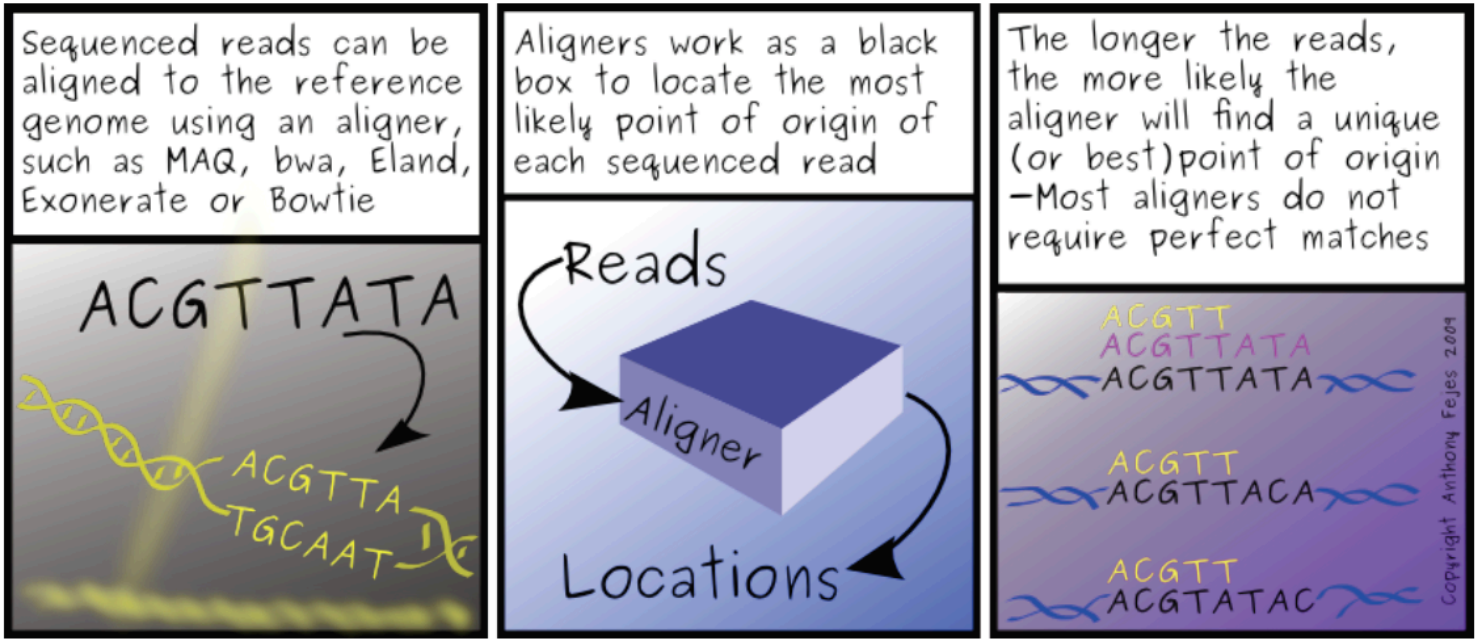
Burrows-Wheeler Transform

Bioinformatica

Read mapping

All the sequencing-based techniques require read mapping as a first step.

Existing alignment tools are not fast enough → need new algorithms!



Mapping Reads

Problem: We are given a read, *R,* and a reference sequence, *S*. Find the best or all occurrences of *R* in *S*.

Example:

R = AAACGAGTTA

S = TTAATGC*AAACGAGTTA*CCCAATATATAT*AAACCAGTTA*TT

Considering no error: one occurrence.

Considering up to 1 substitution error: two occurrences.

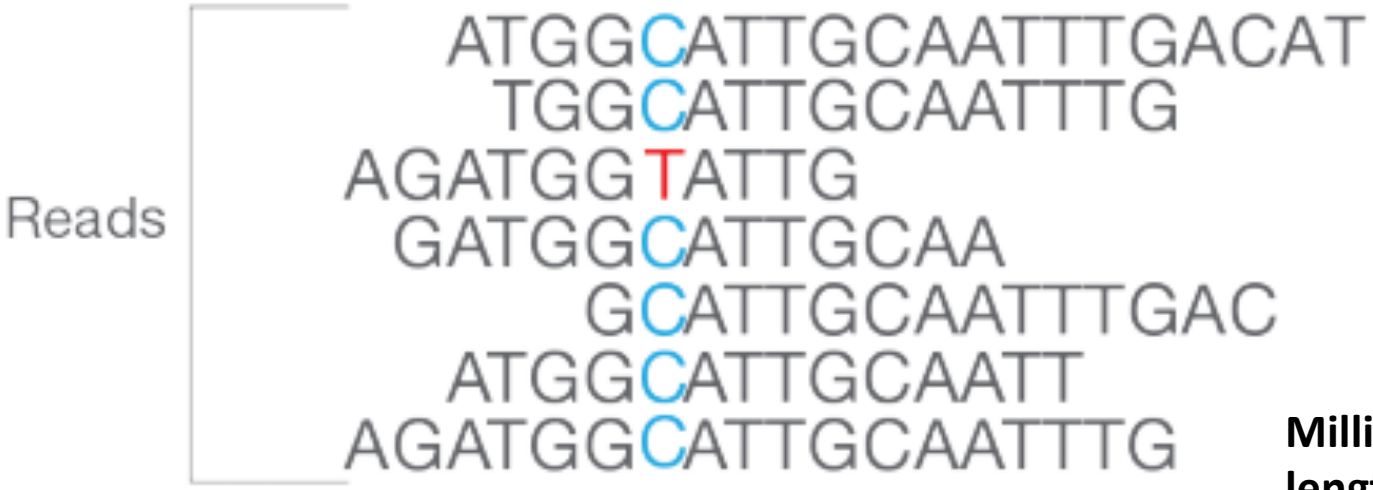
Considering up to 10 substitution errors: many meaningless occurrences!

The problem is….

Category	Detection or Application	Recommended Coverage (x) or Reads (millions)	References
Whole genome sequencing	Homozygous SNVs	15x	Bentley et al., 2008
	Heterozygous SNVs	33x	Bentley et al., 2008
	INDELs	60x	Feng et al., 2014
	Genotype calls	35x	Ajay et al., 2011
	CNV	1-8x	Xie et al., 2009; Medvedev at al., 2010
Whole exome sequencing	Homozygous SNVs	100x (3x local depth)	Clark et al., 2011; Meynert et al., 2013
	Heterozygous SNVs	100x (13x local depth)	Clark et al., 2011; Meynert et al., 2013
	INDELs	not recommended	Feng et al., 2014
Transcriptome Sequencing	Differential expression profiling	10-25M	Liu Y. et al., 2014; ENCODE 2011 RNA-Seq
	Alternative splicing	50-100M	Liu Y. et al., 2013; ENCODE 2011 RNA-Seq
	Allele specific expression	50-100M	Liu Y. et al., 2013; ENCODE 2011 RNA-Seq
	De novo assembly	>100M	Liu Y. et al., 2013; ENCODE 2011 RNA-Seq
DNA Target-Based Sequencing	ChIP-Seq	10-14M (sharp peaks); 20-40M (broad marks)	Rozowsky et al., 2009; ENCODE 2011 Genome; Landt et al., 2012

Human 3.2Gbp

How to solve the problem?



Millions of reads with length = hundreds bp

Reference Genome

AGATGGTATTGCAATTTGACAT

3,000,000,000 bp

How to solve it?

Strings

Read

CTCAAACTCCTGACCTTTGGTGATCCACCCGCCTAGGCCTTC

x billions

Reference

x million

We're going to *need* the right algorithms...

Exact matching

Find places where *pattern P* occurs as a substring of *text T*.
Each such place is an *occurrence* or *match*.

Let $n = |P|$, and let $m = |T|$ Assume $n \leq m$

Alignment: a way of putting *P's* characters opposite *T's*.
May or may not correspond to an match.

P: `word`
T: `There would have been a time for such a word`

Alignment 1:
`word`

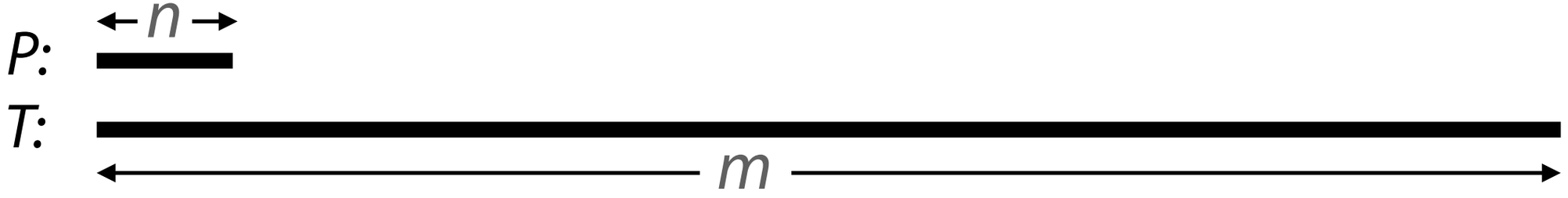
Alignment 2:
`word`

Exact matching: naïve algorithm

$n = |P|$ $\quad$ $m = |T|$

How many alignments are possible?

$$m - n + 1$$



Exact matching: naïve algorithm

$n = |P|$ $\quad$ $m = |T|$

Greatest # character comparisons possible?

$$n(m - n + 1)$$

P: **aaaa**

T: **aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa**

aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa

Exact matching: better algorithms?

P: **word**

T: **There would have been a time for such a word**

word ----------→

u doesn't occur in *P*, so skip next two alignments

P: **word**

T: **There would have been a time for such a word**

word ----------→

word skip!

word skip!

word

We'll take such ideas further when we discuss Boyer-Moore

Boyer-Moore

Learn from character comparisons to skip pointless alignments

1. When we hit a mismatch, move *P* along until the mismatch becomes a match — "Bad character rule"
2. When we move *P* along, make sure characters that matched in the last alignment also match in the next alignment — "Good suffix rule"
3. Try alignments in one direction, but do character comparisons in *opposite* direction — For longer skips

Boyer, RS and Moore, JS. "A fast string searching algorithm." *Communications of the ACM* 20.10 (1977): 762-772.

Naive vs Boyer-Moore

As *m* & *n* grow, # characters comparisons grows with...

$|P| = n \quad |T| = m$

	Naïve matching	Boyer-Moore
Worst case	m·n	m
Best case	m	m / n

Performance comparison

Simple Python implementations of naïve and Boyer-Moore:

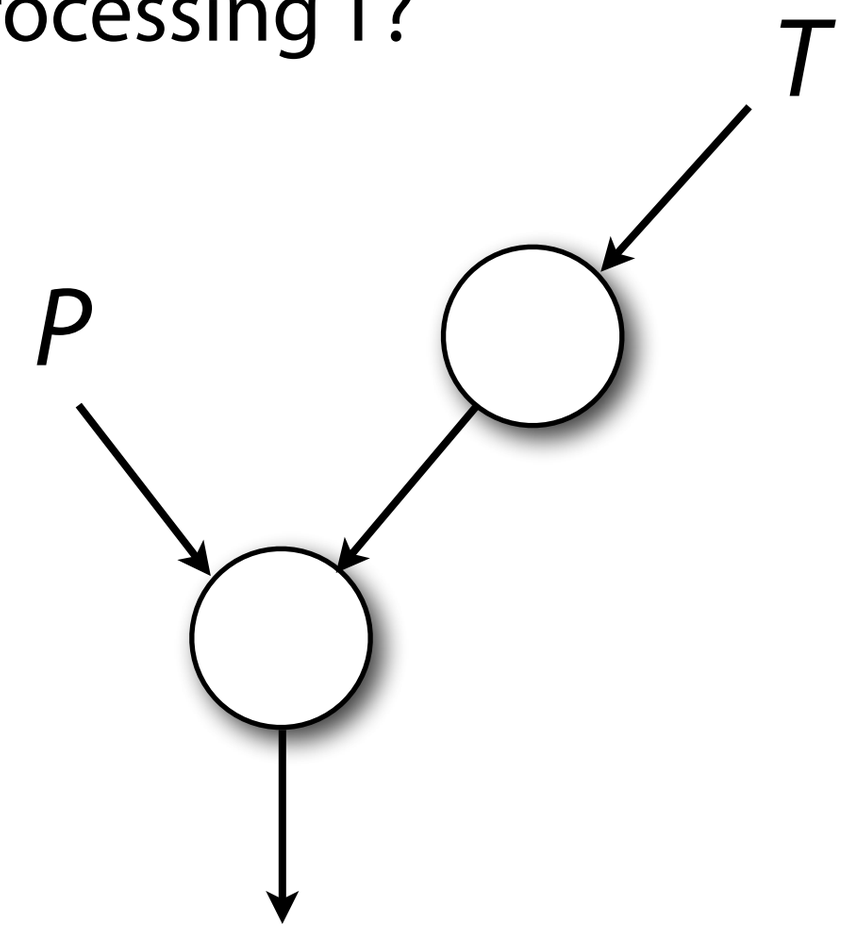
	Naïve matching		Boyer-Moore		
	# character comparisons	wall clock time	# character comparisons	wall clock time	
P: "tomorrow" **T**: Shakespeare's complete works	5,906,125	2.90 s	785,855	1.54 s	17 matches $\|T\| = 5.59$ M
P: 50 nt string from Alu repeat* **T**: Human reference (hg19) chromosome 1	307,013,905	137 s	32,495,111	55 s	336 matches $\|T\| = 249$ M

* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

Preprocessing

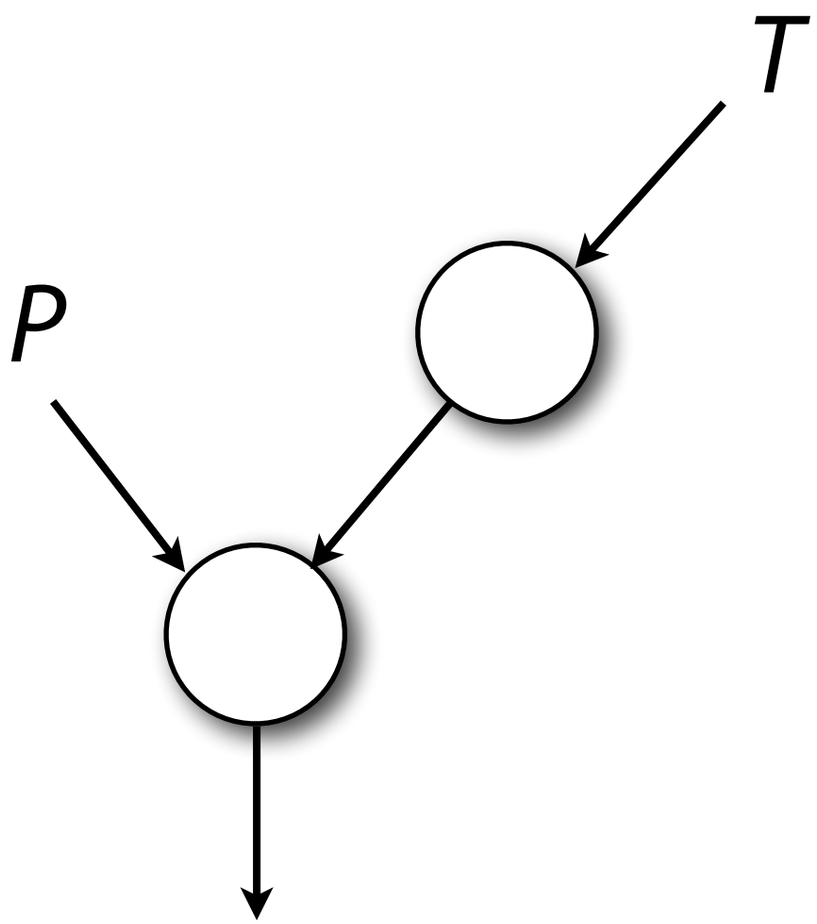
Boyer-Moore preprocessed P

What about preprocessing T?



Preprocessing

Algorithm that preprocesses *T* is *offline*.
Otherwise, algorithm is *online*.



Online or offline?

- Naïve algorithm
- Boyer-Moore
- Web search engine
- Read alignment

Indexing DNA

Index of T

T: C G T G C G T G C T T

Indexing DNA

Index of T

`CGTGC: 0`

T: `C G T G C G T G C T T`

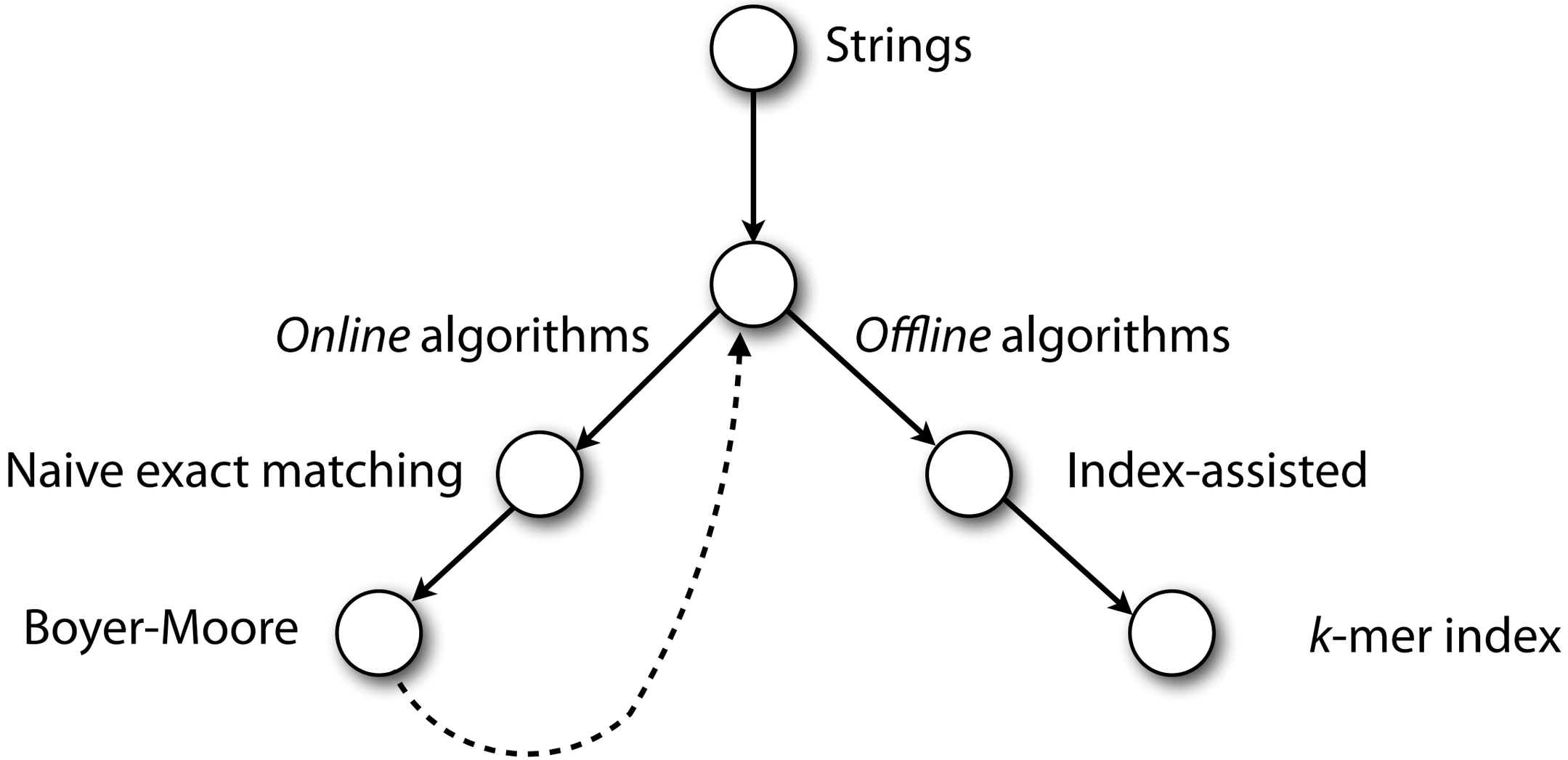
Indexing DNA

Index of T

```
CGTGC:  0,4
GCGTG:  3
GTGCC:  1
GTGCT:  5
TGCCT:  2
TGCTT:  6
```

T: **C G T G C G T G C T T**

Approximate matching



We have focused on *exact* matching...

... in reality, we have to deal with *differences*

Read

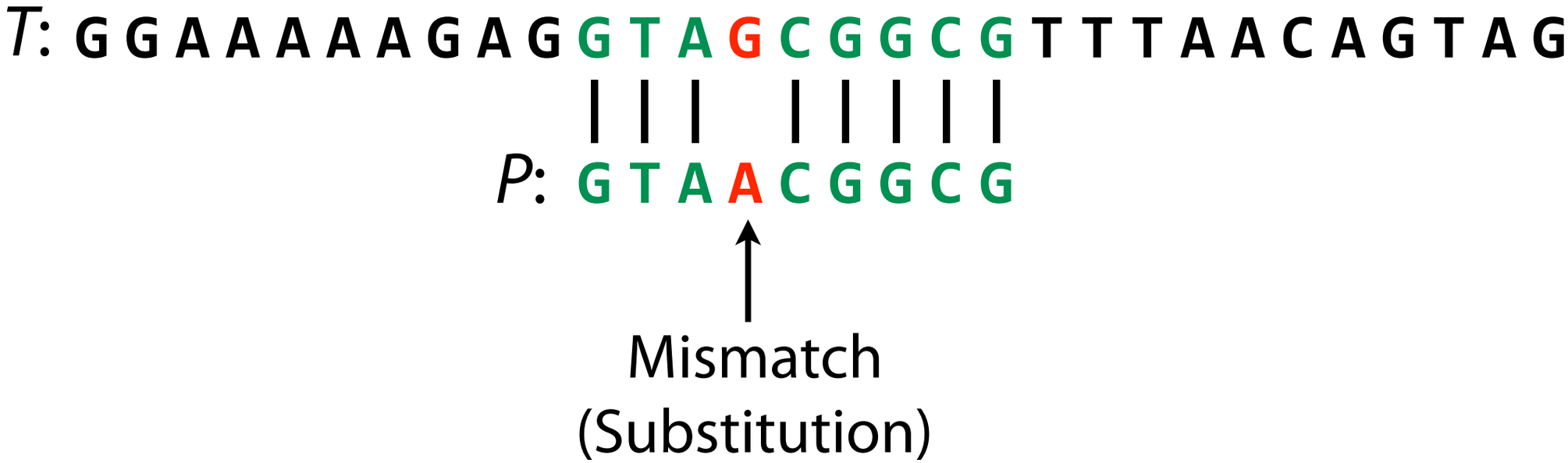
CTCAAACTCCTGACCTTTGGTGATCCACCCGCCTNGGCCTTC

Reference

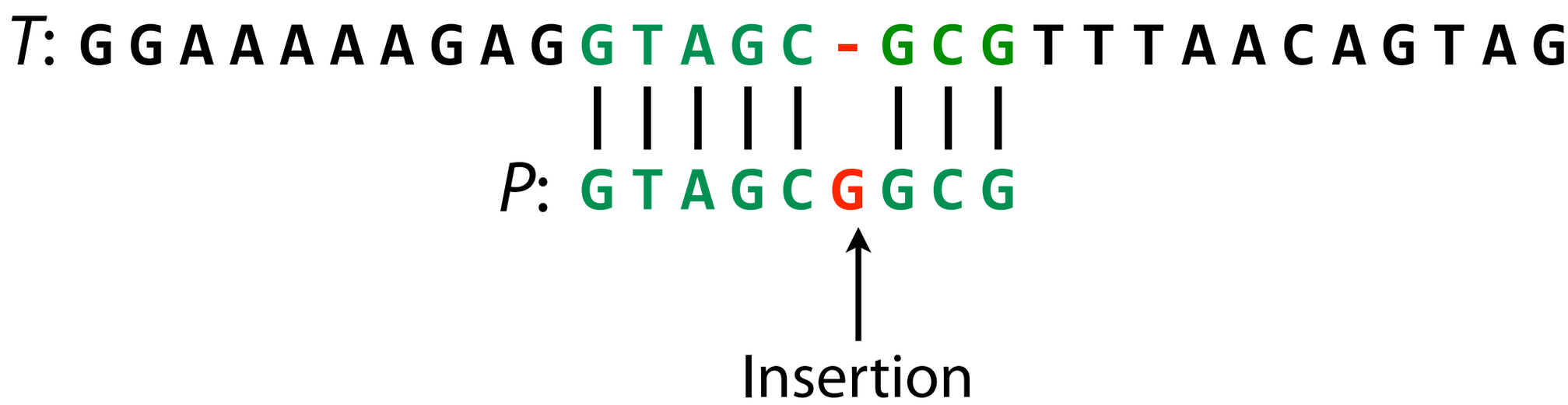
Sequence differences occur because of...

1. Sequencing error
2. Genetic variation

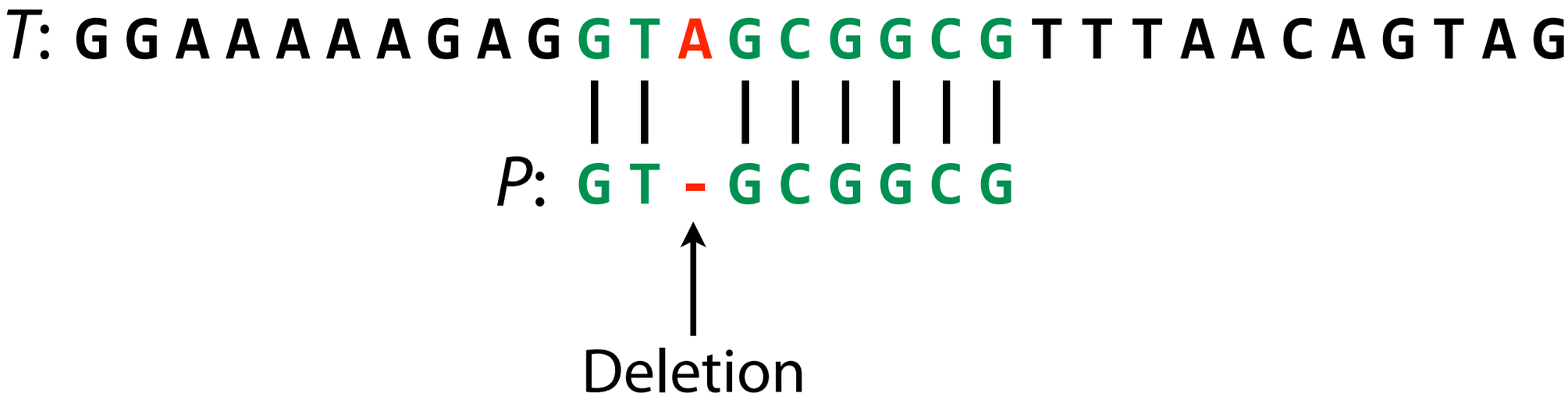
Approximate matching



Approximate matching



Approximate matching



Hamming distance

For X & Y where $|X| = |Y|$, *hamming distance* =
minimum # substitutions needed to turn one into the other

Hamming distance = 3

Edit distance

(AKA Levenshtein distance)

For *X* & *Y*, *edit distance* = minimum # edits (substitutions, insertions, deletions) needed to turn one into the other

```
X: T G G C C G C G C A A A A A C A G C
   | |   | | | | | | | | | |   | | | |
Y: T G A C C G C G C A A A A - C A G C
```

Edit distance = 2

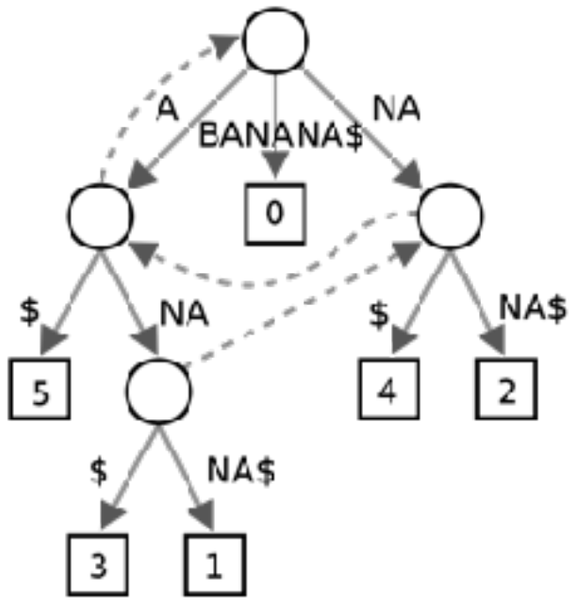
```
X: G C G T A T G C G G C T A - A C G C
   | |   | | | | | | | | | |   | | |
Y: G C - T A T G C G G C T A T A C G C
```

Edit distance = 2

Indexing with suffixes

We studied indexes built over substrings of *T*

Different approach is to index *suffixes* of *T*. This yields surprisingly economical & practical data structures:



Suffix Tree

6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix Array

$ B A N A N **A**
A **$** B A N A **N**
A **N** A **$** B A **N**
A **N** A N A **$** **B**
B A N A N A **$**
N A **$** B A N **A**
N A N A **$** B **A**

FM Index

Suffix trie

How do we check whether a string *S* is a substring of *T*?

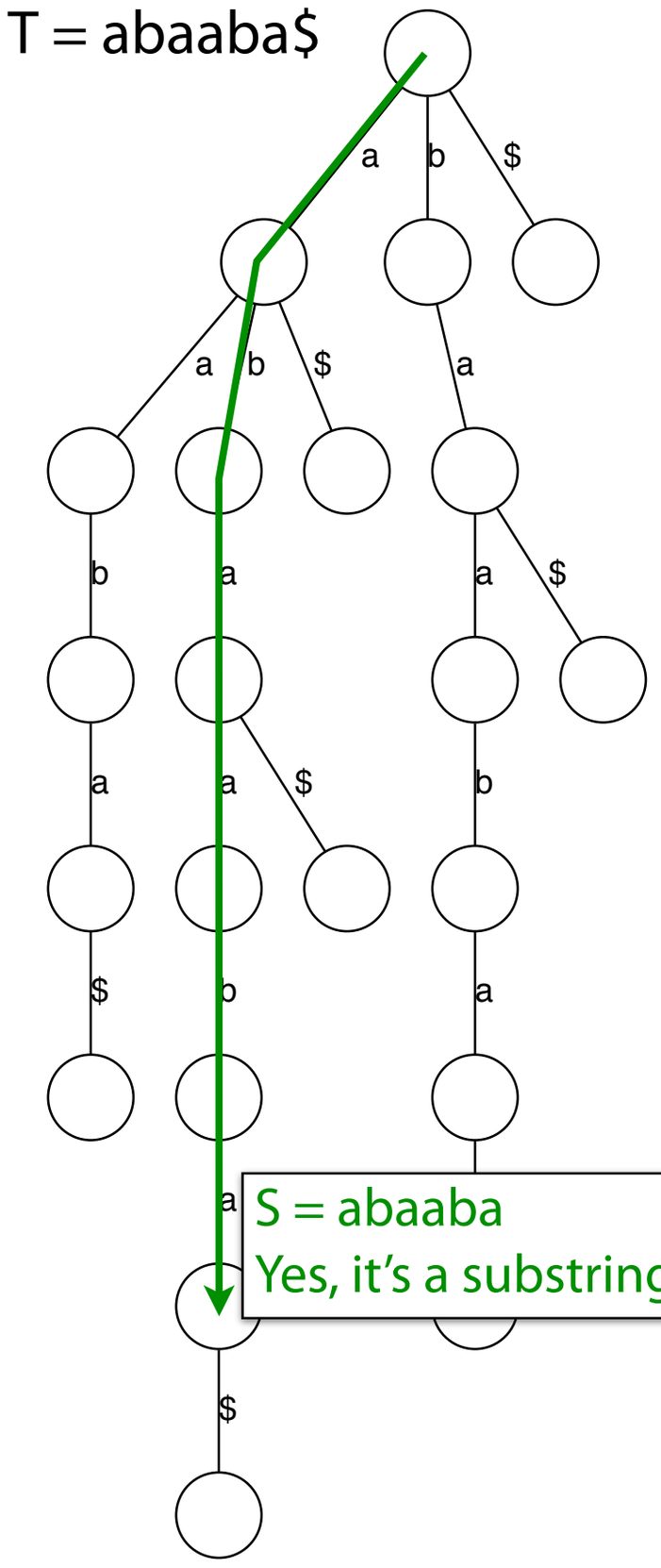
Note: Each of *T*'s substrings is spelled out along a path from the root.

Every *substring* is a *prefix* of some *suffix* of T.

Start at the root and follow the edges labeled with the characters of *S*

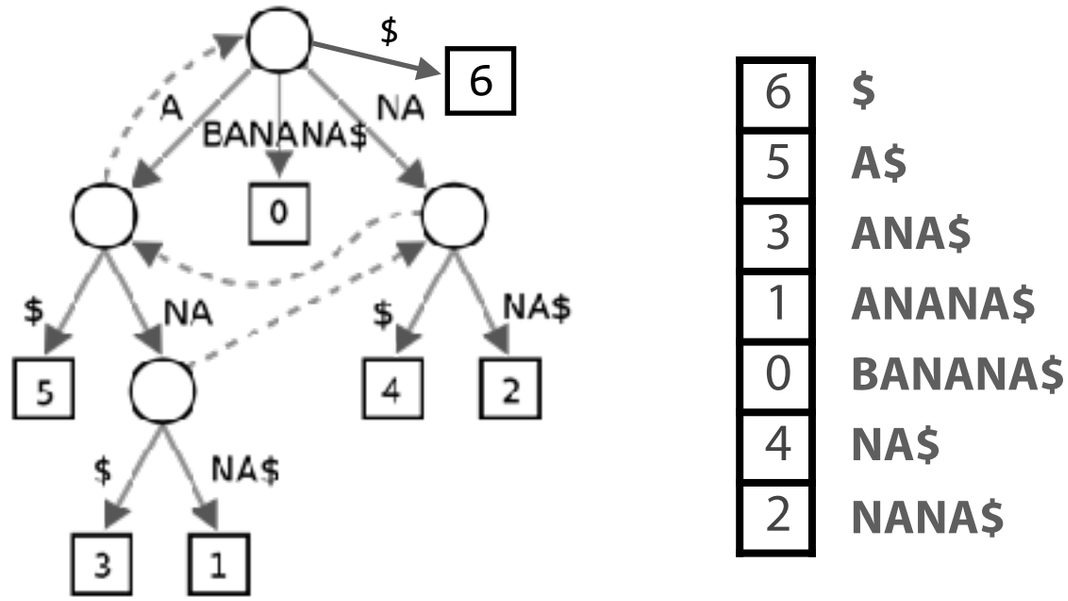
If we "fall off" the trie -- i.e. there is no outgoing edge for next character of *S*, then *S* is not a substring of *T*

If we exhaust *S* without falling off, *S* is a substring of *T*



Suffix array: summary

Just *m* integers, with O(*n* log *m*) query time

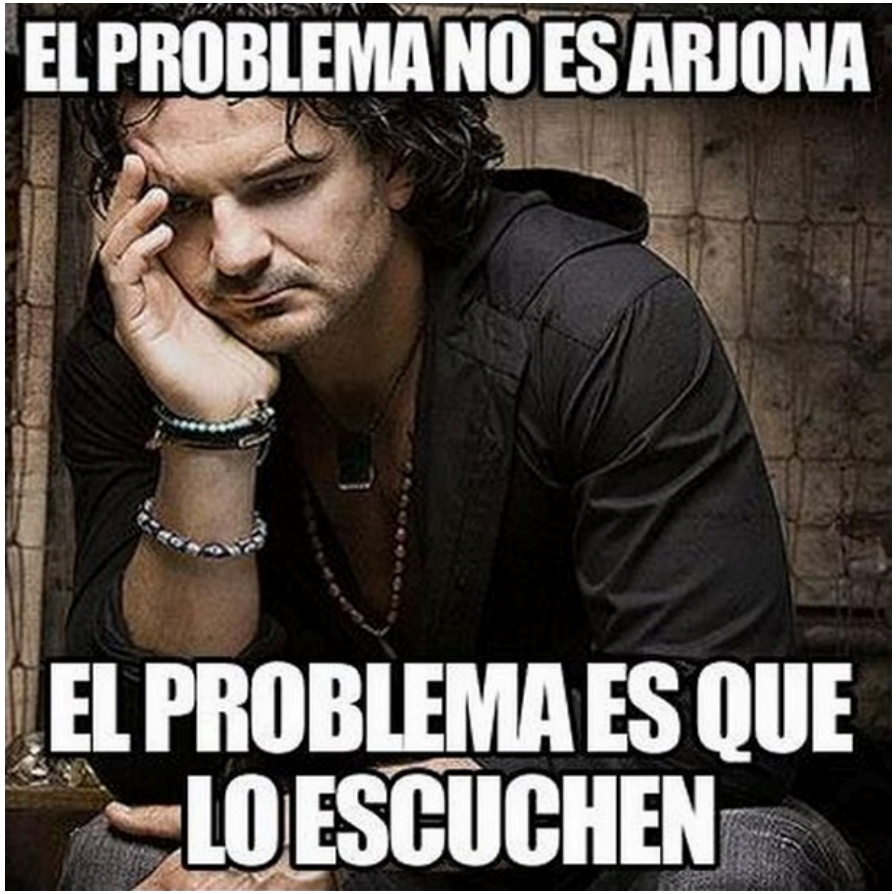


Suffix Tree

6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

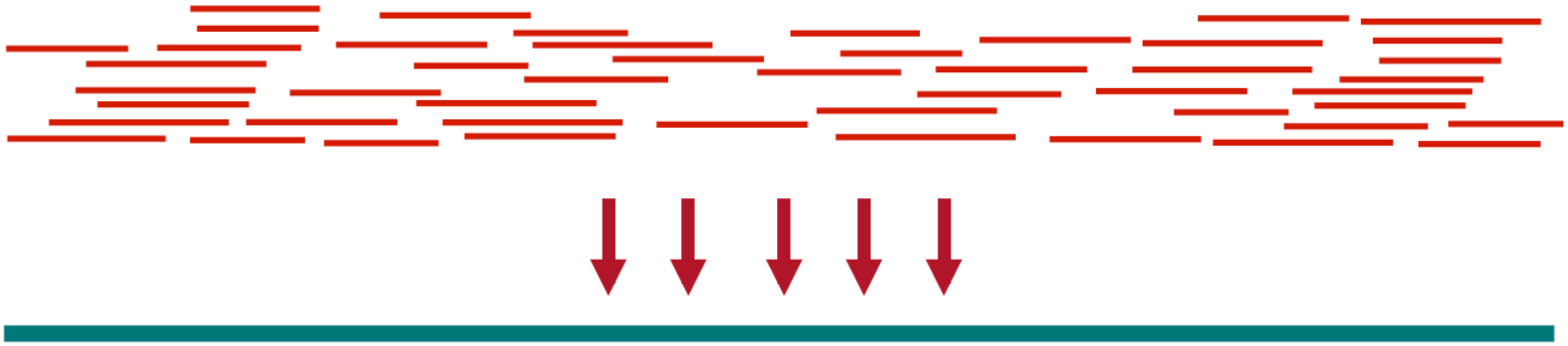
Suffix Array

Constant factor greatly reduced compared to suffix tree:
human genome index fits in ~12 GB instead of > 45 GB



El problema no es la secuencia, el problema es el procesamiento, la memoria, corrección de errores, etc.

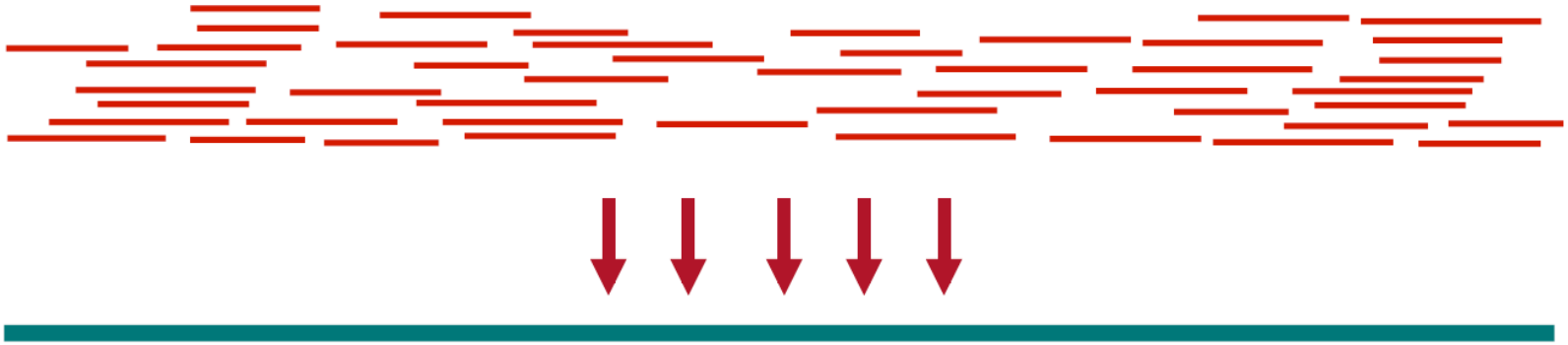
Read Mapping



```
                    CATCGACCGAGCGCGATGCTAGCTAGGTGATCGT......
                 TGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT...
              GCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT
           GTGCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATC
......AGGTGCATGCCGCATCGATCGAGCGCGATGCTAGCTAGCTGATCGT......
```

- Want ultra fast, highly similar alignment
- Detection of genomic variation

Read Mapping – Burrows-Wheeler Transform



```
                    CATCGACCGAGCGCGATGCTAGCTAGGTGATCGT......
                 TGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT...
              GCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT
           GTGCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATC
......AGGTGCATGCCGCATCGATCGAGCGCGATGCTAGCTAGCTGATCGT......
```

- Modern fast read aligners: BWT, Bowtie, SOAP
  - Based on ***Burrows-Wheeler transform***

Burrows-Wheeler Transformation
Example: mississippi

1. Append to the input string a special char, $, smaller than all alphabet.

mississippi$

Burrows-Wheeler Transformation (cnt'd)

Example: mississippi

2. Generate all rotations.

m	i	s	s	i	s	s	i	p	p	i	$
i	s	s	i	s	s	i	p	p	i	$	m
s	s	i	s	s	i	p	p	i	$	m	i
s	i	s	s	i	p	p	i	$	m	i	s
i	s	s	i	p	p	i	$	m	i	s	s
s	s	i	p	p	i	$	m	i	s	s	i
s	i	p	p	i	$	m	i	s	s	i	s
i	p	p	i	$	m	i	s	s	i	s	s
p	p	i	$	m	i	s	s	i	s	s	i
p	i	$	m	i	s	s	i	s	s	i	p
i	$	m	i	s	s	i	s	s	i	p	p
$	m	i	s	s	i	s	s	i	p	p	i

Burrows-Wheeler Transformation (cnt'd)
Example: mississippi

3. Sort rotations according to the alphabetica lorder.

$	m	i	s	s	i	s	s	i	p	p	i
i	$	m	i	s	s	i	s	s	i	p	p
i	p	p	i	$	m	i	s	s	i	s	s
i	s	s	i	p	p	i	$	m	i	s	s
i	s	s	i	s	s	i	p	p	i	$	m
m	i	s	s	i	s	s	i	p	p	i	$
p	i	$	m	i	s	s	i	s	s	i	p
p	p	i	$	m	i	s	s	i	s	s	i
s	i	p	p	i	$	m	i	s	s	i	s
s	i	s	s	i	p	p	i	$	m	i	s
s	s	i	p	p	i	$	m	i	s	s	i
s	s	i	s	s	i	p	p	i	$	m	i

Burrows-Wheeler Transformation (cnt'd)

Example: mississippi

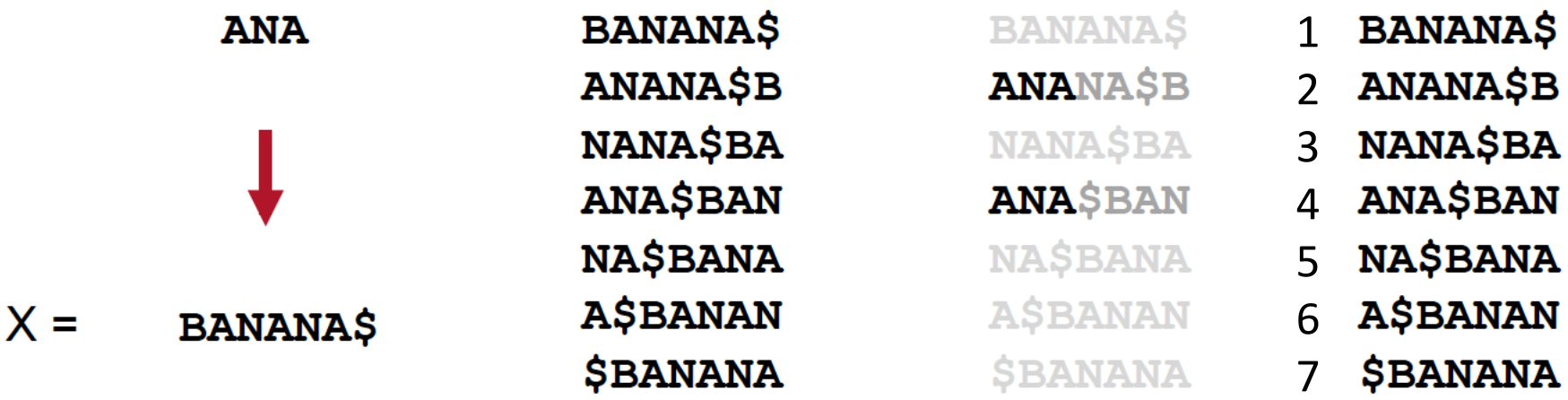
4. Output the last column.

$	m	i	s	s	i	s	s	i	p	p	i
i	$	m	i	s	s	i	s	s	i	p	p
i	p	p	i	$	m	i	s	s	i	s	s
i	s	s	i	p	p	i	$	m	i	s	s
i	s	s	i	s	s	i	p	p	i	$	m
m	i	s	s	i	s	s	i	p	p	i	$
p	i	$	m	i	s	s	i	s	s	i	p
p	p	i	$	m	i	s	s	i	s	s	i
s	i	p	p	i	$	m	i	s	s	i	s
s	i	s	s	i	p	p	i	$	m	i	s
s	s	i	p	p	i	$	m	i	s	s	i
s	s	i	s	s	i	p	p	i	$	m	i

Burrows-Wheeler Transformation (cnt'd)
Example: mississippi

ipssm$pissii

Burrows-Wheeler Transform



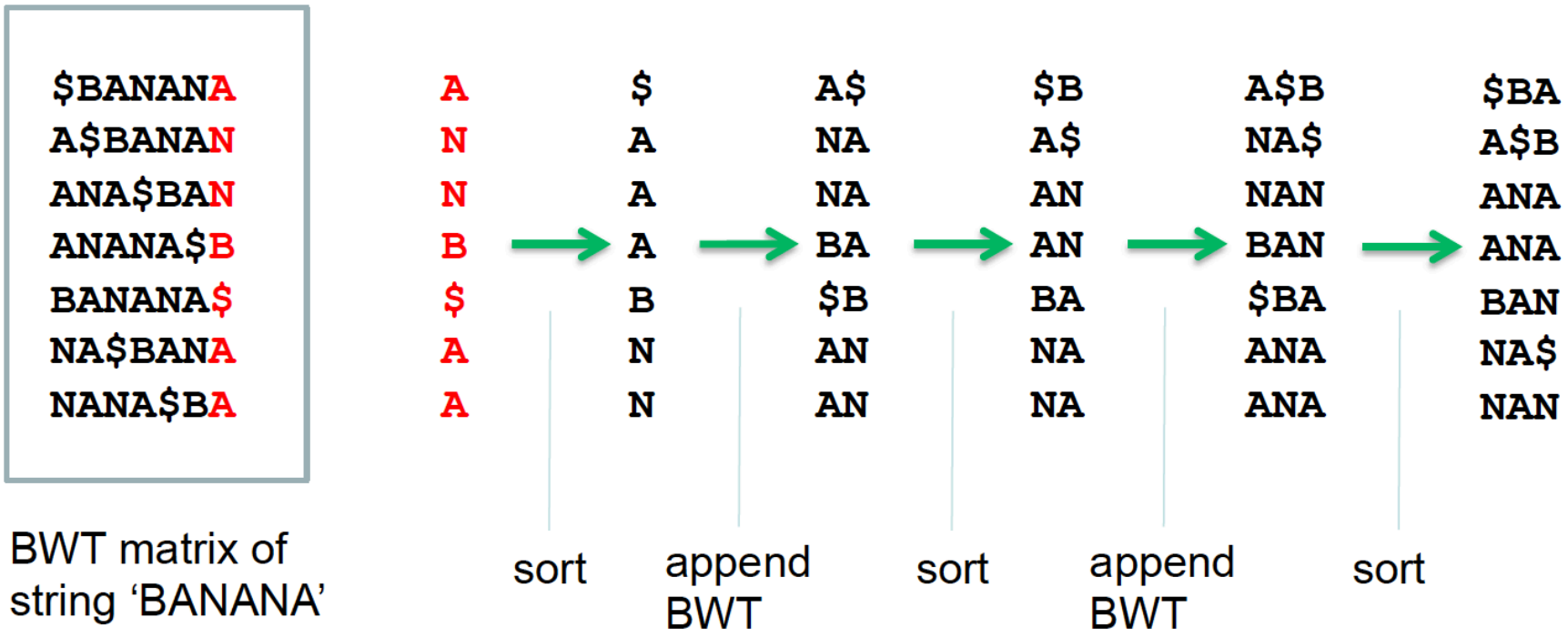
7 $BANANA
6 A$BANAN
4 ANA$BAN
2 ANANA$B
1 BANANA$
5 NA$BANA
3 NANA$BA

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

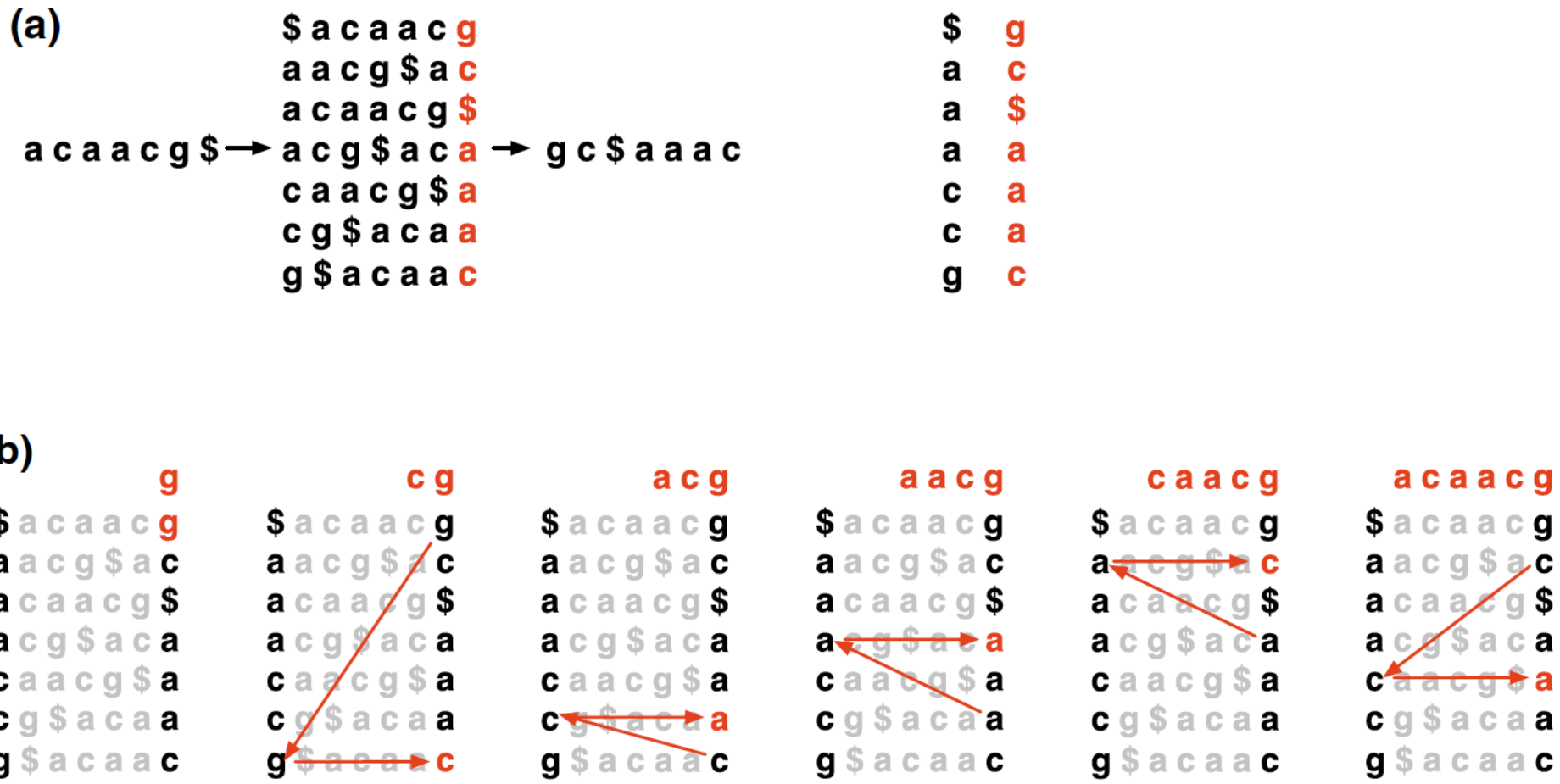
BWT matrix of string 'BANANA'

BWT(BANANA) = ANNB$AA

Reconstructing BANANA

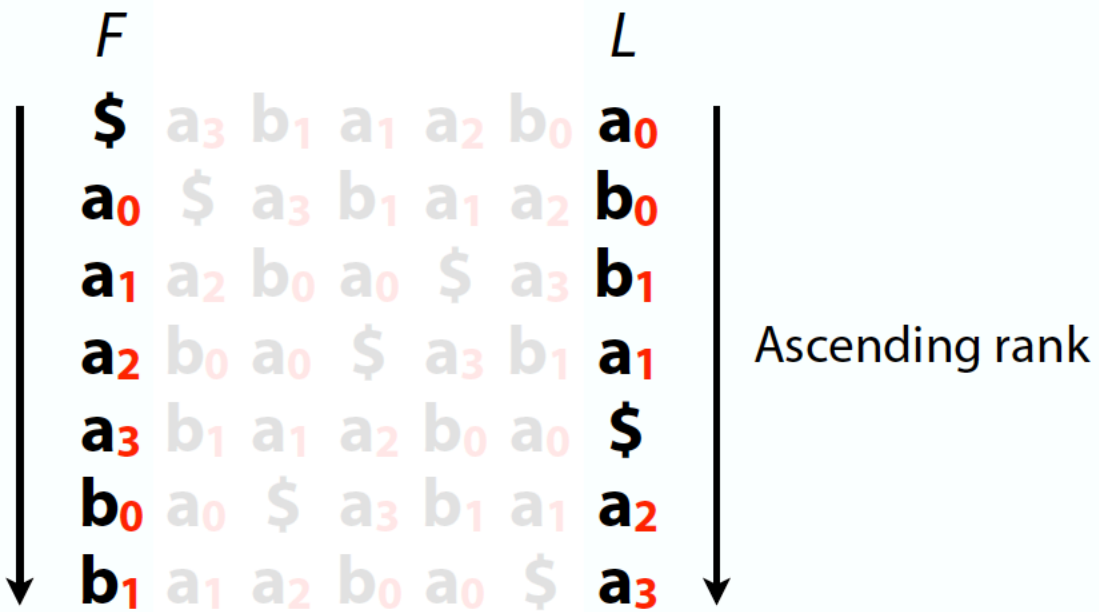


Burrows-Wheeler Transform



Burrows-Wheeler Transform: LF Mapping

BWM with B-ranking:



F now has very simple structure: a **$**, a block of **a**s *with ascending ranks*, a block of **b**s *with ascending ranks*

Burrows-Wheeler Transform

Say *T* has 300 **A**s, 400 **C**s, 250 **G**s and 700 **T**s and **$** < **A** < **C** < **G** < **T**

Which BWM row (0-based) begins with $\mathbf{G}_{100}$? (Ranks are B-ranks.)

Skip row starting with **$** (1 row)
Skip rows starting with **A** (300 rows)
Skip rows starting with **C** (400 rows)
Skip first 100 rows starting with **G** (100 rows)

Answer: row 1 + 300 + 400 + 100 = **row 801**

FM Index

FM Index: an index combining the BWT *with a few small auxiliary data structures*

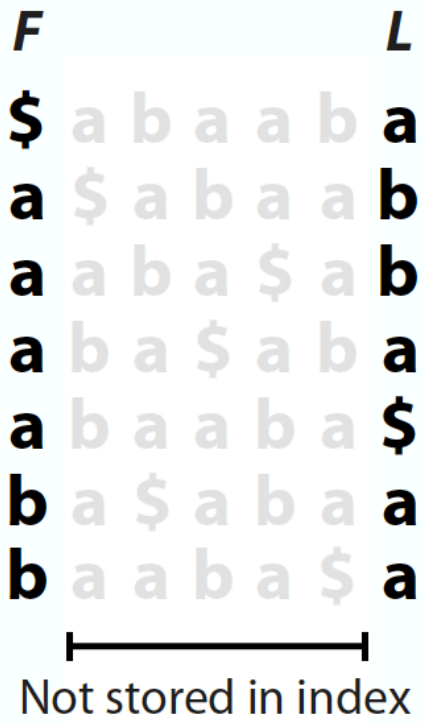
Core of index is ***F*** and ***L*** from BWM:

L is the same size as *T*

F can be represented as array of |Σ| integers

L is compressible (but even uncompressed, it's small compared to suffix array)

We're discarding *T*

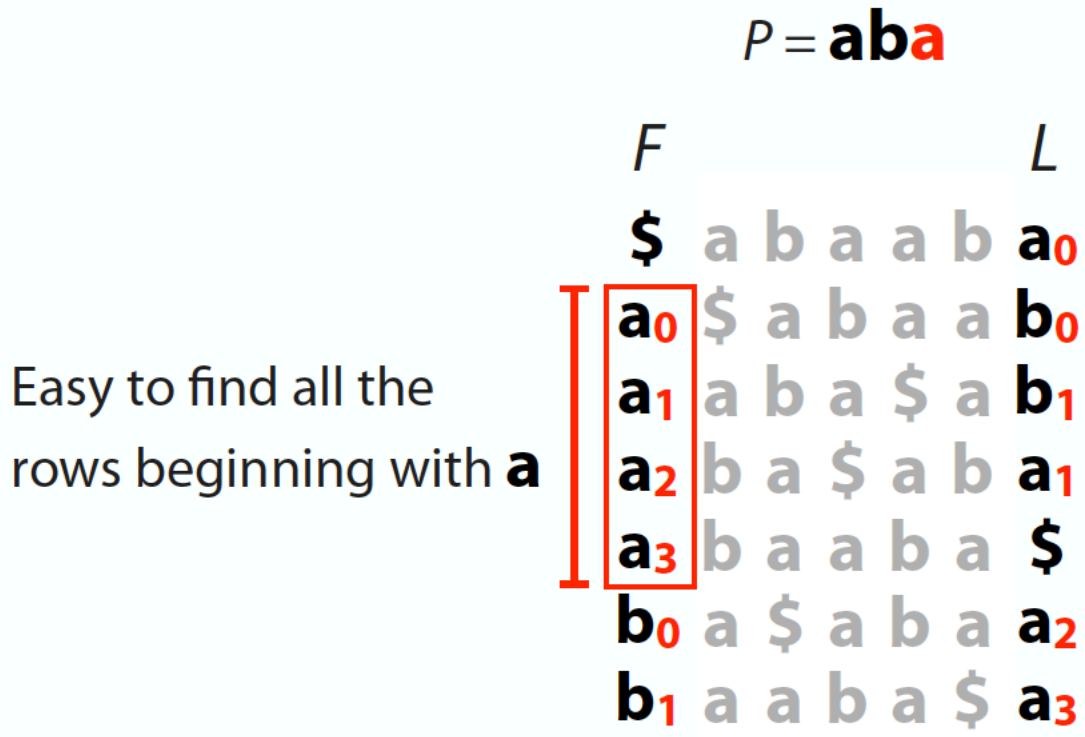


Paolo Ferragina, and Giovanni Manzini. "Opportunistic data structures with applications." *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 2000.

FM Index: querying

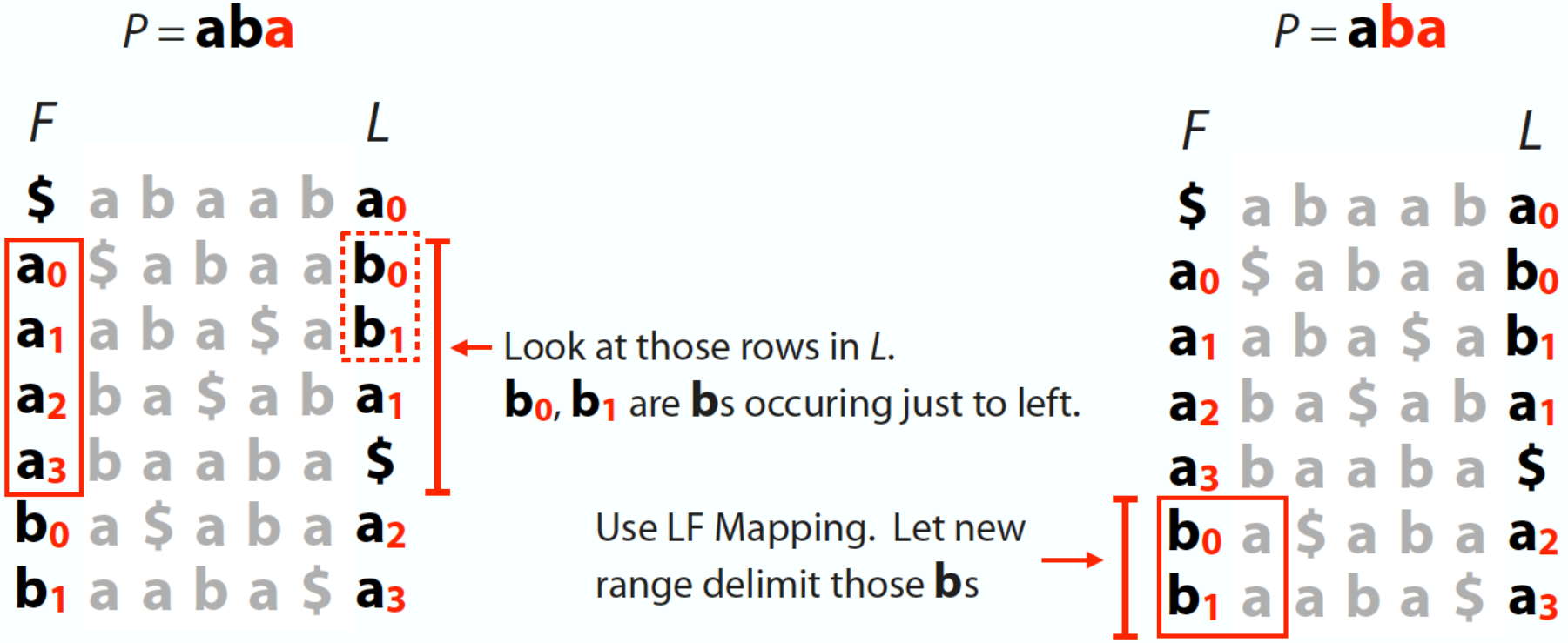
Look for range of rows of BWM(T) with *P* as prefix

Start with shortest suffix, then match successively longer suffixes



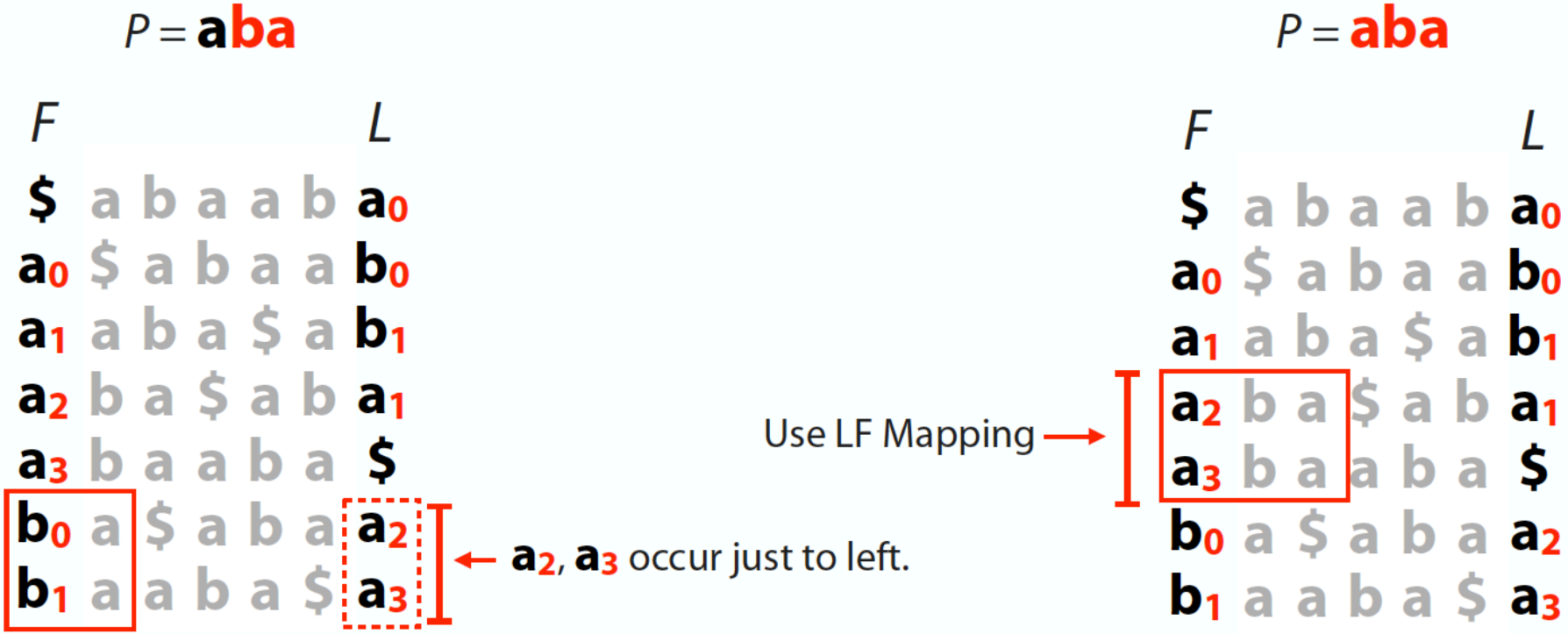
FM Index: querying

We have rows beginning with **a**, now we want rows beginning with **ba**



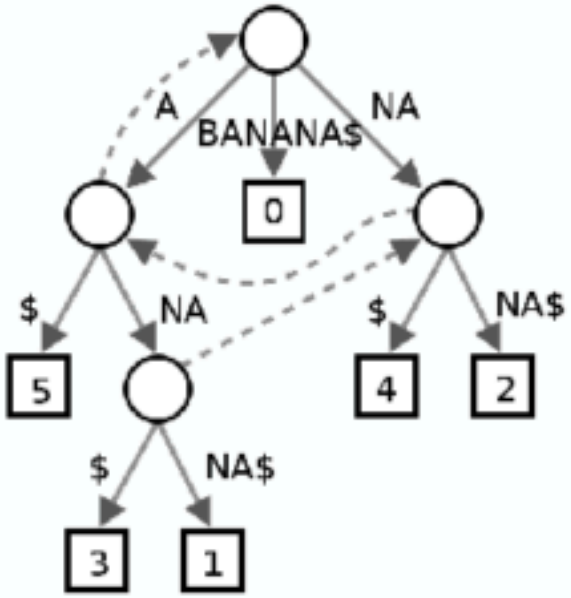
FM Index: querying

We have rows beginning with **ba**, now we seek rows beginning with **aba**



Now we have the rows with prefix **aba**

FM Index: small memory footprint



Suffix tree

≥ 45 GB

6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix array

≥ 12 GB

$ B A N A N A
A $ B A N A N
A N A $ B A N
A N A N A $ B
B A N A N A $
N A $ B A N A
N A N A $ B A

FM Index

~ 1.5 GB

Suffix index bounds

	Suffix tree	**Suffix array**	**FM Index**
Time: Does P occur?	$O(n)$	$O(n \log m)$	$O(n)$
Time: Count k occurrences of P	$O(n + k)$	$O(n \log m)$	$O(n)$
Time: Report k locations of P	$O(n + k)$	$O(n \log m + k)$	$O(n + k)$
Space	$O(m)$	$O(m)$	$O(m)$
Needs T?	*yes*	*yes*	*no*
Bytes per input character	>15	~4	~0.5

$m = |T|, n = |P|, k =$ # occurrences of P in T

References

- https://www.cs.colostate.edu/~cs425/spring17/home_progress.php
- http://www.langmead-lab.org/teaching-materials/
- https://web.stanford.edu/class/cs262/cgi-bin/index.php