Genome Scale Algorithms

Practical Information

Form and expectations

- Lectures are not a substitute for the textbook but a supplement. Read the teaching material each week.

  - At lectures I will often present material in different ways, or present slightly different algorithms.

  - This is intentional. Seeing material presented in two different ways makes it more likely that it is understood.

  - It is usually easier to follow lectures, so you can read the full details after the lecture. Lectures rarely have all the details and special cases, though.

Form and expectations

- Exercises are more important than the lectures. Do them!
  - It is very easy to fool yourself into thinking you understand an algorithm — implementing it will test that assumption.
  - However, an implementation is not a substitute for understanding. At the exam, you will be asked to present algorithms.
  - I suggest that we also prepare for that at the exercises. (If you don’t want to, we won’t, but it will hurt you more than it will hurt me).

Algorithms and implementations

- There are often many choices in an algorithm; some are important and some are not. There is almost always more than one way to do the same thing.

- When you implement an algorithm, you have to make choices. You want to know when you have an implementation choice and when something is essential for the algorithm and can’t be done in any other way.

Algorithms and implementations

- I will not show you much code or pseudocode on the slides. That blurs this distinction. Look at figures and descriptions when you write your own code. It might take a little longer, but then you understand what you are doing.

- In my experience, and I have some experience, the (pseudo)code shortcut does more harm than good.

- If you cannot implement an algorithm without (pseudo)code, you do not understand it at all. You are only fooling yourself. Let’s try to avoid that.

- It is much better if you ask for hints than you google for solutions. The former will help you; the latter will harm you.

Algorithms and implementations

- However! You don’t know how to write algorithmic code yet, and the best way to learn is to read actual code.

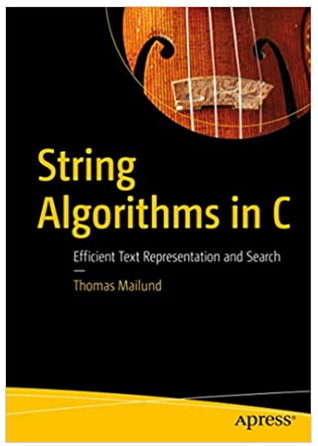
- Algorithms are only half the story; there is a lot to how you implement them.

- Your choice of programming language affects this (sometimes), so ideally you want to see implementations in different languages. Even if you don’t know all of them.

- I will show you how I implement them, just not until after the exercises/projects. We need to talk programming as well as algorithms.

Teaching material

Textbook: **String Algorithms in C**



Contains the algorithms we will see in class. The code is mostly useful for implementation details and if you know the C programming language, but it is a low-level language and anything you can do there, you can probably do in your language of choice.

In addition, I will hand out selected papers or extracts from other textbooks when relevant.

Exercises

- Mostly programming exercises that you can do in groups or individually
- Try to get them done before the TA sessions so you can use those sessions actively to discuss and get help
  - If you are unprepared, you will just be listening to the others
  - You do not learn how to program by watching others do it
  - You might easily fool yourself into thinking it, though. Don’t be a fool
- There is a large overlap between exercises and projects, so often you will have solved part of a project if you do the exercises.

Exercises

- Once in a while, I want to go through algorithms at the exercises by having you present.

- This exposes when understanding is incomplete, which can be uncomfortable but is something you want to catch early so we can fix it. You learn more from 15 mins presentation than 15 hours of lectures.

- Some people find this uncomfortable, so we can drop it. I will leave it up to you. But if we choose to do this, then I expect that prepare for it and will be willing to present.

Projects

- There will be 5 mandatory projects
- Do them in groups of 1-3 people (it is the same projects regardless of your group size, so you probably want 3)
- We are going to build a real readmapper. That adds some work beyond implementing algorithms, but stuff you need to know if you ever have to implement a tool in real life.
  - You need to build command-line tools that take command-line options
  - You need to parse the file formats used for readmappers
  - If you don’t know how to do this, I’ll help you in the first week’s exercises
- More later

Exercises, projects and GitHub

- Exercises/Projects: [“GitHub Classroom” ->] GitHub
  - Templates for exercises
  - Automatic testing
- Discussion forum: https://github.com/orgs/birc-gsa-2022/discussions (experimental)

Slides and notes

- Some of the following slides are very verbose, so they can be used as notes as well as visualisation aids.

- We will only provide such detailed notes for information that is *not* in the teaching material. When possible, we will use less cluttered presentations.

- This first week, we have a lot of information to cover that is either not mentioned in the book, or too briefly mentioned, so the slides are text heavy. It gets better in future weeks.

String notation and terminology

- An *alphabet* is a set of *characters*. We frequently use Σ to denote the alphabet we work with. We assume that Σ is finite, and we use σ to denote its size. Elsewhere in the literature it is allowed to be even uncountable — but never empty.

- A *string, S,* is a sequence of characters. The book assumes it is finite, so it has a beginning and an end. Both assumptions are sometimes relaxed — with streams you assume there is no end; circular strings (such as bacteria genomes) have no beginning and no end.

- For two strings, *u* and *v*, their concatenation is written *uv.*

- The empty string is written ε. For all strings $u = \varepsilon u = u\varepsilon$.

String notation and terminology

- We use Σ^n to denote strings of length *n*.

- Naturally, Σ^0 then contains only the empty string.

- We use Σ^* to denote all strings over the alphabet and Σ^+ to denote all non-empty strings.

- For a string *x*, *x*[*i*] denotes the character at index *i* (undefined if it doesn't exist) and *x*[*i:j*] denotes the substring from *i* to *j.*

Sentinels

- We often use a special letter, that by convention we write as $, called the *sentinel.* Usually it appears at the end of strings, and we can translate a string x into one with a sentinel as x$.

- The sentinel is alphabetically smaller than all other letters, and we always assumes that it is not part of the alphabet.

- Consequently, $ only appears as the last letter in x$ and not inside x.

Sentinels

- Some programming languages have an explicit sentinel. C strings always terminate with the byte that is numerical zero, and while you can have zero bytes inside strings, none of the string functions in C can handle that.

- Other language, e.g. Python, do not have a sentinel. If you want one, you need to add it yourself.

- Don’t worry if you do not see the point of sentinels yet—you will. We use them a lot in string algorithms. Just know that a sentinel is a smallest letter that isn’t anywhere else in a string, it is explicitly there in some languages but not in others.

Unicode

- The standard representation strings today is unicode
- It is a list of ~150,000 broadly defined “characters”
- Mapped to integers in the range 0x0-0x10FFFF
- 17 x 2^{16} = 1,114,112 possible values

Unicode

A vs A

U+0041 (Latin A)

U+0410 (Cyrillic A)

Å vs Å

U+00C5

"LATIN CAPITAL LETTER A WITH RING ABOVE”

U+212B

"ANGSTROM SIGN”

Unicode

ñ vs ñ

U+00F1
"LATIN SMALL LETTER N WITH TILDE”

[U+006E, U+0303]
“LATIN SMALL LETTER N” + “COMBINING TILDE”

ff vs f f

U+FB00
"LATIN SMALL LIGATURE FF”

[U+0066, U+0066]
[“LATIN SMALL LETTER F”, “LATIN SMALL LETTER F”]

Unicode

- The same text can be represented in more than one way, so you cannot index directly to a character

  - Characters such as U+0303 “COMBINING TILDE” modifies the previous, it isn’t a character itself

  - while U0FB00 “LATIN SMALL LIGATURE FF” is really two characters

  - This requires **normalisation** of strings before we compare

We completely ignore this in class; our strings are DNA so it isn’t relevant for us.

Unicode

- It’s a huge character set (you need 21 bits to represent the range)

- Different **encodings** deal with representing unicode strings

- **UTF-32**: a sequence of code points, each represented in 32 bits (4 bytes).

  - Simple, you can index all code points.

  - Problem: strings are huge.

Unicode

- It’s a huge character set (you need 20 bits to represent the range)
- Different **encodings** deal with representing unicode strings
    - **UTF-8**: each code point represented in 1-4 bytes.
        - Uses less memory
        - Indexing into the bytes does not index into the code points
            - You need to scan multiple bytes to work out which code point you have
            - For indexing, you cannot easily know how many bytes a prefix of the string takes

Python

Strings are sequences of code points, represented so you can index them.

```
x = "baño"
print(f"len(x) = {len(x)}")
for a in x:
    print(f"code point: U+{ord(a):0>4x}")
print("Third character:", x[2])
print()
```

```
len(x) = 4
code point: U+0062
code point: U+0061
code point: U+00f1
code point: U+006f
Third character: ñ
```

```
x = "baño"
print(f"len(x) = {len(x)}")
for a in x:
    print(f"code point: U+{ord(a):0>4x}")
print("Third character:", x[2])
print()
```

```
len(x) = 5
code point: U+0062
code point: U+0061
code point: U+006e
code point: U+0303
code point: U+006f
Third character: n
```

The string representation depends on the characters the string contains.

Go

Go represents strings as UTF-8. When you loop over a string, you get the code points (type “rune”), but when you index, you get the bytes

```
x = "你好，世界" // hello, world

fmt.Println("len(x) =", len(x),
  “bytes\nrepresenting", utf8.RuneCountInString(x),
  “code points\n")

// Iterate through code points
for i, a := range x {
    fmt.Printf("code point x[%d]: 0x%x\n", i, a)
}

// Also iterating through code points
codepoints := []rune(x)
for i = range codepoints {
    fmt.Printf("code point x[%d]: 0x%x\n", i, codepoints[i])
}

// Iterate through bytes (x[i] always gives you the byte!)
for i := 0; i < len(x); i++ {
   fmt.Printf("byte x[%d] = 0x%x\n", i, x[i])
}
```

```
len(x) = 15 bytes
representing 5 code points

code point x[0]: 0×4f60
code point x[3]: 0×597d
code point x[6]: 0×ff0c
code point x[9]: 0×4e16
code point x[12]: 0×754c

code point x[0]: 0×4f60
code point x[3]: 0×597d
code point x[6]: 0×ff0c
code point x[9]: 0×4e16
code point x[12]: 0×754c

byte x[0] = 0×e4
byte x[1] = 0×bd
byte x[2] = 0×a0
byte x[3] = 0×e5
byte x[4] = 0×a5
```

C

- No unicode support at all…
- Strings are chunks of “char” that are de-factor bytes (so unicode encoding is something you have to build yourself)
- The type wchar_t also provides some support for encodings, but behaviour is implementation depending

Handle alphabets…

- You might want to handle the alphabet more explicitly than using the support you get from your language.
- You can reduce the alphabet to what you actually use.
- You can ensure that the alphabet is 0, …, σ-1, with $=0 to ensure that the sentinel is smallest.

(show code)

Warning: Indexing

- Different authors and different programming languages use very different indexing conventions! (The same does different file formats).

- Zero or one indexing?
  - Is the first character in the string at position 0 or 1?
  - The language R says 1. This is typical notation amongst mathematicians.
  - Many papers start at 0 as does languages such as C, Java and Python. This is the usual notation for computer scientists.

Warning: Indexing

- What is included in a range?
- In a substring *x*[*i*:*j*], is *x*[*j*] included?
  - Some languages, such as R, say yes. Mathematicians would too.
  - Python would say no. (Typically, 0-index languages do).
  - C doesn’t let you extract substrings that way, but since it index from 0 it would consider the entire string *x*[0:*n*-1] which if we exclude *j* can be written succinctly *x*[0:*n*].

Warning: Indexing

- I will index from zero, so *x*[0] is the first letter in x (assuming it is not empty), and *x*[n-1] is the last letter, assuming that the length of x is n (len(x) = n or |x| = n).
- I will use semi-open intervals, which means that x[i:j] includes x[i] but not x[j]. This won't just be for strings, but for any contiguous sub-array of an array a: a[i:j] starts at a[i] and ends at a[j-1].

Warning: Indexing

- Depending on the programming language you use, you will have to modify algorithms you find in the litterature. On the left here is pseudocode from a textbook that uses 1-indexing, and on the right is the same code in C that uses 0-indexing.

```
for j ← 0 to n - m do
    i ← 1
    while i ≤ m and T[j+i] == P[i] {
        i ← i + 1

    if i = m+1 then
        report(j+1)
```

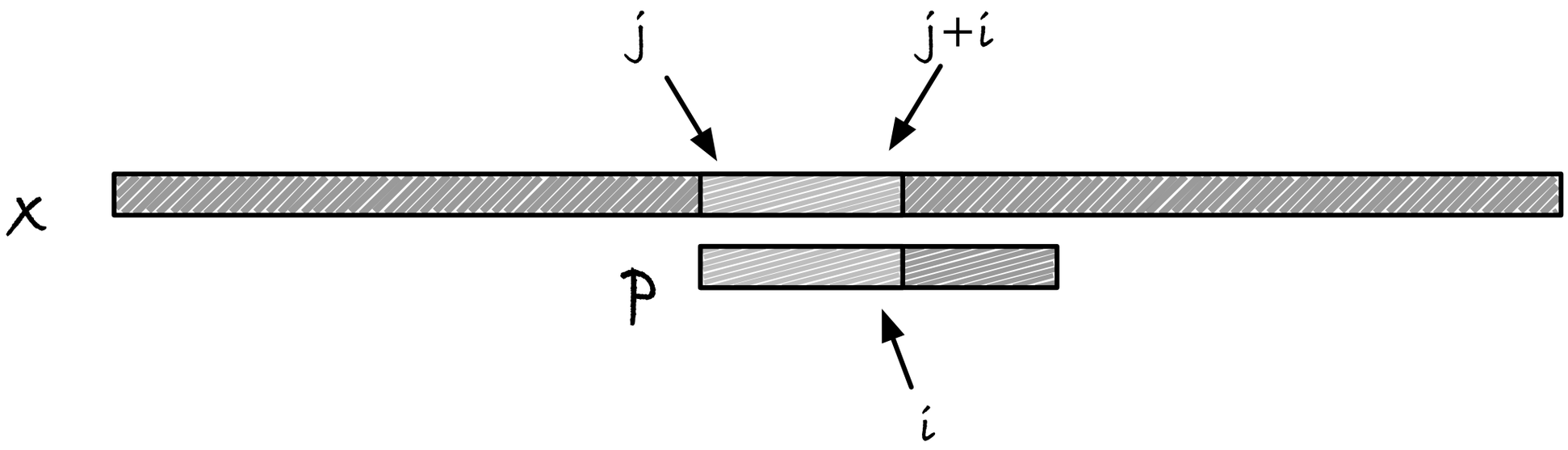
```
for (int j = 0; j <= n - m; j++) {
    int i = 0;
    while (i < m && T[j+i] == P[i]) {
        i++;
    }
    if (i == m) {
        report(j);
    }
}
```

Typical sizes

- Whenever we work with a single string, typically we call it x, unless otherwise stated, n denotes its length, len(x) = n.

- When we work with two strings, x and y or x and p, len(x) = n and len(y) = m / len(p) = m.

Graphical notation



- We draw strings like boxes, and such drawings are your friend! Get used to drawing them.
- When we match substrings, we align the boxes.
- Indexes/pointers into the boxes illustrate characters we are looking at

Strings in biology

- Much of bioinformatics and molecular biology concerns strings.

    - Not actual strings, but strings as abstractions for complex molecules.

    - We can proteins as strings over a letter of 20 amino acids, DNA as strings over four letters (A,C,G,T) and RNA as strings over (another) four letters (A,C,G,U).

    - There is more structure to those molecules, but their string-structure is called the primary, and for good reasons.

Genomes and DNA strings

- A *genome* is the combined genetic material of an organism. The inheritable material that is passed between generations (usually with some modifications).

- In some viruses the genome consist of RNA molecules, but for all other organisms — DNA viruses and all living organisms — the genome is encoded in DNA.

- The single DNA molecule that is part of a genome is called a *chromosome*.

- Almost all bacteria have a single, circular chromosome as their genome.

- Eukaryotes (such as humans) usually have several, linear chromosomes. (We have 23 pairs — stretched out we have about 2m of DNA in each cell)

Genomes and DNA strings

- The computer science abstraction of a genome is one long string per chromosome.

    - By long I mean *long*. The human genome consists of about 3 billion nucleotides (characters in DNA).

    - For reference, 3 billion seconds is about 95 years.

    - The human genome is split into 23 more manageable chapters, so you don’t have 3 billion long strings, but the longest (chromosome 1) is still about a quarter of a billion long, and that is still a pretty long string.

Genomes and DNA strings

- We do not have the technology to read an entire genome as one single string.

- We can, however, read short subsequences — called *reads.* The length depends on the technology.

  - The technology used in the human genome project produced reads with length ~800bp (bp=base pair) but is very expensive

  - Modern methods produce reads with length 100-200bp but can sequence an entire human genome 30 times (30x) in a week (sequencing 30x is not as stupid as it might sound).

  - Newer methods can produce very long reads, but are also very expensive

Genomes and DNA strings

- Figuring out what a genome looks like from reads is a computational problem called *assembly*.

- Unfortunately, it is not something we have time to cover in this class (but involves many of the techniques we do see)

- Instead, we will consider a related problem: *read mapping*

Read mapping

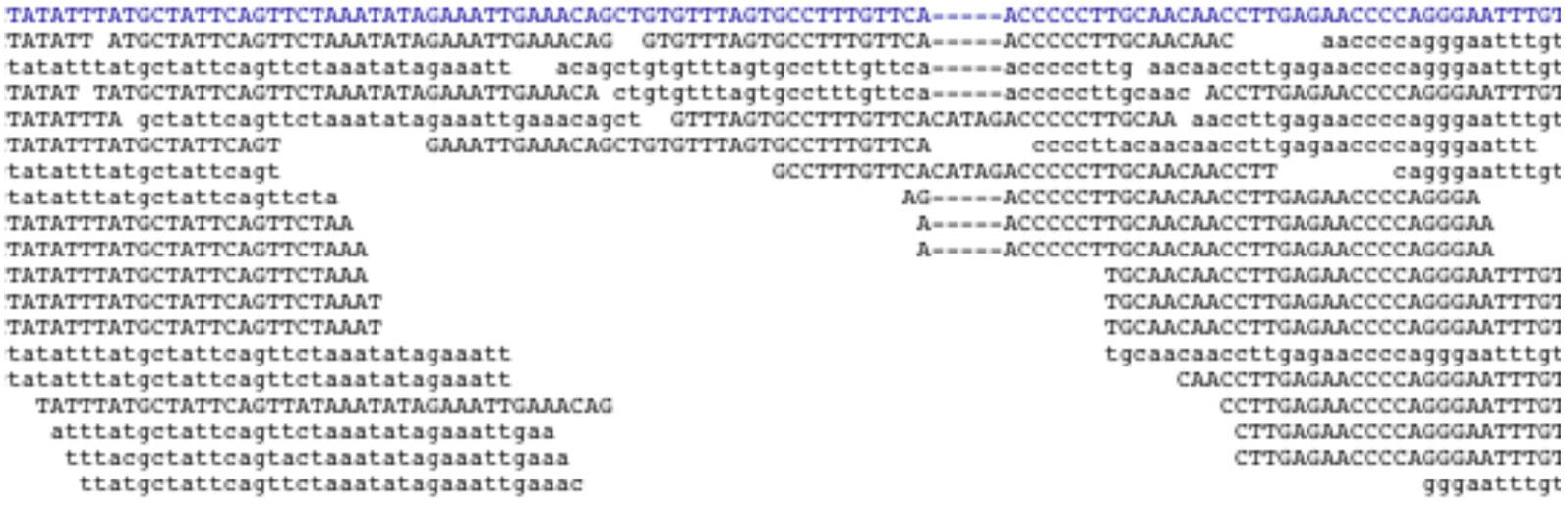
- Read mapping is the first step in many bioinformatics analyses, e.g. figuring out gene expression levels in different tissue or figuring out the genetic variation within a population.

- The basic problem is: *given a genome and a number of reads, find out where the reads occur in the genome.*

- The reason it is a computational problem is that genomes can be billions of basepairs long and you might have billions of reads.

Read mapping

- Also, reads won't necessarily map perfectly — if there is variation in a species (and there is) you will have to map approximately.



Approximate match and pair-wise alignment

- Two strings *approximately* match if you can translate one into the other with only a few changes (“few” depends on context).
- Usually, the operations are: insertion, deletion, and substitution.

Approximate match and pair-wise alignment

- Two strings *approximately* match if you can translate one into the other with only a few changes (“few” depends on context).
- Typically, the operations are: insertion, deletion, and substitution.

```
ACAACTG-CTAGGCTAGCATAGGTGAC
ACAACTGACTAGGCT-GCATATGTGAC
```

CIGAR

- CIGAR strings are representations of pairwise alignments
- Basic CIGAR has three operations: Insertion (I), Deletion (D), and match/mismatch (M).
- A CIGAR string consists of a sequence of operations, all preceded by how many times each operation is performed

```
ACAACTG-CTAGGCTAGCATAGGTGAC
ACAACTGACTAGGCT-GCATATGTGAC
  7M  1I  7M  1D    11M

CIGAR: 7M1I7M1D11M
```

CIGAR

- There are some more symbols you can use in a CIGAR string. We don’t have to worry about them, except when doing exercises with the bwa tool. It uses something called “soft clipping”
- For various reasons, you can have sequences that align well except in the beginning and end. In real applications, we don’t want to penalise this, so we use “free” editing operations there and use the symbol ’S’ in the CIGAR string.

File formats: FASTA

- Full genomes are usually represented in FASTA files. These are text files with one or more sequences, each new sequence preceded by a header.
- A header line starts with ‘>’. Everything else is sequence

```
> chr1
ACCGATACCATACACAGGACTATTACT
CATTACGGAAGTACGAGACTAGTTACA
CTAGGCAT …
> chr2
TGTGTACCATACGGATTCATATCCATA
CATACCTAGCTAGGCATACATTACATA
TACTA …
```

A quick aside…

- We are going to assume that DNA is a sequence over the alphabet A,C,G,T (and it is if we abstract a genome as a string).

- With our current technology, though, we don't have perfect knowledge of what all the characters are. Some parts of the genome cannot be read for chemical reasons and sometimes sequencing machines cannot determine a character with certainty.

- The actual alphabet contains symbols for all subsets of characters. Usually, though, it will either be one of the four or "N", which stands for any of the four.

- The class is assuming a perfect world — if you go out in the real world it is more complicated and you need to learn a bit more.

File formats: FASTQ

- The FASTQ file is used to store reads and resembles the FASTA format.
- A FASTQ file uses four lines per read:
  - Line 1 begins with a '@' character and is followed by a sequence identifier
  - Line 2 is the raw sequence letters.
  - Line 3 begins with a '+' character and is optionally followed by the same sequence identifier
  - Line 4 encodes the quality values for the sequence in Line 2, and must contain the same number of symbols as letters in the sequence.

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

Example read mapping with bwa

- bwa — Burrows-Wheeler Aligner — is a popular tool for read mapping. Let us see how it works in practise

```
$ ls
reads.fq   ref.fa
```

```
$ cat ref.fa
> ref1
ACCTACAGACTACCATGTATCTCCATTTACCTAGTCTAG
CATACTTTCCACACGCTGTGTGTCACTAGTGTGACTACG
AAATACGTGTGTACTACGGACTACCTACTACCTA
> ref2
ACCTACAGACTACCATGTATCTCCATTTACCTAGTCTAG
CATACTTTCCACACGCTGTGTGTCACTAGTGTGACTACG
AAATACGTGTGTACTACGGACTACCTACTACCTA
```

```
$ cat reads.fq
@read1
ACCTACAGACTACCATGTATCTCCATTTACCTAGTCTAG
+
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
@read2
ACCTACAGACTACCATGTATCTCCATTACCTAGTCTAGA
+
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
@read3
CTACAGACTACCATGTATCTCCATTTACCTAGTCTAGTT
+
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Example read mapping with bwa

- bwa — Burrows-Wheeler Aligner — is a common tool for read mapping. Let us see how it works in practise

```
$ ls
reads.fq   ref.fa
```

```
$ cat ref.fa
> ref1
ACCTACAGACTACCATGTATCTCCATTTACCTAGTCTAG
CATACTTTCCACACGCTGTGTGTCACTAGTGTGACTACG
AAATACGTGTGTACTACGGACTACCTACTACCTA
> ref2
ACCTACAGACTACCATGTATCTCCATTTACCTAGTCTAG
CATACTTTCCACACGCTGTGTGTCACTAGTGTGACTACG
AAATACGTGTGTACTACGGACTACCTACTACCTA
```

The quality scores here are fake, so I have just used the maximal possible score, ~

```
$ cat reads.fq
@read1
ACCTACAGACTACCATGTATCTCCATTTACCTAGTCTAG
+
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
@read2
ACCTACAGACTACCATGTATCTCCATTACCTAGTCTAGA
+
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
@read3
CTACAGACTACCATGTATCTCCATTTACCTAGTCTAGTT
+
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Example read mapping with bwa

- With bwa you first have to index the genome (we see what that means later in the class)

```
$ bwa index ref.fa
[bwa_index] Pack FASTA... 0.00 sec
[bwa_index] Construct BWT for the packed sequence...
[bwa_index] 0.00 seconds elapse.
[bwa_index] Update BWT... 0.00 sec
[bwa_index] Pack forward-only FASTA... 0.00 sec
[bwa_index] Construct SA from BWT and Occ... 0.00 sec
[main] Version: 0.7.15-r1140
[main] CMD: bwa index ref.fa
[main] Real time: 0.006 sec; CPU: 0.005 sec

$ ls
reads.fq   ref.fa     ref.fa.amb ref.fa.ann ref.fa.bwt ref.fa.pac ref.fa.sa
```

All these new files describe data structures for efficient read mapping…

Example read mapping with bwa

- The command for map is ‘mem’ (named after an algorithm, it is not just weird)

```
$ bwa mem ref.fa reads.fq > bwa.sam
[M::bwa_idx_load_from_disk] read 0 ALT contigs
[M::process] read 3 sequences (117 bp)...
[M::mem_process_seqs] Processed 3 reads in 0.000 CPU sec, 0.001 real sec
[main] Version: 0.7.15-r1140
[main] CMD: bwa mem ref.fa reads.fq
[main] Real time: 0.006 sec; CPU: 0.004 sec

$ ls
bwa.sam    reads.fq   ref.fa     ref.fa.amb ref.fa.ann ref.fa.bwt ref.fa.pac ref.fa.sa
```

Here’s the file we generated.
It is in SAM format.

http://samtools.github.io/hts-specs/SAMv1.pdf

Example read mapping with bwa

2. mailund@st-d12846: ~/Dropbox/Undervisning/Genome-scale algorithms/bwa (zsh)

```
~/D/U/G/bwa cat bwa.sam
@SQ     SN:ref1 LN:112
@SQ     SN:ref2 LN:112
@PG     ID:bwa  PN:bwa  VN:0.7.15-r1140 CL:bwa mem ref.fa reads.fq
read1   0       ref2    1       0       39M     *       0       0       ACCTACAGACTACCATGTATCTCCATTTACCTAGTCTAG ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ NM:i:0  MD:Z:39 AS:i:39 XS:i:39 XA:Z:ref1,+1,39M,0;
read2   0       ref1    1       0       25M1D14M        *       0       0       ACCTACAGACTACCATGTATCTCCATTACCTAGTCTAGA ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ NM:i:2  MD:Z:25^T13C0   AS:i:31 XS:i:31 XA:Z:ref2,+1,25M1D14M,2;
read3   0       ref1    3       0       37M2S   *       0       0       CTACAGACTACCATGTATCTCCATTTACCTAGTCTAGTT ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ NM:i:0  MD:Z:37 AS:i:37 XS:i:37 XA:Z:ref2,+3,37M2S,0;
~/D/U/G/bwa
```

Example read mapping with bwa

- To simplify our project, we will make it about finding all maps within a given edit distance.

- bwa doesn’t do this, though. It will report (one of) the best.

- So if your program is working, bwa should find a subset of what you find

    - Except that bwa can find soft clipped sequences and we cannot…

File formats: SAM

- A SAM file consists of a header and a list of mapped reads.

- The header contains lines starting with ‘@‘ — but it is not mandatory so we ignore it.

- The mapped sequences are tab-separated lines with 11 mandatory flags (we ignore anything that isn’t mandatory)

File formats: SAM

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	`[!-?A-~]{1,254}`	Query template NAME
2	FLAG	Int	`[0,`2^{16}`-1]`	bitwise FLAG
3	RNAME	String	`\*|[!-()+-<>-~][!-~]*`	Reference sequence NAME
4	POS	Int	`[0,`2^{31}`-1]`	1-based leftmost mapping POSition
5	MAPQ	Int	`[0,`2^{8}`-1]`	MAPping Quality
6	CIGAR	String	`\*|([0-9]+[MIDNSHPX=])+`	CIGAR string
7	RNEXT	String	`\*|=|[!-()+-<>-~][!-~]*`	Ref. name of the mate/next read
8	PNEXT	Int	`[0,`2^{31}`-1]`	Position of the mate/next read
9	TLEN	Int	`[-`2^{31}`+1,`2^{31}`-1]`	observed Template LENgth
10	SEQ	String	`\*|[A-Za-z=.]+`	segment SEQuence
11	QUAL	String	`[!-~]+`	ASCII of Phred-scaled base QUALity+33

File formats: SAM

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	`[!-?A-~]{1,254}`	Query template NAME
2	FLAG	Int	`[0,2`^{16}`-1]`	bitwise FLAG
3	RNAME	String	`\*|[!-()+-<>-~][!-~]*`	Reference sequence NAME
4	POS	Int	`[0,2`^{31}`-1]`	1-based leftmost mapping POSition
5	MAPQ	Int	`[0,2`^{8}`-1]`	MAPping Quality
6	CIGAR	String	`\*|([0-9]+[MIDNSHPX=])+`	CIGAR string
7	RNEXT	String	`\*|=|[!-()+-<>-~][!-~]*`	Ref. name of the mate/next read
8	PNEXT	Int	`[0,2`^{31}`-1]`	Position of the mate/next read
9	TLEN	Int	`[-2`^{31}`+1,2`^{31}`-1]`	observed Template LENgth
10	SEQ	String	`\*|[A-Za-z=.]+`	segment SEQuence
11	QUAL	String	`[!-~]+`	ASCII of Phred-scaled base QUALity+33

QNAME (query name) is the name of the read from the FASTQ file

File formats: SAM

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	`[!-?A-~]{1,254}`	Query template NAME
2	FLAG	Int	`[0,2`^{16}`-1]`	bitwise FLAG
3	RNAME	String	`\*|[!-()+-<>-~][!-~]*`	Reference sequence NAME
4	POS	Int	`[0,2`^{31}`-1]`	1-based leftmost mapping POSition
5	MAPQ	Int	`[0,2`^{8}`-1]`	MAPping Quality
6	CIGAR	String	`\*|([0-9]+[MIDNSHPX=])+`	CIGAR string
7	RNEXT	String	`\*|=|[!-()+-<>-~][!-~]*`	Ref. name of the mate/next read
8	PNEXT	Int	`[0,2`^{31}`-1]`	Position of the mate/next read
9	TLEN	Int	`[-2`^{31}`+1,2`^{31}`-1]`	observed Template LENgth
10	SEQ	String	`\*|[A-Za-z=.]+`	segment SEQuence
11	QUAL	String	`[!-~]+`	ASCII of Phred-scaled base QUALity+33

FLAG We don’t use it — ignore it by setting it to zero

File formats: SAM

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	`[!-?A-~]{1,254}`	Query template NAME
2	FLAG	Int	`[0,2`^{16}`-1]`	bitwise FLAG
3	RNAME	String	`\*|[!-()+-<>-~][!-~]*`	Reference sequence NAME
4	POS	Int	`[0,2`^{31}`-1]`	1-based leftmost mapping POSition
5	MAPQ	Int	`[0,2`^{8}`-1]`	MAPping Quality
6	CIGAR	String	`\*|([0-9]+[MIDNSHPX=])+`	CIGAR string
7	RNEXT	String	`\*|=|[!-()+-<>-~][!-~]*`	Ref. name of the mate/next read
8	PNEXT	Int	`[0,2`^{31}`-1]`	Position of the mate/next read
9	TLEN	Int	`[-2`^{31}`+1,2`^{31}`-1]`	observed Template LENgth
10	SEQ	String	`\*|[A-Za-z=.]+`	segment SEQuence
11	QUAL	String	`[!-~]+`	ASCII of Phred-scaled base QUALity+33

RNAME (Reference name) This is the name of the sequence from the FASTA file

File formats: SAM

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	`[!-?A-~]{1,254}`	Query template NAME
2	FLAG	Int	`[0,2`^{16}`-1]`	bitwise FLAG
3	RNAME	String	`\*|[!-()+-<>-~][!-~]*`	Reference sequence NAME
4	POS	Int	`[0,2`^{31}`-1]`	1-based leftmost mapping POSition
5	MAPQ	Int	`[0,2`^{8}`-1]`	MAPping Quality
6	CIGAR	String	`\*|([0-9]+[MIDNSHPX=])+`	CIGAR string
7	RNEXT	String	`\*|=|[!-()+-<>-~][!-~]*`	Ref. name of the mate/next read
8	PNEXT	Int	`[0,2`^{31}`-1]`	Position of the mate/next read
9	TLEN	Int	`[-2`^{31}`+1,2`^{31}`-1]`	observed Template LENgth
10	SEQ	String	`\*|[A-Za-z=.]+`	segment SEQuence
11	QUAL	String	`[!-~]+`	ASCII of Phred-scaled base QUALity+33

POS The position in the RNAME sequence where we have a match of QNAME. Notice that this is a 1-indexed position!

File formats: SAM

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	`[!-?A-~]{1,254}`	Query template NAME
2	FLAG	Int	`[0,2`^{16}`-1]`	bitwise FLAG
3	RNAME	String	`\*|[!-()+-<>-~][!-~]*`	Reference sequence NAME
4	POS	Int	`[0,2`^{31}`-1]`	1-based leftmost mapping POSition
5	MAPQ	Int	`[0,2`^{8}`-1]`	MAPping Quality
6	CIGAR	String	`\*|([0-9]+[MIDNSHPX=])+`	CIGAR string
7	RNEXT	String	`\*|=|[!-()+-<>-~][!-~]*`	Ref. name of the mate/next read
8	PNEXT	Int	`[0,2`^{31}`-1]`	Position of the mate/next read
9	TLEN	Int	`[-2`^{31}`+1,2`^{31}`-1]`	observed Template LENgth
10	SEQ	String	`\*|[A-Za-z=.]+`	segment SEQuence
11	QUAL	String	`[!-~]+`	ASCII of Phred-scaled base QUALity+33

MAPQ (Mapping quality) — we ignore quality so set this one to 0

File formats: SAM

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	`[!-?A-~]{1,254}`	Query template NAME
2	FLAG	Int	`[0,2`^{16}`-1]`	bitwise FLAG
3	RNAME	String	`\*|[!-()+-<>-~][!-~]*`	Reference sequence NAME
4	POS	Int	`[0,2`^{31}`-1]`	1-based leftmost mapping POSition
5	MAPQ	Int	`[0,2`^{8}`-1]`	MAPping Quality
6	CIGAR	String	`\*|([0-9]+[MIDNSHPX=])+`	CIGAR string
7	RNEXT	String	`\*|=|[!-()+-<>-~][!-~]*`	Ref. name of the mate/next read
8	PNEXT	Int	`[0,2`^{31}`-1]`	Position of the mate/next read
9	TLEN	Int	`[-2`^{31}`+1,2`^{31}`-1]`	observed Template LENgth
10	SEQ	String	`\*|[A-Za-z=.]+`	segment SEQuence
11	QUAL	String	`[!-~]+`	ASCII of Phred-scaled base QUALity+33

CIGAR You can probably guess what this is

File formats: SAM

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	`[!-?A-~]{1,254}`	Query template NAME
2	FLAG	Int	`[0,2`^{16}`-1]`	bitwise FLAG
3	RNAME	String	`\*|[!-()+-<>-~][!-~]*`	Reference sequence NAME
4	POS	Int	`[0,2`^{31}`-1]`	1-based leftmost mapping POSition
5	MAPQ	Int	`[0,2`^{8}`-1]`	MAPping Quality
6	CIGAR	String	`\*|([0-9]+[MIDNSHPX=])+`	CIGAR string
7	RNEXT	String	`\*|=|[!-()+-<>-~][!-~]*`	Ref. name of the mate/next read
8	PNEXT	Int	`[0,2`^{31}`-1]`	Position of the mate/next read
9	TLEN	Int	`[-2`^{31}`+1,2`^{31}`-1]`	observed Template LENgth
10	SEQ	String	`\*|[A-Za-z=.]+`	segment SEQuence
11	QUAL	String	`[!-~]+`	ASCII of Phred-scaled base QUALity+33

RNEXT and **PNEXT** This is for something we do not consider in this class. Set the first to '*' and the second to 0

File formats: SAM

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	`[!-?A-~]{1,254}`	Query template NAME
2	FLAG	Int	`[0,2`^{16}`-1]`	bitwise FLAG
3	RNAME	String	`\*|[!-()+-<>-~][!-~]*`	Reference sequence NAME
4	POS	Int	`[0,2`^{31}`-1]`	1-based leftmost mapping POSition
5	MAPQ	Int	`[0,2`^{8}`-1]`	MAPping Quality
6	CIGAR	String	`\*|([0-9]+[MIDNSHPX=])+`	CIGAR string
7	RNEXT	String	`\*|=|[!-()+-<>-~][!-~]*`	Ref. name of the mate/next read
8	PNEXT	Int	`[0,2`^{31}`-1]`	Position of the mate/next read
9	TLEN	Int	`[-2`^{31}`+1,2`^{31}`-1]`	observed Template LENgth
10	SEQ	String	`\*|[A-Za-z=.]+`	segment SEQuence
11	QUAL	String	`[!-~]+`	ASCII of Phred-scaled base QUALity+33

TLEN We ignore this one as well — set it to 0

File formats: SAM

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	`[!-?A-~]{1,254}`	Query template NAME
2	FLAG	Int	$[0,2^{16}-1]$	bitwise FLAG
3	RNAME	String	`\*|[!-()+-<>-~][!-~]*`	Reference sequence NAME
4	POS	Int	$[0,2^{31}-1]$	1-based leftmost mapping POSition
5	MAPQ	Int	$[0,2^{8}-1]$	MAPping Quality
6	CIGAR	String	`\*|([0-9]+[MIDNSHPX=])+`	CIGAR string
7	RNEXT	String	`\*|=|[!-()+-<>-~][!-~]*`	Ref. name of the mate/next read
8	PNEXT	Int	$[0,2^{31}-1]$	Position of the mate/next read
9	TLEN	Int	$[-2^{31}+1,2^{31}-1]$	observed Template LENgth
10	SEQ	String	`\*|[A-Za-z=.]+`	segment SEQuence
11	QUAL	String	`[!-~]+`	ASCII of Phred-scaled base QUALity+33

SEQ This is the read sequence from the FASTQ file

File formats: SAM

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	`[!-?A-~]{1,254}`	Query template NAME
2	FLAG	Int	`[0,2`^{16}`-1]`	bitwise FLAG
3	RNAME	String	`\*|[!-()+-<>-~][!-~]*`	Reference sequence NAME
4	POS	Int	`[0,2`^{31}`-1]`	1-based leftmost mapping POSition
5	MAPQ	Int	`[0,2`^{8}`-1]`	MAPping Quality
6	CIGAR	String	`\*|([0-9]+[MIDNSHPX=])+`	CIGAR string
7	RNEXT	String	`\*|=|[!-()+-<>-~][!-~]*`	Ref. name of the mate/next read
8	PNEXT	Int	`[0,2`^{31}`-1]`	Position of the mate/next read
9	TLEN	Int	`[-2`^{31}`+1,2`^{31}`-1]`	observed Template LENgth
10	SEQ	String	`\*|[A-Za-z=.]+`	segment SEQuence
11	QUAL	String	`[!-~]+`	ASCII of Phred-scaled base QUALity+33

QUAL This is the read quality sequence from the FASTQ file

Class goal

- At the end of the term, the goal is to have a working (but not necessarily efficient) read mapper.

- The topics I have chosen to cover are those that we find most relevant for a read mapper, but we will from time to time stray a little of course to see some interesting things.

That's all Folks!