Bioinformatics With Rust

Welcome to Bioinformatics with Rust! An unofficial book aimed at introducing the Rust programming language for bioinformatic applications.

Introduction

- This book is **not** in any way, shape or form, an official introduction to neither the Rust programming language, nor bioinformatics.
    - For a comprehensive introduction to Rust, please visit resources.
    - For learning bioinformatics, please visit your favorite university.
- The **purpose** of this book is trifold:
    - Explain common bioinformatic concepts in a (hopefully) clear way.
    - Showcase some basic Rust implementations from scratch.
    - Provide examples of awesome open-source Rust crates for bioinformatics.
- Since `mdbook` is not easily integrated with external Rust crates, the code examples are minimally viable and built with native Rust. Throughout the book, however, examples of GitHub repositories and external Rust crates are provided as a way to showcase more real life applications.
- We'll mostly deal with DNA sequences and canonical nucleotides. However, feel free to request additional topics.
- For any issues related to this book, please file a GitHub issue.
- Currently, I'm a single person working on this project. If this project grows, I most likely need help from other people. Contributions are welcome!

Why Rust?

The different kinds of bioinformaticians

Bioinformatics encompasses lots of programming languages, from high level languages such as Python and R, to low level languages such as C and C++. The choice of language depends entirely on the target application. Below, I'll list my interpretation of the different kinds of bioinformaticians I know of:

- **The tool developer** - usually has a strong background in computer science and writes high-performance, open source tools for others to use. A name that comes to mind is Heng Li.
    - Language of choice is usually C or C++.
- **The pipeline developer** - has a strong sense of what bioinformatic tools are suitable for which task. They are experts in chaining multiple tools together to create complete pipelines for a given application.
    - Language of choice is usually Python, R and/or Bash, preferably in combination with ChatGPT.
- **The yak-shaver** - is interested in the details of things. Does not hesitate to spend weeks or months building custom databases and reading through literature. Usually starts digging into things and has troubles stopping.
    - Language of choice is usually Python and/or Bash.
- **The jack-of-all-trades** - has no prominent strengths nor weaknesses. Good at multi-tasking and knows a bit about everything. Might not have the strongest background in bioinformatics or programming, but has very high versatility.
    - Language of choice is whatever gets the job done.

Where the Rust programming language fits in

In my own experience, programming is complex and difficult. In addition, there are almost countless programming languages to choose from, each with their own pros and cons.

Traditionally, C and C++ have been used to write high-performance code because they are low level languages. You have to manage a lot of things, such as memory, manually. However, this comes with the advantage of experienced developers being able to write blazingly fast programs.

There is a fundamental problem with manual memory management - it is easy to introduce bugs and security vulnerabilities that can be hard to debug. This can be detrimental for performance-critical applications. Check out this blogpost as an example.

What is different about Rust? It prioritizes memory safety in order to reduce the accidental introduction of bugs and security vulnerabilities, whilst maintaining high performance. In my opinion (coming from a Python background), this comes with a cost of added complexity. I would like to reference my favorite quote from some random person on the internet:

"*The Rust compiler is stricter than my high school chemistry teacher.*"

When I started learning Rust, I'd agree with this statement. However, today I'd say it is a blessing rather than a curse.

To conclude, use Rust for bioinformatics if:

- You are interested in learning a low level, high performance language for bioinformatic applications.
- You want to create memory safe and performance critical bioinformatic pipelines.
- You want to traverse a steep learning curve, especially if coming from the Python world.

Why Not Python?

Traditionally, Python has been used as a wrapper around bioinformatic software to generate capable pipelines with great success. If this is your only intent, it makes sense to stick to Python. It is easy to learn and has a straightforward syntax.

However, as soon as one diverges from this and aims for implementing any sort of high-performance library, Python is not your friend. It is usually too slow, even though libraries such as pandas (which is basically C in disguise) improve runtimes. Sure, one can use the C interoperability interface but this is a bit cumbersome and not inherently memory safe.

Bioinformatic tools written in Rust

Finally, I just want to give a quick shoutout to some awesome bioinformatic tools written in Rust. There is actually quite a lot of bioinformatics-related crates available, but here are some of my favorites:

- Bio - General purpose bioinformatic tool for alignment, file processing and much more.
- Sylph - Metagenomic classification tool.
- NextClade - Virus specific tool for alignment, SNP calling, clade assignment and more.

For a more exhaustive list, see resources.

Alternatives To Rust

We'll finish this chapter off with listing some alternative programming languages, outside of Rust, that have been shown to work well within bioinformatics.

- `C/C++` - Lots of bioinformatic software (dare I say the majority?) is written in C and C++. Some examples that come to mind are Minimap2, Freebayes and Flye.
- `Go` - Believe it or not, the fastx toolkit Seqkit is actually written in Go.
- `Python` - Despite its downsides, there are some high performance bioinformatic applications written in Python (with C interoperability) such as Cutadapt. This also includes machine learning modules such as Medaka.
- `Perl` - Yes you read that right. Perl might have been voted one of the ugliest programming languages; regardless, there is at least one awesome tool SAMclip that makes the list.
- `Zig` - There might not be many existing bioinformatic tools written in Zig (yet), but due to its cross compilation functionality with C, I expect we'll see much more of this language in the coming years.
- `Mojo` - If I were to bet my money on any programming language becoming the go-to for bioinformatics, Mojo would be it. Designed to be a superset of Python (similar to what TypeScript is to JavaScript) whilst having similar performance to C and Rust, but with an intuitive GPU acceleration support, Mojo seems particularly promising within the field of bioinformatics.
- `R` - If you want a slow language, R is the way to go. With that said, there are seemingly endless R-packages for various bioinformatic applications, such as transcriptomics and metabolomics. In addition, it is actually awesome for generating beautiful plots with ggplot2.

About AI

I want to start this section by stating that I strongly opted out of vibe-coding this project. If I did, I'd probably been done in a day or two. Instead, I chose the proper (and difficult) path of trying things out, failing, swearing, reading documentation and finally (somewhat) understanding. This probably means there are some text and code-snippets in this book that are not 100% correct. I'm okay with that, because it means there is room for improvement.

With that said, I *have* used AI as a tool for the following:

- Asking questions about my code to find potential bugs, weaknesses and edge cases.
- Explain Rust concepts that I did not fully understand (such as declarative macros).
- Asking for suggestions on performance improvements and implemented them only if I can understand why it makes the code more performant.
- Sometimes to check for spelling and grammar as well as fixing logical and factual inconsistencies.

The Future of AI

My take on AI is that we are in a hype at the moment. I think that the hype might not live up to the expectations. In recent time, we have seen multiple disappointing releases of models and agents from companies that drive the AI train.

Ultimately, I don't think AI is bad. It can be incredibly useful. What I do think is that people are using it incorrectly. Multiple news channels have reported a MIT report, showing that 95% of organizations gain zero return using tools such as ChatGPT and Copilot. The report also shows that the primary productivity gain is on the individual level and not necessarily on large scale.

With that said, Claude is very, very good at writing lots of code.

Prerequisites

If you have never heard of programming or bioinformatics, this book is probably not for you.

If you have a lot of experience with both Rust and bioinformatics, this book is also probably not for you (and you probably have more knowledge than me, consider becoming a contributor!).

However, if you know a bit about bioinformatics and have worked with programming languages such as Python or C++ then this book will give you an introduction to the Rust ecosystem for building bioinformatic pipelines!

Getting Started

First and foremost, we need to install Rust and its package manager Cargo. The easiest way to do this is to use Rustup and follow the installation instructions.

Second, we need a code editor. If you want to make your life a living hell, you can use notepad. However, then we are missing some important stuff, like syntax highlighting, code formatting and much more. Here are some examples of code editors that can get the job done:

- VScode - Easy to use with lots of plugins to make your life easier.
- Zed - A text editor written in Rust!
- Vim - For hardcore programmers.
- NeoVim - For modern, hardcore programmers.

Personally, I prefer Zed. It is fast and has first-class support for Rust.

Rust Basics

Even though I stated that this book wouldn't include an introduction to Rust, here we are. This chapter only covers some of the basics and the reader is strongly encouraged to visit resources for a more comprehensive take on Rust.

Below is a summary of some of the things I personally like and dislike with Rust. This might give the reader some insight into whether or not Rust would be a suitable language for them.

Why I like Rust

- **The Rust Compiler Is Amazing**. The error messages it produces actually teaches you about Rust and why you cannot do certain things. It even gives you suggestions on how to make changes to your code to make it work correctly.
- **Declarative Mutability**. Variables that are not declared with the `mut` keyword are immutable, meaning that they cannot change (disregarding interior mutability, which won't be covered here).
- **Fast Growing Community**. There are endless Rust crates available at crates.io, some of which will make your Rust programming journey much more enjoyable.
- **Cargo**. It just works. Install a crate? Use `cargo install`. Run your code? Use `cargo run`.

Why I dislike Rust

- **Compile Times**. Compared to languages such as Go, Rust takes ages to compile. This is especially true when the dependencies are piling up.

- **Verbose Syntax**. In my personal opinion, Rust is a rather verbose language. Some people might like that, some people don't. Luckily, the rust-analyzer VScode extension helps out a lot with auto-completion and other neat features.
- **Steep Learning Curve**. Coming from the Python world, I had a really difficult time with Rust in the beginning. It was not just switching to a compiled language, but also having to learn about lifetimes, ownership and so on. But trust me, it gets easier.

Create a Project

Time to start! Enter your favorite directory and run `cargo new my_rust_project`. In the generated directory, you'll see one file `Cargo.toml` and one directory `src`.

`Cargo.toml` is where all of your dependencies go. For more information, visit the official reference.

`src` is where all of your Rust scripts go. For now, we only have `main.rs`, which is the entrypoint to the program.

Use `cargo run` to compile and run the program. It should output "Hello, world!". The `main.rs` file is very basic and should look something like this:

```
fn main() {
    println!("Hello, world!");
}
```

Note that we must have a `main()` function, otherwise it won't compile.

Syntax

The Rust syntax is similar to other languages such as C and C++. However, here is a brief overview.

Variable declaration

Rust is a statically typed language, which means that the type of a variable needs to be known, either explicitly or implicitly. The basic syntax for variable declaration is `let name: type = value;`. E.g.,

```
fn main() {
    let x: usize = 0; // unsigned integer.
    let x: &str = "Hello, world!"; // string slice.
    let x: String = "Hello, world!".to_string(); // string.
    let x: &[u8] = b"Hello, world!"; // byte slice.
    let x: Vec<usize> = vec![1, 2, 3, 4, 5]; // vec.
}
```

Scopes

`{` and `}` define scopes. E.g.,

```
fn main() { // start of function scope.
    println!("Hello, world!");
} // end of function scope.
```

We can also have nested scopes. E.g.,

```
fn main() {
    let x: &str = "Hello, world!";

    {
        println!("{x}");
    }
}
```

Scopes are important for ownership and lifetimes, which will be covered later on.

Statement delimiters

`;` is used for statement delimiters. E.g.,

```
fn main() {
    println!("Hello, world!"); // Defines the println! statement.
} // does not need a ";".
```

Note that scopes do not need a `;` terminator.

Comments

`//` is used for code comments.
`///` is used for docstrings.

```
/// This is a docstring.
fn main() {
    // This is a comment.
    println!("Hello, world");
}
```

Keywords

`use` - is used for importing. E.g., `use std::num::ParseIntError`

`let` - initializes something immutable. E.g., `let x: usize = 10;`

`mut` - makes something mutable. E.g., `let mut x: usize = 10;`

`fn` - defines a function. This is analogous to Python's `def` keyword. E.g.,

```
fn main() {
    println!("Hello, world!");
}
```

`struct` - defines a struct. This is kind of analogous to `class` in Python. E.g.,

```
struct MyStruct {
    field1: usize,
    field2: f32,
    field3: bool,
}
```

`enum` - defines an enum. This is kind of analogous to `Enum` in Python. E.g.,

```
enum MyEnum {
    Choice1,
    Choice2,
    Choice3,
}
```

`pub` - makes something like a function or struct public, meaning that other Rust files can access them. E.g.,

```
pub struct MyStruct {
    field1: usize,
    field2: f32,
    field3: bool,
}
```

`loop` - creates an infinite loop until a `break` statement is encountered. E.g.,

```
fn main() {
    let mut x: usize = 0;

    loop{
        x += 1;

        println!("{x}");

        if x >= 5{
            break;
        }
    }
}
```

`for` - creates a loop over an iterator. E.g.,

```
fn main() {
    for i in (0..5){
        println!("{i}");
    }
}
```

`while` - creates a loop that keeps running as long as its condition is true. E.g.,

```
fn main() {
    let mut x: usize = 0;

    while x <= 5 {
        println!("{x}");
        x += 1;
    }
}
```

Macros

Macros should have a dedicated book to themselves. Rust supports both declarative and procedural macros, which are either built-in or user created. In this book, we'll cover some of the most common built-in macros.

For more information about Rust macros, also see The Little Book of Rust Macros

Declarative macros

`println!` - prints to stdout. Requires a formatter depending on the data type. E.g.,

```
fn main() {
    println!("This is a string");
    println!("This is an int: {}", 5);
    println!("{:?}", vec!["This", "is", "a", "vec"]); // {:?} means
debug mode.
}
```

`vec!` - creates a `Vec` based on the provided input.

```
fn main() {
    let x: Vec<usize> = vec![1, 2, 3, 4, 5];
    println!("{:?}", x);
}
```

`panic!` - causes the program to exit and starts unwinding the stack.

```
fn main() {
    panic!("This will exit the program!")
}
```

`assert!` - runtime assert that a boolean expression evaluates to `true`. E.g.,

```rust
fn main() {
    assert!(5 < 6);
}
```

`assert_eq!` - runtime equality assert. E.g.,

```rust
fn main() {
    assert_eq!(6, 5 + 1);
}
```

Implementing our own declarative macro

I want to emphasize that I personally do not know that much about Rust macros. However, in this example we'll try to implement something that resembles Python's `Path`, which is a part of the `Pathlib` module. Using `Path`, there is a very handy way to define a file path by chaining multiple directories.

```python
from pathlib import Path

outdir = Path("my_outdir")
outfile = outdir / "sub_dir" / "another_sub_dir" / "my_file.txt"
```

Essentially, if the top directory `outdir` is of type `Path`, we can generate a file path through `/`. Personally, I think this is way more neat than having to use an f-string or similar. Let's try to implement something similar using a Rust declarative macro.

There are endless ways of implementing this, but below is one example. We'll define our macro `file_path` to require a base directory and at least one more argument. The syntax is a bit strange. It kinda looks like a function, but kinda not.

The expression `($base:expr $(, $sub:expr)+)` defines the pattern that we enforce. In this case, we require one expression `$base`, followed by one or more comma-separated expressions `$sub`.

We use import statements with a leading `::` to signify that we want the root crate `std` to not accidentally use some locally defined crate called `std`.

Finally, we create a `PathBuf` from our base dir and iteratively build up the path.

```
use std::path::PathBuf;

macro_rules! file_path {
    ($base:expr $(, $sub:expr)+) => {{
        use ::std::path::PathBuf;
        use ::std::fs;

        let mut full_path = PathBuf::from($base);

        $(
            full_path.push($sub);
        )*

        full_path
    }};
}

fn main(){

    let outdir = "my_outdir".to_string();
    let outfile = file_path!(outdir, "sub_dir", "another_sub_dir",
"my_file.txt");

    println!("{:?}", outfile);
}
```

The point with the simple example above is not to generate a bullet proof, production ready macro but rather showcase that declarative macros can be very handy for defining custom behaviors. If we'd try to implement `file_path` as a function, we'd probably have to handle the variable number of sub-directories through a `Vec` or similar.

Procedural macros

Are divided into three categories, all of which are outside the scope of this book. Regardless, they are very handy for deriving traits, such as `Debug`. As an example, assume we've created a `Struct` that we'd want to be able to print to stdout using `println!`. In this case, we need to derive the `Debug` trait through `#[derive(Debug)]`.

```
#[derive(Debug)] // Try commenting out this line!
struct MyStruct{
    my_vec: Vec<usize>,
}

fn main() {
    let my_struct = MyStruct { my_vec: vec![1, 2, 3, 4, 5] };

    println!("{:?}", my_struct);
}
```

Data Types

Rust has a lot of data types. Here is a rundown of the ones I use most often:

Data type	Rust type	Example
boolean	`bool`	`let x: bool = true;`
string	`String`	`let x: String = "Hello".to_string();`
string slice	`&str`	`let x: &str = "Hello";`
array	`[type; len]`	`let x: [usize; 3] = [1, 2, 3];`
vec	`Vec<type>`	`let x: Vec<usize> = vec![1, 2, 3];`
byte slice	`&[u8]`	`let x: &[u8] = b"Hello";`
unsigned 32-bit int	`u32`	`let x: u32 = 0;`
unsigned 64-bit int	`u64`	`let x: u64 = 0;`
unsigned (32 or 64)[1]-bit int	`usize`	`let x: usize = 0;`
32-bit float	`f32`	`let x: f32 = 0.0;`
64-bit float	`f64`	`let x: f64 = 0.0;`

For an interactive map of Rust types, please visit RustCurious.

---

1. Depends on computer architecture. ↩

Strings

There are multiple string types in Rust. Two of the ones I use most often are `String` and `&str`.

`String` is an owned, mutable and heap-allocated type. We can allow it to be mutable with the `mut` keyword. E.g.,

```
fn main() {
    let mut seq: String = "ATCG".to_string();
    seq.push_str("ATCG"); // Mutate.

    assert_eq!(seq, "ATCGATCG".to_string());
}
```

`&str` is a borrowed and immutable type. We can read from it, but cannot mutate it. `&str` is suitable when one wants to avoid heap-allocation.

```
fn main() {
    let seq: &str = "ATCG";

    println!("{seq}");
}
```

Array

Arrays in Rust are fixed size that need to be known at compile time. In bioinformatic applications, we can use an array as a lookup table for nucleotide encoding, which we'll see in later chapters.

If we declare the array as mutable, we can change its values but not its size.

```
fn main(){
    let mut arr: [usize; 5] = [1, 2, 3, 4, 5];

    for i in (0..arr.len()){
        arr[i] = arr[i] * 2;
    }

    assert_eq!(arr, [2, 4, 6, 8, 10]);
}
```

Vec

A `Vec` is like an array type with dynamic size. There are two common ways to initialize a `Vec`, either through the `vec!` macro, or through `Vec::new()`.

```
fn main() {
    // Create an empty vec.
    let mut my_vec: Vec<usize> = Vec::new();

    my_vec.push(1);

    assert_eq!(my_vec, vec![1]);
}
```

We can also collect an iterator into a `Vec`, which is very convenient.

```
fn main() {
    let my_iterator = 1..5;

    let my_vec: Vec<usize> = my_iterator.collect();

    assert_eq!(my_vec, vec![1, 2, 3, 4]); // my_iterator is right
exclusive.
}
```

Control Flow

An `if` statement works very similar to other languages. E.g.,

```
fn main() {
    let x: usize = 5;

    if x >= 10 {
        println!("{x} is large");
    }
    else {
        println!("{x} is small");
    }
}
```

A `match` statement works similarly to a `switch` statement in C and C++ and needs to be exhaustive.

```
fn main() {
    let x: usize = 1;

    match x {
        1 => println!("x is 1"),
        2 => println!("x is 2"),
        _ => println!("x is something else"),
    }
}
```

Rust supports relatively advanced pattern matching which is extremely useful.

References

References in Rust are different from pointers in languages such as C and C++. In Rust, references are always valid and cannot be null. Use `&` to reference a variable, and `*` to dereference.

```
fn main() {
    let my_vec: Vec<usize> = vec![1, 2, 3, 4, 5];

    println!("{:?}", &my_vec); // Pass my_vec as a reference to the
println! macro.
}
```

References are useful for passing variables to other functions without needing to clone the data. In the following example below, we'll create a `Vec` and then pass it by reference to a function. Note the syntax here, we actually don't pass a reference `&Vec<usize>` . We could, but a more idiomatic approach (in my opinion) is to pass a slice instead.

```
fn print_a_vec(x: &[usize]) {
    println!("{:?}", x);
}

fn main() {
    let my_vec: Vec<usize> = vec![1, 2, 3, 4, 5];

    print_a_vec(&my_vec[..]);
}
```

There is an important rule when it comes to references, which I'll quote from the official Rust book:

"*At any given time, you can have either one mutable reference or any number of immutable references.*"

This makes perfect sense when you think about it. If we are able to mutate a variable, we do not want a bunch of read-only references with unpredictable values when read.

Functions

As we saw earlier, a function is defined with `fn`.

```
fn main(){
    println!("Hello, world!");
}
```

To define a function that takes arguments, we need to define the argument names and types.

```
fn my_function(a: usize, b: usize) {
    println!("Arguments are {a} and {b}");
}

fn main() {
    let a: usize = 1;
    let b: usize = 2;

    my_function(a, b);
}
```

We also need to take into consideration if we are passing values as references and whether or not they are mutable. In the example below, we define a `Vec` as mutable and pass it by reference to a function `mutate_vec`. In order for this to work, the argument type of `mutate_vec` must be `&mut Vec<usize>` to signify that we are passing a mutable `Vec` by reference. We call `mutate_vec` from the main function with `&mut my_vec` to match the defined argument type `&mut Vec<usize>`.

```
fn mutate_vec(a: &mut Vec<usize>) {
    a[0] = 10;

    println!("{:?}", a);
}

fn main() {
    let mut my_vec: Vec<usize> = vec![1, 2, 3, 4, 5];

    mutate_vec(&mut my_vec);
}
```

Finally, we'll also add a return type, which is done with `->` in the function signature. In this example, we mutate the `Vec` inside `mutate_vec`, return a mutable reference to it and mutate it again.

```
fn mutate_vec(a: &mut Vec<usize>) -> &mut Vec<usize> {
    a[0] = 10;

    return a;
}

fn main() {
    let mut my_vec: Vec<usize> = vec![1, 2, 3, 4, 5];

    let mut my_mutated_vec: &mut Vec<usize> = mutate_vec(&mut
my_vec);

    my_mutated_vec[0] = 20;
    println!("{:?}", my_mutated_vec);

}
```

Enums

Rust enums are awesome and also extremely useful, especially in match statements. Assume we have implemented three different alignment functions: local, semi-global and global. We can use an enum as input to decide what alignment function to run for a given query and subject:

```
enum AlignmentType {
    Local,
    SemiGlobal,
    Global,
}

#[allow(unused)]
fn local_alignment(query: &str, subject: &str){
    println!("Running local alignment...");
}

#[allow(unused)]
fn semi_global_alignment(query: &str, subject: &str){
    println!("Running semi-global alignment...");
}

#[allow(unused)]
fn global_alignment(query: &str, subject: &str){
    println!("Running global alignment...");
}


fn align(query: &str, subject: &str, alignment_type: AlignmentType)
{
    match alignment_type {
        AlignmentType::Local => local_alignment(query, subject),
        AlignmentType::SemiGlobal => semi_global_alignment(query,
subject),
        AlignmentType::Global => global_alignment(query, subject),
    }
}

fn main(){
    align("ATCG", "ATCG", AlignmentType::Local);
    align("ATCG", "ATCG", AlignmentType::SemiGlobal);
    align("ATCG", "ATCG", AlignmentType::Global);
}
```

Another use case of enums could be if we have an alignment between two sequences for which we want to calculate an alignment score. We could create an alignment type enum that is associated with increasing or decreasing an alignment score.

```rust
enum AlignmentCost {
    Match(usize),
    Mismatch(usize),
    DeletionQuery(usize),
    DeletionSubject(usize),
}

fn update_score(score: &mut i32, alignment_cost: AlignmentCost) {
    match alignment_cost {
        AlignmentCost::Match(c) => {
            *score += c as i32;
        }
        AlignmentCost::Mismatch(c) => {
            *score -= c as i32;
        }
        AlignmentCost::DeletionQuery(c) => {
            *score -= c as i32;
        }
        AlignmentCost::DeletionSubject(c) => {
            *score -= c as i32;
        }
    };
}

fn main() {
    let mut score: i32 = 0;
    println!("Initial score: {score}");

    // Match will increase the score.
    update_score(&mut score, AlignmentCost::Match(4));
    println!("Score after match: {score}");

    // Mismatch will decrease the score.
    update_score(&mut score, AlignmentCost::Mismatch(1));
    println!("Score after mismatch: {score}");

    // Query deletion will decrease the score.
    update_score(&mut score, AlignmentCost::DeletionQuery(1));
    println!("Score after query deletion: {score}");

    // Subject deletion will decrease the score.
    update_score(&mut score, AlignmentCost::DeletionSubject(1));
    println!("Score after subject deletion: {score}");
}
```

This is a pretty silly example, but showcases how enums are very convenient for handling and taking action based on a specific set of cases.

Structs

Implementing structs in Rust is a bit different from languages such as Python. In Python, we have the excellent Pydantic module for data validation and other awesome features. In Rust, we can use something like the Validify crate, however in this book we won't bother much with validation.

Pretend we have a fastq parser for filtering and trimming reads. However, we want to change the filtering and trimming parameters based on sequencing platform. Maybe we want different behavior depending on if our data originated from PacBio or Oxford Nanopore. An example of this would be to implement a default function based on a provided platform:

```
enum Platform {
    PacBio,
    Nanopore,
}

#[allow(unused)]
#[derive(Debug)]
struct Parameters {
    min_len: usize,
    max_len: usize,
    min_phred: usize,
}

impl Parameters {
    fn default(platform: Platform) -> Self {
        match platform {
            Platform::PacBio => Self {
                min_len: 100,
                max_len: 1000,
                min_phred: 20,
            },
            Platform::Nanopore => Self {
                min_len: 200,
                max_len: 900,
                min_phred: 15,
            },
        }
    }
}

fn main() {
    let pacbio_parameters = Parameters::default(Platform::PacBio);
    println!("PacBio: {:?}", pacbio_parameters);

    let nanopore_parameters =
Parameters::default(Platform::Nanopore);
    println!("Nanopore: {:?}", nanopore_parameters);
}
```

This works, but might not be very idiomatic. Another way is to leverage Rust's type traits by implementing `from`. By specifying our variable as type `Parameters`, we can call `.into()` directly.

```
// [...]

impl From<Platform> for Parameters {
    fn from(platform: Platform) -> Self {
        match platform{
            Platform::PacBio => Self {
                min_len: 100,
                max_len: 1000,
                min_phred: 20,
            },
            Platform::Nanopore => Self {
                min_len: 200,
                max_len: 900,
                min_phred: 15
            },
        }
    }
}

fn main(){
    let pacbio_parameters: Parameters = Platform::PacBio.into();
    println!("PacBio: {:?}", pacbio_parameters);

    let nanopore_parameters: Parameters = Platform::Nanopore.into();
    println!("Nanopore: {:?}", nanopore_parameters);
}
```

Again, note that these are just examples that might not be real-world applicable. However, the point here is that structuring the code in certain ways will be of help in the long run.

Option and Result

Option

In contrast to Python, there is no `None` type in Rust. There is, however, something called `Option`, which is of type `Option<T>`. An `Option<T>` can either be `Some(T)` (there is a value) or `None` (there is no value). Usually, one would pattern match to extract the value from an `Option` if it exists.

```
fn print_value_if_exist(x: Option<usize>) {
    match x {
        Some(value) => println!("Value is {value}"),
        None => println!("No value"),
    };
}

fn main() {
    let x: Option<usize> = Some(5);
    print_value_if_exist(x);

    // We can define x have the value None, but
    // its type will always be Option<T>.
    let x: Option<usize> = None;
    print_value_if_exist(x);
}
```

Result

Similarly for errors, there is `Result`, which is of type `Result<T, E>`. A `Result<T, E>` can be either `Ok(T)` or `Err(E)`, which we can pattern match against.

```rust
use std::num::ParseIntError;

fn parse_to_usize(x: &str) {
    let parsed: Result<usize, ParseIntError> = x.parse::<usize>();

    match parsed {
        Ok(number) => println!("{number}"),
        Err(err) => println!("{}: {x}", err),
    }
}

fn main() {
    let x: &str = "5";
    parse_to_usize(x);

    let x: &str = "5ab";
    parse_to_usize(x);
}
```

Error Handling

It took a while for me to understand errors in Rust. However, one day, I came across the concept of recoverable and non-recoverable errors and that is when everything started making sense.

An `unrecoverable` error occurs when it makes sense to terminate the code. One example would be failing to read a file because it is corrupt. If our only goal is to parse the file and print its contents, but we cannot even read it, then we'd classify this as an unrecoverable error.

A `recoverable` error you can think of as when it is still safe or okay to proceed executing code. One example would be parsing lines from a file and one line has an unexpected structure. If we are okay with this, we can just skip this line and proceed to the next.

There are different ways of handling errors, some of which are listed below:

- `panic!` - Is a macro that, in single threaded applications, will cause the program to exit.
- `unwrap` - Will panic if an `Option<T>` is `None` or if a `Result<T, Error>` is `Error`.
- `expect` - Is similar to `unwrap` but also displays a provided error message on panic.
- `?` - Is used for error propagation and can be handled by e.g., upstream functions. This is a very elegant way of handling errors and is preferred over `unwrap` and `expect` in real world applications. `?` must always be inside a function that returns the `Result` type.

Unrecoverable errors

In the code snippet below, we try to open a file that does not exist. Using `.expect()` will cause a panic, but this is okay because we allow this to be an unrecoverable error.

```
use std::fs::File;

fn main() {
    let _ = File::open("file_does_not_exist.txt").expect("Failed to
open file.");
}
```

Recoverable errors

In the following example, we implement a recoverable error for integer division using the `?` operator. The code looks quite complex for such a simple example, but the general pattern can be applied to other code as well.

- We define a custom error type called `MathError`. We could define multiple `MathError` types, but in our case, `DivisionByZero` will suffice.
- We implement the `Display` trait for our custom error to avoid having to use `Debug` print.
- We implement a function `divide` that returns a `Result`, containing either a `f32`, or a `MathError`.
- We implement a function `division` that uses the `?` operator. Think of the `?` as "assume no error", then we can return `Ok(result)`. If `result` contains an error, the function `division` will make an early return.
- In `main`, we handle the division result accordingly.

```rust
#[derive(Debug)]
enum MathError {
    DivisionByZero,
}

impl std::fmt::Display for MathError {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        match self {
            MathError::DivisionByZero => write!(f, "Cannot divide by
zero!"),
        }
    }
}

fn divide(a: usize, b: usize) -> Result<f32, MathError> {
    match b {
        0 => Err(MathError::DivisionByZero),
        _ => Ok(a as f32 / b as f32),
    }
}

fn division(a: usize, b: usize) -> Result<f32, MathError> {
    let result = divide(a, b)?;

    return Ok(result);
}

fn main() {
    let values: Vec<(usize, usize)> = vec![(1, 1), (1, 0)];

    for (a, b) in values {
        match division(a, b) {
            Ok(r) => println!("{r}"),
            Err(e) => println!("{e}"),
        }
    }
}
```

The takeaway here is that by handling recoverable errors, we avoid crashing our program when it does not need to.

Visit the official documentation for error handling to learn more. In addition, there are crates such as anyhow and thiserror that simplifies the generation of custom error types.

Ownership and Borrowing

Ownership can initially be a rather tricky topic to understand. The reader is advised to read the official reference on ownership.

I like to think of ownership in terms of scopes. A variable is valid when it is inside the scope it was defined in. When the scope ends, the variable is dropped from memory. This might not always be true, but this way of thinking simplified the ownership concept for me quite a lot. Consider the following example:

```
fn main() {
    // Create a nested scope.
    {
        let x: usize = 0;
        println!("{x}");
    }
}
```

It is perfectly valid to use `println!` whilst we are inside the scope, because `x` is still valid here. Once we move outside of this scope, `x` is dropped. This means we cannot move our `println!` outside of the scope where `x` is defined.

```
fn main() {
    // Create a nested scope.
    {
        let x: usize = 0;
    }

    // try commenting this out!
    // println!("{x}");
}
```

The same goes for heap-allocated variables that are passed by value (i.e., we are not passing a reference). In the following example, we'll create a `Vec` and pass it by value to a function `print_vec`. As we will see, the Rust compiler won't let us print the `Vec` anymore in the main function.

```
fn print_vec(x: Vec<usize>) {
    println!("[print_vec]: {:?}", x);
}


fn main() {
    let x: Vec<usize> = vec![1, 2, 3, 4, 5];

    print_vec(x); // x is passed by value here. Ownership is transferr

    // try commenting this out!
    // println!("[main]: {:?}", x);
}
```

How do we solve this? We can pass `x` by reference. This way, `x` is still owned by `main` and borrowed by `print_vec`.

```
fn print_vec(x: &Vec<usize>) {
    println!("[print_vec]: {:?}", x);
}

fn main() {
    let x: Vec<usize> = vec![1, 2, 3, 4, 5];

    print_vec(&x); // x is passed by reference here. main still owns
x.
    println!("[main]: {:?}", x);
}
```

What about mutable references? Remember that a variable can only have one mutable reference existing at a given time. If we pass `x` as a mutable reference to a new function `mutate_vec`, `main` still has ownership of `x` so we are good.

```
fn mutate_vec(x: &mut Vec<usize>) {
    x[0] = 10;

    println!("[mutate_vec]: {:?}", x);
}

fn main() {
    let mut x: Vec<usize> = vec![1, 2, 3, 4, 5];

    mutate_vec(&mut x); // x is passed by (mutable) reference here.
main still owns r.
    println!("[main]: {:?}", x);
}
```

However, we would run into issues if we try dereferencing `x` inside `mutate_vec`. In the example below, we try dereferencing `x` into a variable `y` inside `mutate_vec`. This is not allowed because we don't own `x`. We have only borrowed it, so we cannot move its value.

```
fn mutate_vec(x: &mut Vec<usize>) {
    x[0] = 10;

    // try commenting this out!
    // let y = *x;

    println!("[mutate_vec]: {:?}", x);
}

fn main() {
    let mut x: Vec<usize> = vec![1, 2, 3, 4, 5];

    mutate_vec(&mut x); // x is passed by (mutable) reference here. ma
    println!("[main]: {:?}", x);
}
```

Lifetimes

The concept of lifetimes is related to how long variables are valid. Even though lifetimes are a rather complex concept, there are two common cases one has to consider:

- Variables are dropped at the end of their defined scope.
- References might need an explicit lifetime notation.

Scopes

By default, variables are dropped at the end of their defined scope.

```
fn main() {
    let x: usize = 0;
} // x is dropped here.
```

The example above gets more interesting if we add a nested scope inside `main`. Try running the code below and see what happens.

```
fn main() {
    {
        let x: usize = 0;
    } // x is dropped here.

    println!("{x}");
}
```

The variable `x` goes out of scope before we try to print it, which is why the compiler complains. Switching the order around by defining `x` in the outer scope and printing it in the inner scope works, because `x` is still valid there.

```
fn main() {
    let x: usize = 0;

    { // x is still valid here.
        println!("{x}");
    }

} // x is dropped here.
```

The same principle applies to function scopes. Unless we `return`, variables defined inside a function scope will be dropped at the end.

```
fn my_function() -> String {
    let x: String = "my_string".to_string();

    // Anything else we define here and don't return
    // will be dropped at the end of the function scope.

    return x;
}

fn main() {
    let x = my_function();

    println!("{x}");
}
```

Lifetime notation

To illustrate explicit lifetime notation, we'll create a `Struct` with a single field `my_string`, which is of type `&str`.

```
struct MyStruct {
    my_string: &str
}

fn main() {
    let my_struct = MyStruct {my_string: "Hello, world!"};
}
```

Try running the code and see what happens. We get a compiler error, stating that we need a lifetime parameter. Why is this? `my_string` is of type `&str`, which means that `MyStruct` does not own it. This also means `MyStruct` does not control when `my_string` is no longer valid. This is dangerous, because if `my_string` would get dropped and we subsequently try to read its value in `MyStruct`, we'd be in trouble. The Rust compiler needs some kind of assurance that `MyStruct` and `my_string` will both be valid for at least as long as each other. This is what lifetimes are for.

Lifetimes are signified with a `'`, followed by a name. E.g., `'a` would be a lifetime called `a`. To make the code run, we'll bind `MyStruct` and `my_string` to the same lifetime, telling the Rust compiler that `MyStruct` will live for at least as long as `my_string`.

```
struct MyStruct<'a> {
    my_string: &'a str
}

fn main() {
    let my_struct = MyStruct {my_string: "Hello, world!"};
}
```

The same concept applies to functions. In the following example, we'll define a function that takes no arguments and returns a `&str`.

```
fn my_function<'a>() -> &'a str{
    let x: &'a str = "my_string";

    return x;
}

fn main() {
    let x = my_function();

    println!("{x}");
}
```

Iterator Chaining

One of my favorite Rust features is its powerful iterator chaining. One basic example is using `map` to apply a custom function to each element in the iterator.

```
fn main() {
    let x: Vec<usize> = vec![1, 2, 3, 4, 5];

    let x_mapped: Vec<usize> = x.iter().map(|v| *v * 2).collect();

    assert_eq!(x_mapped, vec![2, 4, 6, 8, 10]);
}
```

A slightly more advanced example is trying to parse a `Vec` of strings, only keeping the values we successfully parsed. In the example below, we use `filter_map` to both filter and map values at the same time.

```
fn main() {
    let x: Vec<&str> = vec!["3", "hello", "8", "10", "world"];

    let x_parsed: Vec<usize> = x.iter().filter_map(|v| v.parse::
<usize>().ok()).collect();

    println!("{:?}", x_parsed);
}
```

`filter_map` accepts an `Option<T>` to filter on. In our case, `parse` returns a `Result<T, Err>` so we use `.ok()` to convert `Result<T, Err>` to `Option<T>`.

We can also use scopes to create extremely versatile chaining. In the last example, we'll loop over a `Vec` that contains some mock fastq reads, gather some stats, and collect into a new `Vec`. Even though this is a silly example, it shows how we can use iterator chaining to provide structure to unstructured data.

```
#[allow(unused)]
#[derive(Debug, PartialEq)]
struct FastqRead<'a> {
    name: &'a str,
    seq: &'a [u8],
    qual: &'a [u8],
    length: usize,
    mean_error: f64,
}

fn collect_fastq_reads<'a>(fastq_reads: &'a [(&str, &[u8], &[u8])])
-> Vec<FastqRead<'a>> {
    let fastq_stats: Vec<FastqRead<'a>> = fastq_reads
        .iter()
        .map(|(name, seq, qual)| {
            // Calculate mean error rate.
            let error_sum: f64 = qual
                .iter()
                .map(|q| 10_f64.powf(-1.0 * ((q - 33) as f64) /
10.0))
                .sum();

            let fastq_read = FastqRead {
                name: name,
                seq: seq,
                qual: qual,
                length: seq.len(),
                mean_error: error_sum / qual.len() as f64,
            };

            return fastq_read;
        })
        .collect();

    fastq_stats
}

fn main() {
    let fastq_reads: Vec<(&str, &[u8], &[u8])> = vec![
        ("read_1", b"ATCG", b"????"),
        ("read_2", b"AAAAAAA", b"???????"),
    ];

    let fastq_stats = collect_fastq_reads(&fastq_reads);

    assert_eq!(
        fastq_stats,
```

```
        vec![
            FastqRead {
                name: "read_1",
                seq: b"ATCG",
                qual: b"????",
                length: 4,
                mean_error: 0.001
            },
            FastqRead {
                name: "read_2",
                seq: b"AAAAAAA",
                qual: b"???????",
                length: 7,
                mean_error: 0.001
            }
        ]
    );
}
```

Multi-threading

In bioinformatics, multithreading can be vital and has the potential to decrease runtimes. Sometimes even by several magnitudes. The Rust documentation covers concurrency and multithreading in detail and there are also several crates that make multithreading easier to implement. Below is a list of Rust crates that work very well for creating fast and memory-safe bioinformatic applications:

- Bio - General purpose bioinformatic tool for e.g., parsing fastq/fasta files.
- Rayon - Data parallelism library that works well together with the Bio crate. Enables parallel processing of sequences through par_bridge().
- DashMap - Concurrent HashMaps and HashSets.

Tips And Tricks

Below, I've gathered some tips and tricks when it comes to using multithreading with Rust, especially for bioinformatic applications.

Start by creating a MVBA

A MVBA (Minimally Viable Bioinformatic Application) is something that runs and produces the expected output. I've found that starting out this way is easier, because one can always optimize the code later on. For me, it is tempting to start out writing the most optimized code from the beginning. However, I've learned that programming this way takes more time and is less productive.

Optimize the MVBA

Once the MVBA is done, it is time to optimize. We must not forget this if we want an application that performs well under heavy loads. Optimization can be done in several ways, such as:

- Testing the application in release mode `cargo build --release`.
- Use a profiler such as Samply to identify bottlenecks.
- Implement concurrency and multithreading if applicable.
- Using appropriate data structures.

Multithreading is not always the answer

Even though multithreading is a useful tool within bioinformatics, there are cases where it might hurt more than it helps.

Cases where multithreading shines:

- CPU heavy loads (such as genome assembly, read alignment, etc).
- Tasks can be executed relatively independently.

Cases where multithreading might not be the answer:

- Tasks are extremely small and frequent.
- Each task is expected to take up a lot of RAM (risking out of memory error).

Find code examples

One excellent way to learn about concurrency is to look at code examples. There is plenty of well-established open source Rust projects that use multithreading. As a start, here are some of my own not-so-well-established-and-work-in-progress projects:

- sintax_rs - Rust implementation of the SINTAX classifier.
- ani_rs - Fast approximate genome similarity.

Trait Bounds and Generics

Rust supports generic types, but due to its strict type checker we need to put some restrictions on the generic type. In Python, we can use unions to signify that a variable or argument can be of different types.

```python
def display(s: str | int | list[str]):
    print(s)

print('my_string')
print(1)
print(['my', 'string'])

# This will actually run but the linter will complain.
print({'my', 'string'})
```

In Rust, this works a bit differently. To print something, the variable needs to implement either the `Display` trait (“normal” print) or the `Debug` trait (debug print). By using a `trait bound`, we can tell the compiler that only generic types which implement the `Display` or `Debug` trait are allowed as argument(s) to the function.

In the example below, we implement a function that accepts an argument of a generic type `T` that implements the `Debug` trait. Luckily, all Rust types in the `std` library automatically implement `Debug`.

```rust
use std::fmt::Debug;

fn display<T: Debug>(arg: T) {
    println!("{:?}", arg);
}

fn main() {
    display(1);
    display("my_string");
    display("my_string".to_string());
    display(vec!["my", "string"]);
}
```

Deriving traits

What if we have a type that does not implement `Debug` by default? We can derive it using the `derive` macro.

In the following example, we'll create a `Struct` that by default does not implement `Debug`. By using the `derive` macro, we can subsequently call our `display` function.

```
use std::fmt::Debug;

#[derive(Debug)] // Try commenting this out!
struct MyStruct<'a> {
    field_1: usize,
    field_2: &'a str,
    field_3: String,
    field_4: Vec<&'a str>,
}

fn display<T: Debug>(arg: T) {
    println!("{:?}", arg);
}

fn main() {
    let my_struct = MyStruct {
        field_1: 1,
        field_2: "my_string",
        field_3: "my_string".to_string(),
        field_4: vec!["my", "string"],
    };

    display(my_struct);
}
```

Other common Rust traits that can be derived or implemented manually are:

- `Display`
- `Clone`
- `Send`
- `Sync`
- `Ord`
- `PartialOrd`

Smart Pointers

In Rust, there is a set of smart-pointers to make our lives easier when it comes to things such as ownership, references, etc.

Box

The `Box` smart-pointer is used to enforce heap allocation of the value, and stack allocation of the reference. It can be used in various applications, two of which are recursive datatypes and dynamic traits.

Recursive Datatypes

Let's define a `Struct` which has a field `inner` that references itself (i.e., a recursive structure). If we try to run this, we'll get a compiler error.

```
struct MyStruct{
    inner: Option<MyStruct>,
    value: usize
}

fn main(){
    let my_struct = MyStruct {value: 0, inner: Some(MyStruct {
value: 1, inner: None}) };
}
```

The problem is that the size of `MyStruct` is not known at compile time, which is a requirement for stack allocation. However, by using a `Box` we can enforce heap allocation of `MyStruct`, keeping its reference on the stack. Since references are compile-size-known, this works.

```
struct MyStruct{
    inner: Option<Box<MyStruct>>,
    value: usize
}

fn main(){
    let my_struct = Box::new(MyStruct {value: 0, inner:
Some(Box::new(MyStruct { value: 1, inner: None})) });
}
```

The obvious downside here is that the code becomes rather verbose.

Dynamic Traits

Another use of `Box` is for dynamic traits. One example of this is to create a `BufWriter` that writes either to file or stdout depending on if we provide an output file. We'll define a function `get_bufwriter`, that wraps `BufWriter` around either `File` or `Stdout` depending on the argument `outfile`. We want something like this (in pseudo-code):

```
fn get_bufwriter(outfile: Option<PathBuf>) -> ??? {
    match outfile {
        Some(outfile) => return
BufWriter::new(File::create(outfile).unwrap());
        None => return BufWriter::New(stdout());
    }
}
```

However, Rust does not natively allow us to return a value that can be of two different types, `BufWriter<File>` or `BufWriter<Stdout>`. Fortunately, both types have the `Write` trait implemented. By wrapping `File` and `Stdout` in a `Box`, we can change our return type to the more generic `BufWriter<Box<dyn Write>>`. Conceptually, this signature means that the return type is a `BufWriter` wrapped around a type that implements the `Write` trait. The `dyn` keyword is related a trait object's type. Because the exact size of `Write` is not known at compile-time, we need to use `Box`.

```rust
use std::{
    fs::File,
    io::{BufWriter, Write, stdout},
    path::PathBuf,
};

fn get_bufwriter(outfile: Option<PathBuf>) -> BufWriter<Box<dyn
Write>> {
    match outfile {
        Some(outfile) => return
BufWriter::new(Box::new(File::create(outfile).unwrap())),
        None => return BufWriter::new(Box::new(stdout())),
    }
}

fn main() {
    // Create a writer that writes to stdout.
    let mut writer = get_bufwriter(None);
    writer.write(b"This will be written to stdout!\n").unwrap();

    // // Commented out for obvious reasons.
    // let mut writer =
get_bufwriter(Some(PathBuf::from("file.txt")));
    // writer
    //     .write(b"This will be written to the output file!\n")
    //     .unwrap();
}
```

Rc

`Rc` stands for “Reference Counting” and is for single-threaded, multiple ownership. By creating multiple references to a variable (increasing the reference count), we can prevent the variable from being dropped until the reference count reaches zero. A good analogy would be multiple people watching the same TV. We don’t want the TV to turn off until all people stop watching.

```
use std::rc::Rc;

#[allow(unused)]
fn main() {
    let x: Rc<usize> = Rc::new(0);

    assert_eq!(Rc::strong_count(&x), 1);
    println!("{}", Rc::strong_count(&x));

    {
        let x_clone = x.clone();
        assert_eq!(Rc::strong_count(&x), 2);
        println!("{}", Rc::strong_count(&x));
    } // x_clone is dropped here, reference count to x will decrease
by one.

    assert_eq!(Rc::strong_count(&x), 1);
    println!("{}", Rc::strong_count(&x));
}
```

Arc

`Arc` stands for “Atomic Reference Count” and is a thread safe alternative to `Rc`. It is commonly used together with `Mutex` for exclusive read/write access. One example is trying to push elements from different threads to a shared `Vec` instance. We need to ensure that our threads do not read and write at the same time since this can cause lockings and undefined behavior.

In the example below, we create a `Vec` for storing a message from each thread. We wrap it in `Arc<Mutex<>>` to ensure thread safety. Then we spawn four threads, each of which will push a `String` to our `Vec`. By using `.lock()` we can make sure only one thread can access our `Vec` at a given time. Finally, we wait for all threads to finish and print the results.

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let v: Arc<Mutex<Vec<String>>> = Arc::new(Mutex::new(vec![]));

    let mut handles = Vec::new();

    for i in 0..4 {
        // Each thread will get its own reference to the
Arc<Mutex<Vec<String>>>.
        let v_clone = v.clone();

        let s = thread::spawn(move || {
            v_clone
                .lock()
                .unwrap()
                .push(format!("Hello from thread {}", i));
        });

        handles.push(s);
    }

    // Wait for and join the spawned threads.
    for h in handles {
        h.join().unwrap();
    }

    // Extract our Vec from the Arc<Mutex<>>.
    let v_done = Arc::into_inner(v).unwrap().into_inner().unwrap();

    for s in v_done {
        println!("{s}");
    }
}
```

Cow

The Clone-On-Write smart pointer provides immutable access to borrowed data with the ability to lazily clone data when mutability or ownership is required. `Cow` is usable for cases where most of the time, we don't need to mutate data.

Consider the example below, where we want to convert a `&str` to lowercase. If we expect that most of the time our `&str` is already lowercase, we can return it as is most of the time with `Cow::Borrowed()`. However, for those rare cases when we need to modify it, we use `Cow::Owned()`.

```
use std::borrow::Cow;

fn convert_to_lowercase(x: &str) -> Cow<'_, str> {
    if x.chars().any(|c| c.is_uppercase()) {
        return Cow::Owned(x.to_lowercase());
    }

    return Cow::Borrowed(x);
}

fn make_lowercase(x: &str) -> Cow<'_, str> {
    let x_uppercase = convert_to_lowercase(&x);

    match &x_uppercase {
        Cow::Borrowed(_) => {
            println!("Is borrowed.");
        }
        Cow::Owned(_) => {
            println!("Is owned.");
        }
    }

    return x_uppercase;
}

#[allow(unused)]
fn main() {
    // Lowercase conversion not needed.
    let x = "my_string";
    let x_lowercase = make_lowercase(x);

    // Lowercase conversion needed.
    let y = "My_String";
    let y_lowercase = make_lowercase(y);
}
```

To be honest, I still fully do not understand the details of `Cow` and I rarely use it in my own code. However, I'm sure it is useful.

File Formats

In bioinformatics, there are two commonly encountered file formats, FASTA and FASTQ. They both store biological sequences but contain different amounts of information. In this chapter, we'll cover both formats briefly.

FASTA

The FASTA format is a standardized way of storing biological sequences such as nucleotides and aminoacids. Each record occupies two lines:

1. Sequence id with an optional description. Must start with the `>` character.
2. Actual sequence.

A simple example of this is the following record:

```
>sequence_1
ATCG
```

which tells us that there is a record called `sequence_1` with the sequence `ATCG`. In this particular example, we actually do not know if these are nucleotides or aminoacids.

Multi FASTA format

There is an alternative to the canonical FASTA format that is commonly referred to as multi FASTA format. Essentially what this means is distributing the sequence over multiple lines of a defined width. E.g., 60 characters per line.

For example, assume we have an arbitrary sequence of length 180. With width 60, it would look something like this.

```
>sequence_1
ATCG...AGAC # End of bases 1-60.
AGAG...TTTA # End of bases 61-120.
ACGC...AATG # End of bases 121-180.
```

FASTQ

The FASTQ format is similar to FASTA, but also associates each nucleotide with an error probability. This is relevant because it enables us to do things such as trimming, filtering and estimating sample quality. Each record takes up four lines:

1. Sequence id with an optional description. Must start with the `@` character.
2. Actual sequence.
3. `+`.
4. Quality (ASCII encoded phred scores).

In addition, the sequence and quality lines must contain the same number of characters. One simple example of a FASTQ record is:

```
@sequence_1
ATCG
+
????
```

which tells us that there is a record called `sequence_1` with the nucleotide sequence `ATCG` and associated qualities `????`. How do we know that these are nucleotides and not aminoacids? Strictly, we don't. However, the FASTQ format is almost exclusively used for nucleotides.

> **Tip**
>
> The quality line in a FASTQ file uses ASCII-encoded phred scores. To understand how to convert characters like `?` to error probabilities, see the Phred Score chapter.

Phred Score

Within bioinformatics, phred quality scores are used in FASTQ files to estimate the error probability for each base.

There are three concepts we need to understand before proceeding:

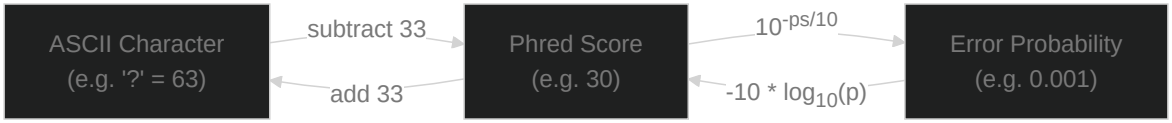
- `Error Probability` - The probability that a particular nucleotide was called incorrectly by the sequencing machine. For example, a nucleotide `A` with error probability `0.001` means there is a 0.1% likelihood that this `A` is actually something else, like a `C`, `G` or `T`. There are two issues with this approach:
    - Assigning a particular error probability to a single called nucleotide might be a bit misleading. The nucleotide is either correct, or it is not. There is no "in between". Consider the analogy of blindly throwing a die and guessing the outcome. Once the die has stopped rolling, it has a particular value (1-6). When making a guess, you can either be 100% wrong, or 100% right. However, if you **repeat** the experiment enough times, you'll be right about `1/6` of the times.
    - Error probabilities are a bit incomplete. For example, how do we estimate a deleted nucleotide? It won't be present in the FASTQ file (because it is deleted) so we cannot assign an error probability to this. To be honest, I'm not sure how this is handled (if it even is) by the sequencing machine.

> **Note**
>
> Error probabilities are statistical estimates across many sequencing events, not absolute truths about individual bases. A base with error probability 0.001 is either correct or incorrect — the probability reflects how often similar bases are miscalled across many reads. Additionally, phred scores only capture substitution errors. Deletion and insertion errors are not represented in the FASTQ quality line, since deleted bases are absent from the read entirely.

- `Phred Score` - Is a logarithmically encoded error probability, expressed as an integer. E.g., a phred score of `30` corresponds to an error probability of `0.001`. Why do we care about phred scores? We don't want to include a bunch of floating point numbers in our FASTQ file because we'll run into issues such as rounding, etc.

- `ASCII Quality` - The ASCII character associated with the error probability, and hence, the phred score of a particular nucleotide. This is what is found in the actual FASTQ file. Why an ASCII character? Because they are fixed length characters (of length 1). This gives a very nice mapping of `one nucleotide -> one ASCII` quality value. The conversion between `ASCII` and `phred score` uses a phred score offset. The reason is that the first 31 ASCII characters are non-printable and the 32nd is the space character `' '`. To account for the fact that our "zero" or lowest quality value starts as ASCII character 33, an offset of 33 is commonly used. E.g., the ASCII character `!` has value 33 equates to a phred score of `33 - 33 = 0`.

Since ASCII, phred scores and error probabilities are related, we can convert between them.



A Quick Look At The Maths

Error Probability

We won't go through the details about the maths regarding phred scores and error probabilities, but the equality looks something like this:

$$error\_probability = 10^{-(ASCII - \text{phred\_offset})/10}$$

where

$$phred\_score = ASCII - \text{phred\_offset} = ASCII - 33$$

We can now test this formula. Assume we'd want to convert `?` to an error probability using `phred_offset = 33`. Since the ASCII value of `?` is `63`, this would equate to:

$$error\_probability = 10^{-(63-33)/10} = 10^{-3} = 0.001$$

Phred Score

Similarly, we can get our `phred_score` through

$$\text{phred\_score} = -10 * log_{10}(\text{error\_probability})$$

E.g., for an error probability of 0.001, we get

$$\text{phred\_score} = -10 * log_{10}(0.001) = 30$$

which would give an ASCII value of `30 + 33 = 63`, or `?`

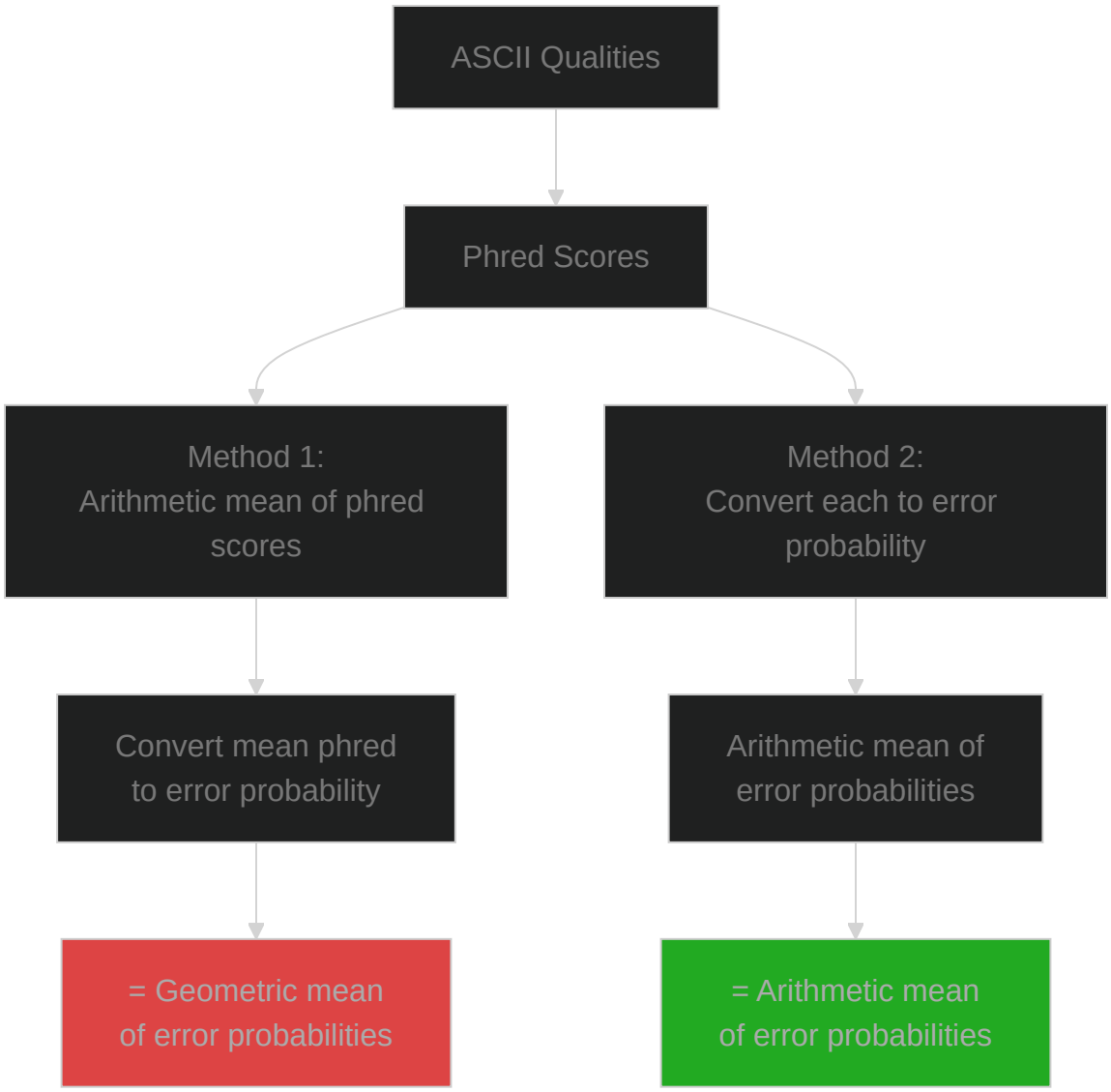
> **Tip**
>
> Now that we understand the relationship between ASCII, phred scores, and error probabilities, the next chapter covers a surprisingly subtle topic: how to correctly calculate mean error probabilities.

Calculating Average Errors

This is (at least according to me) actually a surprisingly interesting topic.

Assume we have a FASTQ record with only three nucleotides with the corresponding ASCII qualities `???`. How do we calculate the mean error probability across these nucleotides? There are basically two different options:

- Convert to phred scores `[30, 30, 30]`, calculate the mean `(30 + 30 + 30) / 3 = 30` and convert to error probability `10^(-30/10) = 10^(-3) = 0.001`.
- Convert all the way to error probabilities first `[0.001, 0.001, 0.001]` and then calculate the mean `(0.001 + 0.001 + 0.001) / 3 = 0.001`.



In this example, we get the same result. This is, however, not always the case. Consider an alternative sequence of only two nucleotides with ASCII qualities `+5`. Our two options give:

- `[10, 20]` -> mean is `(10 + 20) / 2 = 15` which gives an error probability of `10^(-15/10) ≈ 0.0316`.
- `[0.1, 0.01]` -> mean is `(0.1 + 0.01) / 2 = 0.055`.

All of a sudden, the results are quite different based on the method we choose. How do we know which is correct?

Different Kinds Of Means

To investigate this, we need to understand that there are different ways of calculating means, neither of which is incorrect.

Arithmetic Mean

The most common way is the `arithmetic mean`, which has the famous formula

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

E.g., for `n=2` this would give `1/2 * (1 + 2) = 3/2 = 1.5`.

Geometric Mean

A less common, but equally valid way to calculate means is the `geometric mean`:

$$\bar{x} = \sqrt[n]{\prod_{i=1}^{n} x_i} = \prod_{i=1}^{n} x_i^{1/n}$$

E.g., for `n=2` this would give `(1 * 2)^(1/2) = √2 ≈ 1.414`.

Putting It All Together

Our two approaches for calculating mean error probabilities, as defined in the start of this chapter, both use the `arithmetic mean`. However, the first approach has a fundamental flaw. We are calculating the `arithmetic mean` of log-encoded values that are not equidistant from each other with respect to error probabilities. As an example, the difference between phred scores `10`

and `20` (with respect to error probabilities) is `0.1 - 0.01 = 0.09` whilst the difference between `20` and `30` is `0.01 - 0.001 = 0.009`.

Mathematically, we can derive the following formulas for our two approaches. Assume we have a set of phred scores

$$ps_1, ps_2, \ldots, ps_n$$

- In the first method, we calculate a mean phred score and finally convert to an error probability.

  $$\text{mean\_error\_probability} = 10^{-\bar{ps}/10}$$

  where

  $$\bar{ps} = \frac{1}{n}\sum_{i=1}^{n} ps_i$$

  which gives

  $$\text{mean\_error\_probability} = 10^{-\frac{1}{n}\sum_{i=1}^{n} ps_i/10} = \left(10^{\sum_{i=1}^{n} -ps_i/10}\right)^{1/n} =$$

  This is the **geometric mean** of the individual error probabilities!

- In the second method, we first convert each phred score to an error probability and then calculate the arithmetic mean.

  $$\text{mean\_error\_probability} = \frac{1}{n}\sum_{i=1}^{n} 10^{-ps_i/10}$$

  This is simply the **arithmetic mean** of the individual error probabilities.

Which Mean To Choose

The natural question is which mean to choose. I'd argue that the arithmetic mean is correct, and here's why.

One way to think about this is: the **expected number of errors** in a read is the sum of all individual error probabilities.

To illustrate with a concrete example, consider a read of length `100` where `50` bases have phred score `20` and the other `50` have phred score `30`:

$$\underbrace{20, \ldots, 20}_{50}, \underbrace{30, \ldots, 30}_{50}$$

Converting to error probabilities:

$$\underbrace{0.01, \ldots, 0.01}_{50}, \underbrace{0.001, \ldots, 0.001}_{50}$$

The expected number of errors in this read is:

$$50 \times 0.01 + 50 \times 0.001 = 0.5 + 0.05 = 0.55$$

So the mean error probability per base should be `0.55 / 100 = 0.0055`. Now let's see what our two methods give:

- **Geometric mean (method 1):** `mean_phred = (20 * 50 + 30 * 50) / 100 = 25`, so `mean_error_probability = 10^(-25/10) ≈ 0.0032`. This predicts `0.32` expected errors — an **underestimate**.
- **Arithmetic mean (method 2):** `mean_error_probability = (0.01 * 50 + 0.001 * 50) / 100 = 0.0055`. This predicts `0.55` expected errors.

The geometric mean systematically underestimates the error rate because it is always less than or equal to the arithmetic mean (this is known as the AM-GM inequality). In practice, this means that if you average phred scores directly, you will be overly optimistic about your data quality.

Nucleotides

When dealing with DNA sequences, nucleotides are everything. Fundamentally, we are usually dealing with four canonical bases:

- `A` - Adenine.
- `C` - Cytosine.
- `G` - Guanine.
- `T` - Thymine.

However, there are some important concepts to be aware of.

Soft masking

A lower case nucleotide indicates *soft masking* and is used to indicate a soft clipped alignment or a region which is low complexity or repetitive.

In the example below, the last part of the upper sequence is soft masked to indicate that this region is not part of the actual alignment. This is commonly encountered in local read aligners such as Minimap2.

```
AAAGTGCCAGTGACGCTTagtcgatcgatg
||||||||||||||||||
AAAGTGCCAGTGACGCTT
```

Hard masking

A capital `N` indicates *hard masking*. This means there is probably a base here, but we don't know exactly what it is. This is usually for indicating uncertainty or gaps in a sequence.

Ambiguous nucleotides

In addition to our four canonical nucleotides, there are also ambiguous nucleotides. Ambiguous in this case, means uncertainty or ambiguity:

- `R = A | G`
- `Y = C | T`
- `K = G | T`
- `M = A | C`
- `S = G | C`
- `W = A | T`
- `H = A | C | T`
- `V = A | C | G`
- `B = C | G | T`
- `D = A | G | T`
- `N = A | C | G | T`

We won't deal much with ambiguous nucleotides in this book. However, make sure not to confuse these nucleotides with one-letter amino acid abbreviations, which have overlapping naming conventions.

Programmatic representations

There are many different ways to represent nucleotide sequences in a programming language. In this book, we'll mainly deal with these different representations:

- `String` and `&str`.
- `&[u8]` (byte slice).
- Binary.

```
/// Assume we want to represent the sequence ATCG.
fn main() {
  let nt_seq_string: String = "ATCG".to_string();
  let nt_seq_str: &str = "ATCG";
  let nt_seq_byte_slice: &[u8] = b"ATCG";
  let nt_seq_binary: u8 = 0b00110110; // Binary, where A = 00, T =
11, C = 01 and G = 10.

  println!("{}", nt_seq_string);
  println!("{}", nt_seq_str);
  println!("{:?}", nt_seq_byte_slice);
  println!("{:08b}", nt_seq_binary);
}
```

Create A nucleotide sequence

String

There are many different string types in Rust, but the two most common ones are `String` and `&str`. Both can be used to store nucleotide sequences, but they have different characteristics. Usually, use `String` if you intend to mutate the sequence, otherwise use `&str`. For more information, visit the rust docs for `String` and `&str` respectively.

```
fn main() {
    let nt_string: String = "ACGT".to_string();
    let nt_string: &str = "ACGT";
}
```

Byte slice

Usually when reading nucleotide sequences from a FASTA/Q file, we get it as a byte slice, `&[u8]`, which is a more convenient format.

```
fn main() {
    let nt_string: &[u8] = b"ACGT";

    println!("{:?}", nt_string);
}
```

Run the code and examine the output. We get a bunch of numbers. This is the ASCII representation of our nucleotides, where `A/T/C/G` corresponds to an 8-bit representation. For more information, visit this link.

We can check that the following representations are equivalent:

```rust
fn main() {
    assert_eq!(b'A', 65);
    assert_eq!(b'C', 67);
    assert_eq!(b'G', 71);
    assert_eq!(b'T', 84);
}
```

Binary

> **Tip**
>
> Binary encoding of nucleotides is covered in detail in the Encoding chapter, and becomes essential for the Kmer chapters where bit-shift operations enable extremely efficient kmer generation.

In short, using 8-bits is overkill for representing only four nucleotides. Instead, we can map `A/C/G/T` to the corresponding binary representation:

- `A => 00`
- `C => 01`
- `G => 10`
- `T => 11`

Counting nucleotides

We'll start off with something relatively easy - counting nucleotides. We'll create a `HashMap` for storing the counts of the nucleotides we encounter in the sequence.

```
use std::collections::HashMap;

fn count_nucleotides(seq: &[u8]) {
    // Note: HashMap::with_capacity(4) would be better here since we
know
    // there are at most 4 canonical nucleotides. See the Increasing
Performance chapter.
    let mut map: HashMap<&u8, usize> = HashMap::new();

    // Iterate over each nucleotide.
    seq.iter().for_each(|nt| match nt {
        // If we have a canonical nucleotide, we bind it to the
variable c.
        c @ (b'A' | b'C' | b'G' | b'T') => match map.contains_key(c)
{
            // If nucleotide is already in HashMap, increment its
count.
            true => {
                let v = map.get_mut(c).unwrap();
                *v += 1;
            }
            // Otherwise, add it.
            false => {
                map.insert(c, 1);
            }
        },
        _ => panic!("Invalid nt {nt}"),
    });

    assert_eq!(map.values().sum::<usize>(), seq.len());
    println!("{:?}", map);
}

fn main() {
    count_nucleotides(b"ATCG");
    count_nucleotides(b"AAAA");
}
```

Run the code and inspect the output. The resulting HashMap will have the ASCII encoded nucleotides as keys.

Note that there are lots of alternative solutions and further optimizations we can do. For example, when we input `b"AAAA"` we see that our `HashMap` only contains one key, `b'A'`. One alternative here would be to initialize the `HashMap` with empty counts for A, T, C and G.

> **Tip**
>
> For a more efficient approach to nucleotide counting using fixed-size arrays instead of HashMaps, see the Using Appropriate Data Structures chapter.

GC content

With the previous section in mind, it is relatively straightforward to implement a function that calculates the GC-content for a given nucleotide sequence.

```
fn gc_content(nt_seq: &[u8]) -> f32 {
    let gc_count = nt_seq.iter().filter(|&&nt| {
        nt == b'C' || nt == b'G'
    }).count();

    return gc_count as f32 / nt_seq.len() as f32;

}

fn main() {
   assert_eq!(gc_content(b"ATCG"), 0.5);
   assert_eq!(gc_content(b"ATTC"), 0.25);
   assert_eq!(gc_content(b"AAAA"), 0.0);
   assert_eq!(gc_content(b"CGCGCG"), 1.0);
}
```

In this code example, we use the filter method to count the number of Gs and Cs. This is not necessarily the fastest way, but it works for now.

Also, note that we are only filtering for uppercase G and C. In a real life application, we'd probably also check for `b'c'` and `b'g'`, e.g., softmasked bases.

Homopolymers

A *homopolymer* is a defined stretch of consecutive, identical nucleotides. One example is the sequence below, in which there is a A-homopolymer of length 7.

Why are homopolymers important? Biologically, they play a role in multiple functions, such as promoters and other regulatory regions. In addition, homopolymer regions have been shown to introduce systematic errors in Oxford Nanopore data, which is why these regions are important to identify and inspect.

In the code-snippet below, we implement a simple function for identifying homopolymer regions of a defined minimum length in a sequence.

```
fn find_homopolymers(seq: &[u8], min_hp_len: usize) -> Vec<&[u8]> {
    let mut homopolymers: Vec<&[u8]> = Vec::new();

    let seq_len = seq.len();

    // Skip checking if seq length is too short.
    if seq_len < min_hp_len {
        return homopolymers
    }

    let mut i = 0;
    let mut j = 1;

    // We only need to check homopolymers until our start index
    // is closer than min_hp_len to the end of the sequence.
    while i <= seq_len - min_hp_len {
        while j < seq_len && seq[j] == seq[i] {
            j += 1;
        }

        // We have a homopolymer of required length.
        if j - i >= min_hp_len{
            homopolymers.push(&seq[i..j]);
        }

        i = j;
        j += 1;
    }


    return homopolymers
}

fn main() {
    // Find all homopolymers of length >= 5.
    assert_eq!(find_homopolymers(b"AAAAA", 5), vec![b"AAAAA"]);

    // Find all homopolymers of length >= 3.
    assert_eq!(find_homopolymers(b"AAACCCTTTGGG", 3), vec![b"AAA",
b"CCC", b"TTT", b"GGG"]);

    // Find all homopolymers of length >= 5.
    assert_eq!(find_homopolymers(b"ATCGAAAAAAAAAAGCTA", 5), vec!
[b"AAAAAAAAAA"]);

    // Find every nucleotide (makes no sense).
    assert_eq!(find_homopolymers(b"ATC", 1), vec![b"A", b"T",
```

```
b"C"]);

}
```

In a real life application, we'd most likely do more than this such as saving the positions for identified homopolymers.

Entropy

Typically, one associates entropy with the physical term for the measure of disorder. In bioinformatics, entropy can be used for quantifying the diversity or randomness of nucleotide sequences.

Although there are different kinds of entropies, the *Shannon* entropy is probably the most famous one. It is defined by the following equation:

$$-\sum_{i=\{A,T,C,G\}} p_i \cdot log_2(p_i) \quad log_2(p_i) = \begin{cases} log_2(p_i) \; if \; p_i > 0 \\ 0 \; if \; p_i \; == \; 0 \end{cases}$$

That is, we calculate the proportions of each nucleotide {A, T, C, G} and calculate the sum of the probability times its logarithm. For example, consider the sequence “AAAAA”. Calculating the Shannon entropy would result in:

$$-(1 \cdot log_2(1) + 0 + 0 + 0) = 0$$

Which tells us there is very little disorder or randomness. This makes sense, because we have the same nucleotide repeated five times. The code snippet below implements the Shannon entropy for a given nucleotide sequence. We’ll reuse the previous code for counting nucleotides (with a few modifications) and add the entropy calculation.

```rust
// [...]

fn shannon_entropy(counts: &HashMap<u8, usize>) -> f32 {
    let sum_count: usize = counts.values().sum();

    // Probabilities of each nucleotide.
    let probs: Vec<f32> = counts
        .values()
        .map(|count| (*count as f32 / sum_count as f32))
        .collect();

    let shannon: f32 = probs
        .iter()
        .map(|prob| match prob {
            0_f32 => return 0 as f32,
            // This is safe because prob is never negative since
            // both count and sum_count are of type usize.
            _ => {
                return prob * prob.log2();
            }
        })
        .sum();

    return -shannon;
}

fn get_shannon_entropy(seq: &[u8]) -> f32 {
    let counts = count_nucleotides(seq);
    let shannon = shannon_entropy(&counts);

    return shannon;
}

fn main() {
    assert_eq!(get_shannon_entropy(b"AAAAA"), 0.0_f32);
    assert_eq!(get_shannon_entropy(b"ATCG"), 2.0_f32);
    assert_eq!(get_shannon_entropy(b"ATCGATCGATCG"), 2.0_f32);
    assert_eq!(get_shannon_entropy(b"AAAAAAG"), 0.5916728_f32);
}
```

Manipulating

The art of manipulating nucleotide sequences has numerous applications. Essentially what it means is somehow changing or modifying the sequence to better fit the application at hand. In this chapter, we'll get acquainted with two different methods: `compression` and `reverse complement`.

Compression

Compression algorithms have been around for a long time. Examples span everything from Huffman encoding (used in e.g., `gzip`) to Burrows Wheeler. In this section, we'll just cover some very basic ways of compressing a nucleotide sequence.

One very straightforward way to implement nucleotide compression would be to save how many times the same nucleotide appears in a row. E.g., `ATAAAAAGGCGCTTTA -> AT5A2GCGC3TA`. Since we preserve the order, this compression is easily reversible.

Another way of compression is nucleotide encoding, which is covered in more detail later on. It turns out that if we only allow `{A, C, G, T, a, c, g, t}`, we can map each base to 2 bits. E.g., `ATCG -> 00110110`, which is extremely efficient when generating kmers.

Homopolymer Compression

> **Tip**
>
> Homopolymers are discussed in more detail in the dedicated Homopolymers chapter, which covers identification and why they matter for sequencing technologies like Oxford Nanopore.

A similar, non reversible approach is homopolymer compression. We define a length `k`, at which we cap the maximum number of allowed adjacent identical nucleotides. E.g., with `k=3` we get `ATAAAAAGGCGCTTTA -> ATAAAGGCGCTTTA` and with `k=1` we get `ATAAAGGCGCTTTA -> ATAGCGCTA`. If we also store positional information, we can make this compression reversible.

In the code example below, we implement the non-reversible version of homopolymer compression for `k=1`, inspired by isONclust3.

```rust
fn homopolymer_compression(seq: &[u8]) -> String {
    let mut hp_compressed = String::new();

    let mut previous: Option<&u8> = None;

    for nt in seq {
        // We are safe to unwrap because we checked if previous is
None.
        if previous.is_none() || previous.unwrap() != nt {
            hp_compressed.push(*nt as char);
        }
        previous = Some(nt);
    }

    return hp_compressed;
}

fn main() {
    assert_eq!(homopolymer_compression(b"").as_bytes(), b"");
    assert_eq!(homopolymer_compression(b"AAAAAAAAAAAAA").as_bytes(),
b"A");
    assert_eq!(homopolymer_compression(b"AATTCCGG").as_bytes(),
b"ATCG");
    assert_eq!(homopolymer_compression(b"AAATTTTTT").as_bytes(),
b"AT");
}
```

We can improve on this idea slightly to allow for arbitrary numbers of `k`. Instead of just checking if the previous nucleotide is the same as the current, we keep track of the number of adjacent nucleotides and write maximally `k` identical, adjacent nucleotides to our string.

```rust
fn homopolymer_compression(seq: &[u8], k: usize) -> String {
    assert!(k > 0, "value of k must be > 0.");

    let mut hp_compressed: String = String::new();
    let mut i: usize = 0;

    while i < seq.len() {
        let mut j = i + 1;

        while j < seq.len() && seq[j] == seq[i] {
            j += 1;
        }

        for _ in 0..std::cmp::min(j-i, k){
            hp_compressed.push(seq[i] as char);
        }
        i = j;
    }

    hp_compressed
}

fn main() {
    assert_eq!(homopolymer_compression(b"AAAAAAAAAAAA",
1).as_bytes(), b"A");
    assert_eq!(homopolymer_compression(b"AAAAAAAAAAAA",
2).as_bytes(), b"AA");
    assert_eq!(homopolymer_compression(b"AAATTTCCCGGG",
1).as_bytes(), b"ATCG");
    assert_eq!(homopolymer_compression(b"AAATTTCCCGGG",
2).as_bytes(), b"AATTCCGG");
    assert_eq!(homopolymer_compression(b"AAATTTCCCGGG",
3).as_bytes(), b"AAATTTCCCGGG");
    assert_eq!(homopolymer_compression(b"AAATTTCCCGGG",
100).as_bytes(), b"AAATTTCCCGGG");
}
```

Reverse complement

Next, we cover a fundamental topic, which is reverse complementing. Why is this important?

DNA is (generally) double stranded, where bases are paired:

- A pairs with T and vice versa.
- G pairs with C and vice versa.

```
5' [...]ACGAGCTTTGTGACGCGATGCGACGAGCTGCAGCGT[...] 3'
3' [...]TGCTCGAAACACTGCGCTACGCTGCTCGACGTCGCA[...] 5'
```

Pretend this is a bacterial genome we want to sequence. Before sequencing, we need to separate the strands and break this molecule into smaller pieces. When doing this, we don't know which pieces are from which strand.

When we want to align the pieces back to a reference sequence (which is defined in the 5' to 3' direction), we need to take both strands into consideration. Otherwise, we lose out on information. We do this by reverse complementing, in which we first reverse the sequence, and then replace each base with the corresponding matching base.

```rust
fn reverse(nt: &u8) -> u8 {
    match nt {
        b'A' => b'T',
        b'C' => b'G',
        b'G' => b'C',
        b'T' => b'A',
        _ => panic!("Invalid nt {nt}"),
    }
}

fn reverse_complement(nt_string: &[u8]) -> Vec<u8> {
    let rev_comp: Vec<u8> = nt_string
        // Iterate over each character.
        .iter()
        // Reverse the iteration order.
        .rev()
        .map(|nt| {
            return reverse(nt);
        })
        .collect();

    return rev_comp;
}

fn main() {
    assert_eq!(reverse_complement(b"AAA"), b"TTT");
    assert_eq!(reverse_complement(b"GGG"), b"CCC");
    assert_eq!(reverse_complement(b"ATCG"), b"CGAT");
    assert_eq!(reverse_complement(b"ACACGT"), b"ACGTGT");
}
```

> **Tip**
>
> A more elegant approach to reverse complementing uses 2-bit nucleotide encoding, where the complement of a base is simply `3 - encoded_value`. This is covered in the Encoding chapter and becomes critical for efficient kmer generation.

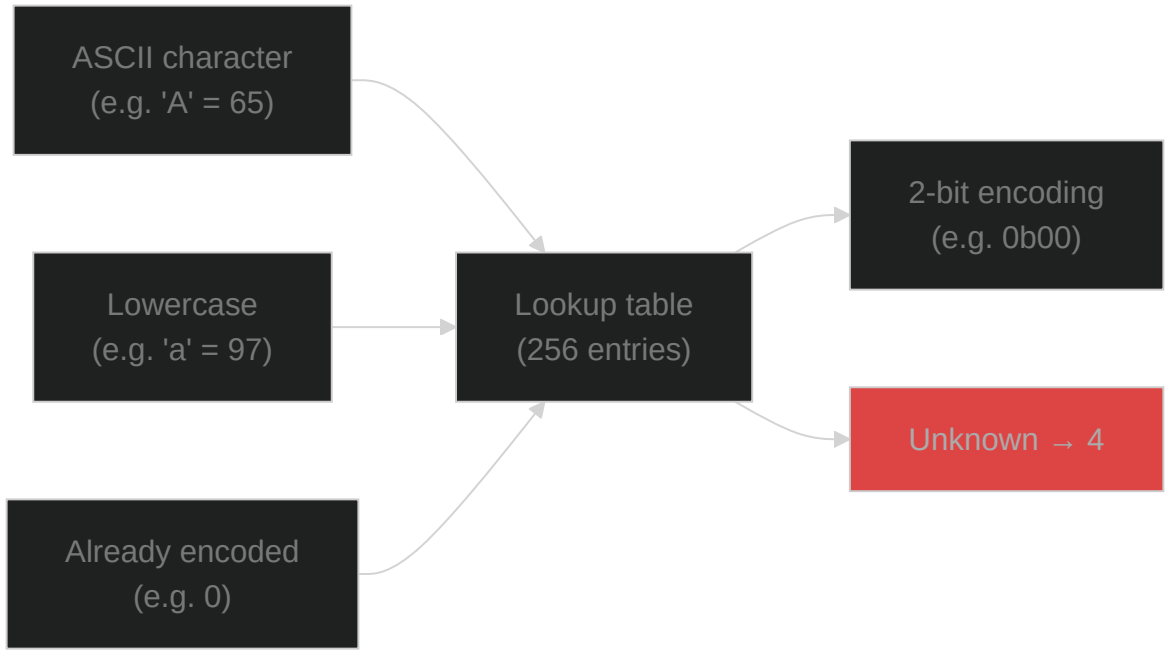
Encoding

Being able to encode and decode nucleotides is a vital part of writing high performance bioinformatic code. What it means is essentially converting nucleotides into a more compact form. There are multiple ways of doing nucleotide encoding. However if we assume we only have to deal with `{A,C,G,T}` then there is a straightforward way for this:

- `A` is encoded as `0` (binary `00`).
- `C` is encoded as `1` (binary `01`).
- `G` is encoded as `2` (binary `10`).
- `T` is encoded as `3` (binary `11`).

The advantages of this approach are:

- Each nucleotide only takes up 2 bits.
- Reverse complementing a base is as easy as:
  - `rev_nt = 3 - nt`
- With some bit-shifting, we can very efficiently generate kmers from our sequences (covered in a later topic).



The following code is just a very straightforward encoding/decoding protocol. However, this enables us to do some more advanced stuff in future topics.

```
/// Convert ASCII to our own 2-bit encoded nt.
fn encode(nt_decoded: u8) -> u8 {
    match nt_decoded {
        b'A' => 0,
        b'C' => 1,
        b'G' => 2,
        b'T' => 3,
        _ => panic!("Invalid nt {nt_decoded}"),
    }
}

/// Convert our own 2-bit encoded nt to ASCII.
fn decode(nt_encoded: u8) -> u8 {
    match nt_encoded {
        0 => b'A',
        1 => b'C',
        2 => b'G',
        3 => b'T',
        _ => panic!("Invalid nt {nt_encoded}"),
    }
}

/// Reverse complement an ASCII base.
fn reverse(nt: u8) -> u8 {
    return decode(3 - encode(nt));
}

/// Reverse complement a nucleotide sequence.
fn reverse_complement(nt_string: &[u8]) -> Vec<u8> {
    nt_string.iter().rev().map(|nt| reverse(*nt)).collect()
}

fn main() {
    // Reverse complement a single nucleotide.
    assert_eq!(reverse(b'A'), b'T');
    assert_eq!(reverse(b'T'), b'A');
    assert_eq!(reverse(b'C'), b'G');
    assert_eq!(reverse(b'G'), b'C');

    // We can also reverse complement a nucleotide sequence.
    assert_eq!(reverse_complement(b"AAAA"), b"TTTT");
    assert_eq!(reverse_complement(b"ATCG"), b"CGAT");
}
```

Using a lookup table

We used match statements to encode and decode nucleotides, which works. However, we only handle the canonical bases `{A,C,G,T}`. This is not ideal, because our FASTA/Q file might contain soft masked bases `{a,c,g,t}` or hard masked bases `N`.

We could just extend our match statement to handle this, but we still have not safe-guarded against any other ambiguous nucleotide that we might encounter. A better approach is to use a compile-time lookup table that supports all 256 ASCII characters, where all irrelevant characters are set to 4.

```
const LOOKUP_TABLE: [u8; 256] = [
    0, 1, 2, 3,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 0, 4, 1,  4, 4, 4, 2,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  3, 3, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 0, 4, 1,  4, 4, 4, 2,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  3, 3, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4
];

fn main() {
    LOOKUP_TABLE.iter().enumerate().for_each(|(index, value)| {
        if *value != 4 {
            println!("[{index}] {} -> {}", index as u8 as char,
value);
        }
    });
}
```

Run the code and inspect the output. Using a lookup table, we are able to map `{A,C,G,T,U,a,c,g,t,u}` to their corresponding encodings. This means we

handle both upper and lowercase nucleotides, and also get U/u for free, meaning that we can now handle RNA as well.

However, we see that the first four values at index `[0], [1], [2], [3]` map to some weird characters. ASCII characters less than 32 are not actually printable characters, but rather control characters where `0, 1, 2, 3` correspond to null, start of heading, start of text and end of text respectively.

That does not make any sense. However, it does enable us to map already encoded nucleotides to themselves, which could serve as some kind of redundancy if we ever would have a mix of encoded and non-encoded nucleotides.

```
// [...]

fn main() {
    // We have a mix of non-encoded and encoded nucleotides.
    let mix_encoded: &[u8] = &[65, 65, 0, 0]; // AAAA

    // Encode all nucleotides.
    let all_encoded: Vec<u8> = mix_encoded.iter().map(|nt|
LOOKUP_TABLE[*nt as usize]).collect();

    assert_eq!(all_encoded, vec![0, 0, 0, 0]);
}
```

Basics Of Alignment

Introduction

In bioinformatics, alignment is the process of determining how well biological sequences match to each other. Usually, we refer to the sequences as the `query` and `subject` respectively. For simplicity, we'll assume that the `query` and `subject` both are single sequences.

There are three important alignment features to understand:

- Match.
- Mismatch.
- Insertion/Deletion.

In the following alignment, matches are shown with a vertical bar `|`, mismatches as asterisks `*` and insertions or deletions as hyphens `-`.

Definitions

- `Query length` - The length of the query sequence. Here, we need to be a bit careful about if we mean either the original query length, or the length of the aligned part of the query.
- `Subject length` - The length of the subject sequence (either original or aligned, same reasoning as for query).
- `Alignment length` - The length of the aligned part between the query and subject.

- `Percent identity` - `100 * (num_matches / alignment_length)`. Here, we also need to be a bit careful since this metric only considers the aligned part of the query and subject. Theoretically, if our query and subject are of length 100, but they align only in the first 10 bases with no mismatches, this would be `percent identity = 100 * ( 10 / 10 ) = 100`.

- `Fraction aligned (query)` - `alignment_length / query_length` (how much of the query is aligned). Here, we use the original query length.

- `Fraction aligned (subject)` - `alignment_length / subject_length` (how much of the subject is aligned). Here, we use the original subject length.

In the example below, we have the following alignment metrics:

- `Query length = 6` (original) or `4` (aligned).
- `Subject length = 4` (original and aligned).
- `Alignment length = 4`.
- `Percent Identity = 100`.
- `Fraction aligned (query) = 4 / 6 = 0.67`.
- `Fraction aligned (subject) = 4 / 4 = 1.0`.

```
query   AGCGACTCGTGCTCGA-CTT
        || |||||*|||||| | |
subject A-CGAGTCGAGCTCGAGCTT
```

Types Of Alignments

There are three basic types of alignments:

- `Global Alignment` - Aligns the entire query against the entire subject. Suitable if query and subject are of similar length, or one expects the entire query to align against the entire subject. An example is aligning two very similar genomes of roughly the same length.

- `Semi Global Alignment` - Fully aligns the shorter of query/subject. An example is trying to align a gene (shorter) against an entire genome (longer).

- `Local Alignment` - Allows partial alignment of the query against the subject. This is the type of alignments that BLAST outputs.

```
query   CATCGT
         ||||
subject  ATCG
```

```
        ATCGATCG
        ||||||||
        ATCGATCG
```

Hamming Distance

The Hamming distance is defined as the number of positions between two strings of equal length that are different. Hence, it measures the number of substitutions needed to convert one string to the other. This also means that it is a kind of `global alignment` that only supports substitutions.

```
GCGATCGTTT
 ||||
 ATCG
```

In the example below, the Hamming distance is 1.

```
CCCATCGTTT
   ||||
GGGATCGAAA
```

```
use std::iter::zip;

fn hamming_distance(query: &str, subject: &str) -> usize {
    assert_eq!(query.len(), subject.len());

    let mut distance = 0;

    for (query_nt, subject_nt) in zip(query.chars(),
subject.chars()) {
        if query_nt != subject_nt {
            distance += 1;
        }
    }

    return distance;
}

fn main() {
    assert_eq!(hamming_distance("ATCG", "ATCG"), 0);
    assert_eq!(hamming_distance("ATCG", "TTCG"), 1);
    // Our function can actually handle non-nucleotide strings.
    assert_eq!(hamming_distance("Hello", "Heiol"), 3);
}
```

Edit Distance

Introduction

In contrast to the Hamming distance, the Edit distance allows for the query and subject to be different lengths, but it is still a global alignment. There are multiple kinds of Edit distances, of which the Levenshtein distance is probably the most common. This distance allows for insertions, deletions and substitutions and is highly suitable for alignment of biological sequences.

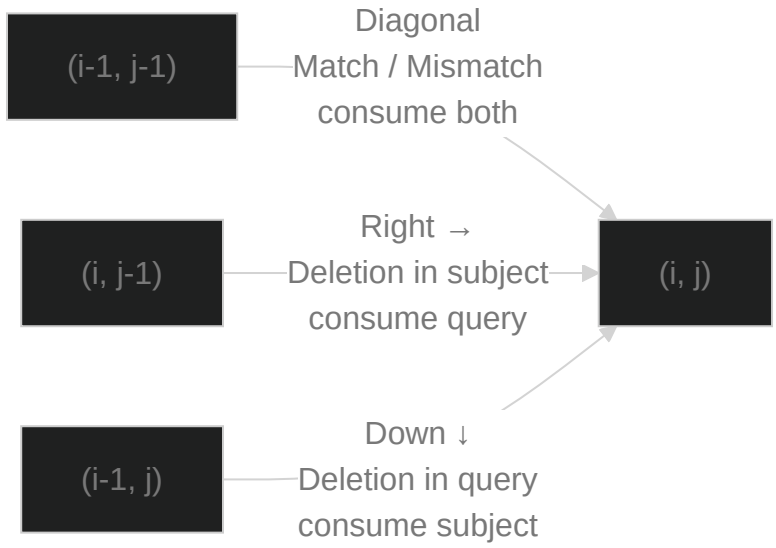
```
query    ATCTACCG
         ||||*-||
subject  ATCTATCG
```

The implementation of the Levenshtein distance is more complex than the Hamming distance and requires us to use some dynamic programming. The goal is to use as few insertions, deletions and substitutions as possible to turn one string into the other. We need to be *exhaustive* since we do not know beforehand what is the optimal solution to this problem.

Setup

To solve this problem, we need a two dimensional array (two dimensions because we have two sequences). The layout will look something like this:

We want to traverse this array from the start, until we have used up the entire query and subject. We can do this in three different ways:



- A diagonal step means we take a step in both the query and subject (we "consume" both the query and the subject). Depending on the nucleotide we have in the current column (query) and row (subject), this is either a *match* or a *mismatch*.
- A step to the right means we take a step only in the query direction (we "consume" the query). This signifies a *deletion* in the subject sequence (we don't consume it).
- A step downwards means we take a step only in the subject direction (we "consume" the subject). This signifies a *deletion* in the query sequence (we don't consume it).

```
  A   T   C   G (query)
A
T
C
G
(subject)
```

The following path corresponds only to matches between the query and the subject:

```
A   -   -   -

T   -   -   -

C   -   -   -

G   -   -   -

(subject)
```

With the equivalent alignment:

The following path corresponds to another alignment between the query and the subject:

With the equivalent alignment:

With this in mind, how do we choose the best alignment? We clearly see that the first alignment is a lot better than the second. However, we need to quantify this somehow. The solution is to associate costs:

- Match costs 0.
- Mismatch costs 1.
- Insertion/Deletion costs 1.

The goal is to minimize this cost to generate the best alignment possible.

We can make another observation, which is that for a given position `(i, j)` the value in this position, `array[i][j]`, is dependent on the three adjacent values `array[i-1][j-1]`, `array[i][j-1]` and `array[i-1][j]`. Namely, we want the minimum value from either of these three values, plus the additional cost to get to `(i, j)`, which can be match, mismatch, insertion or deletion:

```
array[i][j] = min(array[i-1][j-1] + cost_of_match_or_mismatch,
                  array[i][j-1] + cost_of_insertion_deletion,
                  array[i-1,j] + cost_of_insertion_deletion)
```

```
        ||||
subject ATCG
```

Here, we might realize something. What if there are multiple alignments that generate the same final score? This depends on how we define the costs for matches, mismatches and insertions/deletions AND actually on the order of the arguments in the min function. Min functions usually return the first minimum value if there is a tie. This is something to keep in mind. For now however, we'll just ignore this.

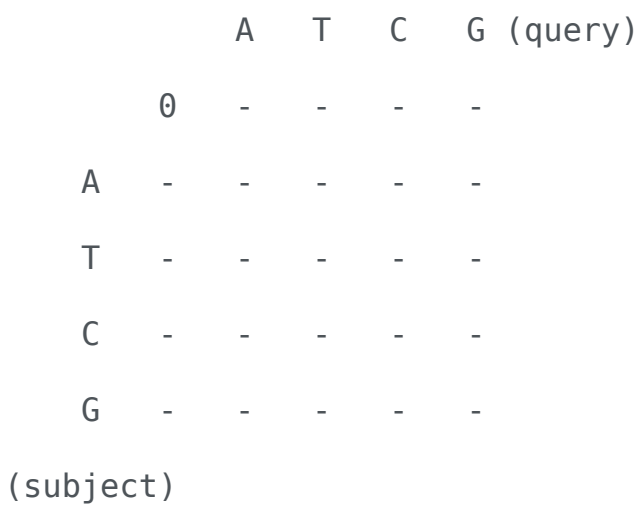
Finally, it also makes sense to initialize a starting position outside of the query and the subject because we could potentially have insertions and deletions at the start. We'll set this value to 0, because there is a match of nothing against nothing. Our final array will look like:

		A	T	C	G
	*	-	-	-	-
A	-	*	-	-	-
T	-	-	*	-	-
C	-	-	-	*	-
G	-	-	-	-	*

```
query   A---TCG
        |
subject ATCG---
```

Implementation

Here is a very naive implementation of the Levenshtein distance. There are many, many ways to optimize this, however it is out of scope in this book.



```rust
#[derive(Clone, Copy)]
enum AlignmentType {
    Match,
    Mismatch,
    DeletionQuery,
    DeletionSubject,
}

fn print_array(array: &Vec<Vec<usize>>) {
    for v in array {
        let values: String = v
            .iter()
            .map(|v| v.to_string())
            .collect::<Vec<_>>()
            .join("\t");
        println!("{values}");
    }
    println!("\n");
}

fn get_alignment_cost(aln: AlignmentType) -> usize {
    match aln {
        AlignmentType::Match => 0,
        AlignmentType::Mismatch => 1,
        AlignmentType::DeletionQuery => 1,
        AlignmentType::DeletionSubject => 1,
    }
}

fn levenshtein_distance(s1: &str, s2: &str) -> usize {
    // We take the number of rows from the subject.
    let m = s2.len();

    // We take the number of columns from the query.
    let n = s1.len();

    // Store array as a vector of vectors.
    let mut array: Vec<Vec<usize>> = Vec::new();

    // Initialize array.
    for _ in 0..m + 1 {
        array.push(vec![0; n + 1]);
    }

    assert!(array[0].len() == s1.len() + 1);
    assert!(array.len() == s2.len() + 1);
```

```
    // We move in the i direction (down), subject is consumed and
query is deleted.
    for i in 1..m + 1 {
        array[i][0] = i *
get_alignment_cost(AlignmentType::DeletionQuery);
    }
    // We move in the j direction (right), query is consumed and
subject is deleted.
    for j in 1..n + 1 {
        array[0][j] = j *
get_alignment_cost(AlignmentType::DeletionSubject);
    }

    for i in 1..m + 1 {
        for j in 1..n + 1 {
            // For a diagonal move, we need to check if we have a
match or mismatch.
            let match_or_mismatch = match s1.chars().nth(j - 1) ==
s2.chars().nth(i - 1) {
                true => array[i - 1][j - 1] +
get_alignment_cost(AlignmentType::Match),
                false => array[i - 1][j - 1] +
get_alignment_cost(AlignmentType::Mismatch),
            };

            // We have moved in the j direction so query is consumed
and subject is deleted
            let deletion_subject =
                array[i][j - 1] +
get_alignment_cost(AlignmentType::DeletionSubject);

            // We have moved in the j direction so subject is
consumed and query is deleted
            let deletion_query = array[i - 1][j] +
get_alignment_cost(AlignmentType::DeletionQuery);

            // Collect these into a vector.
            let previous_values: Vec<usize> =
                vec![match_or_mismatch, deletion_query,
deletion_subject];

            // NOTE - depending on how we define the order of
previous_values
            // and our alignment costs, we might get different
alignment results.
            let previous_min_value =
previous_values.iter().min().unwrap();
```

```
            // Update array for current value.
            array[i][j] = *previous_min_value;
        }
    }

    return array[m][n];
}

fn main() {
    assert_eq!(levenshtein_distance("ATCG", "ATCG"), 0);
    assert_eq!(levenshtein_distance("AAAAA", "A"), 4);
    assert_eq!(levenshtein_distance("ATATAT", "GGGGGG"), 6);
    assert_eq!(levenshtein_distance("ATCGATCG", "ATCGTTCG"), 1);
}
```

Yay! We have now implemented another kind of *global aligner* that supports matches, mismatches, insertions and deletions.

Adding Traceback

Introduction

We have successfully created a basic edit distance aligner! However, we don't just want to return a simple usize of the distance between two strings. We also want to *visualize* the alignment.

To make this work, we need to implement a traceback that enables us to generate the optimal alignment after we are done filling out the array. Let's look at the final array after aligning `ATCG` to `ATCG`:

In this example, we clearly see that the traceback should be just traversing diagonally. But how do we implement this programmatically?

We know the origin for each cell in the array, because we have defined it as `array[i][j] = min(diagonal, left, up)`. We can store the origin of each cell in a `HashMap` and start the traceback from the last cell until we reach the start. This is rather inefficient, but it'll work for now.

For simplicity, we'll save each cells origin along with the alignment type.

Implementation

```rust
use std::collections::HashMap;

#[derive(Clone, Copy)]
enum AlignmentType {
    Match,
    Mismatch,
    DeletionQuery,
    DeletionSubject,
}

fn print_array(array: &Vec<Vec<usize>>) {
    for v in array {
        let values: String = v
            .iter()
            .map(|v| v.to_string())
            .collect::<Vec<_>>()
            .join("\t");
        println!("{values}");
    }
    println!("\n");
}

/// We could modify these if we want.
fn get_alignment_cost(aln: AlignmentType) -> usize {
    match aln {
        AlignmentType::Match => 0,
        AlignmentType::Mismatch => 1,
        AlignmentType::DeletionQuery => 1,
        AlignmentType::DeletionSubject => 1,
    }
}

fn levenshtein_distance(
    s1: &str,
    s2: &str,
) -> (
    Vec<Vec<usize>>,
    HashMap<(usize, usize), ((usize, usize), AlignmentType)>,
) {
    // We take the number of rows from the subject.
    let m = s2.len();

    // We take the number of columns from the query.
    let n = s1.len();
```

```
// Store array as a vector of vectors.
let mut array: Vec<Vec<usize>> = Vec::new();

// Initialize array.
for _ in 0..m + 1 {
    array.push(vec![0; n + 1]);
}

assert!(array[0].len() == s1.len() + 1);
assert!(array.len() == s2.len() + 1);

// We store the origin of each element in the array.
let mut traceback: HashMap<(usize, usize), ((usize, usize), Alignm

// We move in the i direction (down), subject is consumed and quer
for i in 1..m + 1 {
    array[i][0] = i * get_alignment_cost(AlignmentType::DeletionQu
    // Remember to add trace.
    traceback.insert((i, 0), ((i - 1, 0), AlignmentType::DeletionQ
}
// We move in the j direction (right), query is consumed and subje
for j in 1..n + 1 {
    array[0][j] = j * get_alignment_cost(AlignmentType::DeletionSu
    // Remember to add trace.
    traceback.insert((0, j), ((0, j - 1), AlignmentType::DeletionS
}

for i in 1..m + 1 {
    for j in 1..n + 1 {
        // For a diagonal move, we need to check if we have a matc
        let match_or_mismatch = match s1.chars().nth(j - 1) == s2.
            true => (
                (i - 1, j - 1),
                array[i - 1][j - 1] + get_alignment_cost(Alignment
                AlignmentType::Match,
            ),
            false => (
                (i - 1, j - 1),
                array[i - 1][j - 1] + get_alignment_cost(Alignment
                AlignmentType::Mismatch,
            ),
        };

        // We have moved in the j direction so query is consumed a
        let deletion_subject = (
            (i, j - 1),
            array[i][j - 1] + get_alignment_cost(AlignmentType::De
            AlignmentType::DeletionSubject,
```

```
            );

            // We have moved in the j direction so subject is consumed
            let deletion_query = (
                (i - 1, j),
                array[i - 1][j] + get_alignment_cost(AlignmentType::De
                AlignmentType::DeletionQuery,
            );

            // EDIT ME! Try switching the order of the
            // elements and see if this changes the traceback.
            let previous_values: Vec<((usize, usize), usize, Alignment
                vec![match_or_mismatch, deletion_query, deletion_subje


            let (previous_index, previous_value, alignment_type) =
                previous_values.iter().min_by_key(|x| x.1).unwrap();

            // Add trace for current element.
            traceback.insert((i, j), (*previous_index, *alignment_type

            // Update array for current value.
            array[i][j] = *previous_value;
        }
    }

    return (array, traceback);
}

fn get_traceback(
    traceback: HashMap<(usize, usize), ((usize, usize), AlignmentType)
    s1: &str,
    s2: &str,
) {
    let mut m = s2.len();
    let mut n = s1.len();

    // Aligned part of s1 and s2 (including deletions).
    let mut s1_aln: Vec<char> = Vec::new();
    let mut s2_aln: Vec<char> = Vec::new();

    // We'll use "|" for match, "*" for mismatch and " " for deletion.
    let mut matches_aln: Vec<char> = Vec::new();

    loop {
        if (m, n) == (0, 0) {
            break;
        }
```

```
        let ((i, j), aln_type) = traceback.get(&(m, n)).unwrap();

        match aln_type {
            AlignmentType::Match => {
                let s1_char = s1.chars().nth(*j).unwrap();
                let s2_char = s2.chars().nth(*i).unwrap();
                s1_aln.push(s1_char);
                s2_aln.push(s2_char);
                matches_aln.push('|');
            }
            AlignmentType::Mismatch => {
                let s1_char = s1.chars().nth(*j).unwrap();
                let s2_char = s2.chars().nth(*i).unwrap();
                s1_aln.push(s1_char);
                s2_aln.push(s2_char);
                matches_aln.push('*');
            }
            AlignmentType::DeletionQuery => {
                s1_aln.push('-');
                s2_aln.push(s2.chars().nth(*i).unwrap());
                matches_aln.push(' ');
            }
            AlignmentType::DeletionSubject => {
                s1_aln.push(s1.chars().nth(*j).unwrap());
                s2_aln.push('-');
                matches_aln.push(' ');
            }
        }
        m = *i;
        n = *j;
    }

    let s1_aln_fwd: String = s1_aln.iter().rev().collect();
    let s2_aln_fwd: String = s2_aln.iter().rev().collect();
    let matches_aln_fwd: String = matches_aln.iter().rev().collect();

    println!("{}", s1_aln_fwd);
    println!("{}", matches_aln_fwd);
    println!("{}\n", s2_aln_fwd);
}

fn align(s1: &str, s2: &str) {
    let (_, traceback) = levenshtein_distance(s1, s2);

    get_traceback(traceback, s1, s2);
}
```

```
fn main() {
    align("ATCG", "ATCG");
    align("A", "T");
    align("ATCG", "ATCGATCG");
    align("TTTTTTTTTTTTTTTTA", "ATTTTTTTTTTTTT");
}
```

This is awesome! We have created a basic aligner that uses the Levenshtein distance and supports non-equal length strings. Some good exercises (left up to the reader) would be:

- Calculating percent identity and other relevant alignment metrics.
- Thinking about how the code can be optimized (trust me, it is not).
    - For example, do we really need to keep track of all rows and columns at the same time?
    - How can we optimize the traceback strategy?
- Writing a bunch of tests to make sure our code works (and fix it if it doesn't).

Smith-Waterman algorithm

Introduction

In the previous section, we implemented a global aligner with traceback. Now, we want to improve on this approach to handle local alignment. The difference here is that we do not require the entire query and subject to be aligned end-to-end.

In the example below, we just align the middle part of the query and subject and ignore (softmask) the surrounding regions because we have no significant match there.

Modifications

We need to make some changes to our global aligner in order for it to handle local alignments. We'll change the concept of cost and instead call it a score.

- We define the scoring procedure as:
    - A match increases the score by 1.
    - A mismatch decreases the score by 1.
    - An insertion/deletion decreases the score by 1.
- We also make the following changes:
    - A score value must be non-negative (>= 0).
    - All cells (i, 0) and (0, j) are initialized to 0.
    - Traceback starts at the cell with the highest score and ends when we reach a 0.

A local alignment array for aligning `TTATCGTT` to `GGATCGGG` would look like:

```
query   ttATCGtt
          ||||
subject ggATCGgg
```

The procedure will be:

- Fill out the entire array.
- Identify the cell with the highest score.
- Traceback from there until we reach a 0.

Implementation

It turns out that implementing the Smith-Waterman algorithm and then printing out the alignment, including the soft masked parts of the query and subject, is not straightforward and includes a bit of extra lines of code. However, here is an example of an implementation:

```rust
use std::collections::HashMap;

#[derive(Clone, Copy)]
enum AlignmentType {
    Match,
    Mismatch,
    DeletionQuery,
    DeletionSubject,
}

fn print_array(array: &Vec<Vec<i32>>) {
    for v in array {
        let values: String = v
            .iter()
            .map(|v| v.to_string())
            .collect::<Vec<_>>()
            .join("\t");
        println!("{values}");
    }
    println!("\n");
}

/// We could modify these if we want.
fn get_alignment_cost(aln: AlignmentType) -> i32 {
    match aln {
        AlignmentType::Match => 1,
        AlignmentType::Mismatch => -1,
        AlignmentType::DeletionQuery => -1,
        AlignmentType::DeletionSubject => -1,
    }
}

fn levenshtein_distance(
    s1: &str,
    s2: &str,
) -> (
    Vec<Vec<i32>>,
    HashMap<(usize, usize), ((usize, usize), AlignmentType)>,
    (usize, usize),
) {
    // We take the number of rows from the subject.
    let m = s2.len();

    // We take the number of columns from the query.
    let n = s1.len();

    assert!(m > 0);
    assert!(n > 0);
```

```
// Store array as a vector of vectors.
let mut array: Vec<Vec<i32>> = Vec::new();

// Initialize array.
for _ in 0..m + 1 {
    array.push(vec![0; n + 1]);
}

assert!(array[0].len() == s1.len() + 1);
assert!(array.len() == s2.len() + 1);

let mut max_score: (i32, (usize, usize)) = (0, (0, 0));

// We store the origin of each element in the array.
let mut traceback: HashMap<(usize, usize), ((usize, usize), Alignm

// We move in the i direction (down), subject is consumed and quer
for i in 1..m + 1 {
    array[i][0] = 0;
    // Remember to add trace.
    traceback.insert((i, 0), ((i - 1, 0), AlignmentType::DeletionQ
}
// We move in the j direction (right), query is consumed and subje
for j in 1..n + 1 {
    array[0][j] = 0;
    // Remember to add trace.
    traceback.insert((0, j), ((0, j - 1), AlignmentType::DeletionS
}

for i in 1..m + 1 {
    for j in 1..n + 1 {
        // For a diagonal move, we need to check if we have a matc
        let match_or_mismatch = match s1.chars().nth(j - 1) == s2.
            true => (
                (i - 1, j - 1),
                std::cmp::max(
                    0,
                    array[i - 1][j - 1] + get_alignment_cost(Alignm
                ),
                AlignmentType::Match,
            ),
            false => (
                (i - 1, j - 1),
                std::cmp::max(
                    0,
                    array[i - 1][j - 1] + get_alignment_cost(Alignm
                ),
```

```
                        AlignmentType::Mismatch,
                    ),
                };

                // We have moved in the j direction so query is consumed a
                let deletion_subject = (
                    (i, j - 1),
                    std::cmp::max(
                        0,
                        array[i][j - 1] + get_alignment_cost(AlignmentType
                    ),
                    AlignmentType::DeletionSubject,
                );

                // We have moved in the j direction so subject is consumed
                let deletion_query = (
                    (i - 1, j),
                    std::cmp::max(
                        0,
                        array[i - 1][j] + get_alignment_cost(AlignmentType
                    ),
                    AlignmentType::DeletionQuery,
                );

                // EDIT ME! Try switching the order of the
                // elements and see if this changes the traceback.
                let previous_values: Vec<((usize, usize), i32, AlignmentTy
                    vec![match_or_mismatch, deletion_query, deletion_subje

                let (previous_index, previous_value, alignment_type) =
                    previous_values.iter().max_by_key(|x| x.1).unwrap();

                // Add trace for current element.
                traceback.insert((i, j), (*previous_index, *alignment_type

                // Update array for current value.
                array[i][j] = *previous_value;

                // Update max array value and its index
                max_score = *vec![(array[i][j], (i, j)), max_score]
                    .iter()
                    .max_by_key(|x| x.0)
                    .unwrap();
            }
        }

        return (array, traceback, max_score.1);
    }
```

```
fn to_lowercase(nt: char) -> char {
    match nt {
        'A' => 'a',
        'C' => 'c',
        'G' => 'g',
        'T' => 't',
        _ => panic!(),
    }
}

fn get_traceback(
    array: &Vec<Vec<i32>>,
    traceback: HashMap<(usize, usize), ((usize, usize), AlignmentType)>
    max_index: (usize, usize),
    s1: &str,
    s2: &str,
) {
    let (mut m, mut n) = max_index;

    // Aligned part of s1 and s2 (including deletions).
    let mut s1_aln: Vec<char> = Vec::new();
    let mut s2_aln: Vec<char> = Vec::new();

    // We'll use "|" for match, "*" for mismatch and " " for deletion.
    let mut matches_aln: Vec<char> = Vec::new();

    let mut m_c = m.clone();
    let mut n_c = n.clone();

    // Fill the left unaligned, we do this first because we iterate th
    while m_c <= s2.len() - 1 || n_c <= s1.len() - 1 {
        match s1.chars().nth(n_c) {
            // We are still within s1, so we push the soft masked base
            Some(nt) => s1_aln.push(to_lowercase(nt)),
            // We have reached the end of s1, so we push a placeholder
            // Must be empty, otherwise the end of the alignment looks
            None => s1_aln.push('\0'),
        }

        match s2.chars().nth(m_c) {
            // We are still within s2, so we push the soft masked base
            Some(nt) => s2_aln.push(to_lowercase(nt)),
            // We have reached the end of s2, so we push a placeholder
            // Must be empty, otherwise the end of the alignment looks
            None => s2_aln.push('\0'),
        }
```

```
    matches_aln.push(' ');
    m_c += 1;
    n_c += 1;
}

s1_aln.reverse();
s2_aln.reverse();

loop {
    if array[m][n] == 0 {
        break;
    }

    let ((i, j), aln_type) = traceback.get(&(m, n)).unwrap();

    match aln_type {
        AlignmentType::Match => {
            let s1_char = s1.chars().nth(*j).unwrap();
            let s2_char = s2.chars().nth(*i).unwrap();

            s1_aln.push(s1_char);
            s2_aln.push(s2_char);
            matches_aln.push('|');
        }
        AlignmentType::Mismatch => {
            let s1_char = s1.chars().nth(*j).unwrap();
            let s2_char = s2.chars().nth(*i).unwrap();
            s1_aln.push(s1_char);
            s2_aln.push(s2_char);
            matches_aln.push('*');
        }
        AlignmentType::DeletionQuery => {
            s1_aln.push('-');
            s2_aln.push(s2.chars().nth(*i).unwrap());
            matches_aln.push(' ');
        }
        AlignmentType::DeletionSubject => {
            s1_aln.push(s1.chars().nth(*j).unwrap());
            s2_aln.push('-');
            matches_aln.push(' ');
        }
    }
    m = *i;
    n = *j;
}

// Fill the right unaligned part, we do this last because we itera
let mut m = m as i32;
```

```
    let mut n = n as i32;

    // We iterate until we have reached the end of both s1 and s2.
    while m >= 1 || n >= 1 {
        // We are still within s1, so we push the soft masked base.
        if n >= 1 {
            match s1.chars().nth((n-1) as usize) {
                Some(nt) => s1_aln.push(to_lowercase(nt)),
                None => panic!("Position {n} is invalid."),
            }
        }
        // We have reached the end of s1, so we push a placeholder.
        else {
            s1_aln.push(' ');
        }

        // We are still within s2, so we push the soft masked base.
        if m >= 1 {
            match s2.chars().nth((m-1) as usize) {
                Some(nt) => s2_aln.push(to_lowercase(nt)),
                None => panic!("Position {m} is invalid."),
            }
        }
        // We have reached the end of s2, so we push a placeholder.
        else {
            s2_aln.push(' ');
        }

        matches_aln.push(' ');
        m -= 1;
        n -= 1;
    }

    let s1_aln_fwd: String = s1_aln.iter().rev().collect();
    let s2_aln_fwd: String = s2_aln.iter().rev().collect();
    let matches_aln_fwd: String = matches_aln.iter().rev().collect();

    println!("{}", s1_aln_fwd);
    println!("{}", matches_aln_fwd);
    println!("{}\n", s2_aln_fwd);
}

fn align(s1: &str, s2: &str) {
    let (array, traceback, max_index) = levenshtein_distance(s1, s2);

    get_traceback(&array, traceback, max_index, s1, s2);
}
```

```
fn main() {
    align("ATCG", "ATCG");
    align("CCCATCGCCC", "GGGATCGGGTT");
    align("AAAAATAAAAA", "CCCCCTCCCCC");
    align("ATCG", "CCCATCGTTT");
}
```

This is a very simple implementation of a local aligner. In practice, as a bioinformatician, one would use a highly optimized and widely established tool. Some examples are:

- Parasail - A SIMD accelerated C library.
- BLAST - Uses the seed-and-extend approach.
- Minimap2 - One of the fastest aligners out there.

Creating a desktop app

We have now made our own local aligner, implemented entirely in Rust. It might not be the most efficient, but it works. Now, let us take this a step further and generate a desktop application that can visualize the alignment.

What we need to build it from scratch:

- Our alignment code (which we implemented in the previous section).
- The Dioxus framework.
- A bit of knowledge about HTML and CSS.

We won't go through the entire implementation from scratch. Instead:

- Make sure you are using a Linux operating system.
- Install the Dioxus cli version `v0.7.0-alpha.3`.
- Clone the repository.
- Enter the `alignment_rs` directory and run `dx serve`.
- It might take several minutes to compile, but when done the desktop app should launch.

Inspect the code to familiarize yourself with Dioxus. If you have used React before, the syntax might look familiar. Think of Dioxus as React for Rust. In short, the code:

- Checks the input for updates to the query and the subject.
- Calls the aligner when Dioxus detects that the query or subject has changed.
- Renders the alignment in real time.

Note that calling the aligner every time either the query or subject has changed might be extremely inefficient for long sequences. However, in our case we have limited the input length to 80 nucleotides, which is short enough for the UI to be responsive.

Below is a preview of what the desktop app looks like.

Resources

Understanding the details of alignment can be tricky. However, here are some good resources for further reading:

- Ben Langmead
- Wikipedia
- Original BLAST paper

Kmers

The concept of kmers is widely used in bioinformatics and is applied in concepts such as alignment and genome assembly. Here, we'll just go through the basics.

Basically, kmers are just subsequences of a specific length. For example, in the following sequence we generate all consecutive kmers of length 3:

Note that our sequence length is 12 and the kmer length is 3. How many consecutive kmers can we generate? The answer is `len(sequence) - kmer_size + 1`, which in our case would `12 - 3 + 1 = 10`. But why this exact formula?

If we had `kmer_size = 1`, the number of kmers would be equal to the sequence length. We just slide along the sequence with a window size of 1. We are losing out on *zero* nucleotides.

If we had `kmer_size = 2`, we use a sliding window of length 2. However, we cannot use the last nucleotide in the sequence, because we need two nucleotides for our sliding window. We are losing out on *one* nucleotide.

We see a pattern here, which is that the number of kmers we can generate is the length of our sequence minus how many nucleotides in the end we are missing out on (which is one less than our kmer size).

`num_kmers = len(sequence) - (kmer_size - 1) = len(sequence) - kmer_size + 1.`

A First Implementation

For a naive implementation of kmers, we'll just use a sliding window of the specified kmer size in the forward direction. For now, we skip the reverse complement.

```
fn kmerize(nt_string: &[u8], kmer_size: usize) -> Vec<&[u8]> {
    assert!(kmer_size <= nt_string.len());

    // Rust has a very handy windows function that works perfectly
here.
    let kmers: Vec<&[u8]> = nt_string.windows(kmer_size).collect();

    // Make sure we generated the correct number of kmers.
    assert_eq!(kmers.len(), nt_string.len() - kmer_size + 1);
    return kmers;
}

fn main() {
    assert_eq!(kmerize(b"AAAA", 2), vec![b"AA", b"AA", b"AA"]);
    assert_eq!(kmerize(b"ATCGATCG", 7), vec![b"ATCGATC",
b"TCGATCG"]);
    assert_eq!(
        kmerize(b"AATTCCGG", 2),
        vec![b"AA", b"AT", b"TT", b"TC", b"CC", b"CG", b"GG"]
    );
}
```

This naive implementation has several flaws that we need to handle:

- We currently don't consider the reverse complement.
- Once the reverse complement is handled, should we use all forward and all reverse kmers, or can we be smart about which kmers to pick?
- We still use ASCII encoding, which takes up unnecessary amounts of storage.
- Using a window function is not feasible when dealing with huge amounts of data. We need another approach.

> Tip

> These flaws are addressed in the following chapters: Bit Shift Encoding replaces ASCII with 2-bit encoding, Forward Strand and Reverse Strand handle both strands, and the Final Implementation combines everything into a canonical kmer generator.

Using Phred Scores

Before we proceed with more efficient nucleotide encoding strategies, we'll cover how phred scores can be used in kmer applications. For samples such as Oxford Nanopore, where the quality generally is lower than say Illumina, we can use phred scores to identify highly erroneous kmers. Use cases could be:

- Only keep high quality kmers for downstream analyses.
- Sort a FASTQ file based on the number of high quality kmers in each read.
- Calculate the expected number of error free kmers.

isONclust3 fundamentally uses some of these approaches as preprocessing steps. In the code below, we'll re-implement isONclust3's implementation of calculating the expected number of error free kmers.

In essence, we convert phred scores to error probabilities `p_e` for every nucleotide in the sequence. We can calculate the probability of the nucleotide being correctly called as `1-p_e`. For an arbitrary kmer of length `k`, we can calculate the probability of the entire kmer being correctly called as the product of the individual nucleotide probabilities.

$$\prod_{i=1}^{k} 1 - \mathrm{p\_e}_i$$

By repeating this calculation for every kmer across a sequence, we get the collection of all kmer probabilities. To get the expected number of error free kmers, we simply calculate the sum. Since we can generate `l-k+1` kmers of size `k` from a sequence of length `l`, we get:

$$\sum_{n=1}^{l-k+1} \prod_{i=1}^{k} 1 - \mathrm{p\_e}_i$$

where `n` is the position in the sequence and `i` is the position in a kmer.

```rust
fn phred_to_err(phred: u8) -> f64 {
    10_f64.powf(-1.0 * ((phred - 33) as f64) / 10.0)
}

/// Re-implementation of
/// https://github.com/aljpetri/isONclust3/blob/main/src/main.rs#L59
fn exp_error_free_kmers(qual: &[u8], kmer_size: usize) -> f64 {
    let mut sum_exp = 0.0_f64;

    // Current probability product for a rolling kmer.
    let mut current_prod = 1.0_f64;

    // We'll use a circular buffer to store up to one kmer at a
time.
    let mut buf = vec![1.0_f64; kmer_size];

    // We need to keep track of the index to know when to circle
back in our buffer.
    let mut idx = 0_usize;

    for i in 0..qual.len() {
        let q = qual[i];
        let p_e = phred_to_err(q);

        // Probability that the base is correct.
        let p_corr = 1.0_f64 - p_e;

        // Include new base in kmer probability.
        current_prod *= p_corr;

        // We have reached the capacity of our circular buffer.
        // Adjust by dividing (remove) the value we'll overwrite.
        if i >= kmer_size {
            let to_remove = buf[idx];
            current_prod /= to_remove;
        }

        // Add to our expected probability sum only for whole kmers.
        if i >= kmer_size - 1 {
            sum_exp += current_prod;
        }

        // Add base probability to our circular buffer and adjust
index.
        buf[idx] = p_corr;
        idx = (idx + 1) % kmer_size;
    }
```

```
    sum_exp
}

fn main() {
    let high_qual = exp_error_free_kmers(b"??????????", 5);
    let mid_qual = exp_error_free_kmers(b"5555555555", 5);
    let low_qual = exp_error_free_kmers(b"++++++++++", 5);

    println!("{}", high_qual);
    println!("{}", mid_qual);
    println!("{}", low_qual);

    // Higher quality scores should yield more expected error-free
kmers.
    assert!(high_qual > mid_qual);
    assert!(mid_qual > low_qual);

    // All values should be between 0 and the maximum possible kmers
(10 - 5 + 1 = 6).
    assert!(high_qual > 0.0 && high_qual <= 6.0);
    assert!(low_qual > 0.0 && low_qual <= 6.0);
}
```

We see from the output that we get `f64` values out for the expected number of error free kmers. This might look odd, but it is in fact the *expected* value. Depending on what we want to do with the expected value, we may or may not want to round it.

Bit Shift Encoding

Introduction

To streamline our kmer generation function, we need to understand a bit about bit shifting and how computers interpret data. Computers are ridiculously fast at bitwise operations. We won't cover the details in this book, but we'll go over the things we need in order for our kmer script to work properly.

In our case, we'll use 2-bit encoding for our nucleotides:

- `A` => `0b00` (0 in base 10)
- `C` => `0b01` (1 in base 10)
- `G` => `0b10` (2 in base 10)
- `T` => `0b11` (3 in base 10)

Bit shift

A `left shift` is defined as an operation in which the bits in a binary number are shifted to the left. The most significant bit (leftmost) is lost, and the least significant bit (rightmost) is shifted after which a zero is added.

- Example: `0010 << 1 = 0100`

A `right shift` does the opposite.

- Example: `0100 >> 1 = 0010`

```
fn main() {
    // Perform a left shift.
    assert_eq!(0b0010 << 1, 0b0100);

    // Perform a right shift.
    assert_eq!(0b0100 >> 1, 0b0010);
}
```

A left shift by one is equivalent to multiplying by 2. It makes sense by considering 10-based numbers. Left shifting the number 10 by one results in 100, which is equivalent to multiplying by 10. The same is true for binary numbers.

BitOR

The bitor operation (usually denoted with a pipe character `|`) applies the OR operation to two binary numbers. Assume we want to insert a `T` (`0b11`) into an integer with value `0b00`. We apply the bitor operation for this:

```
0b00 // Storage.
bitor
0b11 // T.
=
0b11 // Result.
```

because applying the OR bitwise, we'll get 0b(0 OR 1)(0 OR 1) = 0b11

```
fn main() {
    // Insert A
    assert_eq!(0b00 | 0b00, 0b00);
    // Insert C
    assert_eq!(0b00 | 0b01, 0b01);
    // Insert G
    assert_eq!(0b00 | 0b10, 0b10);
    // Insert T.
    assert_eq!(0b00 | 0b11, 0b11);
}
```

Bit masks

Bit masks can be used to manipulate a binary number certain ways. In our context, we'll use it to mask certain parts of our storage integer to ensure proper kmer length `k`. Say we have inserted three Gs `0b101010`, but we want to "mask" the upper two bits (the "oldest" `G`) because `k=2`. Masking the upper two bits is the same as saying we only want to keep the lower 4 bits (two Gs).

For this, we'll use the AND operator, which only returns 1 if both bits at a given position in our numbers are 1. This way, we can use 1 for every bit we want to keep, and 0 for the rest.

```
fn main() {
    // Only keep the lower 4 bits, mask the rest (e.g., set to
zero).
    assert_eq!(0b101010 & 0b001111, 0b001010);
}
```

How do we construct this mask programmatically? If we know our kmer size, we can do it. In the previous example, if our kmer size is 2, we want to keep 4 bits and mask the upper two.

We start with the number 1 (`0b000001`) and shift it 4 bits to the left, we get `0b010000`. This is not what we want. Our target is `0b001111`. However, subtracting 1 from `0b010000` gives us the correct answer.

```
fn main() {
    // Kmer size.
    let k = 2;

    // Equivalent to multiplying by 2.
    let nbits = k << 1;
    assert_eq!(nbits, 4);

    // We start with a 1 (0b000001) and shifts it nbits to the left.
    // this results in 0b010000, hence we overshoot since we wanted
0b001111.
    // This is why we subtract one, because 0b010000 - 0b000001 =
0b001111.
    let mask: u64 = (1 << nbits) - 1;

    assert_eq!(mask, 0b1111);
}
```

Choosing storage size

We use unsigned integers to store our kmers. Remember that each nucleotide, with our encoding, occupies two bits. The following types are available in Rust:

- `u8` - can store kmers of max size 8/2 = 4.
- `u16` - can store kmers of max size 16/2 = 8.
- `u32` - can store kmers of max size 32 / 2 = 16.
- `u64` - can store kmers of max size 64/2 = 32.
- `u128` - can store kmers of max size 128/2 = 64.

Can we store a kmer of size 2 in, say, a `u16`? Yes we can, but we'll waste space. Unfortunately, Rust does not yet provide arbitrary integer size, so these are the choices.

Forward Strand

In order to insert a nucleotide, we need two things:

- A left shift by two to make room for the two new bits.
- Insert the actual nucleotide, which is done with the `|` operator (BitOR).

Hence, for the forward strand we add nucleotides from the **right side**.

```
fn main() {
    let mut storage: u32 = 0b0;

    // Insert a T.
    storage = storage << 2 | 0b11;
    assert_eq!(storage, 0b11);

    // Insert another T.
    storage = storage << 2 | 0b11;
    assert_eq!(storage, 0b1111);

    // Insert a G.
    storage = storage << 2 | 0b10;
    assert_eq!(storage, 0b111110);

    println!("{:032b}", storage);
}
```

Note - it seems like new digits magically appear in our test cases. However, when we print the full u32, we see the leading zeros.

Handling the kmer size

Our approach kinda works, but it has a fundamental flaw. We want our storage variable to only contain k nucleotides at one time, all other leading bits should be zero. As an example:

```
nt_string = "GTGT"
kmer_size = 2

# start
0b00000000

# insert G
0b00000010

# insert T
0b00001011
```

At this point, we have inserted two nucleotides, which also is our target kmer length. In order to keep our target kmer size of 2, we need to:

- Insert the next nucleotide, G, resulting in a kmer of length 3.
- Mask anything above our kmer length to keep the length of 2.

We solve this by applying a bit-mask (as discussed previously). In the code example below, we also take care of invalid nucleotides.

```
// [...]

/// Print a u64 encoded nucleotide with some bit manipulation.
fn print_nt_string(kmer: u64, k: usize) {
    let mut result = String::with_capacity(k);

    for i in 0..k {
        // Shift to extract the 2 bits corresponding to the current
nucleotide
        let shift = 2 * (k - i - 1);
        let bits = (kmer >> shift) & 0b11;

        result.push(decode(bits));
    }

    println!("{}", result);
}

fn kmerize(kmer_size: usize, nt_string: &[u8]) {
    assert!(kmer_size <= nt_string.len());

    // Forward related kmer stuff
    let mut storage: u64 = 0;

    // Mask for bits above kmer size.
    let nbits = kmer_size << 1;
    let mask: u64 = (1 << nbits) - 1;

    // We keep track of how many valid nucleotides we
    // have in our storage and reset if we find an invalid nt.
    let mut valid_kmer_index: usize = 0;

    nt_string.iter().for_each(|nt_char| {
        // Forward kmer.
        let nt = LOOKUP[*nt_char as usize] as u64;

        // Reset if we found an invalid nucleotide.
        if nt >= 4 {
            valid_kmer_index = 0;
            storage = 0;
            return;
        }
        storage = (storage << 2 | nt) & mask;

        if valid_kmer_index >= kmer_size - 1 {
            print_nt_string(storage, kmer_size);
        }
```

```
            valid_kmer_index += 1;
        });
    }

    fn main() {
        // We expect just one kmer.
        kmerize(5, b"AAAAA");

        // We expect no kmers.
        kmerize(5, b"AAAANAAAA");

        // We expect AAA, AAT, ATT, TTT.
        kmerize(3, b"AAATTT");
    }
```

Converting kmer to string

Finally, it would also be nice to be able to convert an encoded kmer to a string. We can do this by leveraging the `kmer_size` and a suitable bitmask.

Consider the kmer `0b00111010`. Here, we are using an `u8` as storage and `kmer_size = 3`. We have inserted nucleotides in the following order: `T`, `G`, `G` and would like to get the same order back. Even though there are multiple ways to do this, one is to extract the nucleotides in the order they appear in the kmer, which is the reverse of how they were inserted, and then reverse the result. We would like to:

- Find the lowest two bits (latest inserted nucleotide).
- Convert these to a stringified nucleotide and append it to something like a `Vec` or `String`.
- Eject these bits from the kmer by a left shift.
- Continue until we have processed all nucleotides (which is `kmer_size` number of times).

We need a suitable bitmask for this. To only keep the lowest two bits, we'll use `0b11` with and `&` operator. This roughly looks like:

```
0b00111010
&
0b00000011
---
0b00000010
```

The result can subsequently be matched against, and converted to the appropriate nucleotide. `0b10` in this case would translate to `G`. The code below is one way of achieving this:

```
fn extract_nucleotides(mut kmer: u8, kmer_size: u8) -> String {
    let mut s = String::with_capacity(kmer_size as usize);
    let mask: u8 = 0b11;

    for _ in 0..kmer_size {
        let lowest_two_nts = kmer & mask;

        match lowest_two_nts {
            0b00 => s.push('A'),
            0b01 => s.push('C'),
            0b10 => s.push('G'),
            0b11 => s.push('T'),
            _ => unreachable!(),
        }
        kmer >>= 2;
    }

    s.chars().rev().collect()
}

fn main(){
    assert!(extract_nucleotides(0b00111010, 3) == "TGG");
    assert!(extract_nucleotides(0b00000010, 3) == "AAG");
    assert!(extract_nucleotides(0b00000011, 1) == "T");
}
```

Reverse Strand

As mentioned in a previous section, we also need to handle the reverse complement. How do we do this in an efficient way? We can insert the reverse complement from the left side instead of the right, ensuring the correct order. To insert from the left side, we first need to shift the two least significant bits, our nucleotide, to the upper most significant bits of our kmer. Then, we shift our storage to the right by 2 and finally apply BitOR to insert.

The following pseudo-code shows how to insert a nucleotide A whilst using `k=4`.

```
// Define variables.
k = 4
nt      =   0b0000000000 # A
nt_rev  =   0b0000000011 # T (reverse complement)
storage =   0b0000000000

// Shift reverse nucleotide to the upper two bits of the kmer size.
0b0000000011 << (k-1) * 2 = 0b0011000000

// Shift storage to the right to make room (empty at the moment).
0b0000000000 >> 2 = 0b0000000000

// Insert.
0b0000000000 | 0b0011000000 = 0b0011000000
```

The following code is an example of inserting the reverse complement of `AGT` into a `u32`. We'll make it easy for us and use a `k=3` to exactly fit the entire reverse complement into the kmer.

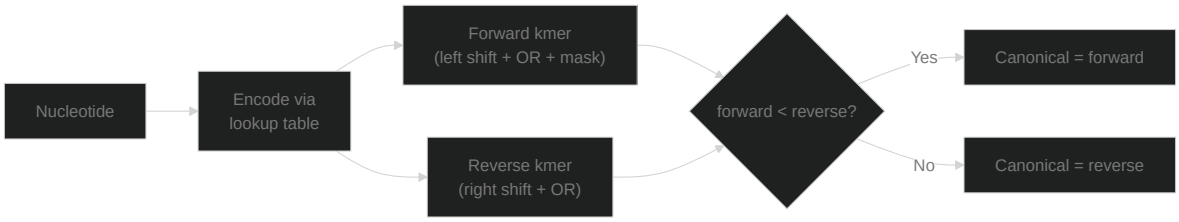
```rust
fn main() {
    let mut storage: u32 = 0b0;

    // Use kmer size 3 to exactly fit our three nucleotides
    // In the least significant bits.
    let k: u32 = 3;

    let forward = b"AGT";

    let shift: u32 = (k - 1) * 2;

    forward.iter().for_each(|nt| {
        let nt_encoded = match nt {
            b'A' => 0 as u32,
            b'C' => 1 as u32,
            b'G' => 2 as u32,
            b'T' => 3 as u32,
            _ => panic!(""),
        };
        // Use 3 - nt_encoded to get the reverse base.
        storage = storage >> 2 | (3 - nt_encoded) << shift;
    });

    // Print the full u32-bit.
    println!("{:032b}", &storage);

    // Verify: reverse complement of AGT is ACT, encoded as 00 01
11.
    assert_eq!(storage, 0b000111);
}
```

Run the code and inspect the result. Our output is:

Which is the reverse complement of `AGT`, inserted in the correct order.

Final Implementation

The code below combines the previous sections and adds an additional feature, which is canonical kmers. We define a canonical kmer as the lexicographically smallest kmer of the forward and reverse. This is a way to avoid keeping all kmers from the forward and the reverse strand.




```
00 [...] 00 01 11
         A  C  T
```

```rust
// [...]

pub fn kmerize(k: usize, nt_string: &[u8]) -> Vec<u64> {
    assert!(k <= nt_string.len());

    // Forward related kmer stuff
    let mut kmer_forward: u64 = 0;

    let nbits = k << 1;
    let mask: u64 = (1 << nbits) - 1;

    // Reverse related kmer stuff.
    let mut kmer_reverse: u64 = 0;
    let shift = ((k - 1) * 2) as u64;

    let mut valid_kmer_index: usize = 0;
    let mut canonical_kmers: Vec<u64> = Vec::new();

    nt_string.iter().for_each(|nt_char| {
        let nt = LOOKUP[*nt_char as usize] as u64;

        // Check for invalid nucleotides.
        if nt >= 4 {
            valid_kmer_index = 0;
            kmer_forward = 0;
            kmer_reverse = 0;
            return;
        }
        // Forward kmer.
        kmer_forward = (kmer_forward << 2 | nt) & mask;

        // Reverse kmer.
        let nt_rev = 3 - nt;
        kmer_reverse = kmer_reverse >> 2 | nt_rev << shift;

        if valid_kmer_index >= k - 1 {
            let canonical = match kmer_forward < kmer_reverse {
                true => kmer_forward,
                false => kmer_reverse,
            };

            print_nt_string(canonical, k);
            canonical_kmers.push(canonical);
        }

        valid_kmer_index += 1;
    });
```

```
    canonical_kmers
}

fn main(){
    let kmers_a = kmerize(5, b"AAAAAA");
    println!("");

    let kmers_t = kmerize(5, b"TTTTTT");
    println!("");

    // AAAAAA and TTTTTT are reverse complements, so they
    // should produce the same canonical kmers.
    assert_eq!(kmers_a, kmers_t);

    // Expected to not generate any kmers since we have an
    // invalid nucleotide interrupting every kmer.
    let kmers_n = kmerize(5, b"AAAANTTTT");
    assert!(kmers_n.is_empty());
    println!("");
}
```

When we run the code, we see that `AAAAAA` and `TTTTTT` generates the same canonical kmers which is expected since they are reverse complements of each other.

FracMinHash

Previously, we implemented an efficient way of generating canonical kmers from nucleotide strings. Now, we'll take this a step further by covering FracMinHash. Briefly, FracMinHash is a clever way of downsampling a large set of kmers into a representative set. For a more detailed explanation, please check out this paper.

Essentially, we only add two steps to our canonical kmer pipeline:

- We hash our canonical kmer using an appropriate hashing function.
- We add our hashed kmer only if its hash is less than or equal to a defined threshold.

We define our threshold as the maximum possible integer value (in our case we'll use `u64`), divided by a downsampling factor.

```rust
// [...]

/// https://github.com/bluenote-1577/sylph
fn mm_hash64(kmer: u64) -> u64 {
    let mut key = kmer;
    key = !key.wrapping_add(key << 21);
    key = key ^ key >> 24;
    key = (key.wrapping_add(key << 3)).wrapping_add(key << 8);
    key = key ^ key >> 14;
    key = (key.wrapping_add(key << 2)).wrapping_add(key << 4);
    key = key ^ key >> 28;
    key = key.wrapping_add(key << 31);
    key
}

fn kmerize(k: usize, ds_factor: u64, nt_string: &[u8]) ->
HashSet<u64> {
    if k >= nt_string.len() {
        panic!("kmer: {k}, nt_string: {}", nt_string.len());
    };

    // Forward related kmer stuff
    let mut kmer_forward: u64 = 0;

    let nbits = k << 1;
    let mask: u64 = (1 << nbits) - 1;

    // Reverse related kmer stuff.
    let mut kmer_reverse: u64 = 0;
    let shift = ((k - 1) * 2) as u64;

    // Storage.
    let mut canonical_hashes: HashSet<u64> =
HashSet::with_capacity(nt_string.len() - k + 1);

    let mut valid_kmer_index: usize = 0;

    nt_string.iter().for_each(|nt_char| {
        // Forward kmer.
        let nt = LOOKUP[*nt_char as usize] as u64;

        if nt >= 4 {
            valid_kmer_index = 0;
            kmer_forward = 0;
            kmer_reverse = 0;
            return;
        }
```

```
        kmer_forward = (kmer_forward << 2 | nt) & mask;

        // Reverse kmer.
        let nt_rev = 3 - nt;
        kmer_reverse = kmer_reverse >> 2 | nt_rev << shift;

        if valid_kmer_index >= k - 1 {
            let canonical = match kmer_forward < kmer_reverse {
                true => kmer_forward,
                false => kmer_reverse,
            };
            // MinFracHash
            if canonical <= u64::MAX / ds_factor {
                canonical_hashes.insert(mm_hash64(canonical));
            }
        }

        valid_kmer_index += 1;
    });

    return canonical_hashes;
}

fn print_canonical_hashes(canonical_hashes: &HashSet<u64>) {
    for canonical_hash in canonical_hashes{
        println!("{canonical_hash}");
    }
}

/// In these examples, we don't downsample because our
/// nucleotide strings are very short and have low complexity.
fn main(){
    let canonical_hashes_a = kmerize(5, 1, b"AAAAAAAAAA");
    print_canonical_hashes(&canonical_hashes_a);

    let canonical_hashes_t = kmerize(5, 1, b"TTTTTTTTTT");
    print_canonical_hashes(&canonical_hashes_t);

    // Both should produce non-empty hash sets.
    assert!(!canonical_hashes_a.is_empty());
    assert!(!canonical_hashes_t.is_empty());

    // Reverse complements should produce the same canonical hashes.
    assert_eq!(canonical_hashes_a, canonical_hashes_t);
}
```

The result is a seemingly nonsensical number for each sequence. However, we note two important things:

- Each sequence only generated one hash.
- Both sequences generated the same hash.

The reasons for this are:

- The sequences are reverse complements, so they generate the same canonical kmers.
- Both sequences generate only one unique canonical kmer, `AAAAA`

Minimizers

We saw earlier how FracMinHash could be used to downsample the number of kmers generated from our sequences. Another approach is to use so called minimizers. First introduced in 2004, minimizers are very commonly used in bioinformatic applications to reduce storage requirements for DNA sequences.

The basic idea is to use a sliding window of w consecutive kmers in a sequence and in each window identify one representative kmer to keep. Since we choose a reduced set of kmers, these will act as an approximate representation for the original sequence. There are multiple ways to choose a representative kmer inside the sliding window, but typically the lexicographically smallest kmer is chosen. We need to define some terms to make things more clear:

- `k` - length of a kmer.
- `w` - number of consecutive kmers to check for minimizers in.
- `|w|` - The actual length (in nucleotides) we need for our sliding window to accommodate `w` consecutive kmers of length `k`.

We can calculate $|w|$ since we know how many kmers we can generate from a given sequence.

```
w = |w| - k + 1
```

See the example below, where we set w=4 and k=3, hence calculating $|w| = 6$.

Let's consider the first window `AAACCC`. The possible kmers we can generate in the forward direction are `AAA`, `AAC`, `ACC`, `CCC`. Out of these, the lexicographically smallest one is AAA and we choose this kmer as this windows minimizer. We then do the same for the remaining windows.

We can get even more space efficient by storing the minimizers in a hashset, since this removes duplicates. However, this is not suitable if we also want to

store information such as the minimizers positions. We also have to take the reverse complement into consideration, similarly to what we did in the bit shift encoding section.

There are several Rust crates, such as Needletail and bio-seq that implement minimizers quite efficiently. In the code snippet below, we just implement a minimally viable prototype.

```
AAACCCGGGAAACCCGGGAAACCCGGG
AAACCC
 AACCCG    ...         ...
  ACCCGG
   CCCGGG            CCCGGG
```

```rust
use std::{cmp::min, vec};

fn reverse(nt: &u8) -> u8 {
    match nt {
        b'A' => b'T',
        b'C' => b'G',
        b'G' => b'C',
        b'T' => b'A',
        _ => panic!("Invalid nt."),
    }
}

/// Find the lexicographically smallest kmers from
/// either the forward or reverse window.
fn minimizer_from_windows<'a>(
    w_forward: &'a [u8],
    w_reverse: &'a [u8],
    kmer_size: usize,
) -> &'a [u8] {
    let min_fwd = w_forward.windows(kmer_size).min().unwrap();
    let min_rev = w_reverse.windows(kmer_size).min().unwrap();

    return min(min_fwd, min_rev);
}

fn get_minimizers(seq: &[u8], window_size: usize, kmer_size: usize)
-> Vec<String> {
    // This is the actual length (in nucleotides) of the sliding
    // window we need for w consecutive kmers of length k.
    let sliding_window_size = window_size + kmer_size - 1;
    assert!(sliding_window_size <= seq.len());

    // We'll store the minimizers as strings convenience.
    let mut m: Vec<String> = Vec::new();

    let rev_comp: Vec<u8> = seq.iter().rev().map(|nt|
reverse(nt)).collect();

    // Create windows for both forward and reverse sequences.
    seq.windows(sliding_window_size)
        .zip(rev_comp.as_slice().windows(sliding_window_size))
        // Iterate over forward/reverse windows at the same time.
        .for_each(|(w_forward, w_reverse)| {
            let minimizer = minimizer_from_windows(w_forward,
w_reverse, kmer_size);
            m.push(String::from_utf8(minimizer.to_vec()).unwrap());
        });
```

```
    return m;
}

fn main() {
    assert_eq!(get_minimizers(b"AAATTT", 4, 3), vec!["AAA"]);

    // Use all canonical kmers as minimizers.
    assert_eq!(
        get_minimizers(b"AAATTT", 1, 3),
        vec!["AAA", "AAT", "ATT", "TTT"]
    );
}
```

For a more thorough review on minimizers, check out this awesome paper.

Syncmers

Minimizers are widely used in bioinformatics by softwares such as Minimap2 and Kraken2. Recently, the concept of syncmers was proposed as an alternative to minimizers. To quote the paper:

"*Syncmers are defined here as a family of alternative methods which select k-mers by inspecting the position of the smallest-valued substring of length s < k within the k-mer.*"

Basically what this means is:

- Take a kmer of length `k`.
- Check for the smallest substring (by value) of length `s` in the kmer.
- If the location of this substring fulfills a given criteria, classify the kmer as a syncmer.

A very simplified example is a nucleotide `ATCG` of length 4. Let `k = 3` and `s = 2`. Let's assume that our criteria is that the smallest valued substring must be located at the start of the kmer.

We can generate two kmers and for each of them, we check if the smallest valued substring of length `s = 2` is located at the start of the kmer.

```
ATCG    # nucleotide sequence.

ATC     # kmer_1
 TCG    # kmer_2
```

We see that:

- `ATC` has `AT` as its smallest valued substring and `AT` is located at the start. `ATC` is a syncmer.
- `TCG` has `CG` as its smallest valued substring and `CG` is not located at the start. `TCG` is not a syncmer.

Closed syncmers

In this section, we'll go through *closed syncmers* where the location criteria is that the smallest value substring must be located at either the start or end of the kmer. Expanding our previous example with this location criteria, we get

```
ATCG    # nucleotide sequence.

ATC     # kmer_1
 TCG    # kmer_2
```

We see that:

- `ATC` has `AT` as its smallest valued substring and `AT` is located at the start. `ATC` is a closed syncmer.
- `TCG` has `CG` as its smallest valued substring and `CG` is located at the end. `TCG` is a closed syncmer.

Implementation

In the example below, we won't bother with nucleotide encoding. Rather, we'll just find closed syncmers by iterate over the kmers, using the `.windows()` function and check the location of the smallest valued substring.

```
fn get_closed_syncmers<'a>(seq: &'a [u8], k: usize, s: usize) ->
Vec<&'a [u8]> {
    assert!(k <= seq.len());
    assert!(s <= k);

    let closed_syncmers: Vec<&[u8]> = seq
        .windows(k) // Generate kmers.
        .filter_map(|kmer| {

            let (smallest_index, _) = kmer
                .windows(s) // Substrings of length s for a given
kmer.
                .enumerate()
                .min_by_key(|substring| substring.1) // Find
smallest valued substring.
                .unwrap();

            // Location criteria.
            if smallest_index == 0 || smallest_index == (k - s) {
                return Some(kmer);
            }

            return None;
        })
        .collect();

    return closed_syncmers;
}

fn main() {
    // Generate one single kmer. Syncmer "AT" is not in the start or
end.
    assert_eq!(get_closed_syncmers(b"TTATT", 5, 2), Vec::<&
[u8]>::new());

    // The example from the introduction to this chapter.
    assert_eq!(get_closed_syncmers(b"ATCG", 3, 2), vec![b"ATC",
b"TCG"]);

    // Example from the syncmer paper.
    assert_eq!(get_closed_syncmers(b"GGCAAGTGACA", 5, 2), vec!
[b"GGCAA", b"AAGTG", b"AGTGA", b"GTGAC"]);
}
```

Advanced Topics

In this section, we explore more advanced techniques that build on the foundations covered earlier in the book. These topics focus on pushing performance further and solving problems that arise when working with large-scale genomic data. We cover SIMD vectorization for parallel kmer processing and reverse indices for efficient sequence search.

SIMD Vectorization

Full disclosure - I'm not a computer science expert. Not even close actually. This will not be a heavy theoretical introduction. Rather, I will try to explain how SIMD can be used to significantly speed up bioinformatic analyses. Finally, I think these resources are valuable with respect to SIMD and Rust:

- SIMD on Wikipedia.
- Rust std::simd.
- Rust core::arch.
- Rust x86_64.

Introduction

SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) enables certain CPU instructions to be executed in parallel. In contrast to threads, SIMD is more primitive, low level and allows for a more restricted set of operations.

What makes SIMD a bit tricky is that it is architecture specific. Hence, the instruction sets we can use depend on what architecture our computer runs. We'll skip the details here, partially because it is out of scope in this book and partially because I personally don't know enough on this topic.

In the subsequent paragraphs, we assume we are running on the `x86_64` architecture.

SIMD Registers and Instruction Set Extensions

SIMD operates on fixed-width registers. On `x86_64`, the main SIMD extensions are:

Even though AVX-512 might be the widest and most powerful, it is not universally supported. SSE2 is effectively ubiquitous on `x86_64`.

A Pragmatic View

Going back to the previous chapter, we constructed a relatively efficient algorithm for generating kmers from a nucleotide sequence. How can we make this algorithm even more efficient with SIMD?

First, let's look at some pseudo code, inspired by the previous chapter for processing the forward strand.

```
fn kmerize(kmer_size: usize, nt_string: &[u8]) {

    let nbits = kmer_size << 1;
    let mask: u64 = (1 << nbits) - 1;

    // iterate over each nt.
    nt_string.iter().for_each(|nt_char| {
        // encode
        let nt = LOOKUP[*nt_char as usize] as u64;

        // bit shift (add nt)
        storage = (storage << 2 | nt) & mask;

    });
}
```

It is obvious that this function handles a single sequence. What if we could process multiple sequences at once? Conceptually (and with pseudo code) it could look something like this

```
fn kmerize(kmer_size: usize, nt_strings: &[&[u8]]){
    let nbits = kmer_size << 1;
    let mask: u64 = (1 << nbits) - 1;

    let storage_simd = create_simd_vector[0_u64; nt_strings.len()];

    // assume all nt_string have equal length.
    let seq_len = nt_strings[0].len();

    for i in 0..seq_len {
        let nt_simd = simd_vector::from(nt_strings.iter().map(|s: &
[u8]| s[i]));

        storage_simd = (storage_simd << 2 | nt_simd) & mask;
    }
}
```

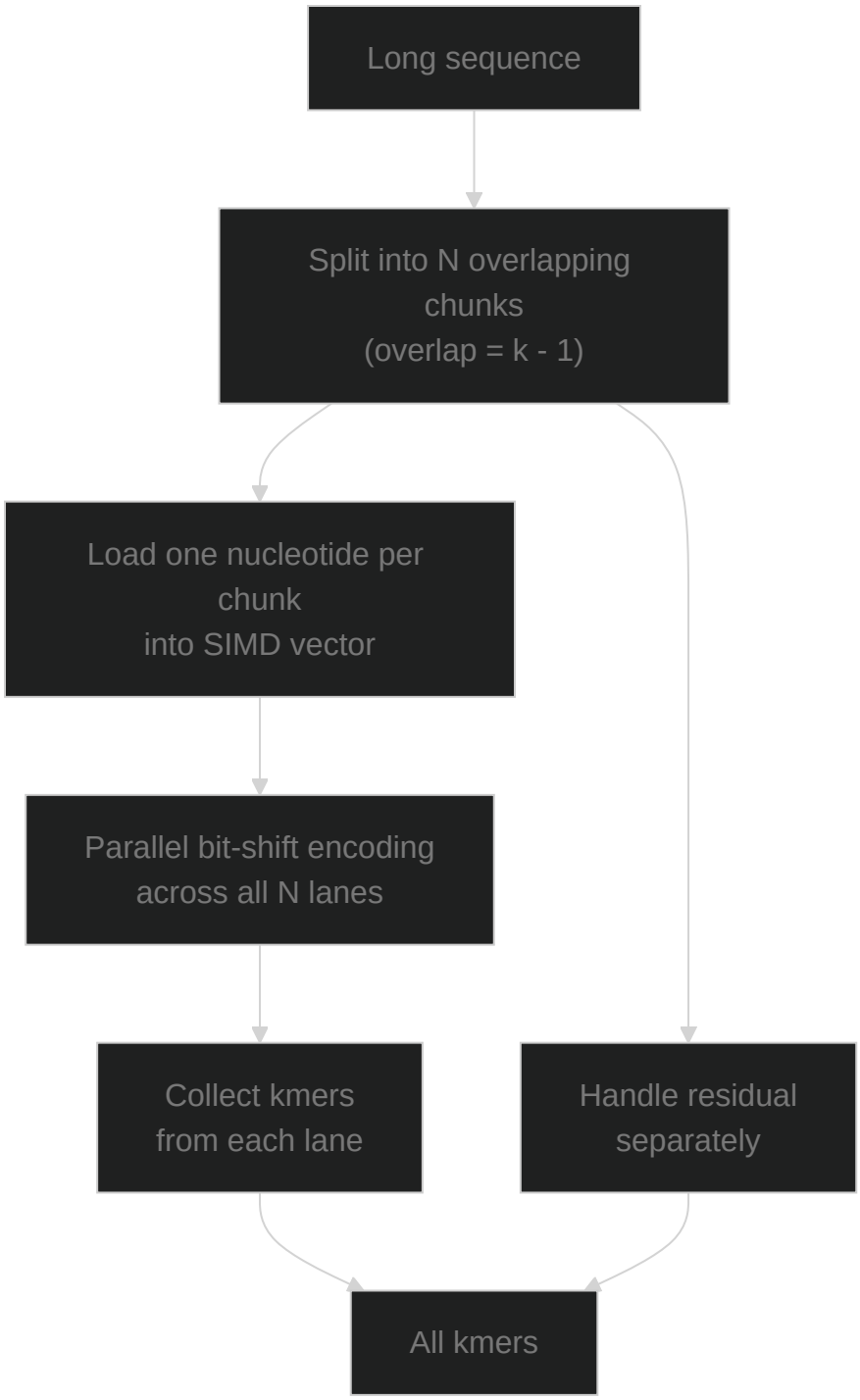
We start with an initial storage SIMD vector the same length as the number of sequences. This way, each sequence gets its own `slot`. In each iteration `i` we extract the `ith` nucleotide from every sequence, create a nucleotide SIMD vector and apply the bit shift in parallel. Schematically, it could look something like this:

```
nt_strings = [b"ATCA", b"GTGA", b"TCGA"]

storage_simd = [0_u64, 0_u64, 0_u64]

for i in 0..4{
    // for i=0, we extract:
    // * the b'A' from b"ATCA"
    // * the b'G' from b"GTGA"
    // * the b'T' from b"TCGA".
    nt_simd = [0, 2, 3] = [0b00, 0b10, 0b11];
    storage_simd = ([0_u64, 0_u64, 0_u64] << 2 | [0b00, 0b10, 0b11])
& mask = [0b00, 0b10, 0b11]

    // for i=1, we extract:
    // * the b'T' from b"ATCA"
    // * the b'T' from b"GTGA"
    // * the b'C' from b"TCGA".
    nt_simd = [3, 3, 1] = [0b11, 0b11, 0b01];
    storage_simd = ([0b00, 0b11, 0b11] << 2 | [0b11, 0b11, 0b01]) &
mask = [0b0011, 0b1011, 0b1101]
}
```

Our implementation has a fundamental flaw. What if the sequences don't have the same length? We have to rethink our approach.

Chunking

Instead of trying to process separate, unrelated sequences at once, what if we can process a single sequence at once?



What if we can cleverly chop our sequence into N equal size chunks (where N is the number of SIMD lanes available) and process them in parallel? We can, with some requirements:

- The sequence has to be reasonably long for this to make sense.

- We have to handle cases where a sequence is not evenly divisible into exactly 8 chunks.

We must first investigate how to chop our sequence. As an example, take the sequence `b"AAATTTCCC"`. With a kmer size of length 3, we want to generate

```
AAA, AAT, ATT, TTT, TTC, TCC, CCC
```

Also, pretend we have 3 SIMD lanes available. We need to chop our sequence into 3 chunks, each of which can be processed in parallel. We must ensure that:

- Each chunk is longer than, or equal to the kmer size.
- We have to handle the residual if the sequence cannot be exactly chunked into 3.

The key is to generate chunks that overlap by `kmer_size - 1` nucleotides. A short motivation for this is that it generates our target kmers. The more nuanced motivation is the following - try chunking into non-overlapping chunks. This would yield `AAA`, `TTT` and `CCC`. Kmerizing these (k=3) would simply give `AAA`, `TTT` and `CCC`, which is fewer kmers than the `sequence_length - kmer_size + 1 = 9 - 3 + 1 = 7` kmers we listed above. One (kinda) correct way to chunk would be `AAAT`, `ATTT` and `TTCC`, where each sub-sequence is overlapping by `kmer_size - 1 = 3 - 1 = 2`. Kmerizing these would give

```
AAAT -> AAA, AAT
ATTT -> ATT, TTT
TTCC -> TTC, TCC
```

Which is 6 out of the 7 kmers we wanted. We are missing one kmer because the last `C` was excluded from the chunking. But since we know this, we can easily generate the last kmer.

Regardless, with some clever maths (of which the formula I still find hard to derive) we can design our chunks and residual so that all kmers can be easily extracted.

In Practice

Assume we are running on `AVX-512`, which means we can have (at most) a 512-bit SIMD register. Since we use `u64` as storage for each kmer, we can have at most `512 / 64 = 8` SIMD lanes. If we chop our sequence into 8 equal-length chunks, we can process them in parallel and then handle the residual separately.

In practice though, one would typically use a crate that fully supports all of this out of the box. One example is simd-minimizers.

Building a Reverse Index

Enough with SIMD, let's talk about another very useful concept within bioinformatics - the reverse index.

But before this, what even is a "forward" index? Imagine you have a FASTA file, which contains your database sequences. This could be e.g., resistance genes, MLST alleles or something else. A "forward" index stores information about what database sequence contains what kmer hash. For example a simple `HashMap` with keys and values.

```
{
    "seq_1": [14184540469240097163, 18446744073709551615, ...],
    "seq_2": [4512398701234987123, 3141592653589793238, ...],
    ...
    "seq_n": [6672914039128457702, 14184540469240097163, ...],
}
```

Remember, a kmer hash is simply a kmer (e.g., `b"AAA"`) that is u64 encoded and that has been fed into a hash function to generate a new u64.

Why would we store information like this? One reason is that if we have a bunch of kmer hashes from a query sequence, we can check which database sequence matches and how well. One way would be to loop over each (key, value) pair in the index (possibly in parallel) and check how many of the query kmer hashes are identical. This gives us an approximate sequence similarity.

Why A Reverse Index Is Better

A reverse index is simply the reverse of a "forward" index, meaning that kmer hashes are the keys and the IDs of the database sequences that contain each hash are the values. E.g.,

```
{
    "14184540469240097163": ["seq_1", "seq_n"],
    "18446744073709551615": ["seq_1"],
    "45123987012349871 23":  ["seq_2"],
    ...
    "66729140391284577 02":  ["seq_n"]
}
```

For the first entry, the reverse index above reads: “kmer hash `14184540469240097163` is found in `seq_1` and `seq_n`”.

We can do better. If we know the number of sequences, e.g., from reading the FASTA file, we can define a fixed size for the length of the value arrays. We can set them to exactly length `n` since each kmer hash can be present in at most `n` unique sequences. Also, let’s switch out the array of strings to a bitset. A bitset is essentially an array where each element can have one of two values, either 0 or 1:

- `0` at index `i` means that sequence `i` does not contain the kmer hash.
- `1` at index `i` means that sequence `i` does contain the kmer hash.

This refined reverse index would look something like:

```
{
    "14184540469240097163": [1, 0, ..., 1],
    "18446744073709551615": [1, 0, ..., 0],
    "45123987012349871 23":  [0, 1, ..., 0],
    ...
    "66729140391284577 02":  [0, 0, ..., 1]
}
```

For the first entry, the reverse index now reads: “kmer hash `14184540469240097163` exists at index `0` and `n-1`”. If we originally had all sequences stored as something like a `sequences: Vec<FastaRecord> = [record_1, record_2, ..., record_n]` it would be as easy as to access the ids as `sequences[0].id` and `sequences[n-1].id`.

Why is this better than a forward index? Because using fixed size bitsets enables very efficient processing and minimal storage.

Using A Reverse Index In Practice

Reverse indices are a cornerstone of many state-of-the-art bioinformatics tools that need to search or classify sequences at scale. For example, COBS (Compact Bit-Sliced Signature Index) uses a compressed reverse index to enable fast approximate membership queries across massive sequence collections. Similarly, sourmash leverages FracMinHash sketches with reverse index structures for rapid genome search and taxonomic classification. Other tools like BIGSI and sylph also rely on variations of this pattern. The core idea remains the same: by indexing kmer hashes and mapping them back to their source sequences, we can quickly identify which database entries share content with a query — without aligning every sequence pair.

Increasing Performance

In this chapter, we'll look at a few ways to improve the performance of our code. Some of these methods can be used to make significant improvements to code we have seen previously in this book (an exercise left to the reader).

Optimizing code is not always straightforward and might require something like flamegraph to identify bottlenecks. Usually, however, there are a few things that are always good to keep in mind. We'll explore some of these in the following sections.

Using Appropriate Data Structures

It is easy to default to using e.g., a `HashMap` or other variable sized data structures for convenience. This is usually fine until it isn't. Below we'll go through some data structures and why they might not be an optimal choice.

HashMap

A `HashMap` is a very convenient way of storing data as key-value pairs. For example, if we want to count nucleotides in a string, we can use the nucleotide as the key and the count as the value. For the sequence `ACTTCC` it would look something like (pretty printed):

```
{
    "A": 1,
    "C": 3,
    "G": 0,
    "T": 2
}
```

Performance wise, a `HashMap` might provide some overhead due to:

- The need of hashing the key.
- Potential memory re-allocation when it reaches its maximum capacity.

These are not really of concern in the example above because the sequence is short and we are only concerned with four unique keys. In other instances however, it might be more relevant.

We can improve our `HashMap` by:

- Choosing a fast hash function such as FxHasher.
- Initializing our `HashMap` with a specified capacity. In our case, we could use `HashMap::with_capacity(4)` to ensure it can accommodate all our keys without having to re-allocate.

With that said, there are cases when a `HashMap` is probably justified, such as in the chapter about building a reverse index.

Vec

`Vec` is another familiar and convenient data structure. Similar to a `HashMap`, a `Vec` is also dynamically sized and requires re-allocation when its capacity is reached. Consider the case where we'd like to kmerize the sequence `ATCATC` with `k=3` and store the kmers in a `Vec`. Since we know that the number of kmers we can generate is `6 - 3 + 1 = 4`, we can initialize a `Vec` with a capacity of `4` to avoid re-allocations when adding kmers.

Fixed Size Array

In the case of counting nucleotides, using a fixed size array is much better than both a `HashMap` and a `Vec`. This data structure is of type `[<dtype>; <length>]` where `<length>` must be known at compile time.

The trick here is to utilize the nucleotide encoding, first encountered in the encoding chapter. If we assume that our sequence only consists of `{"A", "C", "G", "T"}` we can use a fixed size array of length `4`. The encoding maps each nucleotide `A, C, G, T` to `0, 1, 2, 3` which exactly corresponds to the indices we have in our array. Conceptually, we'd:

- Initialize a fixed size array of length `4` with all values set to `0`.
- Loop over each nucleotide and encode.
- Increment that index in the array.

```rust
fn main(){
    let seq = b"AATCG";

    let mut counts: [usize; 4] = [0; 4];

    for nt in seq{
        let encoding = match nt{
            b'A' => 0,
            b'C' => 1,
            b'G' => 2,
            b'T' => 3,
            _ => continue
        };

        counts[encoding] += 1;
    }

    assert_eq!(counts, [2, 1, 1, 1]);
}
```

It is not elegant to just skip unexpected characters. In practise, we could use a lookup table to map:

- canonical bases `{A, C, G, T}` (and possibly `{a, c, g, t}`) to `{0, 1, 2, 3}`.
- ambiguous bases `N` to `4`.
- everything else to `5`.

This, however, requires us to use an array of length `6`.

Favor Compile Time

A general rule I've found to work quite nicely is to favor compile time when possible. An excellent example of this is our lookup tables for nucleotide encoding and phred-score-to-error.

Lookup Tables

> **Tip**
>
> As of recent Rust versions, many use cases for `lazy_static!` can be replaced with `std::sync::LazyLock` (stabilized in Rust 1.80). Consider this alternative for simpler, dependency-free compile-time initialization.

We can use the lazy_static crate to define static lookup tables. In the code example below, we define two lookup tables:

- One for converting ASCII characters to nucleotide encoding.
- One for converting ASCII qualities to error probabilities.

```rust
use lazy_static::lazy_static;

const PHRED_OFFSET: usize = 33;
const MAX_PHRED_INDEX: usize = 93;

lazy_static! {
    pub static ref NT_LOOKUP: [u8; 256] = {
        let mut table = [4u8; 256];

        for i in 0u8..=255 {
            table[i as usize] = match i {
                b'A' | b'a' => 0,
                b'C' | b'c' => 1,
                b'G' | b'g' => 2,
                b'T' | b't' | b'U' | b'u' => 3,
                _ => 4,
            };
        }

        table
    };

    pub static ref PHRED_TO_ERROR: [f64; MAX_PHRED_INDEX + 1] = {
        let mut error_lookup = [1.0; MAX_PHRED_INDEX + 1];

        for (i, entry) in
error_lookup.iter_mut().enumerate().skip(PHRED_OFFSET) {
            *entry = 10_f64.powf(-((i - PHRED_OFFSET) as f64) /
10.0);
        }

        error_lookup
    };
}
```

At least for `PHRED_TO_ERROR`, the advantage of using a static lookup table is obvious. We avoid repeated calculations of

$$\text{error\_probability} = 10^{-phred/10}$$

since the values are now "cached" in the lookup table. Note that in the code above, we also cap the ASCII value `93`, which corresponds to a phred score of `93 - 33 = 60` (an error probability of `10e-6`). This is optional, but avoids storing non-sensically low error probabilities that are very rarely encountered.

The disadvantage of this approach is that when iterating over the ASCII qualities in a FASTQ record, we must make sure to cap the quality at `93` before indexing into the lookup table.

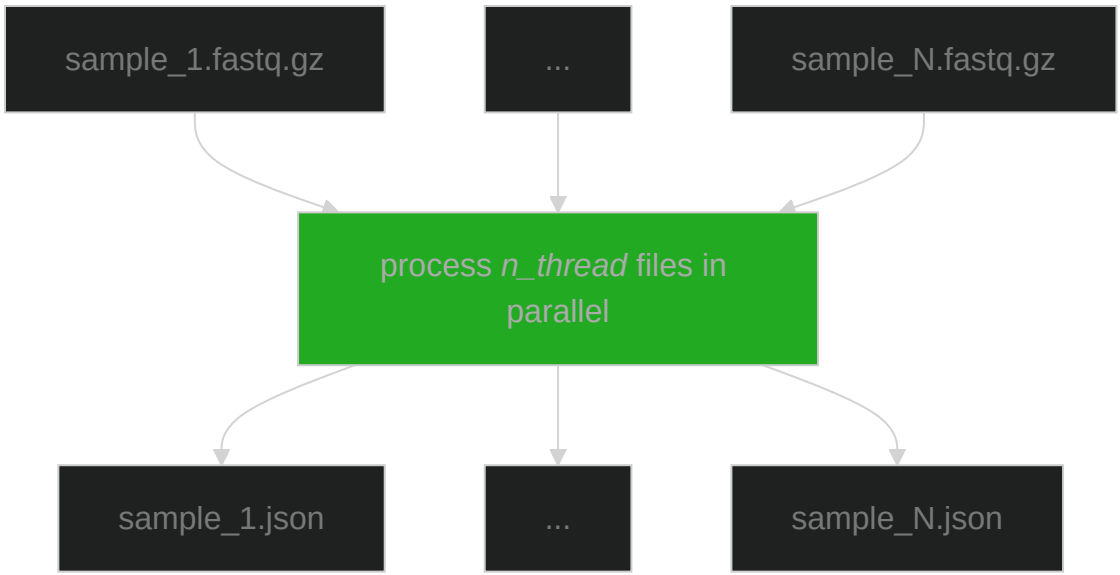
Multithreading

We've discussed multithreading in an earlier chapter, but it is worth revisiting.

Multithreading enables us to literally run code in parallel, which is sometimes advantageous within bioinformatics. However, we should also be a bit careful about when multithreading helps and also when it hurts.

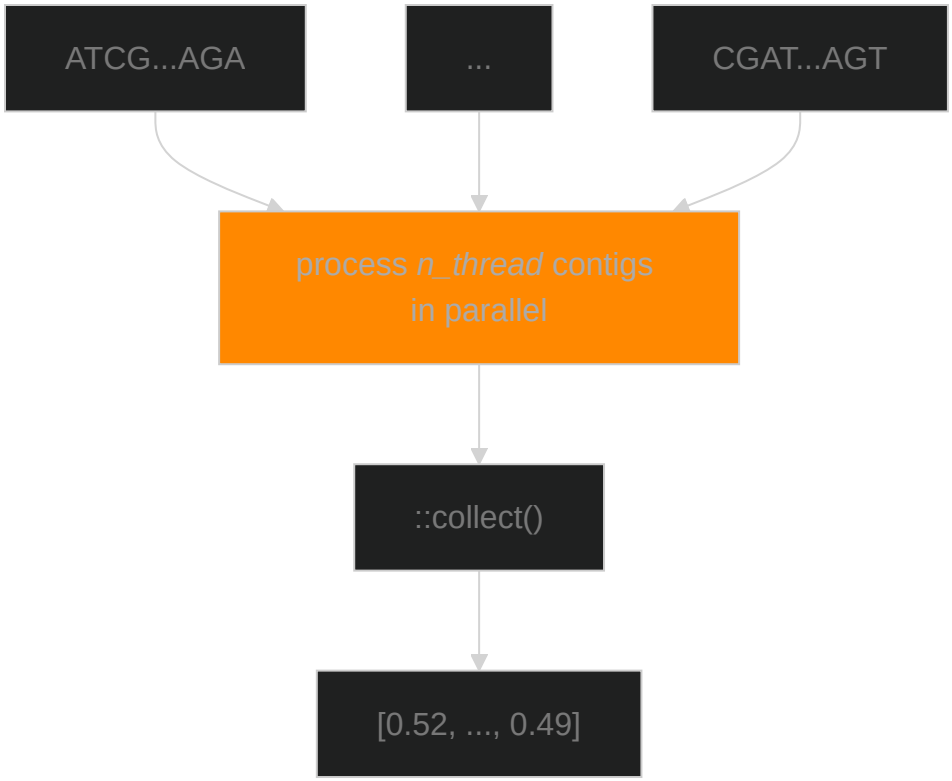
Ideal Use Cases

Multithreading shines during CPU-heavy workloads when the outputs are independent of each other. A good example of this is calculating stats for multiple FASTQ files in parallel. Since the files (and their outputs) are completely independent, adding multithreading can be a huge win.



Conditional Benefits

There are cases when multithreading might help, but the benefit depends on the specifics of the workload. Consider a single FASTA file, for which we'd like to process individual sequences in parallel. Maybe we'd like to calculate the `GC` content for each contig and collect the result in to a `Vec`. Here, we have a tradeoff between the time it takes to calculate the GC content for each contig vs the overhead of maintaining the logistics of a threadpool.



If the contigs are very short, the bottleneck probably isn't calculating GC content but rather the overhead of distributing nucleotide sequences to each thread. We also need to wait for every calculation to finish before we can access the resulting `Vec`. In this case, multithreading adds more overhead than it saves.

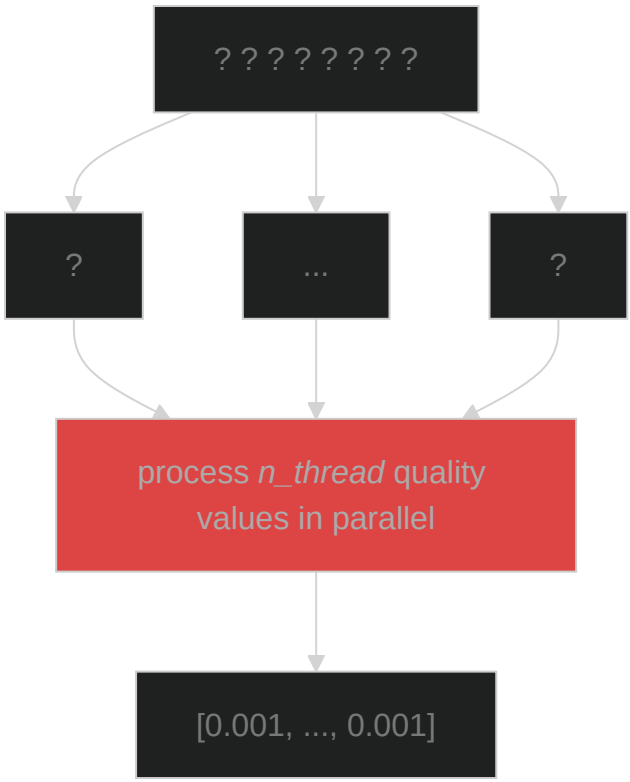
If the contigs are very long, e.g., if the FASTA file contains multiple large single contig genomes, the bottleneck might actually be the GC calculation itself. In this case, multithreading can provide a meaningful speedup.

The takeaway is to consider the ratio of useful computation to coordination overhead. When the work per unit is small relative to the cost of dispatching it to a thread, you might not see a benefit.

When To Avoid Multithreading

A rather questionable example of using multithreading would be trying to convert a FASTQ ASCII quality string to error probabilities. E.g., trying to map `b"????????"` to `[0.001, 0.001, 0.001, 0.001, 0.001, 0.001, 0.001, 0.001]`. In theory, we could use `8` threads to attempt to do this in parallel. However, the arithmetic operation of converting `b'?'` to `0.001` is very fast and the bottleneck here is most likely maintaining the logistics around the threadpool we need to run this operation in parallel.

As a general rule, if the per-item computation takes less time than the overhead of scheduling work onto a thread, multithreading will make things slower, not faster. For cheap operations like arithmetic conversions or table lookups, a simple sequential loop will outperform a parallel one.


? ? ? ? ? ? ? ?
?
...
?
process *n_thread* quality values in parallel
[0.001, ..., 0.001]

Aminoacids

In the past chapters, we have gone through some fundamental ways to analyze and manipulate nucleotide sequences. Now, we'll take a brief look at aminoacid sequences. Luckily, some of the concepts we have implemented for nucleotides also apply to aminoacids, with some minor tweaking of the code. Examples are:

- Counting aminoacids.
- Identifying homopolymers.
- Hamming distance.
- Global and local aligner (with suitable substitution matrix).

Codon Table

For aminoacids, we have to think triplets of nucleotides because this is what encodes aminoacids. In Rust, we can use something like the bio_seq crate. However, for fun we'll create our own very basic `HashMap` using the standard NCBI codon table.

```
use std::collections::HashMap;

fn generate_codon_table<'a>() -> HashMap<[u8; 3], u8> {
    let aa =
b"FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG";

    let base1 =
b"TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG";
    let base2 =
b"TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG";
    let base3 =
b"TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG";

    let map: HashMap<[u8; 3], u8> = (0..aa.len())
        .map(|i| {
            let value = aa[i];
            let key = [base1[i], base2[i], base3[i]];

            return (key, value);
        })
        .collect();

    return map;
}

fn main() {
    let codon_table = generate_codon_table();

    assert_eq!(codon_table.get(b"ATG"), Some(&b'M'));
}
```

Translation

When it comes to translation, there is a couple of things we need to consider:

- `sequence_length` - Is the length of our nucleotide string divisible by 3? If no, then we need to handle this. Otherwise, we might encounter a partial codon.

  E.g., `ATGTTTTAG -> ATG TTT TAG` is a well behaved nucleotide string.

- `frames` - The forward strand has three reading frames, so does the reverse strand. Ideally, we'd try translating all six frames.

  E.g., `...ATGTTTTAG...` can be read in the forward direction as:
  `...ATG TTT TAG...` or
  `....TGT TTT AG...` or
  `.....GTT TTA G...`

  where `...` is the remaining part of the string.

- `ambiguous nucleotides` - We need to decide how to handle ambiguous and softmasked nucleotides.

For this first implementation, we'll use the first frame of the forward strand and `panic!` if the length is not divisible by 3.

```
// [...]

fn translate(seq: &[u8], codon_table: &HashMap<[u8; 3], u8>) ->
Vec<u8> {
    if seq.len() % 3 != 0 {
        panic!("Length of sequence must be divisible by three.");
    }

    let translation: Vec<u8> = seq
        .chunks(3)
        .filter_map(|codon| match codon_table.get(codon) {
            Some(aa) => Some(*aa),
            None => None,
        })
        .collect();

    return translation;
}

fn main() {
    let codon_table = generate_codon_table();

    assert_eq!(translate(b"ATGTAG", &codon_table), b"M*");
    assert_eq!(translate(b"", &codon_table), b"");
}
```

There is lots of room for improvement here, such as accounting for frames and also ending the iteration when we encounter a stop codon.

Accounting For Frames

We can refine our approach for translating a nucleotide sequence by considering the six frames (three in the forward direction and three in the reverse).

The Rust code becomes a bit complex, but basically we:

1. Iterate over our three start positions at (zero-based) indices 0, 1 and 2.
2. From each start position, we chunk the sequence by length 3 to produce complete codons.
3. For each codon we translate to an aminoacid.
4. Finally, we extract the longest translated sequence from the three different frames.

We use `chunk_exact` to skip the last chunks that are not of length 3. `map_while` makes sure we stop iterating when we reach a stop codon (or invalid codon, e.g., if we have ambiguous nucleotides).

The code below does not take the reverse complement into consideration (the reader is encouraged to implement this).

```rust
use std::collections::HashMap;

// [...]

fn get_longest_translation(codon_table: &HashMap<Vec<u8>, u8>, seq:
&[u8]) -> Vec<u8> {
    let frames: usize = 3;

    let longest_translations: Vec<u8> = (0..frames)
        .map(|start_pos| {
            // Process nt string in chunks of three and stop when we
don't have enough nucleotides for a codon.
            let translated: Vec<u8> = seq[start_pos..]
                .chunks_exact(3)
                .map_while(|codon| {
                    let aa = match codon_table.get(codon) {
                        None | Some(b'*') => None,
                        Some(valid) => Some(*valid),
                    };

                    aa
                })
                .collect();

            translated
        })
        .max_by_key(|translation| translation.len())
        .expect("Failed to extract longest translated sequence");

    longest_translations
}

fn main() {
    let codon_table = generate_codon_table();

    // Single codon in first frame.
    assert_eq!(get_longest_translation(&codon_table, b"ATG"), vec!
[b'M']);

    // Two valid codons in first frame.
    assert_eq!(
        get_longest_translation(&codon_table, b"ATGGGG"),
        vec![b'M', b'G']
    );

    // In frame 1 -> M*PP -> M
    // In frame 2 -> CNPP
```

```
    // In frame 3 -> VTP
    assert_eq!(
        get_longest_translation(&codon_table, b"ATGTAACCCCCCC"),
        vec![b'C', b'N', b'P', b'P']
    );
}
```

There is still room for improvement. First, we end the iteration when we encounter a stop codon, but don't actually include it in the return value. Second, using a `HashMap` is not ideal.

Improving Translation Algorithm

The previous approach for mapping codons to amino acids works, but it is not the most efficient. Mainly because of the `HashMap`. There is a performance penalty involved in having to hash input keys and find them.

There is a more brilliant approach, which (you guessed it) involves bit shifts. Let's look again at our codon table:

```
    let aa =
b"FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG";

    let base1 =
b"TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG";
    let base2 =
b"TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG";
    let base3 =
b"TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG";
```

The order is not random, but rather deliberate. The order of the codons is:

```
TTT, TTC, TTA, TTG, ...,
CTT, CTC, CTA, CTG, ...,
ATT, ATC, ATA, ATG, ...,
GTT, GTC, GTA, GTG, ...,
GGG
```

From this, we can derive an order where `T < C < A < G`. We have a total of 64 amino acids, so we need a way to map `TTT -> index 0` and `GGG -> index 63`. Recall that a codon is essentially a kmer of length 3, so we can take inspiration from the bit shift encoding we did in section 6.3. Here, however, we need to map nucleotides accordingly:

- `T => 0b00`
- `C => 0b01`
- `A => 0b10`
- `G => 0b11`

Each nucleotide occupies 2 bits and since we need three of them to form a codon, we need 6 bits in total (we'll use `usize` for convenience though). We have a total of `4^3 = 64` combinations of triplets than can form codons. In addition, using a 6 bit encoding, we cover numbers up to `0b111111 = 2⁵ + 2⁴ ... + 2⁰ = 63`. The formula we'll use to pack the nucleotides is `(base_1 << 4) | (base_2 << 2) | base_3`.

Consider the case of `b"GGG"`, which gives us `base_1 = b'G' => 0b11`, `base_2 = b'G' => 0b11` and `base_3 = b'G' => 0b11`. Doing the bit-shifts (without the ORs) gives:

```
0b...000011 << 4 = 0b...110000  # base_1
0b...000011 << 2 = 0b...001100  # base_2
0b...000011      = 0b...000011  # base_3
```

Now, including the ORs, we get `0b...110000 | 0b...001100 | 0b...000011 = 0b...111111`, which is the number `63` in base 10. You can visually check that `b"GGG"` maps to the amino acid `G`. Similarly, using the 6 bit encoding for `b"TTT"` results in `0b...000000`, which is the number `0` in base 10. We can also visually check that `b"TTT"` corresponds to the first amino acid.

Putting all of this into code, it would look something like this

```rust
const CODON_STANDARD: &[u8; 64] =

b"FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG";

const NT_CODON_MAP: [u8; 256] = {
    let mut map = [0u8; 256];
    map[b'T' as usize] = 0;
    map[b'C' as usize] = 1;
    map[b'A' as usize] = 2;
    map[b'G' as usize] = 3;
    // softmask
    map[b't' as usize] = 0;
    map[b'c' as usize] = 1;
    map[b'a' as usize] = 2;
    map[b'g' as usize] = 3;

    map[b'U' as usize] = 0;

    map
};

pub enum CodonTable {
    Standard,
}

impl CodonTable {
    pub fn table(&self) -> &[u8; 64] {
        match self {
            CodonTable::Standard => CODON_STANDARD,
        }
    }
}

enum Frame {
    First,
    Second,
    Third,
}

impl Frame {
    pub fn start_pos(&self) -> usize {
        match self {
            Frame::First => 0,
            Frame::Second => 1,
            Frame::Third => 2,
        }
    }
```

```
}


fn translate(codon_table_type: CodonTable, frame: &Frame, seq: &
[u8]) -> Vec<u8> {
    let start_pos = frame.start_pos();

    if seq.len() < 3 {
        return vec![];
    }

    let codon_table = codon_table_type.table();

    let mut translated: Vec<u8> = Vec::with_capacity(seq.len() / 3);

    for codon in seq[start_pos..].chunks_exact(3) {
        let b1 = NT_CODON_MAP[codon[0] as usize] as usize;
        let b2 = NT_CODON_MAP[codon[1] as usize] as usize;
        let b3 = NT_CODON_MAP[codon[2] as usize] as usize;

        let index = (b1 << 4) | (b2 << 2) | b3;

        let aa = codon_table[index];

        translated.push(aa);

        if aa == b'*' {
            break;
        }
    }

    translated
}

fn main(){
    let seq = b"ATG";
    let translated = translate(CodonTable::Standard, &Frame::First,
seq);
    assert_eq!(&translated[..], b"M");

    let seq = b"ATGTGA";
    let translated = translate(CodonTable::Standard, &Frame::First,
seq);
    assert_eq!(&translated[..], b"M*");
}
```

A possible improvement here would be to handle ambiguous nucleotides and not just map them to `0` (which is done implicitly since we initialize `[0_u8; 256]` before overwriting with the nucleotide specific encodings).

Amplicon

As we all know, there are multiple different approaches to genome sequencing. I'll list my interpretation of the different approaches below:

- `WGS` - Is a rather broad term, but generally refers to single isolate sequencing. This could be for example sequencing the entire genome from a single bacterial colony.
- `Shotgun` - Generally refers to sequencing the entire genomes from multiple taxa (metagenomic sample). For example, sequencing a gut sample from a patient.
- `Amplicon` - Is a targeted approach, commonly used with PCR. The goal here is to sequence a part of the target genome. One example is amplicon sequencing of the 16S bacterial rRNA region.

Amplicon sequencing has several advantages compared to other sequencing protocols:

- Reduced costs per sample due to sequencing less DNA for a shorter period of time.
- Enables sequencing more samples in parallel due to smaller sample sizes.
- The bioinformatic analysis is generally less computationally heavy.

However, this does not come without disadvantages:

- PCR can introduce artifacts.
- Off targets, depending on the primer design.
- Reduced genomic resolution.

In the following chapters, we'll go through some very basic Rust implementations of common amplicon based analyses.

In Silico PCR

The first amplicon based analysis we'll cover is in silico PCR. The goal is to find certain genomic regions, not by searching for the sequences themselves, but rather through identifying flanking primer regions.

Consider a case where we are looking for a genomic region which can be quite diverse across taxa, but has very conserved flanking sites. This is the case for e.g., the 16S rRNA region in bacteria. Alignment approaches might not be suitable if we are unsure of, or expect a large diversity in the target region.

The following example tries to illustrate this, where the conserved flanking regions are `ATATAT` and `GTGTGT`.

```
...ATATAT ACGTGACGTGACGGAGAT GTGTGT...  # taxa_1
...ATATAT ACCTAGCGTAGTCGAGTG GTGTGT...  # taxa_2
...ATATAT ACCTAGCGTACGAGTG GTGTGT...    # taxa_3
```

Instead of looking directly at the target region we can search for flanking sites, extract the target region and use some kind of length cutoff to prevent outliers. In the example above, using flanking sites `ATATAT` and `GTGTGT` with a length threshold of `>= 15` and `<= 20` would capture all three target regions with some margin.

```rust
fn primer_search(primer: &[u8], seq: &[u8]) -> Option<usize> {
    for (i, window) in seq.windows(primer.len()).enumerate() {
        if window == primer {
            return Some(i);
        }
    }
    return None;
}

fn is_pcr<'a>(start: &'a [u8], end: &'a [u8], seq: &'a [u8]) ->
Option<&'a [u8]> {
    let start_index = primer_search(start, seq);

    let end_index = primer_search(end, seq);

    match (start_index, end_index) {
        (Some(s), Some(e)) => {
            let actual_start = s + start.len();

            if actual_start < e {
                return Some(&seq[actual_start..e]);
            }

            return None;
        }
        _ => return None,
    }
}

fn main() {
    assert_eq!(is_pcr(b"A", b"G", b""), None);
    assert_eq!(is_pcr(b"AT", b"CG", b"ATCG"), None);
    assert_eq!(is_pcr(b"AT", b"CG", b"ATTCG"), Some(&b"T"[..]));
    assert_eq!(
        is_pcr(b"AAA", b"TTT", b"CGCGCGAAACCCCCCTTTCGCGCG"),
        Some(&b"CCCCCC"[..])
    );
}
```

In the code example above, our naive implementation just uses an exact string search for our flanking regions. We then check if the start primer is located prior to the end primer. If it is, we extract the interjacent sequence. Some good improvements to the code would be:

- Enable multiple matches to the start and end primer for finding multiple interjacent regions.
- Add a `min_len` and `max_len` criteria to filter out potential outliers.
- Check both forward and reverse complements.
- Replace `.windows()` with something faster. For exact matches, memchr is a good alternative.
- Add fuzzy search to allow for a few mismatches between the primers and the sequence. A good alternative here is myers from the bio crate since it supports ambiguous nucleotides.

Clustering

Within amplicon analysis, read clustering is commonly applied as a type of dimensionality reduction. A typical example of this is prior to taxonomic classification. Usually, it is redundant and computationally expensive to classify every single read in a sample. Especially if multiple reads belong to the same taxa.

Consider a theoretical example with a known prior taxonomic distribution where we have 100,000 reads, half of which belong to *Escherichia coli* and the rest belong to *Staphylococcus aureus*. Instead of classifying all reads, we apply read clustering and get two distinct clusters, each containing 50,000 reads. From each cluster, we pick one representative sequence (E.g., the read with the highest quality) and classify only that one. Pretend that our two representatives (one for each cluster) classifies as *Escherichia coli* and *Staphylococcus aureus* respectively. We then extrapolate the classification for both clusters and say that 50,000 reads belong to *Escherichia coli* and 50,000 reads belong to *Staphylococcus aureus*, even though we only classified two sequences.

Obviously, we don't always know the taxonomic distribution beforehand. Maybe if we use a mock sample, otherwise generally we don't. There are also several questions that need to be addressed regarding our theoretical example:

- What algorithm should we use for read clustering?
- What thresholds are suitable for considering a read part of a cluster?
- How do we pick a suitable representative sequence from each cluster?

Algorithms

There are multiple different approaches we can use for read clustering, each with its pros and cons. Common methods include:

Alignment based methods rely on some version of global, semi-global or local alignment. This approach is relatively slow but can be highly accurate.

Kmer based methods rely on kmers for sequence similarity. This is an alignment free method that is fast but not always as sensitive as alignment based methods. In order to save even more space and time, we can use minimizers or syncmers.

Cluster free methods do not really belong in this chapter, but we'll briefly mention them anyways. These methods usually try to classify reads directly without an intermediary clustering step. One example is EMU, which leverages read error rates and an iterative maximum likelihood algorithm.

Thresholds

We need some kind of metric to decide if a read belongs to an already existing cluster, or if it should initiate a new cluster. Obviously, this depends on if we use an alignment based method, or a kmer based method.

Historically, for alignment methods a threshold of 97% similarity has been used to classify two sequences as belonging to the same Operational Taxonomic Unit (OTU). The obvious downside to the OTU approach is that taxa with >97% similarity are collapsed.

A more recent approach is to use Amplicon Sequence Variants, which is a more high resolution approach. With the improvements in NGS data quality, it is much easier to distinguish sequencing errors from true biological variation. This means we can relatively accurately identify the exact taxonomic variation in the sample and leverage that. The ASV approach is commonly used for Illumina data, but might be unsuitable for error prone Nanopore data.

For kmer approaches, we can use set theory to calculate how many, and the fraction of, shared kmers we have between the read and the cluster. The jaccard index is a good example of this.

Read error rates are also something that must be considered. This is not generally a problem for Illumina data, but it can be for Nanopore. For example, if the average error rate in a sample is 3%, we need to take this into consideration when we choose a threshold. Error rates also become a problem

during classification. If the average error rate is 3%, but the taxa we are trying to distinguish are >97% similar, we might run into issues.

Representatives

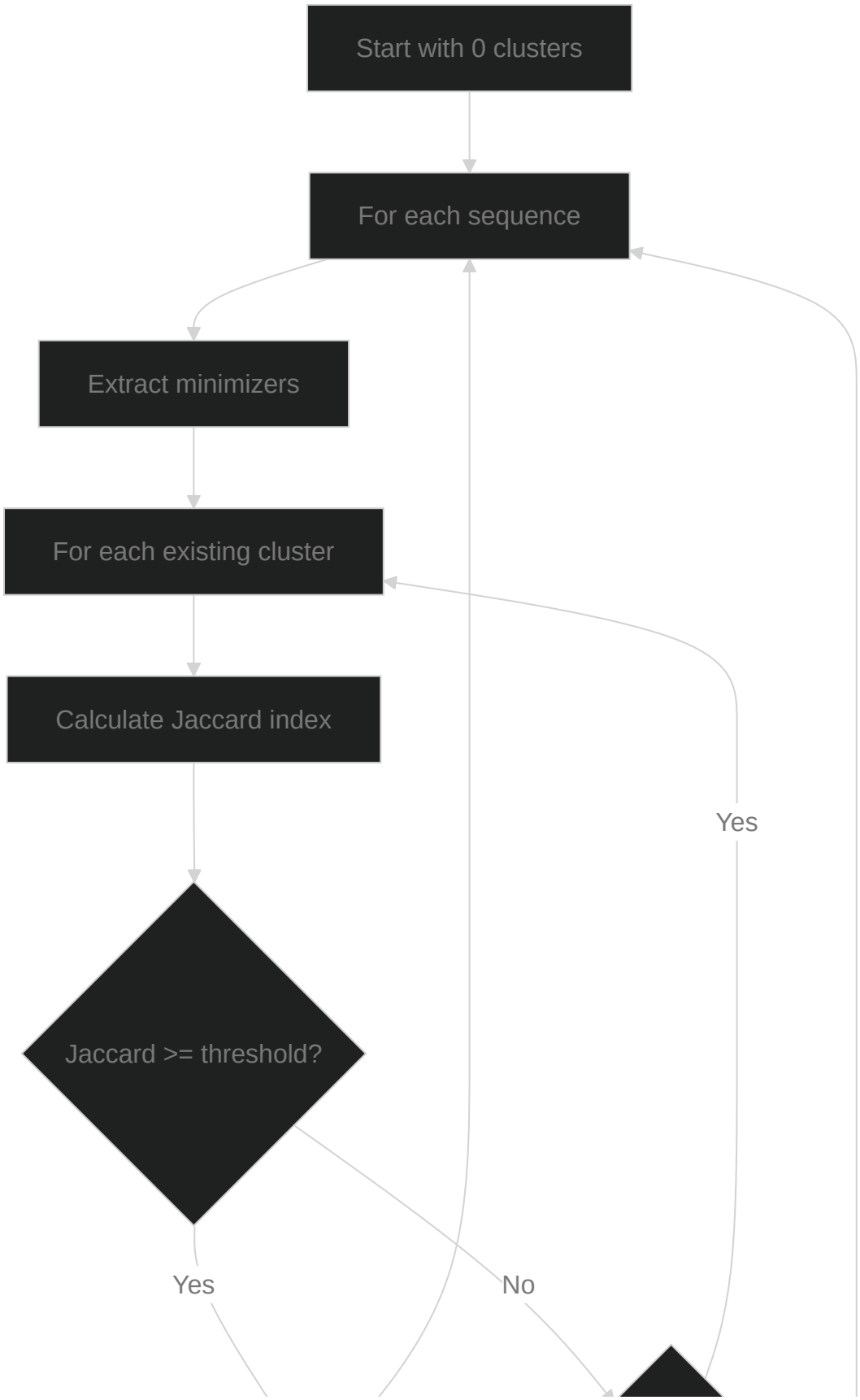
How to choose a representative sequence from a cluster varies across bioinformatic tools. A few common ways are:

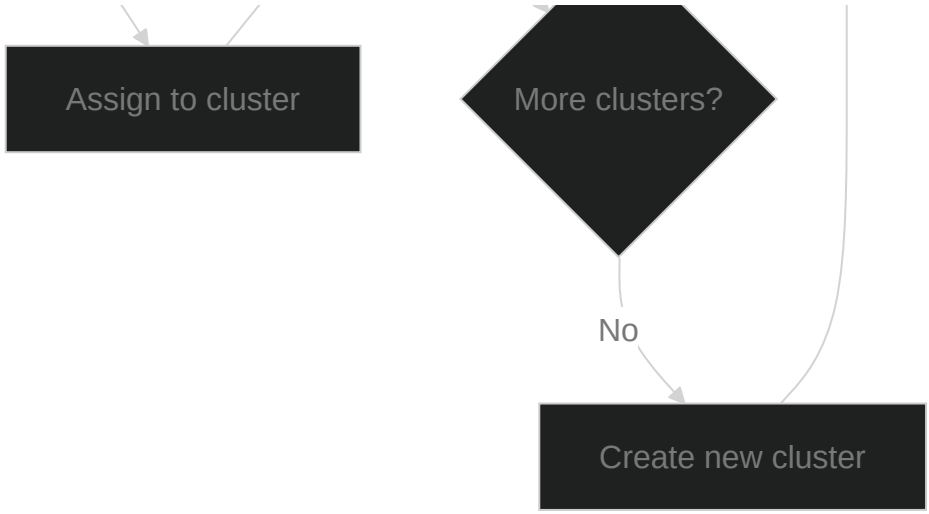
- **First seen in cluster**. The most straightforward way is to choose the read that initiated the cluster. However, since we don't consider read length, error rate or anything else, this method might not be appropriate unless we have done any prior read sorting.
- **Choosing the longest sequence**. One could argue that the longer the representative, the longer alignment or the most kmers we can generate. However, we probably need to consider factors such as outliers and sequence errors as well.
- **Choosing a random sequence**. This approach introduces a bit of stochasticity unless we use a seed.
- **Choosing the highest quality sequence**. For Nanopore samples, this is a highly suitable approach because the sequence with the lowest error rate generally has the potential to generate the best classification.
- **Least intercluster distance**. For each cluster, we choose the sequence that has the least total distance to all other sequences in the cluster. This approach uses some kind of concept of a "mean" and is a bit more robust towards accidentally picking outliers, but might be a bit more computationally heavy.
- **Generating a consensus sequence**. If we expect the sequences in a cluster to be highly similar, we could generate a consensus sequence. This essentially means we generate a new sequence, based on some kind of multiple sequence alignment of all reads in the cluster, and take the majority vote in each position.

Code example

Clustering algorithms can be quite complex so we'll create a *very* basic native Rust implementation here, which uses minimizers (using code from our sloppy, earlier implementation). We won't bother with sorting sequences or choosing representatives but rather just keeping track of how many clusters we generate and their members.

Conceptually what we do is:


Start with 0 clusters
For each sequence
Extract minimizers
For each existing cluster
Calculate Jaccard index
Jaccard >= threshold?
Yes
No
Yes




```
// [...]

fn jaccard_index(h1: &HashSet<&[u8]>, h2: &HashSet<&[u8]>) -> f64 {
    let num_common = h1.intersection(h2).count();
    let num_total = h1.union(h2).count();

    return num_common as f64 / num_total as f64;
}

fn cluster<'a>(
    seqs: &[(&'a str, &[u8])],
    kmer_size: usize,
    window_size: usize,
    threshold: f64,
) -> Vec<(HashSet<String>, Vec<&'a str>)> {
    let mut clusters: Vec<(HashSet<String>, Vec<&str>)> =
Vec::new();

    for (seq_name, seq) in seqs {
        let minimizers = get_minimizers(seq, window_size,
kmer_size);
        let minimizer_set: HashSet<&[u8]> =
minimizers.iter().map(|m| m.as_bytes()).collect();

        let mut assigned: bool = false;

        for (cluster_hashset, cluster_members) in &mut clusters {
            let cluster_hashset: HashSet<&[u8]> =
                cluster_hashset.iter().map(|h|
h.as_bytes()).collect();

            let d = jaccard_index(&minimizer_set, &cluster_hashset);
```

```
            if d >= threshold {
                assigned = true;
                cluster_members.push(seq_name);
                break;
            }
        }

        if !assigned {
            clusters.push((minimizers, vec![seq_name]));
        }
    }

    clusters
}

fn main() {
    let seqs: Vec<(&str, &[u8])> = vec![
        ("seq_1", b"AAACACCGTGTGGGGCTAGCTATTTCACATGTGTCATGCAT"),
        ("seq_2", b"AAACACCGTGTGGGGCTAGCTATTTCACATGTGTCATGCAT"),
        ("seq_3", b"TACGTACGTACGTACGTACGATCGATCGTACGATCGATCGT"),
        ("seq_4", b"TACGTACGTCCGTACGTACGATCGATCGTACGATCGTTCGT"),
    ];

    let window_size: usize = 3;
    let kmer_size: usize = 3;
    let threshold: f64 = 0.6;

    let clusters = cluster(&seqs[..], kmer_size, window_size,
threshold);

    println!("Num clusters: {}", clusters.len());

    for (_, members) in clusters {
        println!("{:?}", members)
    }
}
```

In this example, we have plenty of room for improvement. Firstly, we'd probably want to filter out low quality sequences if we have access to phred scores. Second, we could sort by quality in descending order. This ensures that new clusters are initiated with the highest quality sequences. In addition, our current approach is greedy, meaning that a sequence is assigned to the first best cluster we find. We don't know if there is a better matching cluster later in the iteration. Finally, we obviously want a more efficient approach for

generating minimizers (such at minimizer_iter or maybe even our own bit-shift encoded implementation).

In summary, this example is crap when it comes to performance. It does, however, conceptually illustrate how clustering algorithms work. isONclust3 and USEARCH are great examples of fast and high performance implementations.

Classification

The final part of the amplicon chapter is classification. In essence, classification means we have some kind of confidence that a sequence originates from a specific taxon

Alignment based methods

Usually, we'd use a global, semi-global or local aligner. The advantage of alignment based methods is that we can easily extract alignment metrics, such as the number of mismatches and indels, as well as alignment length, percent identity and other metrics. This is relevant if we not only want to find the best database hit, but also know how and where this database hit differs from our sequence.

The downside to alignment based methods is that they are slow when the number of sequences and database entries grow. Another downside, not commonly talked about, is the fundamental issue with choosing only the best database hit. Pretend that our sequence matches to a particular database entry (taxa X) with 99.9% identity, but also matches to another database entry (taxa Y) with 99.85% identity. Can we really be sure that our sequence belongs to taxa X? The difference in identities between the hits could be as little as a single nucleotide. We better make sure our sequence does not contain any errors, since a single sequencing error theoretically could flip the classification from taxa Y to taxa X. If taxa X and taxa Y are different species from the same genus, maybe it makes sense to classify this sequence on genus level. This is especially true for Nanopore data, where sequencing errors can reach several percent.

One algorithm that is worth mentioning here is EMU, which uses minimap2 to align Nanopore reads to the entire database. Through an iterative maximum likelihood algorithm, based on alignment metrics, it can accurately estimate taxonomic abundances in the sample.

Alignment free methods

These typically use kmers and are based on exact matches. There are numerous ways to use kmers for classification, but one algorithm in particular is worth mentioning. SINTAX from the USEARCH toolkit uses a rather interesting kmer based classification approach that relies on bootstrapped subsampling of kmers to generate a classification confidence score for each sequence.

In the code example below, we'll implement our own working prototype of a kmer based classifier. To keep things simple, we'll just compare and extract common kmers between our sequences and database entries. For each sequence, we keep the database hit that had the highest match to our sequence.

```
fn main() {
    println!("");

    // Define some mock sequences.

    // Define some mock database entries.

    // Extract kmers from both sequences and database entries.

    // For each sequence, return the database entry that had the
most number of common kmers.
}
```

Blueprints

This section acts as a collection of templates that can be used to build bioinformatic applications in Rust. Each chapter contains a brief introduction, followed by non-runnable template code that can be used as inspiration for building your own application.

Argument Parsing

There are multiple ways to handle argument parsing in Rust. One easy way is to use `std::env`, but it quickly becomes rather complex when the number and types of arguments increase.

An alternative approach is to use clap, which has worked really well for me personally. Defining arguments is as easy as defining a struct with a clap specific derive macro.

For reproducibility purposes, the code example uses the following Cargo.toml dependency:

```
[dependencies]
clap = { version = "4.5.39", features = ["derive"] }
```

Pretend we are creating a CLI called `fasta_cli` for filtering and parsing a FASTA file. These might be some of the arguments we think are relevant.

```
use clap::Parser;
use clap::value_parser;
use std::path::PathBuf;

#[derive(Parser, Debug)]
struct Args {
    #[arg(short, long, help = "Path to fasta file.")]
    fasta: PathBuf,

    #[arg(long, default_value_t = 100, help = "Min allowed read
length.")]
    min_len: usize,

    #[arg(long, default_value_t = 1000, help = "Max allowed read
length.")]
    max_len: usize,

    #[arg(long, default_value_t = 15, value_parser = value_parser!
(u16).range(7..31))]
    kmer_size: u16,

    #[arg(short, long, default_value_t = 8)]
    threads: usize,
}

fn main() {
    let args = Args::parse();

    // Now, we can access the values as args.fasta, args.min_len,
etc.
}
```

Once compiled, we can run our binary as `fasta_cli --fasta <file.fasta> --min_len <min_len> --max_len <max_len> --kmer_size <kmer_size> --threads <threads>`.

Clap also supports more complex argument parsing, such as global flags, subcommands and enums. See e.g., fastq_rs for examples of this.

Commands

In Python, commands can easily be run with `subprocess` or through the very neat `sh` module.

In Rust, we can use `std::process::Command` to achieve something similar to `subprocess`. The example below shows how to call `minimap2` to align reads against a genome. We'll use `thiserror` to create two custom errors, the last of which will capture stderr if the command exits with a non-zero exit code.

For reproducibility purposes, the code example uses the following Cargo.toml dependency:

```
[dependencies]
thiserror = { version = "2.0.16" }
```

Note, we obviously need `minimap2` installed in order for this code to work properly.

```rust
use std::{path::PathBuf, process::Command};
use thiserror::Error;

#[derive(Debug, Error)]
enum RunCommandError {
    #[error("Failed to initialize child process.")]
    CommandInitError,

    #[error("Command exited with non-zero exit code.")]
    NonZeroExitCodeError(String),
}

fn minimap2_align(fastq: PathBuf, fasta: PathBuf, outfile: PathBuf)
-> Result<(), RunCommandError> {
    let result = Command::new("minimap2")
        .arg(fasta)
        .arg(fastq)
        .arg("-o")
        .arg(outfile)
        .arg("-a")
        .output()
        .map_err(|_| RunCommandError::CommandInitError)?;

    match result.status.success() {
        true => Ok(()),
        false => Err(RunCommandError::NonZeroExitCodeError(
            String::from_utf8(result.stderr).unwrap(),
        )),
    }
}

fn main() {
    let fastq = PathBuf::from("reads.fastq.gz");
    let fasta = PathBuf::from("genome.fasta");
    let outfile = PathBuf::from("out.sam");

    minimap2_align(fastq, fasta, outfile).unwrap();
}
```

Why would we want to call `minimap2` from Rust instead of e.g., Python or Bash? In many cases, we wouldn't. If the goal is to simply align reads and parse the generated .sam file with SAMtools, then Python or Bash are probably better alternatives.

However, maybe our goal is to align reads and parse the .sam file with rust_htslib to calculate some more advanced alignment statistics that require high performance. Maybe we also had a Rust preprocessing step for the fastq file prior to alignment. In those cases, it *could* be justified to also call `minimap2` from Rust to make the codebase more unified.

DataFrames

Reading and manipulating dataframes in Rust is actually not that easy. Polars is the crate to use for dataframes but honestly, the Rust API is not that good. In my opinion, it is much easier to either use the Python API, or simply use pandas and completely skip Rust.

With that said, here is a small example of reading a .tsv file in Rust, using polars.

For reproducibility purposes, the code example uses the following Cargo.toml dependency:

```
[dependencies]
polars = { version = "0.50.0", features = ["lazy", "csv"]}
```

```
use polars::prelude::*;

/// Assumes tab separated values and that the first line is the
header.
fn tsv_to_df(tsv: &PathBuf) -> LazyFrame {
    let df = LazyCsvReader::new(PlPath::new(tsv.to_str().unwrap()))
        .with_separator(b'\t')
        .with_has_header(true)
        .with_truncate_ragged_lines(true)
        .finish()
        .expect("Failed to read tsv to DataFrame.");

    df
}

fn main() {
    let tsv: PathBuf = PathBuf::from("my_file.tsv");

    let df = tsv_to_df(&tsv);

    // Do stuff with the dataframe...
}
```

Needletail

The needletail crate is perfectly suited for reading and parsing FASTA and FASTQ files. It is very fast and efficient but not easily parallelized. Here, we'll outline a template for reading a fastq file and looping over each record.

For reproducibility purposes, the code example uses the following Cargo.toml dependency:

```
[dependencies]
needletail = { version = "0.6.3" }
```

```
use needletail::parse_fastx_file;
use std::path::PathBuf;

fn main() {
    let fastx_file = PathBuf::from("file.fastq.gz");

    let mut reader = parse_fastx_file(&fastx_file).expect("Failed to
initialize FastxReader.");

    while let Some(record) = reader.next() {
        let record = match record {
            Ok(record) => record,
            Err(_) => continue,
        };
    }

    // Do stuff with the record...
}
```

The advantage of using `parse_fastx_file` is that we can read both .fasta and .fastq files in plain or gzip format, which is very convenient.

Note that in this example, we just skip invalid records. In practice, we probably want to log that as a warning or error.

Bio

An alternative needletail is the bio crate. This fastq reader is not as fast as needletail and does not natively handle both gzipped files. It is however easily parallelized with rayon using `par_bridge()`. In the following example, we use flate2 together with rayon and `bio::io::fastq::Reader` to enable multi-thread support for gzipped fastq files. For a FASTA equivalent reader, check out `bio::io::fasta::Reader`.

For reproducibility purposes, the code example uses the following Cargo.toml dependencies:

```
[dependencies]
bio = { version = "2.3.0" }
flate2 = { version = "1.1.2" }
rayon = { version = "1.10.0" }
```

```
use bio::io::fastq::Reader;
use flate2::read::MultiGzDecoder;
use rayon::prelude::*;
use std::{fs::File, path::PathBuf};

fn main() {
    let fastq_file = PathBuf::from("file.fastq.gz");

    let f = File::open(fastq_file).expect("Failed to open provided
file.");

    // Wrap in GzDecoder since file is in gzip format.
    let gzip_reader = Reader::new(MultiGzDecoder::new(f));

    gzip_reader.records().par_bridge().for_each(|record| {
        let record = match record {
            Ok(record) => record,
            Err(_) => return,
        };

        // Do stuff with the record...
    });
}
```

Since `.records()` returns an iterator, we can apply loads of different iterator chaining steps here, such as `.map()` or `.filter_map()` followed by `.collect()`.

A word of caution - multithreading is great in certain circumstances, but not all. If the processing time for each record is very short, for example if we only calculate the length of each record, multithreading probably does not help. It might actually be slower. In those cases, needletail is probably a better alternative.

Resources

Reading References

- The Rust Programming Language - Official Rust book.
- Rust By Practice - Practice Rust.
- The Rustonomicon - Even more Rust!
- The Rust Performance Book - How to optimize Rust for performance.
- Rustfinity - Learn Rust through interactive problems.

Viewing References

- Jon Gjengset
- Bogdan from Let's Get Rusty
- Dave from You Suck at Programming
- Max Taylor
- Rust Curious

Listening References

- Rust in Production

Awesome Rust crates

General purpose awesome Rust crates! For a mega list of even more crates, see Awesome Rust.

- Clap - Command line argument parsing.
- Rstest - Rust test fixtures.
- Rayon - Multithreading library for iterators.
- Dashmap - Concurrent HashMaps.
- Serde - Serialization/Deserialization.
- Thiserror - Easily create custom error types.
- Anyhow - Idiomatic error handling.
- Log and SimpleLogger - Switch those pesky println! macros for proper logging.
- Flate2 - Compression/Decompression library.
- Bindgen - Rust bindings for C and C++.
- Polars - Blazingly fast dataframes. **NOTE** - Using polars with native Rust can be a bit cumbersome. An alternative is to use the Python bindings.
- Pyo3 - Generate Rust bindings to Python or vice versa.
- Plotly-rs - Rust bindings for the popular Plotly plotting library.
- Linfa - Closely resembles Python's scikit-learn for Machine Learning applications.
- Statrs - Statistical utilities such as distributions, etc.
- Validator - Struct validation.
- Dioxus - Fullstack framework in Rust that resembles React. Build your own (bioinformatic) web or desktop applications!
- Ratatui - Build TUI applications in Rust.
- Iced - Build GUI applications in Rust.

Awesome bioinformatic tools

Alignment related:

- Minimap2 [C] - Pairwise aligner. Written by the legendary Heng Li. The go-to for Oxford Nanopore and PacBio data.
- BWA-MEM [C] - Pairwise aligner suitable for Illumina data.
- Parasail [C] - General purpose pairwise aligner.
- Parasail-rs [Rust] - Rust bindings for the parasail library.
- BLAST [C++] - The usual go-to for local sequence alignment.
- MAFFT - Multiple sequence aligner.
- Clustal Omega - Multiple sequence aligner.

Assembly related:

- Flye [C/C++] - Genome assembler for Oxford Nanopore or PacBio data.
- IDBA [C++] - Illumina specific genome assembler.
- SPAdes [C++] - Genome assembler suitable for Illumina or IonTorrent data.
- Myloasm [Rust] - Longread metagenome assembler.

Variant calling related:

- Clair3 [Python] - Variant caller suitable for Illumina, Oxford Nanopore or PacBio data.
- Medaka [Python] - Variant caller and polishing tool specifically for Oxford Nanopore data.
- Freebayes [C++] - Variant caller suitable for Illumina and IonTorrent data. Questionable use for Oxford Nanopore data.

Misc:

- SAMtools [C] - SAM file manipulation.
- BCFtools [C] - VCF file manipulation.
- Kmer-cnt [C] - Several kmer counting algorithms.
- Seqkit [Go] - Parsing and processing FASTA/Q files.
- Tablet [Java] - Graphical alignment visualizer.

- Bandage [C++] - Assembly graph visualizer.

Tools written in Rust:

- Needletail [Rust] - Parsing and processing FASTA/Q files.
- Bio [Rust] - General purpose bioinformatic tool for alignment, file processing and much more.
- Bio-seq [Rust] - Toolbox for bit-packed biological sequences.
- Sylph [Rust] - Metagenomic classification tool.
- Rust Htslib [Rust] - Rust bindings for Htslib.
- Nextclade [Rust] - Virus specific tool for alignment, SNP calling, clade assignment and more.
- Herro [Rust] - Deep-learning based error correction tool for Oxford Nanopore data.

Thank You

This was it. If you have made it to this point, I sincerely want to thank you for spending your time reading through this book.

Initially, this started off as a private project with the goal of learning more about the Rust programming language. Since my background is in biotech/bioinformatics, the combination Rust + bioinformatics felt very natural to me. Along the way, I realized that even though there are some bioinformatic tools written in Rust, there was no really good tutorial on just how well Rust integrates with the bioinformatic landscape.

I strongly opted out of vibe-coding this entire book. If I'd used ChatGPT and something like Claude, this project would probably have been done in a few days. Instead, I chose the difficult but proper way of reading a lot of resources and documentation. I did a lot of testing and failing until I reached something I thought was good enough. It is not perfect, nowhere close actually.

Contributions

This project is far from done. It probably never will be. However, I stand by the concept of open-source and people working together to create software that is available for everyone. If you feel like you could contribute in any shape, way or form, then I'd be more than open to this.