LLM Assisted Library Development

Rust@Wellcome meetup Apr. 13
jeff-k

A library for manipulating sequencing data:

```
let sequence: String = “ACTGACTGCAT”.to_string();
let slice: &str = &sequence[2..];
```

Versus:

```
let sequence: Seq<Dna> = “ACTGACTGCAT”.into()?;
let slice: &SeqSlice<Dna> = &sequence[2..];
```

The Seq type should protect the user from the clever bit-twiddling hacks and protect the computer from the user’s possible misuse of types

What operations should a library support?

Let’s work through some Rosalind exercises to shake down the Seq implementation using chatgpt4:

“Genome Assembly as Shortest Superstring (solved by 3388)”

```
>Rosalind_56
ATTAGACCTG
>Rosalind_57
CCTGCCGGAA
>Rosalind_58
AGACCTGCCG
>Rosalind_59
GCCGGAATAC

Output:
ATTAGACCTGCCGGAATA
```

:o

```
fn find_best_overlap(s1: &SeqSlice<Dna>, s2: &SeqSlice<Dna>) -> usize {
    for overlap in (1..=s1.len()).rev() {
        if s1.len() >= overlap && s2.len() >= overlap && s1[s1.len() - overlap..] == s2[0..overlap] {
            return overlap;
        }
    }
    0
}

fn shortest_superseq(seqs: &[Seq<Dna>]) -> Seq<Dna> {
    let mut superseq = seqs[0].clone();
…
    for _ in 1..n {
…
        for i in 1..n {
…
                let overlap = find_best_overlap(&superseq, &seqs[i]);
…
        superseq.extend(seqs[best_index][best_overlap..].into_iter());
        used[best_index] = true;
```

…but wait

```
seqs[0].clone()
```

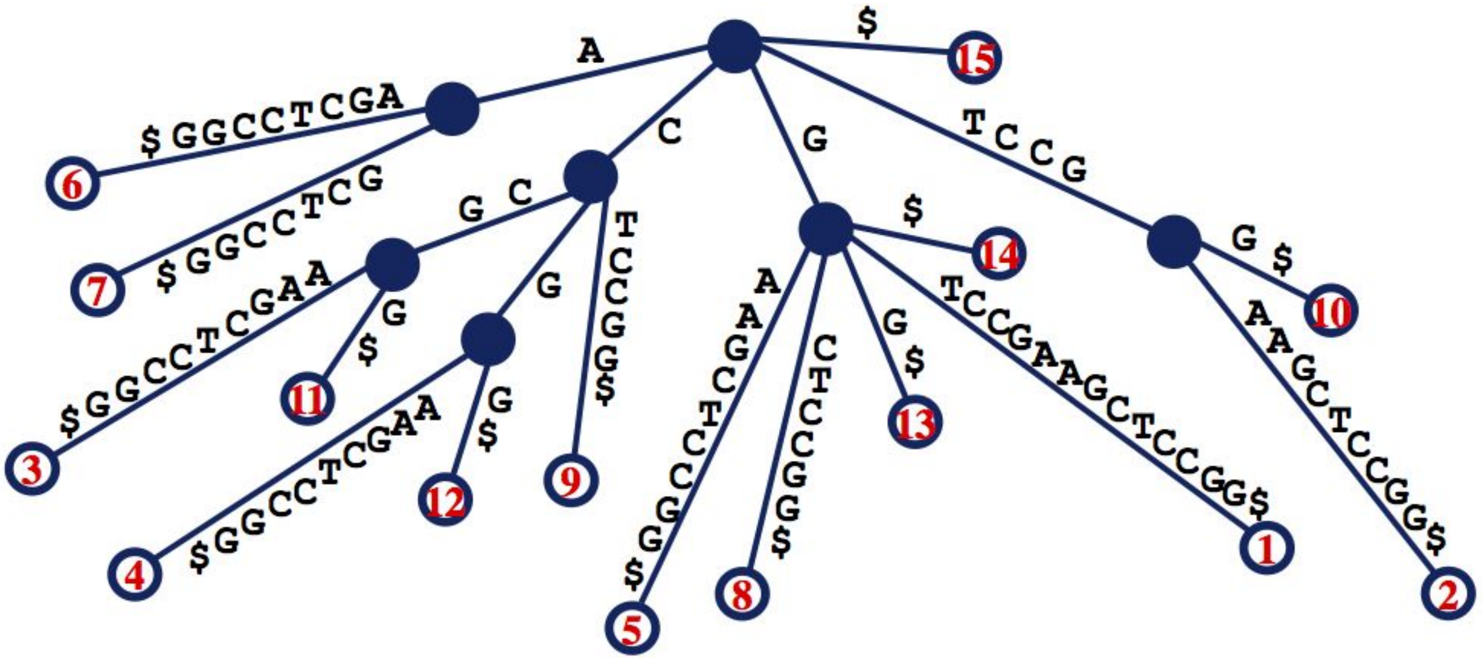
```
superseq.extend(...)
```

These didn’t exist.

Finding the Longest Multiple Repeat (solved by 555)

```
Input:
CATACATAC$

Output:
CATAC
```



First attempt

```
impl Codec for Dna {
    type D = Dna;

    fn code(&self) -> u8 {
        match *self {
            Dna::A => 0b00,
            Dna::C => 0b01,
            Dna::G => 0b10,
            Dna::T => 0b11,
            Dna::Eos => 0b100, // Add this line
        }
    }
…

let dna = Seq::<Dna>::try_from("CATACATAC$").unwrap();
```

Human feedback

“Instead of augmenting the Dna codec, we augment Seq<Dna>. There's two advantages: firstly, using `Eos` as a token requires adding another bit to the encoding and secondly, there will only ever be at most 1 `Eos` token per sequence. So what if we created a struct that contains a Seq<Dna> and an optional `eos_position` member that encodes where in the sequence that end-of-sequence token should be?”

Suggestion: `prefix.chain(suffix)`

```
pub struct DnaSeqWithEos {
    seq: Seq<Dna>,
    eos_position: Option<usize>,
}
…
let seq_slice = if let Some(eos_pos) = seq.eos_position {
    if i <= eos_pos {
        seq.seq.slice(i, eos_pos)
    } else {
        let prefix = seq.seq.slice(i, seq.seq.len());
        let suffix = seq.seq.slice(0, eos_pos);
        prefix.chain(suffix)
```

Conclusion

- Excellent for exploring use-cases
- Excellent for unittesting
- Don’t trust it with the implementation