



SevenBridges

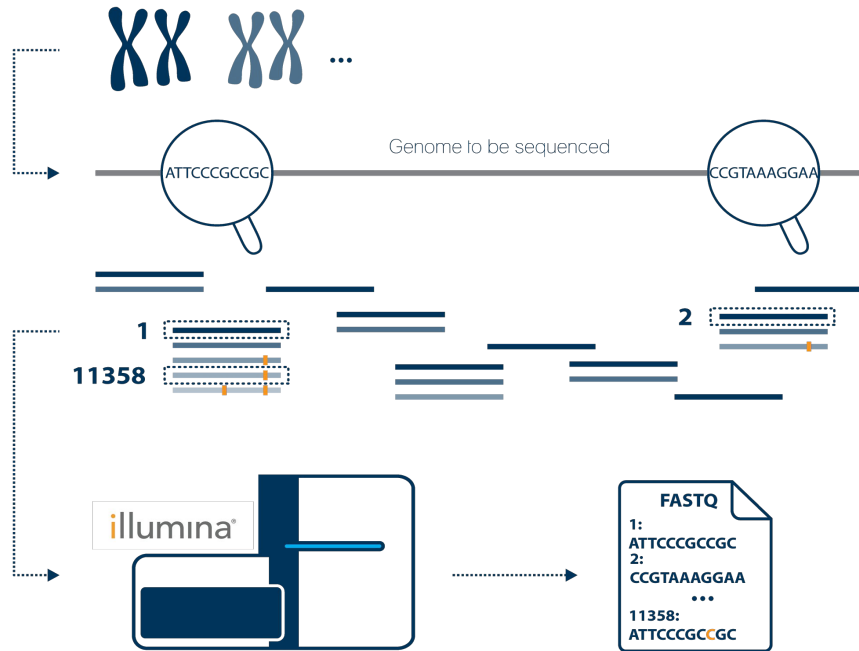
Short read alignment

March 2021

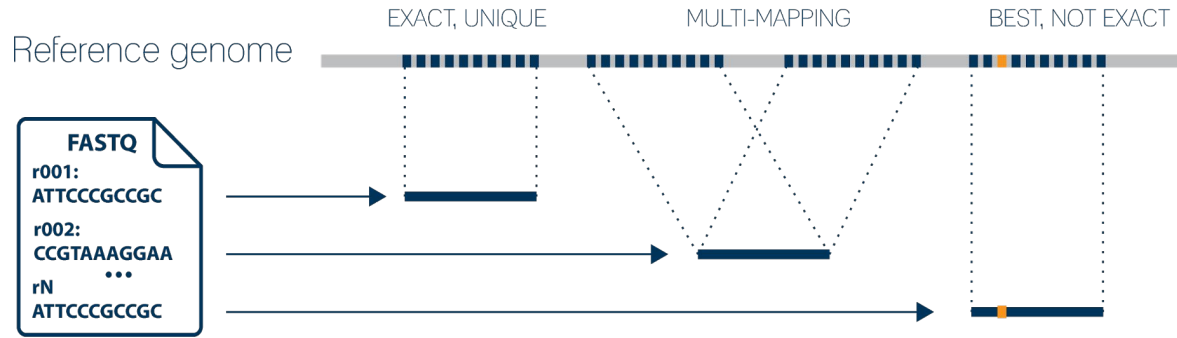
sanja.mijalkovic@sbgenomics.com

DNA Sequencing - Reminder

- We got a FASTQ files with the “reads” - little pieces of the genome



Alignment



How to align reads against the reference?



Problem description

- Scale
 - 1 billion reads for one human genome in average
 - Human Genome fasta has 3.000.000.000 bp
 - Computationally and memory intensive problem
- Let's look at simplified situation - only two sequences
 - Read
 - Reference genome
- We need to find where is this read coming from
 - Starting position of the read on the reference genome
 - How does it align
- What are possible approaches?

Brute Force

Brute Force: At every possible offset in the reference check if all of the characters of the query match

Offset	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
Ref.	T	G	A	T	T	A	C	A	G	A	T	T	A	C	C	...
Query	G	A	T	T	A	C	A									

Brute Force

Brute Force: At every possible offset in the reference check if all of the characters of the query match

Offset	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
Ref.	T	G	A	T	T	A	C	A	G	A	T	T	A	C	C	...
Query		G	A	T	T	A	C	A								

Brute Force

Brute Force: At every possible offset in the reference check if all of the characters of the query match

Offset	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
Ref.	T	G	A	T	T	A	C	A	G	A	T	T	A	C	C	...
Query			G	A	T	T	A	C	A							

Brute Force

This is too slow: For the reference length n and query length m we would have $(n - m + 1) * m$ comparisons.

Run time: $O(nm)$

How can we improve it?

Suffix arrays

Split the reference into n suffixes and sort them alphabetically.

Offset	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Ref.	T	G	A	T	T	A	C	A	G	A	T	T	A	C	C
1	T	G	A	T	T	A	C	A	G	A	T	T	A	C	C
2		G	A	T	T	A	C	A	G	A	T	T	A	C	C
3			A	T	T	A	C	A	G	A	T	T	A	C	C
4				T	T	A	C	A	G	A	T	T	A	C	C
...								...							
15															C

#	Sequence	Pos
1	ACAGATTACC...	6
2	ACC...	13
3	AGATTACC...	8
4	ATTACAGATTACC...	3
5	ATTACC...	10
6	C...	15
7	CAGATTACC...	7
8	CC...	14
9	GATTACAGATTACC...	2
10	GATTACC...	9
11	TACAGATTACC...	5
12	TACC...	12
13	TGATTACAGATTACC...	1
14	TTACAGATTACC...	4
15	TTACC...	11

Suffix arrays

Where is 'GATTACA'?

#	Sequence	Pos
1	ACAGATTACC...	6
2	ACC...	13
3	AGATTACC...	8
4	ATTACAGATTACC...	3
5	ATTACC...	10
6	C...	15
7	CAGATTACC...	7
8	CC...	14
9	GATTACAGATTACC...	2
10	GATTACC...	9
11	TACAGATTACC...	5
12	TACC...	12
13	TGATTACAGATTACC...	1
14	TTACAGATTACC...	4
15	TTACC...	11

Suffix arrays

Where is 'GATTACA'? - **Binary search!**

Step 1

Lo = 1; Hi = 15; Mid = $(1+15)/2 = 8$

Suffix[8] = CC...

'GATTACA' is higher than 'CC...'

=> Lo = Mid + 1

Step 2

Lo = 9; Hi = 15; Mid = $(9+15)/2 = 12$

Suffix[12] = TACC...

'GATTACA' is lower than 'TACC...'

=> Hi = Mid - 1

Step 3

Lo = 9; Hi = 11; Mid = $(9+11)/2 = 10$

Suffix[10] = GATTACC...

'GATTACA' is lower than 'GATTACC...'

=> Hi = Mid - 1

Step 4

Lo = 9; Hi = 9; Mid = $(9+9)/2 = 9$

Suffix[9] = GATTACA...

=> **Match at position 2!**

Suffix arrays

This is more complicated approach for implementation and takes a lot of memory, but much faster!

Looking up a query 32 times instead of 3 billion as with Brute Force (for the reference with 3 billion base pairs).

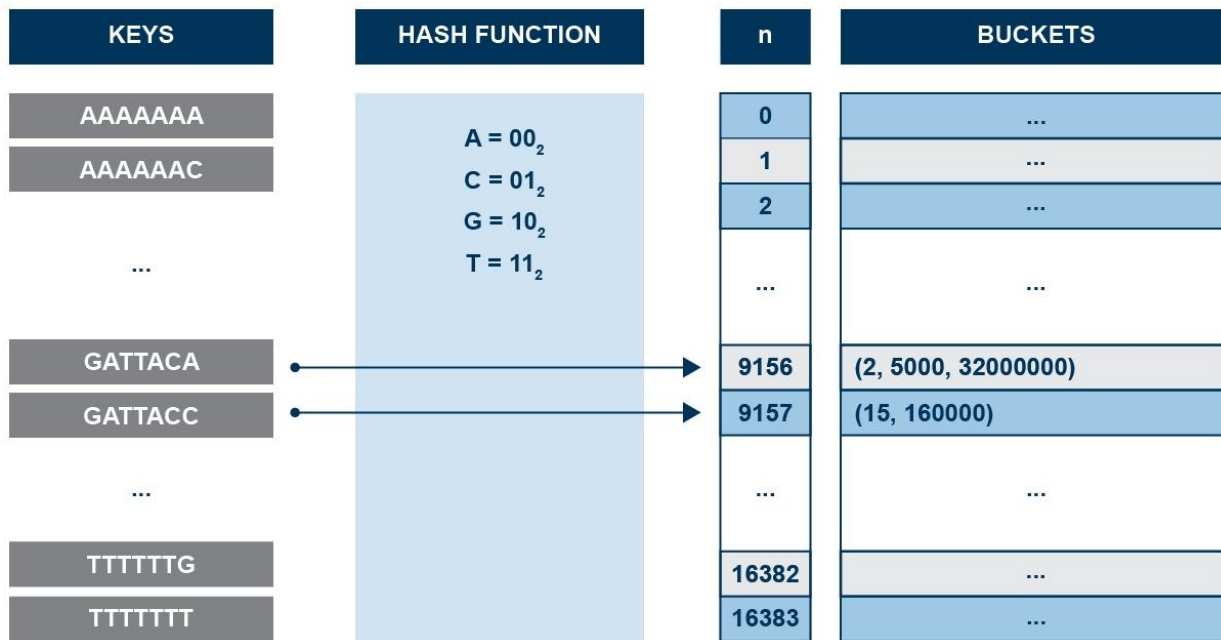
Run time: $O(m \log_2 n)$

Can we do it any better?

It would be great if would know in advance the index at which the query is located in the array.

Hash map

Idea: Build a table with every k-mer in the reference.



Hash map

Number of possible sequences of length k ?

$$4^k$$

$$4^7 = 16,384 \text{ (easy to store)}$$

$$4^{20} = 1,099,511,627,776 \text{ (impossible to directly store in RAM!)}$$

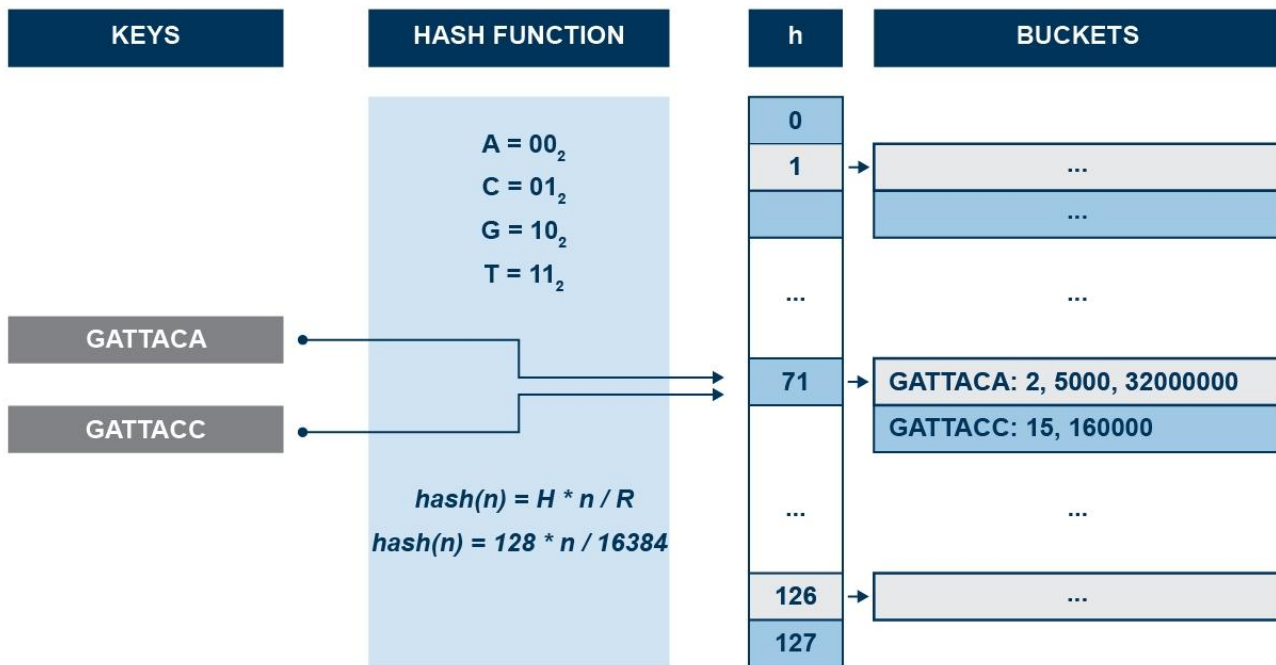
But! There are only 3 billion 20-mers in the reference - 99.7% of the table would be empty anyways.

Hash map

- Use a **hash function** to shrink the possible range
 - Maps a number n in $[0,R]$ to h in $[0,H]$
 - Use 128 buckets instead of 16,384 or 1 billion instead of 1 trillion
 - Division: $\text{hash}(n) = H * n / R$,
 - H - # of buckets
 - n is sequence number
 - R - total # of sequences
 - $\text{hash}(\text{GATTACA}) = 128 * 9156 / 16384 = 71$
 - Modulo: $\text{hash}(n) = n \% H$
 - $\text{hash}(\text{GATTACA}) = 9156 \% 128 = 68$
- By construction, multiple keys have the same hash value

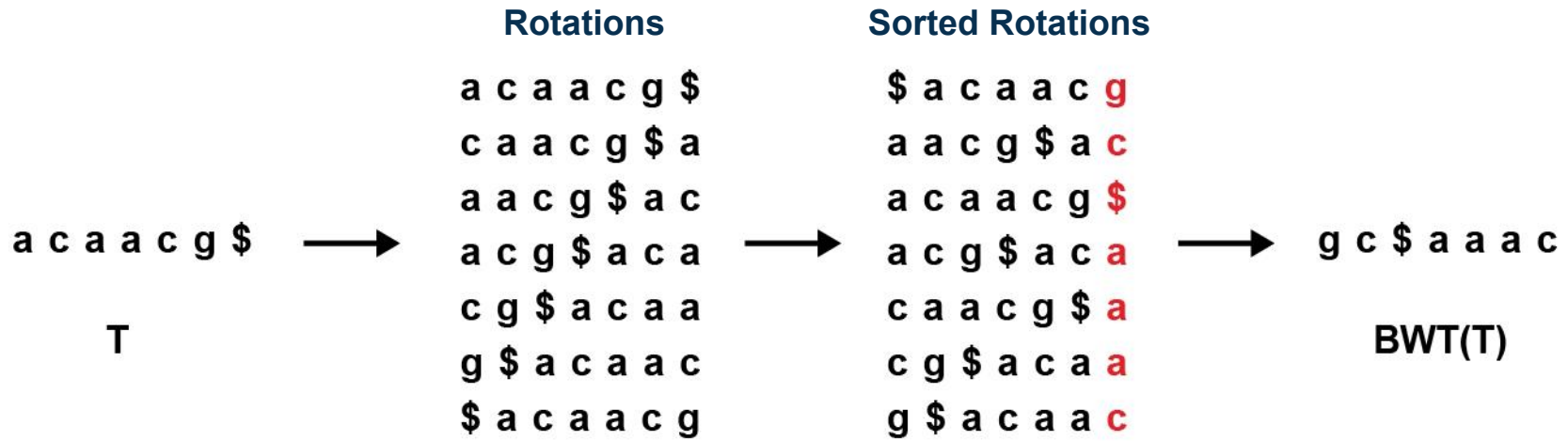
Hash map

- Store elements with the same key in a bucket chained together
- Looking up a value scans the entire bucket
- Run time:
 - Construction: $O(n)$
 - Lookup: $O(1) + O(z)$
 - Sorting the genome mers in linear time



Burrows-Wheeler Transformation

- Reversible lossless transformation algorithm which permutes an input string into a new string
- **BWT** string lends itself to an **effective compression**



Burrows-Wheeler Transformation

- Rank preserving property - needed for **LF** (Last-First) mapping

¹ ¹ ² ³ ² ¹
a c a a c g \$

\$	a ₁	c ₁	a ₂	a ₃	c ₂	g ₁
a ₂	a ₃	c ₂	g ₁	\$	a ₁	c ₁
a₁	c ₁	a ₂	a ₃	c ₂	g ₁	\$
a ₃	c ₂	g ₁	\$	a ₁	c ₁	a ₂
c ₁	a ₂	a ₃	c ₂	g ₁	\$	a₁
c ₂	g ₁	\$	a ₁	c ₁	a ₂	a ₃
g ₁	\$	a ₁	c ₁	a ₂	a ₃	c ₂

Burrows-Wheeler Transformation

- BWT is reversible
- Recreating T from BWT(T): Start in the first row and apply **LF** repeatedly, accumulating predecessors along the way:

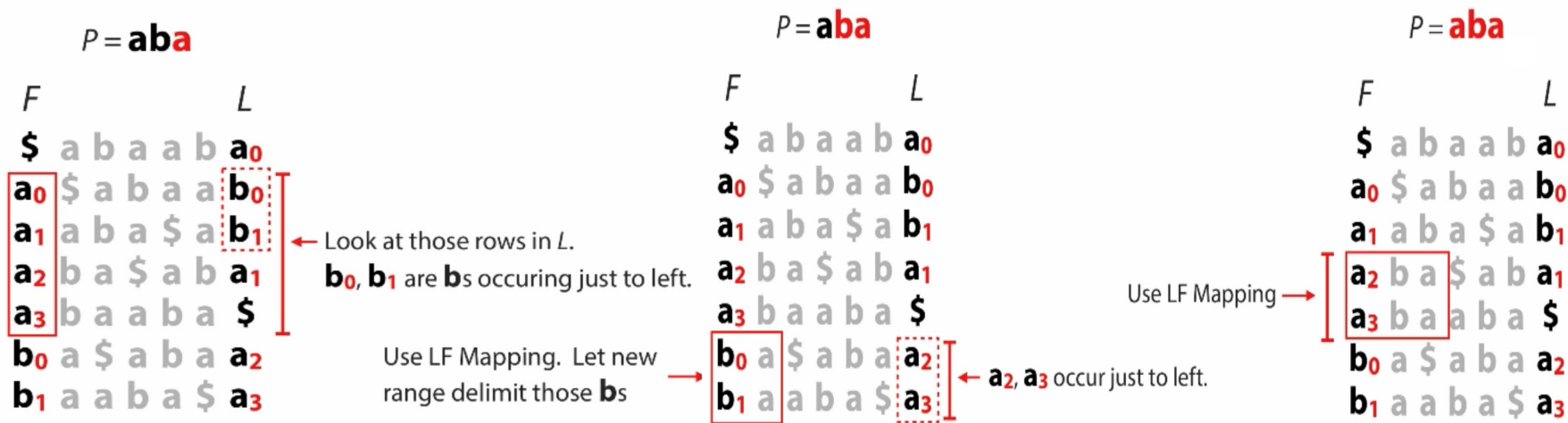


FM index

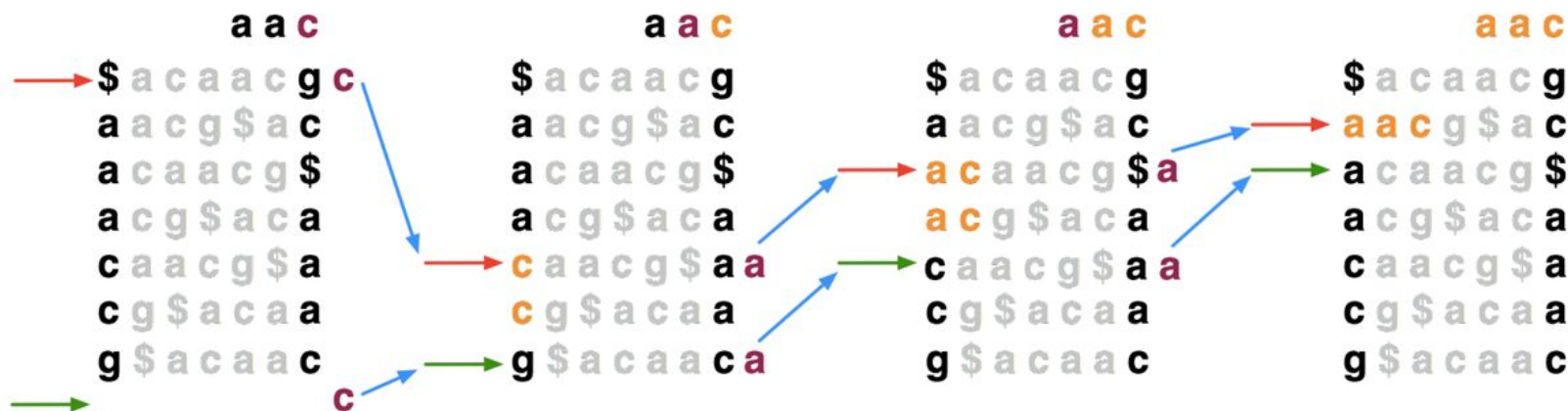
- Ferragina & Manzini proposed “**FM Index**” based on BWT
- Observed:
 - LF Mapping also allows ***exact matching*** within T
 - LF(i) can be made fast with ***checkpointing***

FM index - Idea

- Algorithm that finds all occurrences of a pattern P in text T by using $BWT(T)$
- Look for range of rows of $BW(T)$ with P as prefix
- Do this for P 's shortest suffix, then extend to successively longer suffixes until range becomes empty or we exhausted P



FM index - another example



FM index - checkpoints

- **LF**(i, **qc**) must determine the rank of **qc** in row i
 - Naïve solution: Count occurrences of **qc** in all previous rows
- Better solution: Pre-calculate occurrences of A, C, G and T in L up to periodic **checkpoints**:

	a	c	g	t	
\$ a c a a c g	0	0	1	0	← Lookup here succeeds as usual
a a c g \$ a c					
a c a a c g \$					
a c g \$ a c a					
c a a c g \$ a					← Oops: not a checkpoint
c g \$ a c a a	3	1	1	0	← But there is one nearby
g \$ a c a a c					

FM index - Position in reference - Suffix array sample

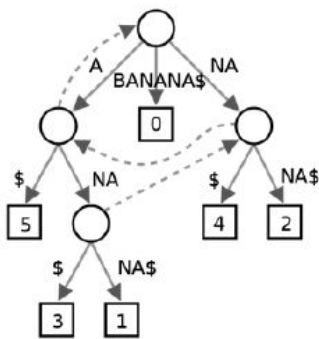
- Once we know a row contains a legal alignment, how do we determine its position in the reference?
- Hybrid solution: Store sample of **suffix array**; “walk left” to next sampled (“marked”) row to the left:



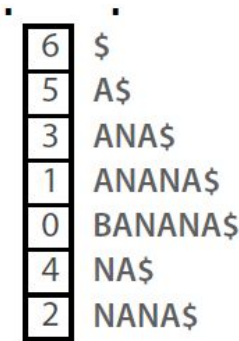
FM index

- Entire FM Index of the reference consists of:
 - Reference BWT (same size as reference)
 - Suffix array sample (50% size of the reference)
 - Checkpoints (15% size of the reference for every 448th row)
- Example: DNA alphabet (2 bits per nucleotide), human genome (3 billion nucleotides), SA sample - every 32nd row, checkpoints - every 128th row:
 - First column: 16 bytes
 - Last column: 2 bits x 3 billion chars ~ 750MB
 - SA sample: 3 billion x 4 bytes / 32 ~ 400MB
 - Checkpoints: 3 billion x 4 bytes x 4 / 128 ~ 400MB

Comparison of different seeding techniques



Suffix tree
≥ 45 GB



Suffix array
≥ 12 GB

\$ BANANA
A \$ BANAN
ANA \$ BAN
ANANA \$ B
BANANA \$
NA \$ BANA
NANA \$ BA

FM Index
~ 1 GB

Score-based alignment

- So far we mentioned only exact match alignments
- Some of the reads will match perfectly to the reference
- Many reads will not:
 - Genomic variations ('mutations', SNP, Indel, etc.)
 - Sequencing errors
- Aligners commonly take a score-based approach:
 - Calculate a score, related to the distance between the read and the local reference sequence
 - Place the read so that the score is maximised
- Many algorithms exist, there is a speed/precision tradeoff

Two-step aligners

- Most modern aligners take a two-step approach (also called “seed and extend”)
- First find “coarse” alignments or seeds
- Then do fine detailed alignments in the vicinity of the seeds
- Choose the best scoring fine grain alignment

Seeding step

- Find a set of possible coordinates
- Many false-positives, but very is usually fast
- Usually aligners use one of mentioned data structures to create index for the references
 - Suffix arrays
 - Hash maps
 - BWT
- A common tradeoff is speed vs RAM footprint

Seeding step example

- Example: a simple hash-based scheme
- Create a hash-table which holds the positions, where each kmer in the genome occurs
- For each kmer in the read find the list of locations
- Regions with hits from many different kmers are hits
- Some kmers will have no hits, some many

Extending step

- Calculates a precise sequences match score
- Often use a lot of RAM (related to sequence size)
- The extend step also needs to produce the information on **how** the sequences match
- Mostly based on **dynamic programming**

Smith-Waterman aligner

- Dynamic programming local alignment algorithm

	-	P	E
-	0	0	0
E	0	0	1

Annotations for the bottom-right cell (E, E):

- From (E, P): $0 + 1 = 1$
- From (P, E): $0 - 2 = -2[0]$
- From (E, -): $0 - 2 = -2[0]$

$$\text{Value}_{i,j} = \begin{cases} 0 \\ \text{Value}_{i-1,j-1} + M \\ \text{Value}_{i-1,j} + G \\ \text{Value}_{i,j-1} + G \end{cases}$$

M = Match score if letters match, otherwise mismatch penalty (+/- 1 in this example)

G = Gap penalty (-2 in this example)

Smith-Waterman aligner

	-	P	E	R	I	C	A	A
-	0	0	0	0	0	0	0	0
E	0	0	1	0	0	0	0	0
R	0	0	0	2	0	0	0	0
C	0	0	0	0	1	1	0	0
A	0	0	0	0	0	0	2	1
A	0	0	0	0	0	0	1	3

P | E | R | I | C | A | A
 - | E | R | - | C | A | A

- Calculate scores for each field
 - Remember the path to each field
 - Backtrack from maximum to zero
 - Diagonal step is match/mismatch
 - Horizontal step is a deletion
 - Vertical step is an insertion
- Match=1, Mismatch=-1, Gap =-2

CIGAR strings

- Run length encoding:

AAAABBBAAAD = 4A2B3A1D

- CIGAR Codes:

- **M** - alignment match (Match or Mismatch)
- **I** - insertion, **D** - Deletion, **S** - soft clip
- H, X, =, P, N - rare in DNA Seq

P	E	R	I	C	A	A			
-	E	R	C	A	A	-	=>	1S5M1S	

P	E	R	I	C	A	A			
-	E	R	-	C	A	A	=>	1S2M1D2M	

Exercise 1 - a simple seeding aligner step

- We will create seed part of alignment algorithm
- This will be done in 4 steps
- Data we are going to use for tests
 - Read:
'ACATACACATGTCCTGTTTTGATGTCCTATAATTAATTTTCTCTCCGTTTTTAA
CTTTTATCTATCTTATTAATGT'
 - Reference sequence: "example_reference.fasta" located in the project
- Seed phase of alignment algorithm
 - Implement a simple hash-based aligner in Python
 - A dict can be used to create the index
 - Create an index for each kmer in a sequence
 - To map a read, find locations for each kmer in the read
 - Find the region with most kmers mapping to it
 - Mind the offset from the beginning of the read

Exercise 1 - a simple seeding aligner step

Step 1.

- Create index for provided fasta file, chromosome 20
- `def create_index(fasta, k)`
- Function should return dict
 - Keys - kmers present in reference
 - Values - list of kmer positions in the reference

Step 2.

- Analyse different k-mer sizes (e.g. $k=10$ and $k=6$)
 - Number of unique k-mers
 - Number of collisions

Exercise 1 - a simple seeding aligner step

Step 3.

- Create seed function
- `def seed_read(index, k, read)`
- Returns dict with
 - reference positions as keys
 - number of supporting kmers as values

Step 4.

- Possible improvements?
 - Filter out all kmers that have more than n mapping positions
 - `def seed_read2_with_improvements(index, k, read, n=2)`

Exercise 2 - Smith Waterman by hand

- Sequence1 - "attcagct"
- Sequence2 - "atcagtct"

Scoring:

- $\text{indel_score} = -2$
- $\text{match_score} = 2$
- $\text{mismatch_score} = -1$

Exercise 2 - Smith Waterman by hand

		A	T	T	C	A	G	C	T
	0	0	0	0	0	0	0	0	0
A	0	2							
T	0								
C	0								
A	0								
G	0								
T	0								
C	0								
T	0								

Exercise 2 - Smith Waterman by hand

		A	T	T	C	A	G	C	T
	0	0	0	0	0	0	0	0	0
A	0	2	0	0	0	2	0	0	0
T	0	0	4	2	0	0	1	0	2
C	0	0	2	3	4	2	0	3	1
A	0	2	0	1	2	6	4	2	2
G	0	0	1	0	0	4	8	6	4
T	0	0	2	3	1	2	6	7	8
C	0	0	0	1	5	3	4	8	6
T	0	0	2	2	3	4	2	6	10

Exercise 3 - Smith Waterman simplified implementation

Implement algorithm that will fill in score table for two sequences

Let's test it on previous example:

- Sequence1 - "attcagct"
- Sequence2 - "atcagtct"

Scoring:

- $\text{indel_score} = -2$
- $\text{match_score} = 2$
- $\text{mismatch_score} = -1$

BWA aligner

- A widely used aligner for DNA Sequencing
- <http://bio-bwa.sourceforge.net> for more info
- BWA requires an index to be built
 - It uses BWT transformation for reference index creation
- Seed and extend approach

BAM file format

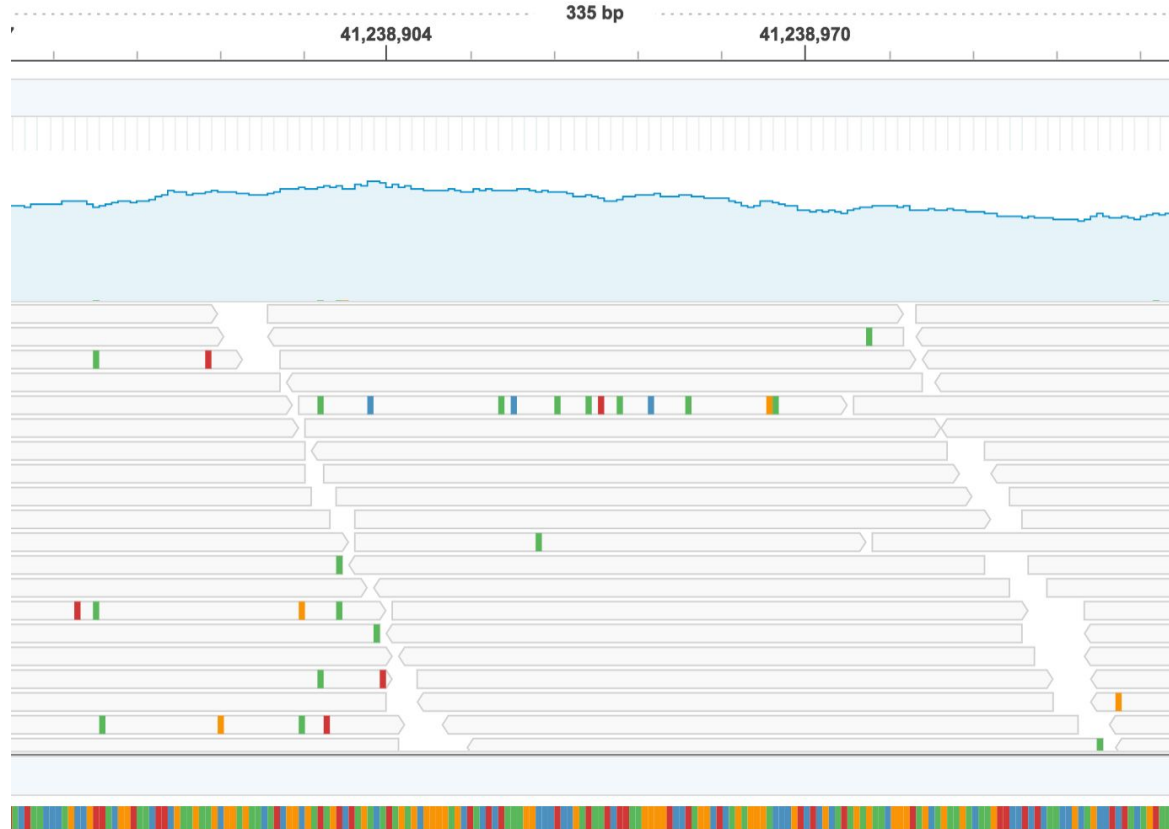
- Standardized format for holding aligned reads
 - The read sequence and qualities
 - Position (chromosome and the first matching base)
 - CIGAR string
 - Flag (various info, like *read has a pair*, *read is aligned*, etc.)
 - Read pair position
 - Other optional tags
- Bgzip compression (roughly $\frac{1}{3}$ of raw text)
- Besides BAM, SAM (plain text) also exists
- Specification: <https://samtools.github.io/hts-specs/SAMv1.pdf>

BAM file format

- BAM is 0-based, SAM is 1-based
- BAM files can be sorted:
 - Coordinate sort, by read position
 - Query name (read name), read pairs are placed together
- Coordinate sorted BAM files can be *indexed*
- By convention, BAM index files:
 - Have .bai appended to the name
 - Are placed in the same folder as the BAM
 - Usually not explicitly passed to tools
 - Majority of the tools requires .bai file

Genome browser

- A visualization tool for genomic data
- See how the reads actually align to the reference



Pysam

- Pysam can be used to process BAM files
- `pysam.AlignmentFile`
 - `AlignmentFile(path_to_file)`
 - Iteration through reads in BAM file:
 - `for read in AlignmentFile(path_to_file):`
- Reads are wrapped in `AlignedSegment` objects
 - `AlignedSegment` provides access to read fields and helpers
- `pysam.AlignmentFile` supports fetching regions
 - The BAM file needs to be sorted and indexed!

Exercise 5 - Pysam

- Create an `AlignmentFile` object for “merged-tumor.bam”
 - Take the first read from the `AlignmentFile`
 - Inspect the fields in the `AlignedSegment`
 - Inspect the flag field
 - Check out the flag for some reads using:
<https://broadinstitute.github.io/picard/explain-flags.html>
- Calculate:
 - How many unmapped reads are in the file?
 - Total number of reads
 - Number of reads with mapping quality 0
 - Average mapping quality for all the reads
 - Average mapping quality if reads with 0 map quality are filtered out

Questions?