word: {}", final_word);
}
```

Listing 9: Using Closure Function to Halt the Program

This concludes the list of functions necessary to implement the Extender program. The Classifier program relies on similar functions, however its analogs of AA2F checks are slightly more complex. As it was noted in subsection 4.1, instead of checking halves

of factors for equality, Classifier needs to perform more complex calculations. The Classifier's analogs of AA2F checking function need to see if factor can be divided into two such parts, that difference between them equals to a user-defined template vector. The code of template vector checking functions is shown below. Another thing to note about the Classifier program is that trying to extend a word to the left can be implemented as trying to extend reversed word to the right.

```
pub fn is_template_vector_free_full_check(word: &Word, templates: &Vec<[isize; cfg::CARDINALITY]>) -> bool {
    let l = word.len();
    //it is possible to shorthen the full_check by calling partial_check from it
    for part_len in 1..(l + 1) {
        for window in word.windows(part_len) {
            if !is_template_vector_free_partial_check(&window.to_vec(), templates) {
                return false;
            }
        }
    }
    true
}

pub fn is_template_vector_free_partial_check(word: &Word, templates: &Vec<[isize; cfg::CARDINALITY]>) -> bool {
    let l = word.len() as isize;
    let cumul_pv = calculate_cumul_pv(word);
    for templ_vec in templates.iter() {
            //norm of the template vector represents by how many character
            //should one part be longer than the other
        let norm = templ_vec.iter()
           .fold(0, |acc, &x| acc + x);
        let mut i: isize = 1;
        while (i * 2) as isize + norm <= l as isize {
            let diff1 = parikh_diff(&cumul_pv[(l - 2 * i - norm) as usize], &cumul_pv[(l - i) as usize]);
            let diff2 = parikh_diff(&cumul_pv[(l - i) as usize], &cumul_pv[l as usize]);+9
            let mut are_eq = true;
            for j in 0..cfg::CARDINALITY {
                if diff1[j] - diff2[j] != templ_vec[j] {
                    are_eq = false;
                }
            }
            if are_eq {
                return false;
            }
            i += 1;
        }
    }
    true
}
```

Listing 10: Functions Checking if Word is Template Vector Free

Listing 10, which provides the code of template-checking functions used by the Classifier, concludes this subsection. The code used to process command line parameters and read file input provides no interest in the scope of this thesis. Nevertheless, it can be found in Appendices 1 and 4 for the Extender and the Classifier respectively.

4.3. Parallelization

The next step after writing a single-threaded program is to make full use of modern multi-core CPUs by parallelizing the program. Two major problems needed to be solved: how to communicate between threads and how to organize their work in the optimal manner.

4.3.1. Thread Communication

Because the task requires constant updates on the current state (i.e., which words need to be tested), the communication between threads is requisite. Two primary methods of achieving this exist in Rust: communication over channels and communication by sharing memory.

Channels in Rust are one-way "multi-producer, single-consumer FIFO queue communication primitives"[24]. They can transfer ownership of almost any data type without any preparations required. To be more precise, data types implementing trait **Send** can have their ownership transferred between threads in a safe manner. The trait is derived by compiler automatically when appropriate (that is, when all fields of a data structure implement **Send** themselves). However using channels for two-way communication is problematic. It can be emulated to some extent by using two one-way channels, but such a solution can be hard for a programmer to write and, later, to read.

Communicating by sharing memory has stricter requirements: shared data should implement both **Send** and **Sync** traits. **Sync** trait marks data that can be safely shared between threads. It can be formally expressed as "a type **T** is **Sync** if **&T** is thread-safe"[24]. **Sync** is not as common as **Send**. The simplest solution is to wrap data type in a **Mutex** (mutual exclusion device) which will implement both **Send** and **Sync** if wrapped type implements **Send**. However, the **Mutex** itself is an owned data type and as such cannot be easily shared; its only purpose is to prevent simultaneous access to the underlying data. To overcome this problem, the **Mutex** itself should wrapped in an **Arc** (atomically reference counted wrapper) to make it possible to share data without violating Rust's memory model. The final type can be denoted as **Arc<Mutex<T>>**.

To determine which of those approaches results in better performance, a test was conducted. In this test the master thread was distributing tasks between worker threads. The time needed to complete the tasks for measured for the single-threaded

code, for the multithreaded code that used channels and for the multithreaded code that used shared memory.

The testing code also employs bulking: transferring more than one task at a time. Such technique can reduce the communication overhead to some extent. Listing 11 displays the code that was used to measure the performance of different communication methods. For the sake of simplicity, the computational task being parallelized was unrelated to AA2F research. The program attempted to run 4 threads simultaneously. The tests were conducted on a 2-core 4-thread (due to the Intel Hyper-Threading Technology) CPU. The full code is available in Appendix 7.

```
fn test_shared_memory() {
        //global task queue that is shared between threads
    let numbers_to_process:Arc<Mutex<Vec<i64>>> =
↪   Arc::new(Mutex::new((1..NUM_COUNT+1).collect()));
    let mut sum = 0i64;
    let start: u64 = precise_time_ns();
    let mut threads: Vec<JoinHandle<_>> =
↪   Vec::with_capacity(SPAWNDED_THREAD_COUNT);
    for _ in 0..SPAWNDED_THREAD_COUNT {
        let nums = numbers_to_process.clone();
        let handle = spawn(move || {
            let mut local_sum = 0i64;
            //local task queue in which bulked tasks  are stored
            let mut task_queue:Vec<i64> = Vec::with_capacity(BULK_SIZE);
            loop {
                if task_queue.len() > 0 {
                    let num = task_queue.pop().expect("Pop result should
↪   not be empty");
                    //collatz_length is the computation-heavy function
                    local_sum += collatz_length(num);
                } else {
                    let mut lock = nums.lock().expect("Mutex was
↪   poisoned");

                    if lock.len() == 0 {
                        return local_sum;
                    } else {
                        for _ in 0..min(lock.len(), BULK_SIZE) {
                            task_queue.push(lock.pop().unwrap());
                        }
                    }
                }
            }
        });
        threads.push(handle);
    }

    for join_handle in threads {
        sum += join_handle.join().unwrap();
    }
    let end: u64 = precise_time_ns();
    println!("Sum is: {:?}", sum);
    println!("Took {} ns to compute in parallel using shared memory with
↪   bulk size of {}", end - start, BULK_SIZE);
}

fn test_channels() {
        //global task queue
    let mut numbers_to_process:Vec<i64> = (1..NUM_COUNT+1).collect();
    let mut sum = 0i64;
    let mut threads: Vec<JoinHandle<_>> =
↪   Vec::with_capacity(SPAWNDED_THREAD_COUNT);
    let start: u64 = precise_time_ns();
```

```
    //the channel via which the main thread receives results from working threads
    //the Sender of the worker's channel is passed as well, so that the main thread
    //can send new task to the worker.
    let (main_tx, main_rx): (Sender<(Sender<Option<Vec<i64>>>, i64)>, Receiver<(Sender<Option<Vec<i64>>>, i64)>) = channel();
    for _ in 0..SPAWNDED_THREAD_COUNT {
        let tx = main_tx.clone();
        let handle = spawn(move || {
            let (local_tx, local_rx): (Sender<Option<Vec<i64>>>, Receiver<Option<Vec<i64>>>) = channel();
            tx.send( (local_tx.clone(), 0) );

            while let Ok(new_tasks_opt) = local_rx.recv() {
                if let Some(new_tasks) = new_tasks_opt {
                    let mut local_sum = 0i64;
                    //new_tasks is the Vec that was sent by the main thread
                    //serves as the local task queue
                    for i in new_tasks.iter() {
                        local_sum += collatz_length(*i);
                    }
                    //send to the main thread the local channel's Sender
                    //and results of computation
                    tx.send( (local_tx.clone(), local_sum) );
                } else {
                    return;
                }
            }
        });

        threads.push(handle);
    }
    drop(main_tx);
    while let Ok( (tx, collatz_result) ) = main_rx.recv() {
        //receive the results from workers
        sum += collatz_result;
        if numbers_to_process.len() == 0 {
            tx.send(None);
        } else {
            let mut data_bulk: Vec<i64> = Vec::with_capacity(BULK_SIZE);
            for _ in 0..min(BULK_SIZE, numbers_to_process.len()) {
                data_bulk.push(numbers_to_process.pop().unwrap());
            }
            //and send new tasks to worker if there were some left
            tx.send(Some(data_bulk));
        }
    }

    for join_handle in threads {
        join_handle.join();
    }

    let end: u64 = precise_time_ns();
    println!("Sum is: {:?}", sum);
    println!("Took {} ns to compute in parallel using channels with bulk size of {}", end - start, BULK_SIZE);
}
```

Listing 11: Testing Tread Communication Methods

Multiple runs with different bulk sizes were conducted; the details can be found in Appendix 8. Each run provided results about the time taken by the shared memory implementation, by the channels implementation and, as a baseline, by the serial program. On the figure below the reader can see the chart showing how many times faster the parallelized versions were than the serial one on average.

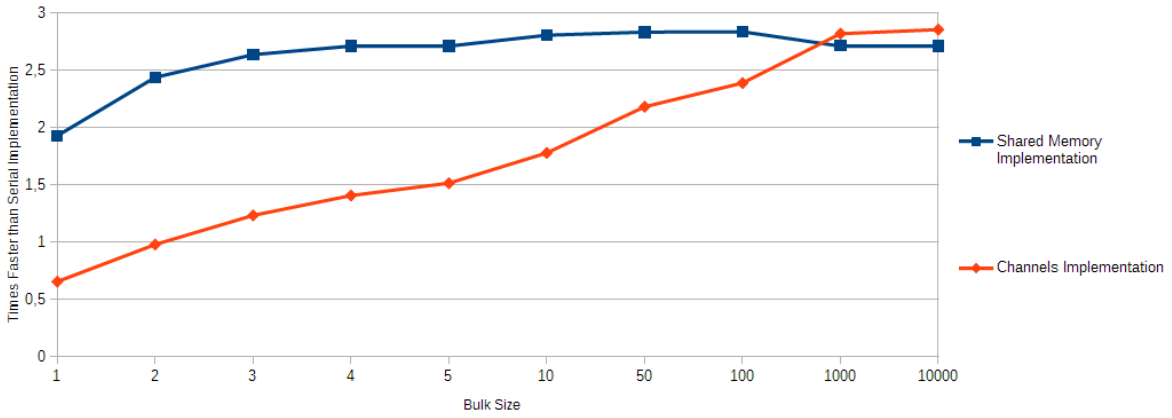

Figure 2: Comparison of Performance of Thread Communications Methods

It can be seen that the channels implementation has comparable performance to that of the shared memory implementation only at large bulk sizes. Furthermore, when only one task at a time was transferred, the channels version had lower performance than the serial program. Therefore the choice was between employing bulking together with using channels to communicate between threads, and using shared memory. However, using bulking in the either of programs developed in the scope of this project is unadvised. The reason is that such model of communication can lead to a single thread exhausting the task queue at low queue sizes and, consequently, to the distribution of the tasks in a suboptimal manner. As a result, the shared memory model was chosen.

4.3.2. Generator Structure

At the core of both programs lies the "Generator" **struct** with minor differences. This custom data structure only has 2 or 3 fields:

1. **task_queue**

   Stores words that are awaiting to be processed. Implemented as Binary Heap and therefore acts as a Priority Queue, which allows it to find the best word based on some criteria. The elements of Binary Heap need to implement **Ord** (ordering) trait so that Binary Heap is able to sort elements. To solve this, the custom wrapper for words needed to be written.

2. **thread_number**

Specifies how many additional worker threads this instance of the Generator is allowed to spawn.

3. **template_vecs**

Is a list (**Vec**) of template vector that need to be avoided. Exclusive to Classifier program.

Listing 12 contains the definition of the Generator **struct** as well as implementation of its "main" method (taken from the Classifier program). Note that the **generate_until** method heavily resembles the **generate_words** from Listing 9, but includes multiple mutex locks needed to keep the threads synchronized. In the Extender program the method is to some extent different and needs to employ atomic access operations as well (see Appendix 2).

```
#[derive(Debug)]
pub struct Generator {
    task_queue: Arc<Mutex<BinaryHeap<Task>>>,
    max_len: Arc<atomic::AtomicUsize>,
    thread_number: usize,
    template_vecs: Vec<[isize; cfg::CARDINALITY]>,
}

impl Generator {
            pub fn generate_until<F>(&mut self, should_halt: F) -> String
            where F: 'static + Send + Sync + Fn(&WordWrapper) -> bool {
                let mut threads = Vec::with_capacity(self.thread_number);
                let thread_number = self.thread_number;
                let arced_should_halt = Arc::new(should_halt);
                for _ in 0..thread_number {
                    let mut missed_pops = 0i32;
                    let queue = self.task_queue.clone();
                    let max_len = self.max_len.clone();
                    let should_halt = arced_should_halt.clone();
                    let local_templates = self.template_vecs.clone();
                    let new_thread = thread::spawn(move || {
                        loop {
                            let current_word: WordWrapper;
                            {
                                let mut queue_lock =
↪   queue.lock().expect(&*format!("Failed at line {}", line!()));
                                match queue_lock.pop() {
                                    Some(task) => {
                                        missed_pops = 0;
                                        match task {
                                            Task::ProcessWord(word) =>
↪   current_word = word,
                                            Task::Halt => return None,
                                        }
                                    }
                                    None => {
                                        //if too many "misses" happen,
↪   than the queue is simply empty and generator is stuck
                                        if missed_pops > 1_000_000 {
                                            return None;
                                        } else {
                                            missed_pops += 1;
                                        }
                                        continue;
                                    }
```

```
                                }
                            }
                            let children: Vec<WordWrapper> =
    ↪ current_word.spawn_children()
                                .into_iter()
                                .filter(|wrd|
    ↪ is_template_vector_free_partial_check(wrd, &local_templates))
                                .collect();
                            let mut queue_lock =
    ↪ queue.lock().expect(&*format!("Failed at line {}", line!()));
                            for child in children.iter() {
                                if should_halt(child) {
                                    for _ in 1..thread_number {
                                        queue_lock.push(Task::Halt);
                                    }
                                    return Some(word_to_string(child));
                                }
                            }
                            for child in children.into_iter() {
                                queue_lock.push(Task::ProcessWord(child));
                            }
                        }
                    });
                    threads.push(new_thread);
                }
                let mut result: Option<String> = None;
                for thread in threads {
                    if let Some(res) = thread.join().expect("Could not
    ↪ join thread") {
                        result = Some(res);
                    }
                }
                //return empty string if the generator stopped by
    ↪ exhausting its queue
                result.unwrap_or("".to_owned())
            }

        }
}
```

Listing 12: Generator Definition and Implementation

5. Discussion

The thesis presents the process of development of two high-performance programs in a relatively young programming language. The application of the programs hopefully can advance the ongoing research of almost abelian square-free word on Three letters.

Before the implementation phase could begin, the author needed to conclude extensive research both in the mathematical and technical areas. In the technical research information needed to be acquired about the unusual concepts utilized by the Rust programming language (e.g. memory model and traits).

It can also be noted that the author's previous Extender program, written in JavaScript, utilized the similar architecture and managed to generate an almost abelian square-free word of length 15796 (compare with previous longest known word having the length of about 3150). This new long almost abelian square-free word can be found

in Appendix 9. Hopefully, the program written in Rust can provide even better result due to more efficient thread communication and the language itself being closer to hardware.

The research continues, currently the approach using template vector is more focused when compared to the direct generation of words. If the need arises, the source codes of the programs (which can be found at https://bitbucket.org/AlGvrl/aa2f-rs/src) should be easy to modify so that they will be more applicable to future research tasks.

References

[1] A. Thue, Über unendliche Zeichenreihen, Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiania 7 (1906) 1–22.

[2] P. Erdős, Some unsolved problems, Magyar Tud. Akad. Mat. Kutat Int. Kzl 6 (1961) 221–254.
URL http://www.emis.ams.org/classics/Erdos/cit/87411003.htm

[3] P. A. Pleasants, Non-repetitive sequences, in: Mathematical Proceedings of the Cambridge Philosophical Society, Vol. 68, Cambridge Univ Press, 1970, pp. 267–274.
URL http://journals.cambridge.org/abstract_S0305004100046077

[4] R. C. Etringer, D. E. Jackson, J. A. Schatz, On nonrepetitive sequences, Journal of Combinatorial Theory 16 (1974) 159–164.

[5] F. M. Dekking, Strongly non-repetitive sequences and progression-free sets, Journal of Combinatorial Theory, Series A 27 (2) (1979) 181–185. doi:10.1016/0097-3165(79)90044-X.
URL http://www.sciencedirect.com/science/article/pii/009731657990044X

[6] V. Keränen, Abelian squares are avoidable on 4 letters, in: W. Kuich (Ed.), Automata, Languages and Programming, no. 623 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1992, pp. 41–52, doi: 10.1007/3-540-55719-9_62.

URL http://link.springer.com/chapter/10.1007/3-540-55719-9_62

[7] A. Carpi, On abelian power-free morphisms, International Journal of Algebra and Computation 3 (02) (1993) 151–167.
URL http://www.worldscientific.com/doi/pdf/10.1142/S0218196793000123

[8] A. Carpi, On the number of abelian square-free words on four letters, Discrete Applied Mathematics 81 (13) (1998) 155–167. doi:10.1016/S0166-218X(97)88002-X.
URL http://www.sciencedirect.com/science/article/pii/S0166218X9788002X

[9] V. Keränen, A powerful abelian square-free substitution over 4 letters, Theoretical Computer Science 410 (3840) (2009) 3893–3900. doi:10.1016/j.tcs.2009.05.027.
URL http://www.sciencedirect.com/science/article/pii/S0304397509003892

[10] The Rust programming language.
URL https://www.rust-lang.org/

[11] Which programs are fastest?
URL http://benchmarksgame.alioth.debian.org/u64q/which-programs-are-fastest.html

[12] Some benchmarks of different languages.
URL https://github.com/kostya/benchmarks

[13] The Rustonomicon.
URL https://doc.rust-lang.org/nomicon/

[14] J. Blandy, Why Rust? Trustworty, concurrent systems programming, O'Reilly Media, 2015.
URL http://www.oreilly.com/programming/free/files/why-rust.pdf

[15] J. Regehr, Race condition vs. data race (Mar. 2011).
URL http://blog.regehr.org/archives/490

[16] R. Poss, Rust for functional programmers (Jul. 2014).
URL http://science.raphael.poss.name/rust-for-functional-programmers.html

[17] The Rust Programming Language.
URL https://doc.rust-lang.org/book/

[18] Rust by example.
URL http://rustbyexample.com/

[19] The Rust reference.
URL https://doc.rust-lang.org/reference.html

[20] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, J. Cheney, Region-based memory management in Cyclone, in: ACM Sigplan Notices, Vol. 37, ACM, 2002, pp. 282–293.
URL http://dl.acm.org/citation.cfm?id=512563

[21] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, T. Jim, Safe manual memory management in Cyclone, Science of Computer Programming 62 (2) (2006) 122–144.
URL http://www.sciencedirect.com/science/article/pii/S0167642306000785

[22] Frequently Asked Questions. The Rust Programming Language.
URL https://www.rust-lang.org/faq.html

[23] N. Schärli, S. Ducasse, O. Nierstrasz, A. P. Black, Traits: Composable units of behaviour, in: ECOOP 2003 Object-Oriented Programming, Springer, 2003, pp. 248–274.
URL http://link.springer.com/chapter/10.1007/978-3-540-45070-2_12

[24] Rust Standard Library API Reference.
URL https://doc.rust-lang.org/std/

Appendices

1. Contents of the src/main.rs File (Extender)

```rust
extern crate clap;

use clap::{Arg, App};
use std::fs::File;
use std::io::{BufRead, BufReader};

pub mod gen;
pub mod cfg;

fn main() {
    let matches = App::new("AA2F Generator")
        .about("Attemtps to create longest AA2F word")
        .arg(Arg::with_name("thread_num")
            .short("t")
            .long("threads")
            .help("Number of worker threads to be spawned")
            .default_value("4")
            .takes_value(true))
        .arg(Arg::with_name("input_path")
            .short("i")
            .long("input")
            .help("(Optional) Path to the file containing word(s) that should be extended")
            .takes_value(true))
        .arg(Arg::with_name("backtrack")
            .short("b")
            .long("backtrack")
            .help("Whether or not word(s) in the input file should be backtracked")
            .requires("input_path"))
        .get_matches();

    let mut initial_words: Vec<String> = Vec::new();
    let thread_num: usize = matches.value_of("thread_num").expect("Could not ge thread number")
        .parse::<usize>().expect("Thread number should be a positive number");
    if let Some(input_path) = matches.value_of("input_path") {
        let file = File::open(input_path).expect("Could not open the specified input file");
```

```
        let reader = BufReader::new(&file);
        for line in reader.lines() {
            initial_words.push(line.expect("Something is very wrong with the input file"));
        }
    } else {
        initial_words.push("a".to_string());
    }
    let should_backtrack: bool = matches.is_present("backtrack");
    if should_backtrack {
        let mut bt_groups: Vec<Vec<gen::Word>> = Vec::new();
        for wrd in initial_words.iter() {
            bt_groups.push(gen::generate_backtrack(&gen::string_to_word(wrd)));
        }
        for group in bt_groups {
            for word in group.iter() {
                initial_words.push(gen::word_to_string(word));
            }
        }
    }
    println!("Starting gen");

    let mut my_gen = gen::Generator::new(&initial_words, should_backtrack, thread_num);
    my_gen.generate();
}
```

2. Contents of the src/gen.rs File (Extender)

```
use std::collections::binary_heap::BinaryHeap;
use std::sync::{Mutex, Arc};
use std::sync::atomic;
use std::thread;
use std::ops::{Deref, DerefMut};
use std::cmp::{PartialEq, Eq, PartialOrd, Ord, Ordering};
use cfg;

///An alias for a `Vec<u8>`, represents a word.
pub type Word = Vec<u8>;

///A wrapper for a Word, which implements Cmp based on a given heuristic.
///Internally caches the result of [priority-calculating function](fn.calculate_comparison_key.html).
///Wrapper can be automatically dereferenced to the wrapped `Word` under some circumstances.
#[derive(Debug)]
pub struct WordWrapper {
    data: Word,
    comparison_key: i64,
}

///Represents the action that a thread should execute.
#[derive(Debug)]
pub enum Task {
    ///Spawn word's children and push those of them that are AA2F back to the queue.
    ProcessWord(WordWrapper),
    ///Kill the thread.
    Halt
}

impl Deref for WordWrapper {
    type Target = Word;

    fn deref(&self) -> &Word {
        &self.data
    }
```

```
}

impl DerefMut for WordWrapper {
    fn deref_mut(&mut self) -> &mut Word {
        &mut self.data
    }
}

impl PartialEq for WordWrapper {
    fn eq(&self, other: &Self) -> bool {
        self.comparison_key == other.comparison_key
    }
}

impl Eq for WordWrapper {

}

impl PartialOrd for WordWrapper {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        self.comparison_key.partial_cmp(&other.comparison_key)
    }
}

impl Ord for WordWrapper {
    fn cmp(&self, other: &Self) -> Ordering {
        self.comparison_key.cmp(&other.comparison_key)
    }
}

impl WordWrapper {
    pub fn new(word: Word) -> WordWrapper {
        WordWrapper{
            comparison_key: calculate_comparison_key(&word),
            data: word,
        }
    }

    pub fn spawn_children(&self) -> Vec<WordWrapper> {
        let mut result = Vec::with_capacity(cfg::CARDINALITY);
        for i in 0..cfg::CARDINALITY {
            let mut new_word: Word = Vec::with_capacity(self.data.len() + 1);
            new_word.clone_from(&self.data);
            new_word.push(i as u8);
            result.push(WordWrapper::new(new_word));
        }
        result
    }
}

pub fn calculate_comparison_key(word: &Word) -> i64 {
    let length = word.len() as i64;
    let mean: i64 = length / (cfg::CARDINALITY as i64);
    let mut unbalancedness = 0i64;
    unbalancedness -= length % (cfg::CARDINALITY as i64);
    let parikh = calculate_parikh_vector(word);
    for i in parikh.iter() {
        unbalancedness += (*i as i64 - mean).abs() as i64;
    }

    length * length - unbalancedness * unbalancedness * unbalancedness * 27 / 8 / length
}

impl PartialEq for Task {
    fn eq(&self, other: &Self) -> bool {
        match (self, other) {
            (&Task::ProcessWord(ref word_1), &Task::ProcessWord(ref word_2)) => {
                word_1 == word_2
            },
            (&Task::Halt, &Task::Halt) => true,
            _ => false
        }
    }
}

impl Eq for Task {

```

```
}

impl PartialOrd for Task {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        match (self, other) {
            (&Task::ProcessWord(ref w_1), &Task::ProcessWord(ref w_2)) => w_1.partial_cmp(&w_2),
            (&Task::Halt, &Task::Halt) => Some(Ordering::Equal),
            (&Task::Halt, _) => Some(Ordering::Greater),
            (_, &Task::Halt) => Some(Ordering::Less),
            //_ => panic!()
        }
    }
}

impl Ord for Task {
    fn cmp(&self, other: &Self) -> Ordering {
        match (self, other) {
            (&Task::ProcessWord(ref w_1), &Task::ProcessWord(ref w_2)) => w_1.cmp(&w_2),
            (&Task::Halt, &Task::Halt) => Ordering::Equal,
            (&Task::Halt, _) => Ordering::Greater,
            (_, &Task::Halt) => Ordering::Less,
            //_ => panic!()
        }

    }
}

#[derive(Debug)]
pub struct Generator {
    task_queue: Arc<Mutex<BinaryHeap<Task>>>,
    max_len: Arc<atomic::AtomicUsize>,
    thread_number: usize,
}

impl Generator {
    pub fn new(strings: &[String], backtrack_input: bool, threads: usize) -> Generator {
        let max_len: Arc<atomic::AtomicUsize> = Arc::new(atomic::ATOMIC_USIZE_INIT);
        let queue: Arc<Mutex<BinaryHeap<Task>>> = Arc::new(Mutex::new(BinaryHeap::with_capacity(cfg::HEAP_CAPACITY)));
        {
            let mut queue_lock = queue.lock().expect(&*format!("Failed at line {}", line!()));
            for s in strings {
                let word = string_to_word(s);
                if backtrack_input {
                    let bt_words = generate_backtrack(&word);
                    for bt_word in bt_words.into_iter() {
                        queue_lock.push(Task::ProcessWord(WordWrapper::new(bt_word)));
                    }
                } else {
                    queue_lock.push(Task::ProcessWord(WordWrapper::new(word)));
                }
            }
        }
        Generator{
            max_len: max_len,
            task_queue: queue,
            thread_number: threads,
        }
    }

    pub fn generate(&mut self) {
        self.generate_until(|_: &WordWrapper| false);
    }

    pub fn generate_until<F>(&mut self, should_halt: F) -> String
    where F: 'static + Send + Sync + Fn(&WordWrapper) -> bool {
        let mut threads = Vec::with_capacity(self.thread_number);
        let thread_number = self.thread_number;
        let arced_should_halt = Arc::new(should_halt);
        for _ in 0..thread_number {
```

```
182             let mut missed_pops = 0i32;
183             let queue = self.task_queue.clone();
184             let max_len = self.max_len.clone();
185             let should_halt = arced_should_halt.clone();
186             let new_thread = thread::spawn(move || {
187                 loop {
188                     let current_word: WordWrapper;
189                     {
190                         let mut queue_lock =
    ↪ queue.lock().expect(&*format!("Failed at line {}", line!()));
191                         match queue_lock.pop() {
192                             Some(task) => {
193                                 missed_pops = 0;
194                                 match task {
195                                     Task::ProcessWord(word) =>
    ↪ current_word = word,
196                                     Task::Halt => return None,
197                                 }
198                             }
199                             None => {
200                                 if missed_pops > 1_000_000 {
201                                     return None;
202                                 } else {
203                                     missed_pops += 1;
204                                 }
205                                 //thread::sleep_ms(5);
206                                 continue;
207                             }
208                         }
209                     }
210                     let children: Vec<WordWrapper> =
    ↪ current_word.spawn_children()
211                         .into_iter()
212                         .filter(|wrd| is_aa2f_partial_check(wrd))
213                         .collect();
214                     //Comment out from here...
215                     for child in children.iter() {
216                         if child.len() >
    ↪ max_len.load(atomic::Ordering::Relaxed) {
217                             max_len.store(child.len(),
    ↪ atomic::Ordering::Relaxed);
218                             println!("{} {}",child.len(),
    ↪ word_to_string(child));
219                         }
220                     }
221                     //...to here to avoid printing words
222                     let mut queue_lock =
    ↪ queue.lock().expect(&*format!("Failed at line {}", line!()));
223                     for child in children.iter() {
224                         if should_halt(child) {
225                             for _ in 1..thread_number {
226                                 queue_lock.push(Task::Halt);
227                             }
228                             return Some(word_to_string(child));
229                         }
230                     }
231                     for child in children.into_iter() {
232                         queue_lock.push(Task::ProcessWord(child));
233                     }
234                 }
235             });
236             threads.push(new_thread);
237         }
238         let mut result: Option<String> = None;
239         for thread in threads {
240             if let Some(res) = thread.join().expect("Could not join
    ↪ thread") {
241                 result = Some(res);
242             }
243         }
244         result.unwrap_or("".to_owned())
245     }
246
247 }
```

```rust

fn calculate_parikh_chain(word: &Word) -> Vec<[usize; cfg::CARDINALITY]> {
    let mut result = Vec::with_capacity(word.len() + 1);
    result.push([0; cfg::CARDINALITY]);
    for i in 0..word.len() {
        let mut new_elem = result[i].clone();
        new_elem[word[i] as usize] += 1;
        result.push(new_elem);
    }
    result
}
pub fn calculate_parikh_vector(word: &Word) -> [usize; cfg::CARDINALITY] {
    let mut result = [0; cfg::CARDINALITY];
    for letter in word.iter() {
        result[*letter as usize] += 1;
    }
    result
}

fn parikh_diff(p1: &[usize; cfg::CARDINALITY], p2: &[usize;
 ↪  cfg::CARDINALITY]) -> [usize; cfg::CARDINALITY] {
    let mut result = [0; cfg::CARDINALITY];
    for i in 0..p1.len() {
        result[i] = p2[i] - p1[i];
    }
    result
}

pub fn is_aa2f_partial_check(word: &Word) -> bool {
    if word.len() < 4 {
        return true;
    }
    let l = word.len();
    let parikh_chain = calculate_parikh_chain(word);
    for i in 2..(l / 2 + 1) {
        let diff2 = parikh_diff(&parikh_chain[l - i], &parikh_chain[l]);
        let diff1 = parikh_diff(&parikh_chain[l - 2 * i], &parikh_chain[l
 ↪  - i]);
        if diff1 == diff2 {
            return false;
        }
    }
    true
}

pub fn is_aa2f_full_check(word: &Word) -> bool {
    let l = word.len();
    for part_len in 1..(l + 1) {
        for window in word.windows(part_len) {
            if !is_aa2f_partial_check(&window.to_vec()) {
                return false;
            }
        }
    }
    true
}

pub fn word_to_string(word: &Word) -> String {
    let utf: Vec<u8> = word.iter().map(|&x| x + 97).collect();
    String::from_utf8(utf).expect(&*format!("Failed at line {}", line!()))
}

pub fn string_to_word(in_str: &str) -> Word {
    in_str.as_bytes().iter().map(|x| x - 97).collect()
}

pub fn generate_backtrack(word: &Word) -> Vec<Word> {
    let mut result = Vec::with_capacity(word.len() - 1);
    for i in (1..word.len()).rev() {
        result.push(word[0..i].to_vec());
    }
    result
}
```

3. Contents of the src/cfg.rs File (Extender)

```rust
pub const CARDINALITY: usize = 3;
pub const HEAP_CAPACITY: usize = 1_000_000;
```

4. Contents of the src/main.rs File (Classifier)

```rust
extern crate clap;

use clap::{Arg, App};
use std::fs::File;
use std::io::{BufRead, BufReader, Write};

pub mod gen;
pub mod cfg;

use gen::Generator;

fn main() {
    let matches = App::new("AA2F Classifier")
        .about("Classifies words into extendable and ")
        .arg(Arg::with_name("thread_num")
            .short("t")
            .long("threads")
            .help("Number of worker threads to be spawned")
            .default_value("4")
            .takes_value(true))
        .arg(Arg::with_name("input_path")
            .short("i")
            .long("input")
            .help("Path to the file containing word(s) that should be classified")
            .required(true)
            .takes_value(true))
        .arg(Arg::with_name("good_name")
            .short("g")
            .long("good")
            .help("(Optional) Name of the file that will be used as an output for\"good\" words")
            .takes_value(true))
        .arg(Arg::with_name("bad_name")
            .short("b")
            .long("bad")
            .help("(Optional) Name of the file that will be used as an output for\"bad\" words")
            .takes_value(true))
        .get_matches();

    let thread_num: usize = matches.value_of("thread_num").expect("Could not ge thread number")
        .parse::<usize>().expect("Thread number should be a positive number");
    let good_name = matches.value_of("good_name").unwrap_or("sfsg.txt");
    let bad_name = matches.value_of("bad_name").unwrap_or("bad.txt");
    let mut good_out = File::create(good_name).expect("Could not create output file");
    let mut bad_out = File::create(bad_name).expect("Could not create output file");
    let input_path = matches.value_of("input_path").expect("Could not retrieve input file's name");
```

```
46
47      let input = File::open(input_path).expect("Could not open the input
    ↪   file");
48      let reader = BufReader::new(&input);
49      let mut lines = reader.lines();
50      let temp_vec_num: usize = lines.next().expect("Iterator
    ↪   error").expect("File input error")
51          .parse::<usize>().expect("Number parsing error");
52      let mut temp_vecs: Vec<[isize; cfg::CARDINALITY]> =
    ↪   Vec::with_capacity(temp_vec_num);
53      good_out.write_all(temp_vec_num.to_string().as_bytes()).expect("Could
    ↪   not write to the \"good\" file");
54      good_out.write_all("\n".as_bytes()).expect("Could not write to the
    ↪   \"good\" file");
55      for _ in 0..temp_vec_num {
56          let line: String = lines.next().expect("Iterator
    ↪   error").expect("File input error");
57          good_out.write_all(line.as_bytes()).expect("Could not write to the
    ↪   \"good\" file");
58          good_out.write_all("\n".as_bytes()).expect("Could not write to the
    ↪   \"good\" file");
59          let nums: Vec<isize> = line.split_whitespace()
60              .map(|num_str: &str| num_str.parse::<isize>().expect("Number
    ↪   parsing error"))
61              .collect();
62          let mut temp_vec: [isize; cfg::CARDINALITY] = [0;
    ↪   cfg::CARDINALITY];
63          for i in 0..temp_vec.len() {
64              temp_vec[i] = nums[i];
65          }
66          temp_vecs.push(temp_vec);
67      }
68
69      let extensions_str: String = lines.next().expect("Iterator
    ↪   error").expect("File input error");
70      good_out.write_all(extensions_str.as_bytes()).expect("Could not write
    ↪   to the \"good\" file");
71      good_out.write_all("\n".as_bytes()).expect("Could not write to the
    ↪   \"good\" file");
72      let extensions: Vec<usize> = extensions_str
73          .split_whitespace()
74          .map(|num_str: &str| num_str.parse::<usize>().expect("Number
    ↪   parsing error"))
75          .collect();
76      let extension_len_l: usize;
77      let extension_len_r: usize;
78      if extensions.len() != 2 {
79          panic!("Wrong input file format");
80      } else {
81          extension_len_l = extensions[0];
82          extension_len_r = extensions[1];
83      }
84
85      //only words should be left in the file at this point
86      for line in lines {
87          let word: String = line.expect("Could not read a word");
88          let reversed: String = word.chars().rev().collect();
89          let needed_len_l = word.len() + extension_len_l;
90          let needed_len_r = word.len() + extension_len_r;
91          let mut gen_1 = gen::Generator::new(&vec![word.clone()], false,
    ↪   thread_num, temp_vecs.clone());
92          let right_res: String = gen_1.generate_until(move |x| x.len() >=
    ↪   needed_len_r);
93          let can_be_ext_right = (right_res.len() > 0);
94          let mut gen_2 = gen::Generator::new(&vec![reversed], false,
    ↪   thread_num, temp_vecs.clone());
95          let left_res: String = gen_2.generate_until(move |x| x.len() >=
    ↪   needed_len_l);
96          let can_be_ext_left = (left_res.len() > 0);
```

```
        if can_be_ext_right && can_be_ext_left {
            good_out.write_all(left_res.chars().rev().collect::<String>().as_bytes()).expect("Could not write to the \"good\" file");
            good_out.write_all("\n".as_bytes()).expect("Could not write to the \"good\" file");
            good_out.write_all(right_res.as_bytes()).expect("Could not write to the \"good\" file");
            good_out.write_all("\n".as_bytes()).expect("Could not write to the \"good\" file");
        } else {
            bad_out.write_all(word.as_bytes()).expect("Could not write to the \"bad\" file");
            bad_out.write_all("\n".as_bytes()).expect("Could not write to the \"bad\" file");
        }
    }
}
```

5. Contents of the src/gen.rs File (Classifier)

```
use std::collections::binary_heap::BinaryHeap;
use std::sync::{Mutex, Arc};
use std::sync::atomic;
use std::thread;
use std::ops::{Deref, DerefMut};
use std::cmp::{PartialEq, Eq, PartialOrd, Ord, Ordering};
use cfg;

pub type Word = Vec<u8>;

#[derive(Debug)]
pub struct WordWrapper {
    data: Word,
    comparison_key: i64,
}

#[derive(Debug)]
pub enum Task {
    ///Spawn word's children and push those of them that are AA2F back to the queue.
    ProcessWord(WordWrapper),
    ///Kill the thread.
    Halt
}

impl Deref for WordWrapper {
    type Target = Word;

    fn deref(&self) -> &Word {
        &self.data
    }
}

impl DerefMut for WordWrapper {
    fn deref_mut(&mut self) -> &mut Word {
        &mut self.data
    }
}

impl PartialEq for WordWrapper {
    fn eq(&self, other: &Self) -> bool {
        self.comparison_key == other.comparison_key
    }
}

impl Eq for WordWrapper {

}
```

```rust
impl PartialOrd for WordWrapper {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        self.comparison_key.partial_cmp(&other.comparison_key)
    }
}
impl Ord for WordWrapper {
    fn cmp(&self, other: &Self) -> Ordering {
        self.comparison_key.cmp(&other.comparison_key)
    }
}
impl WordWrapper {
    pub fn new(word: Word) -> WordWrapper {
        WordWrapper{
            comparison_key: calculate_comparison_key(&word),
            data: word,
        }
    }

    pub fn spawn_children(&self) -> Vec<WordWrapper> {
        let mut result = Vec::with_capacity(cfg::CARDINALITY);
        for i in 0..cfg::CARDINALITY {
            let mut new_word: Word = Vec::with_capacity(self.data.len() + 1);
            new_word.clone_from(&self.data);
            new_word.push(i as u8);
            result.push(WordWrapper::new(new_word));
        }
        result
    }
}
pub fn calculate_comparison_key(word: &Word) -> i64 {
    let length = word.len() as i64;
    let mean: i64 = length / (cfg::CARDINALITY as i64);
    let mut unbalancedness = 0i64;
    unbalancedness -= length % (cfg::CARDINALITY as i64);
    let parikh = calculate_parikh_vector(word);
    for i in parikh.iter() {
        unbalancedness += (*i as i64 - mean).abs() as i64;
    }

    length * length - unbalancedness * unbalancedness * unbalancedness * 27 / 8 / length
}
impl PartialEq for Task {
    fn eq(&self, other: &Self) -> bool {
        match (self, other) {
            (&Task::ProcessWord(ref word_1), &Task::ProcessWord(ref word_2)) => {
                word_1 == word_2
            },
            (&Task::Halt, &Task::Halt) => true,
            _ => false
        }
    }
}
impl Eq for Task {

}
impl PartialOrd for Task {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        match (self, other) {
            (&Task::ProcessWord(ref w_1), &Task::ProcessWord(ref w_2)) => w_1.partial_cmp(&w_2),
            (&Task::Halt, &Task::Halt) => Some(Ordering::Equal),
            (&Task::Halt, _) => Some(Ordering::Greater),
            (_, &Task::Halt) => Some(Ordering::Less),
            //_ => panic!()
        }
    }
}
```

```rust
impl Ord for Task {
    fn cmp(&self, other: &Self) -> Ordering {
        match (self, other) {
            (&Task::ProcessWord(ref w_1), &Task::ProcessWord(ref w_2)) => w_1.cmp(&w_2),
            (&Task::Halt, &Task::Halt) => Ordering::Equal,
            (&Task::Halt, _) => Ordering::Greater,
            (_, &Task::Halt) => Ordering::Less,
            //_ => panic!()
        }

    }
}

#[derive(Debug)]
pub struct Generator {
    task_queue: Arc<Mutex<BinaryHeap<Task>>>,
    max_len: Arc<atomic::AtomicUsize>,
    thread_number: usize,
    template_vecs: Vec<[isize; cfg::CARDINALITY]>,
}

impl Generator {
    pub fn new(strings: &[String], backtrack_input: bool, threads: usize, templates: Vec<[isize; cfg::CARDINALITY]>) -> Generator {
        let max_len: Arc<atomic::AtomicUsize> = Arc::new(atomic::ATOMIC_USIZE_INIT);
        let queue: Arc<Mutex<BinaryHeap<Task>>> = Arc::new(Mutex::new(BinaryHeap::with_capacity(cfg::HEAP_CAPACITY)));
        {
            let mut queue_lock = queue.lock().expect(&*format!("Failed at line {}", line!()));
            for s in strings {
                let word = string_to_word(s);
                if backtrack_input {
                    let bt_words = generate_backtrack(&word);
                    for bt_word in bt_words.into_iter() {
                        queue_lock.push(Task::ProcessWord(WordWrapper::new(bt_word)));
                    }
                } else {
                    queue_lock.push(Task::ProcessWord(WordWrapper::new(word)));
                }
            }
        }
        Generator{
            max_len: max_len,
            task_queue: queue,
            thread_number: threads,
            template_vecs: templates
        }
    }

    pub fn generate(&mut self) -> String {
        self.generate_until(|_: &WordWrapper| false)
    }

    pub fn generate_until<F>(&mut self, should_halt: F) -> String
    where F: 'static + Send + Sync + Fn(&WordWrapper) -> bool {
        let mut threads = Vec::with_capacity(self.thread_number);
        let thread_number = self.thread_number;
        let arced_should_halt = Arc::new(should_halt);
        for _ in 0..thread_number {
            let mut missed_pops = 0i32;
            let queue = self.task_queue.clone();
            let max_len = self.max_len.clone();
            let should_halt = arced_should_halt.clone();
            let local_templates = self.template_vecs.clone();
            let new_thread = thread::spawn(move || {
                loop {
                    let current_word: WordWrapper;
                    {
                        let mut queue_lock = queue.lock().expect(&*format!("Failed at line {}", line!()));
```

```
                        match queue_lock.pop() {
                            Some(task) => {
                                missed_pops = 0;
                                match task {
                                    Task::ProcessWord(word) => current_word = word,
                                    Task::Halt => return None,
                                }
                            }
                            None => {
                                //println!("miss {}", missed_pops);
                                if missed_pops > 100_000 {
                                    return None;
                                } else {
                                    missed_pops += 1;
                                }
                                //thread::sleep_ms(5);
                                continue;
                            }
                        }
                    }
                    let children: Vec<WordWrapper> = current_word.spawn_children()
                        .into_iter()
                        .filter(|wrd| is_template_vector_free_partial_check(wrd, &local_templates))
                        .collect();
                    let mut queue_lock = queue.lock().expect(&*format!("Failed at line {}", line!()));
                    for child in children.iter() {
                        if should_halt(child) {
                            for _ in 1..thread_number {
                                queue_lock.push(Task::Halt);
                            }
                            return Some(word_to_string(child));
                        }
                    }
                    for child in children.into_iter() {
                        queue_lock.push(Task::ProcessWord(child));
                    }
                }
            });
            threads.push(new_thread);
        }
        let mut result: Option<String> = None;
        for thread in threads {
            if let Some(res) = thread.join().expect("Could not join thread") {
                result = Some(res);
            }
        }
        result.unwrap_or("".to_owned())
    }
}

fn calculate_parikh_chain(word: &Word) -> Vec<[usize; cfg::CARDINALITY]> {
    let mut result = Vec::with_capacity(word.len() + 1);
    result.push([0; cfg::CARDINALITY]);
    for i in 0..word.len() {
        let mut new_elem = result[i].clone();
        new_elem[word[i] as usize] += 1;
        result.push(new_elem);
    }
    result
}

pub fn calculate_parikh_vector(word: &Word) -> [usize; cfg::CARDINALITY] {
    let mut result = [0; cfg::CARDINALITY];
    for letter in word.iter() {
        result[*letter as usize] += 1;
    }
    result
}

fn parikh_diff(p1: &[usize; cfg::CARDINALITY], p2: &[usize; cfg::CARDINALITY]) -> [isize; cfg::CARDINALITY] {
```

```
    let mut result = [0; cfg::CARDINALITY];
    for i in 0..p1.len() {
        result[i] = (p2[i] - p1[i]) as isize;
    }
    result
}

pub fn is_aa2f_partial_check(word: &Word) -> bool {
    if word.len() < 4 {
        return true;
    }
    let l = word.len();
    let parikh_chain = calculate_parikh_chain(word);
    for i in 2..(l / 2 + 1) {
        let diff2 = parikh_diff(&parikh_chain[l - i], &parikh_chain[l]);
        let diff1 = parikh_diff(&parikh_chain[l - 2 * i], &parikh_chain[l
 ↪  - i]);
        if diff1 == diff2 {
            return false;
        }
    }
    true
}

pub fn is_aa2f_full_check(word: &Word) -> bool {
    let l = word.len();
    for part_len in 1..(l + 1) {
        for window in word.windows(part_len) {
            if !is_aa2f_partial_check(&window.to_vec()) {
                return false;
            }
        }
    }
    true
}

pub fn is_template_vector_free_full_check(word: &Word, templates:
 ↪  &Vec<[isize; cfg::CARDINALITY]>) -> bool {
    let l = word.len();
    for part_len in 1..(l + 1) {
        for window in word.windows(part_len) {
            if !is_template_vector_free_partial_check(&window.to_vec(),
 ↪  templates) {
                return false;
            }
        }
    }
    true
}

pub fn is_template_vector_free_partial_check(word: &Word, templates:
 ↪  &Vec<[isize; cfg::CARDINALITY]>) -> bool {
    let l = word.len() as isize;
    let parikh_chain = calculate_parikh_chain(word);
    for templ_vec in templates.iter() {
        let norm = templ_vec.iter()
           .fold(0, |acc, &x| acc + x);
        let mut i: isize = 1;
        while (i * 2) as isize + norm <= l as isize {
            let diff1 = parikh_diff(&parikh_chain[(l - 2 * i - norm) as
 ↪  usize], &parikh_chain[(l - i) as usize]);
            let diff2 = parikh_diff(&parikh_chain[(l - i) as usize],
 ↪  &parikh_chain[l as usize]);
            //println!("Diffs are {:?} {:?}", diff1, diff2);
            let mut are_eq = true;
            for j in 0..cfg::CARDINALITY {
                if diff1[j] - diff2[j] != templ_vec[j] {
                    are_eq = false;
                }
            }
            if are_eq {
                return false;
            }
            i += 1;
        }
```

```
    }
    true
}

pub fn word_to_string(word: &Word) -> String {
    let utf: Vec<u8> = word.iter().map(|&x| x + 97).collect();
    String::from_utf8(utf).expect(&*format!("Failed at line {}", line!()))
}

pub fn string_to_word(in_str: &str) -> Word {
    in_str.as_bytes().iter().map(|x| x - 97).collect()
}

pub fn generate_backtrack(word: &Word) -> Vec<Word> {
    let mut result = Vec::with_capacity(word.len() - 1);
    for i in (1..word.len()).rev() {
        result.push(word[0..i].to_vec());
    }
    result
}
```

6. Contents of the src/cfg.rs File (Classifier)

```
pub const CARDINALITY: usize = 3;
pub const HEAP_CAPACITY: usize = 1_000_000;
```

7. The Thread Communication Methods Test

```
extern crate time;

use time::*;
use std::sync::{Arc, Mutex};
use std::sync::mpsc::*;
use std::thread::*;
use std::cmp::min;

const NUM_COUNT:i64 = 5_000_000i64;
const SPAWNDED_THREAD_COUNT:usize = 4;
const BULK_SIZE:usize = 10;

fn collatz_length(mut num:i64) -> i64 {
    if num <= 0 {
        return 0;
    }
    let mut result = 1i64;
    while num != 1 {
        result += 1;
        num = match num % 2 {
            0 => num / 2,
            1 => 3 * num + 1,
            _ => 1,
        }
    }
    result
}

fn test_serial() {
    let numbers_to_process:Vec<i64> = (1..NUM_COUNT+1).collect();
    let mut sum = 0i64;
    let start: u64 = precise_time_ns();
    for num in numbers_to_process.iter() {
        sum += collatz_length(*num);
    }
```

```
    let end: u64 = precise_time_ns();
    println!("Sum is: {:?}", sum); //to try to avoid compiler optimizing it away
    println!("Took {} ns to compute in serial manner", end - start);
}

fn test_shared_memory() {
    let numbers_to_process:Arc<Mutex<Vec<i64>>> = Arc::new(Mutex::new((1..NUM_COUNT+1).collect()));
    let mut sum = 0i64;
    let start: u64 = precise_time_ns();
    let mut threads: Vec<JoinHandle<_>> = Vec::with_capacity(SPAWNDED_THREAD_COUNT);
    for _ in 0..SPAWNDED_THREAD_COUNT {
        let nums = numbers_to_process.clone();
        let handle = spawn(move || {
            let mut local_sum = 0i64;
            let mut task_queue:Vec<i64> = Vec::with_capacity(BULK_SIZE);
            loop {
                if task_queue.len() > 0 {
                    let num = task_queue.pop().expect("Pop result shouldn't be empty at {}");
                    local_sum += collatz_length(num);
                } else {
                    let mut lock = nums.lock().expect("Mutex was poisoned");

                    if lock.len() == 0 {
                        return local_sum;
                    } else {
                        for _ in 0..min(lock.len(), BULK_SIZE) {
                            task_queue.push(lock.pop().unwrap());
                        }
                    }
                }
            }
        });
        threads.push(handle);
    }

    for join_handle in threads {
        sum += join_handle.join().unwrap();
    }
    let end: u64 = precise_time_ns();
    println!("Sum is: {:?}", sum);
    println!("Took {} ns to compute in parallel using shared memory with bulk size of {}", end - start, BULK_SIZE);
}

fn test_channels() {
    let mut numbers_to_process:Vec<i64> = (1..NUM_COUNT+1).collect();
    let mut sum = 0i64;
    let mut threads: Vec<JoinHandle<_>> = Vec::with_capacity(SPAWNDED_THREAD_COUNT);
    let start: u64 = precise_time_ns();
    let (main_tx, main_rx): (Sender<(Sender<Option<Vec<i64>>>, i64)>, Receiver<(Sender<Option<Vec<i64>>>, i64)>) = channel();
    for _ in 0..SPAWNDED_THREAD_COUNT {
        let tx = main_tx.clone();
        let handle = spawn(move || {
            let (local_tx, local_rx): (Sender<Option<Vec<i64>>>, Receiver<Option<Vec<i64>>>) = channel();
            tx.send( (local_tx.clone(), 0) );

            while let Ok(new_tasks_opt) = local_rx.recv() {
                if let Some(new_tasks) = new_tasks_opt {
                    let mut local_sum = 0i64;
                    for i in new_tasks.iter() {
                        local_sum += collatz_length(*i);
                    }
                    tx.send( (local_tx.clone(), local_sum) );
                } else {
                    return;
                }
```

```
            }
        });

        threads.push(handle);
    }
    drop(main_tx);
    while let Ok( (tx, collatz_result) ) = main_rx.recv() {
        sum += collatz_result;
        if numbers_to_process.len() == 0 {
            tx.send(None);
        } else {
            let mut data_bulk: Vec<i64> = Vec::with_capacity(BULK_SIZE);
            for _ in 0..min(BULK_SIZE, numbers_to_process.len()) {
                data_bulk.push(numbers_to_process.pop().unwrap());
            }
            tx.send(Some(data_bulk));
        }
    }

    for join_handle in threads {
        join_handle.join();
    }

    let end: u64 = precise_time_ns();
    println!("Sum is: {:?}", sum);
    println!("Took {} ns to compute in parallel using channels with bulk
↪  size of {}", end - start, BULK_SIZE);
}

fn main() {
    test_serial();
    test_shared_memory();
    test_channels();
}
```

8. The Results of Thread Communication Testing in CSV Format

Task Size,Threads Spawned,Bulk Size,Serial,Shared Memory,Channels

5 000 000, 4,1,3424578206 ns,1778776155 ns,5224623987 ns

5 000 000, 4,1,3414758847 ns,1772402450 ns,5220775848 ns

5 000 000, 4,1,3433412774 ns,1774028450 ns,5226536009 ns,

5 000 000, 4,2,3441499936 ns,1452454907 ns,3634465568 ns

5 000 000, 4,2,3429163043 ns,1382693992 ns,3447613498 ns

5 000 000, 4,2,3419446758 ns,1384678747 ns,3423935213 ns

5 000 000, 4,3,3422137868 ns,1299928651 ns,2759338231 ns

5 000 000, 4,3,3416833290 ns,1297326338 ns,2756429815 ns

5 000 000, 4,3,3420646624 ns,1293847216 ns,2798645566 ns

5 000 000, 4,4,3420009433 ns,1262555202 ns,2435675965 ns

5 000 000, 4,4,3430368711 ns,1259968506 ns,2425848128 ns

5 000 000, 4,4,3429328588 ns,1267493439 ns,2443529312 ns

5 000 000, 4,5,3467159851 ns,1312614481 ns,2314144099 ns

5 000 000, 4,5,3424305570 ns,1248111293 ns,2250099617 ns

5 000 000, 4,5,3434055320 ns,1245624102 ns,2252204849 ns

5 000 000, 4,10,3423886577 ns,1223353174 ns,1924358000 ns

5 000 000, 4,10,3437502314 ns,1220080203 ns,1926680092 ns

5 000 000, 4,10,3421918331 ns,1220700884 ns,1928178921 ns

5 000 000, 4,50,3428462489 ns,1211860069 ns,1568163250 ns

5 000 000, 4,50,3411500601 ns,1208260024 ns,1569898571 ns

5 000 000, 4,50,3434107974 ns,1206328369 ns,1571151091 ns

5 000 000, 4,100,3421820164 ns,1213313385 ns,1438375553 ns

5 000 000, 4,100,3456933992 ns,1216621160 ns,1438201530 ns

5 000 000, 4,100,3433728247 ns,1206304273 ns,1439515627 ns

5 000 000, 4,1000,3570326043 ns,1314629578 ns,1264423942 ns

5 000 000, 4,1000,3557772737 ns,1309710082 ns,1264052692 ns

5 000 000, 4,1000,3551442316 ns,1309506163 ns,1259236717 ns

5 000 000, 4,10000,3542614888 ns,1304465296 ns,1242568436 ns

5 000 000, 4,10000,3541726032 ns,1307653933 ns,1238440522 ns

5 000 000, 4,10000,3540885366 ns,1310332103 ns,1240470344 ns

9. Almost Abelian Square Free Word of Length 15796

abaccbbbcccbaacccaaaccbbaaabbbcbbabbcccacbcccaaaccbbbacccbbaaacaaabaaaccbba
bbbcbbaaccbcaaabbcccbbbccabbcccacccbcaaabaacccbbbaacbaaabbcbbbaaccbcccaccbb
aaabbbcbbbabbbccaaabbbaaacaaabaaacccbcaabaaacaabcbbabbcccaaacbaacccbcccaccc
bbbacccaabbcabbbaaacabcacccbbaaabccaaacccabccaaabaaacaaabbbaaacccbaaabbbaac
cbbbcccabccaabaaacaabbbaacccaaabbcaabbbcbbabbcccbbbccaaabbbaacaabaaaccbabb
cccbaacccaaabaaacaaabbbaacccbaaabbbaccbbbcccbaccbbbaacaabaaacccbccabbcccaccc
bcccaaacbbbabbcccbbbccaabcbbbaaabbcccabccaaacccabbcccbbbacaabaaccbbbaacbbaa
abbbcbbbabbbccacbabbbcbbbaaacaabbccaaacccaabaaacaabbccaaabbbaccbcccaabccaa
abaacbbabbbcccbbaaabcacccbcccaabbbcccbbbabbbcbacaabbbcccaaacbbaaabbbaccbbb

cccabccaaabaacbbbcacbcccaaaccbbacbcccacccbbaccbbbcccaaabacbabbbcccbaaabbbcbb
babbbccabbcccbbbaacbcccacbbbcccbbaaacaaabaaacccbbbabbbcbbbaaabbcaabbbcbbba
bbbcccbacccaabbbcbbbabbbccaaacbbaaabbbaacbbaaacaaabacccaaabbccacccbbabbcba
acaaabaacccbbaaabbbcbbbabbbccabbcccacbcccaaaccbabbbcccbbbaaabbbcbbbabbbcca
bbcbbbaaacccaaabaaacaaabbbcabbcccaaaccbcccaabbbaaacccaaabaaacaaabbbacbbaaac
aaabaaacccbccaaabbbcccbbabbcbbbaaacabacccbbbaaacaaabaaacccabcccbbaacbacccaa
acbbaaabbbaaccbbabcccbaacccabcccbbbabbbcbbbaaacabacccbbabcbbaaacaaabacccab
bcccbacaaabbbcbaacccabbcccbbbccabbcccacccbcccaaacccbbbccacccbccaaabaaacaaabb
cbaacccbabbcbbaacaaabbbcccaaacccbcacccbbbcccaabbbaaabcbbbabbbccaaabbbaaacaa
baacccabbbcbbaaacaabaacccaabbbcccbabbbcbaaacbccabbcccacccbcccaaabaaacabbbaa
acccaaabaaacaaabbcaabbbcaccbaacaaabaacbbabbbcbbaacabbcccbacccaaabaaacaaabbc
aabbbaaabbcccaaabaccbbaaabbbcbbbabbbcccbbaaccbcaaaccbbaaabbbacbbaaacaaabaa
acccbbaccbbbcccacccbcccaaabbccacccbabbbcbacaaabbbacbbbccaaacccabbcccbbbccaaa
bbbcccbabbbcbbbaaabbbcccbbaaabbbaacaabaaaccbacccbbbacaabbbcbbbabbbccacbbb
aaacaabbccacccbbbaaacccaaabaaacaaabbcaabbbcccaaabaaacaaabbbcbbbabbbcccacccb
cccaabaccbbbcccbbabbbcaaaccbbbabbbcbbbaacaaabaacccaaaccbbbaaacaabcccaccbbab
bcbaacccaaabbcaabbbaaacaaabaaaccbaacccaaaccbaaabbcaaacccbaccbbabbcbaaaccbcca
cccbbbcccaaacccbcccacccbbaccbbbcccbbacccaabbbcccbbbabbbcbbbaaacabccaaabaaaca
aabbbaacccaaaccbaacccbbbccabbbaacccaaaccbaacccbcccacccbbbccaaacccbcccacbbbccc
bbaacccaabaacaaabbcccaabbbcbbbabcccbbaacccaaaccbccacccbbbccabbbaacabbcccbbb
abbbcbbbaaabbcaabbbccaabcaaacccbbbccaaabbbaacbbaaacaaabaaacccbcccacccbbabb
cbacaaabbcaabbbcbabbbcccbbaaacccbccabbcccacccbcccaaabaaacabbbaccbcccaccbbaac
babbbcbacbcccaabbbaaacaabbcbbbaaacaaabaaaccbaacccbcabaaacaaabbbacbbaacccbb
bccacccbccaaacccbbaaabbbcbbbabbbccaaabbcaabbbaaacaaabacccabbbaaacabcbbbaaa
bbbcccbbaccbbbabbbcbbbaaabbbcccbbaaabbbacbbbcccabaacccaabbbcccbbacaaabbbc
bbaaacaaabaaacccabbcbbbaaacaabbbcccaaaccbcccaccbbbcccaaaccbbabbcbbbaacccaaa
baaacaaabbbcbbaacccbaaabbbcbbbabbbcccbbbaaacccbcccacbbabbbcccbbaaabbbcbbba
bbbccacccbbbabbbcbbbaaabbbcccbbaaacaaabaccbbbcccbbaaacccbcccacccbbabbcbbba
aabcccaccbabbccaccbcccaabaaacbbbccaaacbaaabbbaacbbbcacbcaaacccabcccbbbcabbb
aaabbcccbaccbbbabbbcbbbaaabcaaaccbbbaacbbabbbcccbacccaaabbcbbbaaabbcccaaab
aaacaaabbcbbabbbcccaaabbbaacaaabbcccaaaccbaacccbcccacbbbaaacabaaaccbacaabbb

cbabbbcccabbbaaaccbacccaabbbcccbbbabbbcbbbaaabbcaabbbccaaabaaacabbbcccaaac
bbaaabbbaacbccaaabbbaaacaaabaaaccbaacccaaabaaacaaabbbcccacccbcccaabbbaaabcc
caaabaaacaaabbbaaacccbbbaacbbaaabbbcbbbabbbccabbcccbbbaacccbaaabbbacbbbcca
bbcccacccbcccaaabaaacaaabbbaacbbaaacccabcccbbbabbbcbbbaaabcccabbcbbbabbccab
cccbbbabccaaacccbcccacccbbacbcccaaaccbaaabbbaaccbcccabbcbbbabbcacccbcccaaacb
bbcacbcccaaabaaacaaabbbaaacccabbbcbbacccaaaccbbbabbbcbbbaaabbcaabbbcbbbabb
bcccbbbaaabbbcbabbbcccbaaacbbbcccabcbbbabbbcccaaacbbbacaaabacccbabbbcbbbaa
acbcccacccbbbcccaaabbbcccbbbabbbcbaaaccbcccaccbbbcccaaacccbcccacccbbaccbbbccc
acccbcccaaabaaacaaabbcaabbbaaabbcaaaccbbbabbbcbaaabbcccaaaccbaaabbbcbbabbc
cacccbbbabbbcbbbaacccaaabbbaacccbcccacccbbbabbcaaaccbacbbbaacaaabaacbcccaaa
cbabbcacccbbbabbbcbbbaacbbaaacaabaacccaaabbbaacbbbcccbbaacccaaabaacbbaaaca
aabaaacccaabcccbbaacccaaabacbabbbccaccbcabaaacaaabbbaacbbbccaaabbbaaccbccca
abbbaccbbbcccbbaacccbccaaacccbbabcbbaaacccaaabacbbaaacaaabaaacccaabcccbabcb
aacaaabaacccbcacccbbbcccaabbbaccbbbcccacccbcccaabcacccbbbaacccaaabaaacaaabbc
ccaaacccbccaabbcccbbbcabbbaaabbcccabbbccaaacccbcccacccbbacbbbcccbbacccaaaccb
bbabbcccbbbccaabbbcbbaaacabaaacccabcccbbbaaccbbabbbcccbaaacccbbbccaccbcccaa
abaaacaaabbcbbbaacaaabbcccaaabaaacabbbcbbacaabbcccbaccbbbabcbbbaaabbccaaab
cbabccaaacccabbcccbbbccabbbaaacccabccaabbbcbbbabbbccaaabbcccbbbcacccbcccaab
bcccbbbccabbbaaacabacccbbbabccaaacccbcccacccbbbcabbcccacccbcccaaabaaacabcbba
aabbcccaaabbbaaccbbbcccbacccaabbcccbbbabbbcbbbaacccbbaaabbbcbbaaacaabaaccc
abbcccbbbacaaabbbacbbbccabbcccacbcccaaabaaccbccacbbbcaabbbaaacaaabaaaccbaac
aaabbbaaccbbbcaabbbaaabbcaabbbcbbbabbbcccbbaacbaaabbbcbbacaaabaaaccbcabbb
aacbcacccbbbabbbcbaaacaabbbcbbbabbbcccbbbaaacbbbcccbbaacccaaaccbbbabbbcbbb
aacccbccaaabaaacaaabbbacbbaaccbbabbbcbbacccbccaaabbbcaccbaacccbcacccbbbcccaa
abbbaccbbbcccbacccaaabaaacaaabbbaaccbbbcccbabbbcbbbaacaaabaacccaaabbbaaaca
aabaaaccbaacccaaabaaacaaabbbcacccbcccaaacccbbabccaabbbaaacbacccbbaaacccaaaba
aacaaabbcabaacccbbacaaabbbaacbbbcccbbacccbaacaaabaacccbbbcccacccbcccaaacccbb
babbcaabbbcbbbabbbcccabbbaaacaabbccaaabbbcabbcccbacaaabaaacccbccaccbbabbbc
ccbaccbbaaacccbcccacccbbbccabbcccaabbbcbbacccbccaaabaacaabbbacbbbccabbbaaacc
cabcccbbabccacccbccaabaaacaabccacccbbaaabbbacaaabaaaccbbabbbcbbaaaccbaacaaa
bbbaccbcccaaabbcbaaccbbbcccbbabbbcbbaacbaaabbbcbaaccbbacbbbcccacccbcccaaabb

cccaccbbabbbcaaabaaccbbbabbbcbbbaacccbccaaabbbaaacaabaaccbabbbcbaacccbcccac
bbbcaaaccbaacccbcccacccbbbcccaaabbcccbbbabbbcbbbaacccbbbcccacccbcccaaabaaccb
baaabcaaaccbaacccbcccacccbbbcccaaacbcccacccbbbabbbcbacaabaacccaaabbcccaccbbb
ccaaabaacccabccaaabaaacaaabbbaaccbabbcaaabaacccbcacbbaaabcccabbcbbbabbcaaba
aacaabcbbaaabbcccaaacccbcacccbbaccbbbabcabaaacabccaabbbcaccbaacccbcccacccbbb
ccabbcccaabbbcbbacccaaabaccbbbcccbbaaabbbcbbabbcccbbbccaaabaaacabbbaaabbcc
aabaaacccabbbcccaccbbaaacbaccbbaaabbbcbbaaccbacccaabbbcbabbbcccacccbcccaabc
abbcccbacccaaabaaacabbbaaabbccaabacccabbcccbbbccabbbacaabccaaabbbaaacaaabaa
acccaaabbccacccbbbaaacaaabaaacccaabccaaabbcccacccbcccaabbccabaaacccaaabbbcccb
baaacccaabaacaaabbcabbbaaaccbcaabbbccabbcccacbcccaaaccbaaabbcaaaccbcccaabbbc
ccbbbabbbcbacaaabbbcbbaacccbccaaabbbacaaabaaacccbcccacbbbcccaaacccbcccacccbb
accbbbaacabbccacccbccaaacbaaabbcaabbbcccaccbbaaabbcbbbabbccabcccbbbccaaacba
aabbbacabaccbbbcccbbabbbcbbaaabbbaacccbbaaabbbcbbaaacaaabaaaccbcaaabbbccab
cccbbabbbcbbaacbaccbbbcccbacccaaabcaccbbaacbabbbcbaaccbbaaabcccaaabaaacaaab
bbaaacccaabbbaaabbcbbabbbcccbbaaacbaacccaaacbaaabbbaacbbbcccaaabbbcbabbbcc
cbaccbbaacbccaaabacabbbaccbbbcccacbcccaaabbbcccaccbaacaaabaacbccaabbbaaabbc
aaacccbbbcccacbbabbbcbbaacabbcbaacccaaabaaacabbbaaabbccabcbbaaacccaabccaaab
aaacaaabbbaacccaaacbbaaabbbacbbbccabaaacccaabbcabbbaaacaaabacccaaabbcccaccc
bcaaacccaabbbaaacaaabaaaccbaacccaaabbbcaaacccbbabbbcaabaaacccaaabbcaabbbaaa
caaabaaaccbaacccaaabbcabaaacaaabbcabbbaaabbcccbabbbcbbbaacbbaaabcaaacccbab
bbcccbaaacccaabbcabaaacaaabbcaabbbaaabbcaccbaaabbbcbbaacccbccaaacbaacccbcca
ccbbbcabbccaaacbbaaabbbaaccbbacbbbcccacbbbaaacccaaabaaacaaabbcaabbbcbbacaa
bbcccaabcaaabbbaaccbbbcaabbbaaabbcaabbbcbbbabbbcccaabbbaaabbcbbbaacccabaa
ccbbbcccbbaaccbacccaaabaaacabcbbaaabbcccaaabaacaabbbcbbbabbbccacbaaabbcaccc
bbbabbcccacccbcccaaaccbaacccbbaaabbbaacaabaaaccbbaaabbbaacbbbccabaacbbbcccb
acccaabaacaaabbcabaaacccbbbaaabbbcbbbabbbccabbcccacccbcccaaabbbccacbbbabbcc
aaabbbacccaabccaaabaaacabcccbbaaabccaaacccbccaccbbbcccbbaaabccaaacccaabccaaa
baaacaaabbbaaacccaaabacaaabbbccacbbaacccaaaccbccacccbbaaccbabbcbbaaabbbaacc
babbbcccbbaacbaaaccbcabaaacaaabbbaaaccbaacccbbbaaabbcaabbbcbbbabbbcccbbbaa
abbbcbaaccbcaccbbbcaabaaacaabcbbaacccbcccacccbbaaacaaabaaaccbbabcacccbcabaac
cbbbaaabbcaabbbcbbbabcccbaaaccbccacccbbbcccaaabaaacabbbaaabbcccaccbbbcccaaa

cccbcccacccbbbcabbbaaacccbcccacccbbbccaabbcbbbabbcccbbbccaaacccaabbbcbbaaab
bbaacbcacccbbbaaabbbcbbbabbbccabcccbbbabbbcbbbaacaaabccacccbbbccaabcbabbbc
ccbbbaacccbcccacbaaccbbaaacbacccbbbacbbaaabcaaaccbaaabcbbbabbbcccbbbaaabbbc
baaacaabbbcccacccbcaaacccaabbbcbbaacaaabaacbbbabbccaaabbbacccbaacccaaacbbba
aacaaabaaacccaaabbbaaacabbbcbbaaacaaabacccaabbcccaccbbbacaabbcccaaacbcccaccc
bbacbbbcccbbacccaaacbbbacaaabacbccaaabbbcbbabbcccbbbccaaaccbccacccbbabcbbaa
acaaabaaacccaaabbcaabbbaccbbbcaabbbaaabcaaaccbcccaabbcccbabbbcbbbaaabbcccaa
bbbcccbbbabbbcbbbaacbbabbbcccaabbbaaabcaaacccbbaaabbbaaccbbbcccbbaacccaaac
baacccbbaaabbbacaaabaaaccbbbabbbcbbbaaaccbacccbbbabbbcbbbaaabbccaaacbbaaa
bbbaacbbaaacaaabaaacccaaabbbaaacabaaacccabbbcacccbcaabaaacccbbbaaabbbcbabb
bccabaaacccaabbcacccbbbcabbccacccbccaaacccbbaaacaaabaaacccaabbbcbabccaaaccca
abcbabcccbbbcaabbbaaabbccabaaacaaabbcccbbbaaacaabccaaabaaacaaabbbaaacccaab
bbaaabbcaabbbcbbbabbbcccacbbaaabbbaaccbacccaabbbaaacaaabaaaccbaacccbabbbcc
cbaacccaaabacaaabbbaaaccbbbabbbcbaccbbbaaacabcccbbaccbbbabbbcbbbaaabbbccca
aabbbaacbbaaacaaabacccabbcbacaaabbbcbabbbcccacbabbcbbaaabccaaacccbcccacccbb
baaacccbabcbbbaacbccaaacbaaabbcaaaccbaacccbcccacccbbbcccaaaccbccacccbbbabbbc
bbbaaaccbaacaaabbbaccbbbcccaccbbaaccbacccaaabaaacaaabbbacccbbbccaabbbbaaabbc
ccaaabbbaaacaaabaaacccaaabbbaacccaaaccbcccabbbaacccbbbcccacccbcccaaaccbaacccb
bacbaacccaaacbaaabbbcaabacccbbbccabbbaaacabbbcccacccbcccaabccaaacccbcccacccb
babcacccbcccaaabaacbbbaccbbbcccbaccbbbaaabbcaccbbbabbcbbaaacabaaacccaabbbaa
acabbcccbbbabbcccaaabaccbbbcccaccbbaacccaaabaaacaaabbbaacbccaaabaacbbabbbcc
caaabbcbbabbbccabbcccbacccaaabaacaabbbaaabbccabcccbbaaabbbcbbbabccacccbbbc
caaabbcccbbabccacccbccaaaccbbacbaaaccbbabbbcbbaaabbcccacbcccaaabaaacabbbaaa
bbccaaacccbbbcccacccbcccaaabaacccbcccacccbbbcaabbbaccbbbcccacbcccaaabaccbbba
bbcbbaaabcaaacccbcaaabbbcccacccbcabbbcccbacaabbbcbbbabbbcccbbbaaabbbcbaaac
aabbcabaaacccaabcccbbaaabbbcbabbbccabbcccabcaaabbbcbaacccbcccacccbbbcccaabb
bcbbabbcccbbbccaabcaaaccbbabbbcccbbbaaabbbcbbbabbbccabcccbbbabbbcbbbaaacc
baacccbccaccbbbaacbcccaaabaaacabcbbbabbbcccabbbaaabbcaaacccbbaacaaabbcbbbaa
acaaabacbbbcccaabaaacaabcccaccbbacbccaaabbbcaccbaacaaabbbaacbbbccaabcabbbcc
cbbacccaaabbbcaaacccbcccacccbbaccbbbaaabbccaccbcccaaabaaacabcbbaaacaaabaaacc
caabccaaabaaacaaabbbcccacccbcabbbaaabcaaaccbbacaaabbbcbbabbccaccbcccaaabaaa

caaabbbaacbccaaacbbaaabbbaccbbbcccacbbabbbcbbaaacccbbbcaabbbaaacaaabaaaccc
bcccacbabbbcccacccbcccaabccaaacccbcccacccbbbcccaabcbbaaabbcccacccbcccaabaaacc
cbccaabbbaaabcacccbbaaabbbcbbbabbbccabcccbbabbbccaabcacccbcccaabbbcccbbbab
cbbbaacabaacccbbbccabaaacccbcccacccbbaccbbbcccbbacccaabbcbbbabbcccbbaaacccaa
bcbbaaacaaabaaaccbccacccbbbcccaabccaaabbcbaccbbabbcbbbaaacaaabaaacccbbbaaac
abcccbbabbbcbbacaaabbbcbbbabbbccacccbbbccaaacccbccaaabaaccbcaabbbaaabbcbbb
abbcccbbbccaabcbbbaaabbccacccbbbcabbcccacccbcccaaacccbbbacccaaabbcbbbacaaab
acccaaabbccacccbbbcaaabbcccbbbabbbcbbbaaabbcaabbbcccbbbabcccaabbbaaacabcca
aabaaacaaabbbcbaacccaaaccbcccabbcbbbacaaabaaacccbbbaaabbcaabbbcccaccbaaabbb
cbbaacccbbbcccacccbcaaacccbbacaaabacccaabbbcccbacaabaaccbbbabbcaaabaacccbbba
caaabaaaccbaacccaaabbcccacccbcaaacccaabbbaccbbbcccacccbcccaabccaaacccaabcccbb
acccaaacbaaabbcbbbabbcccbbbaacccabbcccbbbccabbcccacccbcccaaacccbbabcbbaaccca
aaccbaacccbcccacccbbbacaabbbcbacbcccaabaaaccbbbcabbbaaacaaabaaaccbcaccbbbaa
acaabbcccaaacccbcccacccbbacbbbccaaacbcccacccbbaaacaabbcbbbaacccabcccbbbabbbc
bbbaaabcccbbbabbcaaabaaccbbaaacccbbacbbbcccbbacccaaaccbaaabbbcccaaabbbacaaa
baaaccbaacabbbcccbaacccaaacbaacccbcccacccbbbcccaaabbbcccbaacccaaabaaacaaabbb
cbaacaaabbbcccabcaaacccabcccbbaccbbbabbbcbaaacaabcccbbabccaaabbbaaacaaabaaa
cccabccaaabaaacaaabbbcccbbbabccacccbccaabaaaccbcaabbbaaacaaabaaaccbaacccaaac
cbaaabbccaaacccbcccacccbbacbcccaaabbbcccbbabbcbacaaabaccbcccaccbaacaaabaacbc
caccbbabccaaabcaabbbcbbbabbbcccbacccaabbcccaccbbbcccbbacaaabaaacccaabcccbba
bccaaacccbcccacccbbaccbbbcccbbacccaabcccbbbcabaaacccbbaacaaabaacccaaaccbabbc
aabaaacccbcccacccbbbccabbcccacccbcccaaacccbbbcccaccbccaaabaaccbcaccbbaaabbbcb
aacccabbcccbbbccabbcccacccbcccaaacccbbbcccaccbccaaabbbaaccbbbcccacccbcccaabbb
cbbbabbbcccbbaacccabcccbbbccaabaaacccbbbcaabbbacbbbcccbbaaacabaaacccaabcccb
bbabbccaabbbcccacccbcccaabccaaacbbbccaccbcccaaabaaacaaabbcccbbbccaabbbaacabc
caabaacaaabbbcbbbabbbcccbbbaaccbcccaccbbbccaabcbbbaaacccabccaabaaacbccaaabb
bacbbaaacaaabaaacccbccaaabaacaabbbccacccbccabbbaacbbaaacabaccbbbabcbbbaaab
bbcccabbbaaacaaabaaacccaaabbcccabaacaabbbaaabbcccbbbabbbcbbbaacbabbbcccacc
cbcccaaacccbbacaabbbcccbbbabcbbbaaacbbbcccaaabaaacaaabbbcbbbabbbccabbcbbba
aaccbcabbcccacccbcccaaacccbbaaabbbacbbbcccaccbccaabcbbbaaaccbacaaabaaaccbacc
caabbcccaccbbbcccbbaaacccaabccaaabbcbbabbbcccbbbaacbbaaabcccaaabaacbbaaacaa

abaaacccaaabbcccacccbcccaaaccbbacabaaacccaaabbcccbbbccaabbbaaacccbcccacccbbb
ccabbcccacccbcccaaacccbbbcccacbbaaabccaaacccbcaabbcccaaabaaacaaabbcaabbbaaab
bcaaacccbbbcaaabcbbbabbbcccbaccbbaacccabbcccbbbcabbcccacccbcccaaacccbbbcccac
bbaaabbbaaccbcabaaacaaabbbcbbbabbbcccabaacaabbbaacccabbcccbbbabbbcbbbaacb
babbbcccbbbaaabcccabaaacaaabbcbabbcccbbaacccbccaaacccbbbabccaaabaaacaaabbba
aacccbbbccabaaacccbbbaaabbbcbbabbcccbbbaacccabcccbbbabbccaabaaacccbccaaabaa
acabccaabbbacbbbcccbacbbbaacccaaacbaacccbcccacccbbbacccabacabbbacccbbbccabb
baaacbbabcccbbaacccaaabacabbcccacbcccaaacbbbacaaabaaaccbaacccaaaccbabbbcbbb
aaacaaabaaacccbbaacaaabaacbbabbbcbbacaaabbbcacccbcabbbaaacaabaacccaaabbccac
ccbccabbbcbbaacabbcccacbacaaabbbcbbbabbbccabbcccacccbcccaaabaaccbbbcccbabbb
cbaaabbcccaccbbaaabbcbaacccaaabaaacabcbbaaabbcccaabbbaacabcbbbabbbcccbbbaa
cccbaaabbcbbbaaccbcaabaaaccbcccaccbbacbbbcccaccbbaaabbbcbbaacaaabbbcbbbabb
bcccbbbaacabbcccbbbccabbcccacccbcabbbaaacabcccbbaccbbbabcbbbaaabbbcccbaacaa
abbccabcaaacccabcccbbbccaabcbbbabbbccaaacccbbbccaabaaaccbcccaccbbbcccaaacccb
cccacccbbaccbbbcccacccbcccaaabbbcbabbbcccbbbaacccbccabaacaabbcccbbbabcbaacaa
abaacccabbbcacccbcccaaacccbbaaacaaabacccaaaccbbacbbbaacbbaaacccabcccbbbabccc
aaabbbcbbbabbbccabcccbbbaaacbbbcccacccbcccaabccaaacccbbbcccacbabbbcccbbbaaa
ccbaacaaabbbaaacccbbbabbcaabbbcbbbabcccbbbccaaacbbaaabbbccabaacccbcccacccbb
babbbcbbbaaacabcccbbbaaaccbbaacaaabbbaaaccbbbcabaaaccbcccaccbaaacaabbbccaaa
baacaabbbaaabbcccbacaaabbbcbbabbcccbaacccaaabcbbbaaabcaaaccbaacccbcccacbaac
cbbbcaaabbbcbbbabbbcccbaacccabcbbaaabcaaaccbaacccbcccacccbbbabccacccbccabaac
aabbbccaaabbbaaacaaabaaaccbcccaccbabbcccacccbcccaaabaaccbcccaaaccbbbacbaaabb
baccbbbcccbbacccaaaccbbbaaabbcccaaacccbcccacccbbacbbbcccacccbcccaaacccbbbccaa
acccabbcccbbbabbbcbbbaacbbabbbcccbbbaacaabacccbccaabaacaaabbcccbabbcaaabbb
acbbbccabbcccacbcccaaabcccbbbaccbcaaabbccaaacccbcccacccbbaaacccaaabacabbcccaa
acbcccacbbbcaabbbaaacbbbcaccbbaaacccaaabaacaabbcbabbcccaaabbbaaacabbcbbbab
bcccbbbccaabcacccbcccaabccaaacbabbbcccbaccbbaaacccaaabacaaabbcaabbbcbabbbccc
acbbaaabbbcbbbabbbcccbbbaacccbcccacbaaabbcccbaccbbbabbcaabaaacccbacbbbaaccc
bbbcccacccbcccaabccacccbbbacccaaabacaaabbcbbbacccbccaabcaaabbccabcbbbabbbcca
bbcccbbbccaaacccbcccacccbbaccbbbcccbbaaacaabaacccbbbcccacccbcccaabccaaacccbcc
cacccbbaaabbbacaaabacbcccacccbbabbbccaabacccbbbabcbaacaaabcccaabbbaaabbcbbb

aaccbbbabbcccbbbccaabbbaacabbcccacbcaaabaacbcccacccbbbacccabaaccbbabbbcbbaa
cccbbaccbbbcccacccbcccaabccacccbbbaaacccabbcccbbbcaabaaacaabccacccbccaabaccbb
bcccacccbcccaabccaaacccbbaacaaabaacccaaaccbbabcacccbcccaaaccbbbaaabbcaabbbccc
aaacccbcccacccbbabbcbbbaaacccbcccacccbbbaaacaaabaaacccaabbbcccbbbabcbbbaaccc
aaabbbcbbbabccaaabbbaaacaaabaaacccbcccacccbbbaacccaaaccbaaabbbacccaaabbbaaa
caaabaaacccaaabbbaccbcccaccbaaacaabbccaaabbbcbbbabccaaabbbaacaabaaacccbcccac
ccbbbcccaaabcccbbbaacbbaaacaaabaaacccaabbbcccaaacccbcccacccbbbcccaaaccbbbaaa
bcacccbcccaabccaaabaacaabbbcbbaacccbccabbcbbbabbccaabcaaaccbcccaccbbacbbbcca
abbbaccbbbcccbaccbbbabbbcbbbaaacaabcbbbaaabbbcccaccbbaacccaaabacaaabbcccaa
bbbcccbbbabbcbbaaccbcccaaabaacccbcccacccbbbcccaabcbbbaaaccbbbccacccbccaaabaa
acaaabbbcbbbabcccabbbaaacccaaabaaacaaabbcbbabcccabaaacccabccaabbcbbbaccbccc
accbbaaccbacccaaabaacaabbbaaabbcccbbbabbbcbbbaaabbcaabbbcccbbaaaccbcccaabb
cbbbaacaaabaacccaabbbaaacaaabaaaccbaacccaaaccbbabcccbbaacaaabbcccbbabbbccaa
abaaacaaabbbcccbacaaabaaacccbcccacccbbacbcccaaabcbbabbccaaabbcaabbbcbbbabcc
cbbbaacccbccaaacccbbbabbccaabbbaaacaabaacccaaaccbbaaacbbbccacccbccaaacccbbbc
caabbbaaacabccaabbbacbbbccabcccb