Universität Bielefeld
Technische Fakultät

AG Genominformatik
Dr. Lucas Robidou

The Rust programming language
Summer 2024 / 2025

Exercises

1 k-mers

Checking if two DNA sequences are highly "similar" can be done using the Needleman–Wunsch algorithm (`https://en.wikipedia.org/wiki/Needleman-Wunsch_algorithm`). However, this algorithm is not linear. Fortunately, we can check if two strings are *likely* to be similar with a linear algorithm, which could then act as a filter to prevent running Needleman–Wunsch on obviously dissimilar sequences. This exercise focuses on the first rapid check, so knowledge about Needleman–Wunsch is out of scope.

The fast algorithm relies on k-mers. A k-mer is a word of fixed size k. For instance, the 3-mers of "ABCDE" are "ABC", "BCD", and "CDE". The intuition behind k-mers is that if two strings share the same set of k-mers, they are likely to be similar. Checking if a set contains a k-mer is a constant-time operation, so checking all the k-mers of a string in a set is linear in the size of the string.

Let's write a struct, `KmerIterator`, and make it an iterator over all the k-mers of a `String`.

1. Wite a `from` method that takes a `String` and a value for k and returns a `KmerIterator`.
2. Implement the `Iterator` trait. Make the `next` method return a String.
3. Verify that you can use `collect` on your iterator to collect all k-mers of a String in a `HashSet`.
4. Write some unit tests to check that everything is alright.

2 DNA files

A fasta file is a file that contains two types of lines. The lines that start with a `>` are identifiers. The others are DNA sequences.

Run this Python script twice (e.g. `python generate.py > index.fa` and `python generate.py > query.fa` to generate two fasta files. Then, replace one line of DNA in the query file with one from the index file. Now, your goal is to find which line from the query is found in the index.

```python
import random as rd

DNA_ALPHABET = {"A", "C", "T", "G"}

def rdseq(length):
    s = list(DNA_ALPHABET)
    return "".join((rd.choice(s) for _ in range(length)))

def main():
    length = 500

    for i in range(1000):
        print(f"> {i}")
        print(rdseq(length))


if __name__ == "__main__":
    main()
```

1. Open the index file line by line by using a `BufRead` and its `lines` method.
2. Using only filter, map, flatten, and collect, write a function that returns the set (`std::collections::HashSet`) of k-mers of the DNA content of a file. Hint: use flatten (`https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.flatten`)to turn an iterator of `KmerIterator`s into an iterator of k-mers.
3. Similarly, open the query file, and using only filter, map, and collect, compute a vector of DNA lines that share at least 90% of their k-mers with the index file.
4. How many lines are found in the index when k is 3, 10, 20, and 31? Why does the number change?

3 The Bloom filter

This exercise might be a bit more difficult if you have never used a Bloom filter before. Just go as far as you can!

In the previous exercise, we used a set of `String`s. Depending on the input file, this set might become huge. We can fix that by using Bloom filters.

A Bloom filter is a probabilistic datastructure that can be used to determine if an element is likely to be present in a set or not. A Bloom filter uses less memory than a set, at the cost of a non-zero false positive rate. That is to say, when you ask if an element is in the set:

- if the element is present, the Bloom filter is guaranteed to return `true`
- if the element is absent, the Bloom filter may return `false`, but sometimes it may return `true` (which is a false positive).

Internally, a Bloom filter consists of two things:

- a vector V of booleans of a given, fixed size, noted as $|V|$
- a hash function that maps any element to an integer, which is used to get a position in V

Initially, all booleans in V are set to false. When inserting an element, you hash it and set `V[hash(element)%|V|]` to `true` (the modulo is there to prevent overflows). To check if an element is in the set, you check if `V[hash(element)%|V|]` is `true`. The hash of an element is essentially random, so an absent element may be hashed to the same position as an inserted element, leading to a false positive. The bigger the size, the more possible positions you have, the less likely a false positive is.

Let's implement a Bloom filter.

```
use std::hash::Hash;
use std::marker::PhantomData;

pub struct Bloom<T: Hash> {  // accepts any T that can be hashed
    nb_elem: usize,
    data: Vec<bool>,
    _marker: PhantomData<T>,  // here to tell the compiler it's OK not to use T in any field of Bloom
}
```

1. Write a `new` method that takes the size of the filter and returns an empty filter.
2. Find a way to hash elements in Rust using the standard library.
3. Write an `add` method that adds an element to the filter.
4. Write a `contains` method that checks if an element is (maybe) present.

5. Write a method that takes an `Iterator<Item = T>`, iterates over every element, and returns the proportion of elements that are found in the filter. Use `for_each` and a `lambda`.
6. Write unit tests for your methods.
7. (advanced) Using a vector of booleans actually wastes a lot of space, as a single boolean costs 8 bits, but retains only one bit of information. Use an array of `u64` instead, and treat each bit of these `u64` as a boolean. Modify the methods of the struct accordingly. This should not change the public interface of your filter, but should reduce its size by a factor of 8 (nice!).

Once done, this snippet allows you to call `collect_bloom` on any iterator to collect it in a Bloom filter:

```
pub trait CollectBloom<T: Hash> {
    fn collect_bloom(self, size: usize) -> Bloom<T>;
}

/// Implement the trait for any struct on which you can call `into_iter`.
impl<I, T: Hash> CollectBloom<T> for I
where
    I: IntoIterator<Item = T>,
{
    fn collect_bloom(self, size: usize) -> Bloom<T> {
        let mut filter = Bloom::new(size);
        self.into_iter().for_each(|element| filter.add(&element));
        filter
    }
}
```

8. Write a test to check that you can insert a vector of integers in the filter using `iter` and `collect_bloom`, and that all of them are found again when searched inside.

4 Putting things together

1. Replace the `HashSet` in the previous exercise with a Bloom filter.
2. Set a size of 1. What happens?
3. Set a size of 10000. What happens?
4. Assuming a letter takes 8 bits, how many 31-mers can you represent using 10000 bits ?