FASTX and kmers

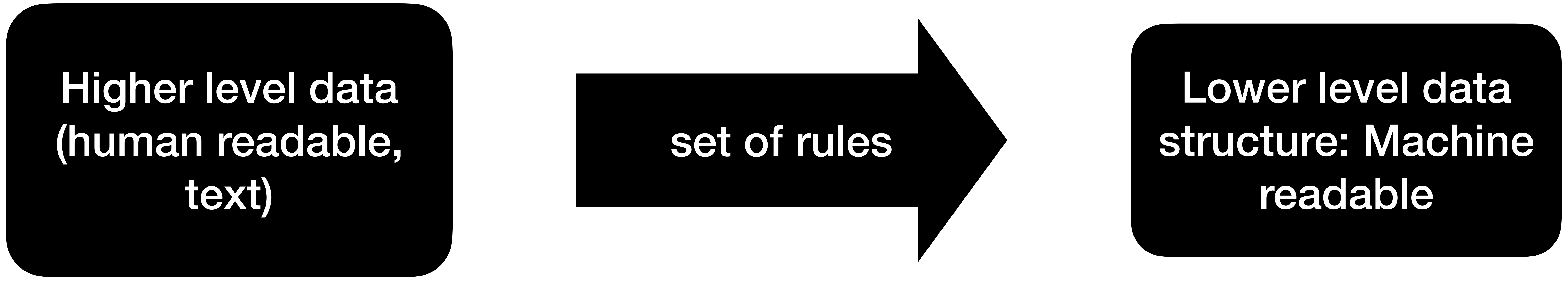
More on biological sequences

Jakob Nybo Nissen, 2021-10-21

BioJulia's parsers

Parsing takes input data and builds a data structure representing the input, while checking for correct syntax

Parsing is hard! But it's a core task of BioJulia


Higher level data (human readable, text)
set of rules
Lower level data structure: Machine readable

This looks awfully similar to code compilation!

- Must be fast, error free and rigorously specified
- Strengths of a machine, not a human!
- Lean on parser generator software

Parser generator software

Automa.jl

- Finite State Machine-based.
    - (it cannot parse structures that are defined recursively, e.g. phylogenetic trees or JSON)
- Rigorous, strict with compile-time check for format ambiguity
- Difficult to use
- Extraordinarily fast

CombinedParsers.jl

- New kid on the block
- Can parse all context-free grammars (including recursive structures!)
- Not yet broadly used, not used in BioJulia
- Easier to use
- Still pretty fast
- Will probably(?) be the future for recursive formats

Automa.jl

- Created by Kenta Sato, Riken, Japan
- Parsers based on FSMs are common and from 1970's at least
- Automa is unique (I think!) because it can inject arbitrary code into parser at compile time.
- Super cool, but a little out of scope for this class!
- BioJulia's website has a tutorial to how Automa works for the interested

How are bioinfo formats typically?

- Many formats are a *series of records*
- *Often* flat (non-recursive) format.
- Often very poorly specified! :(
  - Partly due to incompetence
  - Partly due to wanting wiggle room because science

SO...

- Interface: `AbstractFormattedIO` with Readers and Writers
- These read and write *records,* i.e. not raw data but objects (validation!)
- Can be *streamed* because they are flat
- Opinionated parsers: We can't accept everything!

Example: FASTA format

- Simplest format for biological sequences
- Is not formally specified! So endless edge cases

```
>identifier description
UAGUCUGAUGUGUCUG
UCGUGUAGUGAGAGA
>another_identifier
UAGUCGUGAGGUG
UGCUGUAGUAAGUAGUAG
UUUAGUC
```

- > Identifier
- Optional whitespace + description
- One or more lines with sequence

Example: FASTA format

```
> description          ← Leading whitespace??
UUCGGAUUCGGAAA
UUGAGGCAAACCCA
>                      ← Missing identifier??
UUAGGAGGAAAAAA
>identifier            ← Missing sequence??
>id descr
人類社会のすべての構成員の固    ← Non-ASCII chars??
```

Are these allowed? How should they be parsed?
Who knows?

Example: FASTA parsing

```
reader = FASTA.Reader(io::IO)
for record in reader
    [ ... ]
end
close(reader)
```

```
open(FASTA.Reader, path) do reader
    for record in reader
        [ ... ]
    end
end
```

```
struct FASTA.Record
  data       ::Vector{UInt8}
  filled     ::UnitRange{Int64}
  identifier ::UnitRange{Int64}
  description::UnitRange{Int64}
  sequence   ::UnitRange{Int64}
```

↔

```
struct FASTA.Record
  identifier ::Union{String, Nothing}
  description::Union{String, Nothing}
  sequence   ::Vector{UInt8}
```

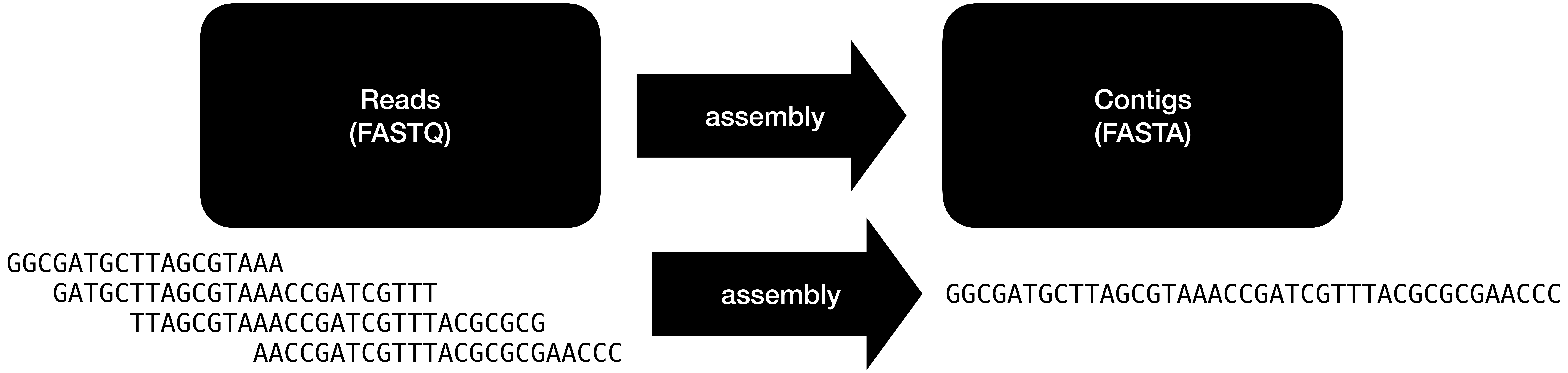
The Record object is a compromise between

- Reading in as raw bytes for speed
- Parsing into a structure for safety

```
FASTA.identifier(record)
FASTA.header(record)
FASTA.sequence(record)
  [ etc ... ]
```

Reads and assemblies

- DNA/RNA sequences machines produce short (e.g. 250 bp) sequences called *reads*
- These are small, random fragments of the real sequence which is much longer.
- To reconstruct original DNA, reads are assembled into *contigs* using assembly software.



```
GGCGATGCTTAGCGTAAA
   GATGCTTAGCGTAAACCGATCGTTT
        TTAGCGTAAACCGATCGTTTACGCGCG
                AACCGATCGTTTACGCGCGAACCC
```

```
GGCGATGCTTAGCGTAAACCGATCGTTTACGCGCGAACCC
```

FASTQ format

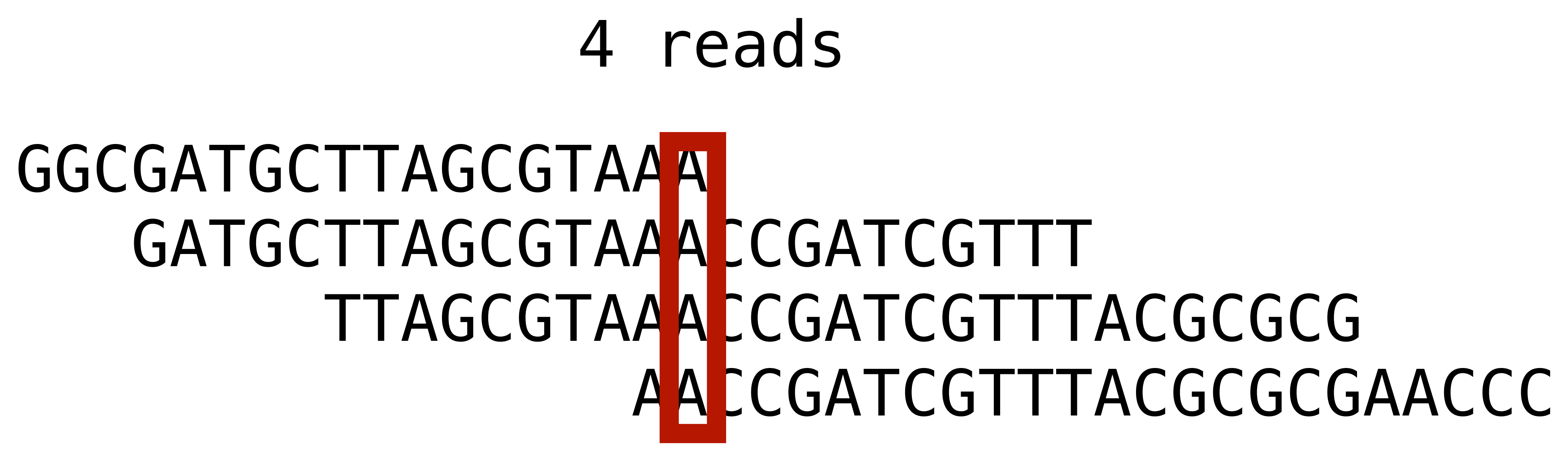
- Analogous to FASTA format, FASTQ format stores reads along with a *quality score* assigned to every single basepair.
- Quality score signals the probability a given base is correct

```
@identifier description
TAGTGCGTGATATT
+
::;91AACFFFFFH
@another_identifier
AGGCTTATAGCGATTTTT
+
110AACCBIJJHHH9911
```

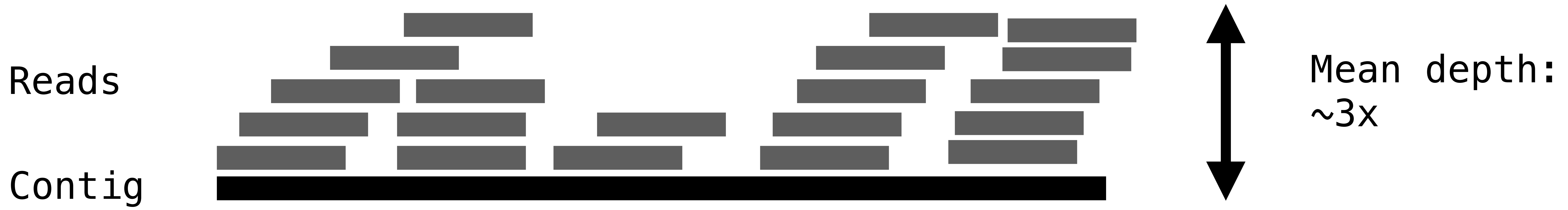
- @Identifier
- Optional whitespace + description
- Single-line sequence with N symbols
- +
- Quality encoded as ASCII with N symbols

Depth

- Outdated assemblers needed the depth of reads before they could assemble to contigs
- Depth: How many reads are at a specific position



```
GGCGATGCTTAGCGTAAA
   GATGCTTAGCGTAAACCGATCGTTT
        TTAGCGTAAACCGATCGTTTACGCGCG
                AACCGATCGTTTACGCGCGAACCC
```

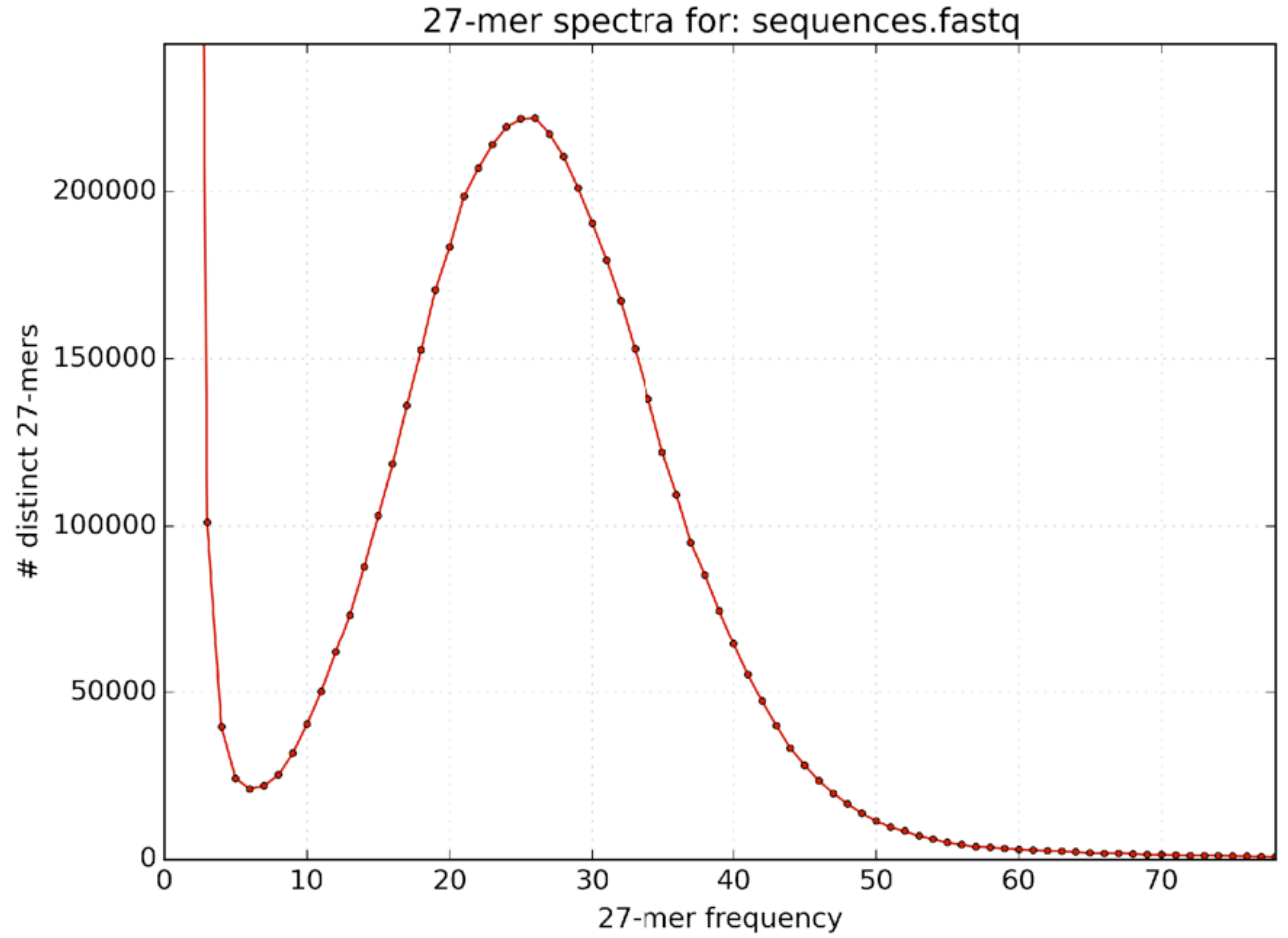


But wait! How do we get the depth BEFORE the assembly?

Before assembly we don't know how reads "stack" on top of the genome!!

Kmer spectrum (exercise)

- Count occurence of *kmers*= all substrings of length K across all reads
- If most kmers occur approx. 50 times, depth is approx 50!
- Need to correct for "missing kmers" at read ends.



- We can use this to calculate mean depth (total kmers / distinct kmers)
- Mode depth (position of peak)
- Genome size (total bases / mean depth)

Kmers

- Kmers are very useful in bioinformatics.
- Lots of algorithms use sequences of fixed length K
- If K is small, you can represent a kmer as a machine integer:

```
A = 00, C = 01, G = 10, T = 11

                          T A G G T C G T G A T G A T G A G G C
0000000000000000000000000011001010110110111000111000111000101001
```

- This can be 100s of times more efficient than using heap-allocated vectors
  - This is why kmers are so popular in practise!
- Due to the `BioSequences.jl` abstraction, a kmer is just another `BioSequence`

Kmers, sequences, views

`Mer{DNAAlphabet{2}, K}`

- Represented by integer (`UInt64`)
- Size fixed at compile time, and limited to small sizes
- Extremely fast
- Immutable

`LongSequence{DNAAlphabet{2}}`

- Represented by heap-allocated vector
- Arbitrary length, length can change at runtime
- Mutable

`LongSubSeq{DNAAlphabet{2}}` Upcoming v3 release only!

- View into an existing `LongSequence`
- Does not own its own data
- Stack-allocated (much more lightweight)
- Arbitrary length, mutable

Questions?

Exercise 3