CS481/CS583: Bioinformatics Algorithms

Can Alkan
EA509
calkan@cs.bilkent.edu.tr

http://www.cs.bilkent.edu.tr/~calkan/teaching/cs481/

Burrows-Wheeler Transformation

- Originally developed for data compression
- Reordering text -> Better locality = better compression
  - Used in bzip2
- Additional data structures for sequence search
  - Ferragina-Manzini index
  - “Summarized” suffix array

Burrows-Wheeler Transformation

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

mississippi$

Burrows-Wheeler Transformation (cnt'd)

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)

3. Sort rotations according to the alphabetical order.

$	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)

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)

mississippi$

↓

ipssm$pissii

Why does BWT boost locality?

mississippi$

ipssm$pissii

Doesn't really seem to help

Why does BWT boost locality?

sorted by right-context:

consists of everything that
comes after it with
a wrap around

final char (L)	sorted rotations
a	n to decompress. It achieves compression
o	n to perform only comparisons to a depth
o	n transformation} This section describes
o	n transformation} We use the example and
o	n treats the right-hand side as the most
a	n tree for each 16 kbyte input block, enc
a	n tree in the output stream, then encodes
i	n turn, set $L[i]$ to be the
i	n turn, set $R[i]$ to the
o	n unusual data. Like the algorithm of Man
a	n use a single set of probabilities table
e	n using the positions of the suffixes in
i	n value at a given point in the vector $R
e	n we present modifications that improve t
e	n when the block size is quite large. Ho
i	n which codes that have not been seen in
i	n with ch appear in the {\em same order
i	n with ch. In our exam
o	n with Huffman or arithmetic coding. Bri
o	n with figures given by Bell~\cite{bell}.

Figure 1: Example of sorted rotations. Twenty consecutive rotations from the sorted list of rotations of a version of this paper are shown, together with the final character of each rotation.

Burrows and Wheeler, 1994

BWT – right context

T = a b a a b a

BWT

```
$ a b a a b a
a $ a b a a b
a a b a $ a b
a b a $ a b a
a b a a b a $
b a $ a b a a
b a a b a $ a
```

Right-context: aba$ab

Slide from Ben Langmead

BWT – right context

T = a b a a b a

BWT

```
$ a b a a b a        a   $ a b a a b
a $ a b a a b        b   a $ a b a a
a a b a $ a b        b   a a b a $ a
a b a $ a b a        a   a b a $ a b
a b a a b a $        $   a b a a b a
b a $ a b a a        a   b a $ a b a
b a a b a $ a        a   b a a b a $
```

Right-context

Slide from Ben Langmead

BWT – alternative construction

T = a b a a b a

BWT	Suffix Array	
$ a b a a b a	6	$
a $ a b a a b	5	a $
a a b a $ a b	2	a a b a $
a b a $ a b a	3	a b a $
a b a a b a $	0	a b a a b a $
b a $ a b a a	4	b a $
b a a b a $ a	1	b a a b a $

$$BWT[i] = \begin{cases} T[SA[i] - 1], & \text{if } SA[i]>0 \\ \$, & \text{if } SA[i]=0 \end{cases}$$

BWT = characters just to the left of characters in SA

Slide from Ben Langmead

L to F map

First column: F

Last column: L

Let's make an L to F map.

Observation: The n^{th} i in L is the n^{th} i in F.

$	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

L to F map

Store/compute a two dimensional Occ(*j*,'c') table of the number of occurrences of char 'c' up to position *j* (inclusive).

and one dimensional Cnt('c') and Rank('c') tables

	$	i	m	p	s
i	0	1	0	0	0
p	0	1	0	1	0
s	0	1	0	1	1
s	0	1	0	1	2
m	0	1	1	1	2
$	1	1	1	1	2
p	1	1	1	2	2
i	1	2	1	2	2
s	1	2	1	2	3
s	1	2	1	2	4
i	1	3	1	2	4
i	1	4	1	2	4

Occ(*j*,'c')

Cnt('c')

$	i	m	p	s
1	4	1	2	4

Rank('c')

$	i	m	p	s
12	2	1	9	3

L to F map

[Cnt(‘$’) +
Cnt(‘i’) +
Cnt(‘m’) +
Cnt(‘p’) = 8]
+ [Occ(9, ‘s’)= 3]
= 11

before ‘s’

‘s’ section

1	$	m	i	s	s	i	s	s	i	p	p	i
2	i	$	m	i	s	s	i	s	s	i	p	p
3	i	p	p	i	$	m	i	s	s	i	s	s
4	i	s	s	i	p	p	i	$	m	i	s	s
5	i	s	s	i	s	s	i	p	p	i	$	m
6	m	i	s	s	i	s	s	i	p	p	i	$
7	p	i	$	m	i	s	s	i	s	s	i	p
8	p	p	i	$	m	i	s	s	i	s	s	i
9	s	i	p	p	i	$	m	i	s	s	i	s
10	s	i	s	s	i	p	p	i	$	m	i	s
11	s	s	i	p	p	i	$	m	i	s	s	i
12	s	s	i	s	s	i	p	p	i	$	m	i

Cnt(‘c’)

$	i	m	p	s
1	4	1	2	4

L to F map

(1) i
(2) p
(7) p
(8) i
(3) s
(9) s
(11) i
(4) s
(10) s
(12) i
(5) m
(6) $

1	$	m	i	s	s	i	s	s	i	p	p	i
2	i	$	m	i	s	s	i	s	s	i	p	p
3	i	p	p	i	$	m	i	s	s	i	s	s
4	i	s	s	i	p	p	i	$	m	i	s	s
5	i	s	s	i	s	s	i	p	p	i	$	m
6	m	i	s	s	i	s	s	i	p	p	i	$
7	p	i	$	m	i	s	s	i	s	s	i	p
8	p	p	i	$	m	i	s	s	i	s	s	i
9	s	i	p	p	i	$	m	i	s	s	i	s
10	s	i	s	s	i	p	p	i	$	m	i	s
11	s	s	i	p	p	i	$	m	i	s	s	i
12	s	s	i	s	s	i	p	p	i	$	m	i

Search with BWT-FM: L to F map

Original sequence

gca

	SA		BWT
		$agcagcagact	t
1	9	act$agcagcag	g
2	7	agact$agcagc	c
3	4	agcagact$agc	c
4	1	agcagcagact$	$
5	6	cagact$agcag	g
6	3	cagcagact$ag	g
7	10	ct$agcagcaga	a
8	8	gact$agcagca	a
9	5	gcagact$agca	a
10	2	gcagcagact$a	a
11	11	t$agcagcagac	c

Search with BWT-FM: FM-index

Auxiliary data structures for efficient pattern matching: how to find the corresponding chars in the first column efficiently, in terms of both time and space.

	a	c	g	t
rank	1	5	8	11

Original sequence

	SA		BWT	a	c	g	t
		$agcagcagact	t	0	0	0	1
1	9	act$agcagcag	g	0	0	1	1
2	7	agact$agcagc	c	0	1	1	1
3	4	agcagact$agc	c	0	2	1	1
4	1	agcagcagact$	$	0	2	1	1
5	6	cagact$agcag	g	0	2	2	1
6	3	cagcagact$ag	g	0	2	3	1
7	10	ct$agcagcaga	a	1	2	3	1
8	8	gact$agcagca	a	2	2	3	1
9	5	gcagact$agca	a	3	2	3	1
10	2	gcagcagact$a	a	4	2	3	1
11	11	t$agcagcagac	c	4	3	3	1

FM indices

Search with BWT-FM: FM-index

Auxiliary data structures for efficient pattern matching: how to find the corresponding chars in the first column efficiently, in terms of both time and space.

	a	c	g	t
rank	1	5	8	11

Original sequence

gca

	SA		BWT	a	c	g	t
		$agcagcagac**t**	t	0	0	0	1
1	9	**a**ct$agcagca**g**	g	0	0	1	1
2	7	**a**gact$agcag**c**	c	0	1	1	1
3	4	**a**gcagact$ag**c**	c	0	2	1	1
4	1	**a**gcagcagact**$**	$	0	2	1	1
5	6	**c**agact$agca**g**	g	0	2	2	1
6	3	**c**agcagact$a**g**	g	0	2	3	1
7	10	**c**t$agcagcag**a**	a	1	2	3	1
8	8	**g**act$agcagc**a**	a	2	2	3	1
9	5	**g**cagact$agc**a**	a	3	2	3	1
10	2	**g**cagcagact$**a**	a	4	2	3	1
11	11	**t**$agcagcaga**c**	c	4	3	3	1

FM indices

Next block:
From 1 + 0 = 1
to 1 + (4-1) = 4

Search with BWT-FM: FM-index

Auxiliary data structures for efficient pattern matching: how to find the corresponding chars in the first column efficiently, in terms of both time and space.

	a	c	g	T
rank	1	5	8	11

Original sequence

gca

	SA		BWT	a	c	g	t
		$agcagcagac**t**	t	0	0	0	1
1	9	**a**ct$agcagca**g**	g	0	0	1	1
2	7	**a**gact$agcag**c**	c	0	1	1	1
3	4	**a**gcagact$ag**c**	c	0	2	1	1
4	1	**a**gcagcagact**$**	$	0	2	1	1
5	6	**c**agact$agca**g**	g	0	2	2	1
6	3	**c**agcagact$a**g**	g	0	2	3	1
7	10	**c**t$agcagcag**a**	a	1	2	3	1
8	8	**g**act$agcagc**a**	a	2	2	3	1
9	5	**g**cagact$agc**a**	a	3	2	3	1
10	2	**g**cagcagact$**a**	a	4	2	3	1
11	11	**t**$agcagcaga**c**	c	4	3	3	1

FM indices

Next block:
From 5 + 0 = 5
to 5 + (2-1) = 6

Search with BWT-FM: FM-index

Auxiliary data structures for efficient pattern matching: how to find the corresponding chars in the first column efficiently, in terms of both time and space.

	a	c	g	T
rank	1	5	8	11

Original sequence

gca

	SA		BWT	a	c	g	t
		$agcagcagac**t**	t	0	0	0	1
1	9	**a**ct$agcagca**g**	g	0	0	1	1
2	7	**a**gact$agcag**c**	c	0	1	1	1
3	4	**a**gcagact$ag**c**	c	0	2	1	1
4	1	**a**gcagcagact**$**	$	0	2	1	1
5	6	**c**agact$agca**g**	g	0	2	2	1
6	3	**c**agcagact$a**g**	g	0	2	3	1
7	10	**c**t$agcagcag**a**	a	1	2	3	1
8	8	**g**act$agcagc**a**	a	2	2	3	1
9	5	**g**cagact$agc**a**	a	3	2	3	1
10	2	**g**cagcagact$**a**	a	4	2	3	1
11	11	**t**$agcagcaga**c**	c	4	3	3	1

FM indices

Next block:
From 8 - 1 = 9
to 8 + (3-1) = 10

FM-index issues

Scanning is slow

$ a b a a b a
a $ a b a a b
a a b a $ a b
a b a $ a b a
a b a a b a $
b a $ a b a a
b a a b a $ a

O(m) scan

Occ table is big

a	b
1	0
1	1
1	2
2	2
2	2
3	2
4	2

O(n) space for Suffix array

6
5
2
3
0
4
1

Slide from Ben Langmead

Sparse Occ table

Pre-calculate only some rows; for example, every 5th row

$ a b a a b a
a $ a b a a b
a a b a $ a b
a b a $ a b a
a b a a b a $
b a $ a b a a
b a a b a $ a

a	b	
1	0	checkpoint 1
3	2	checkpoint 2

Slide from Ben Langmead

Sparse Occ table

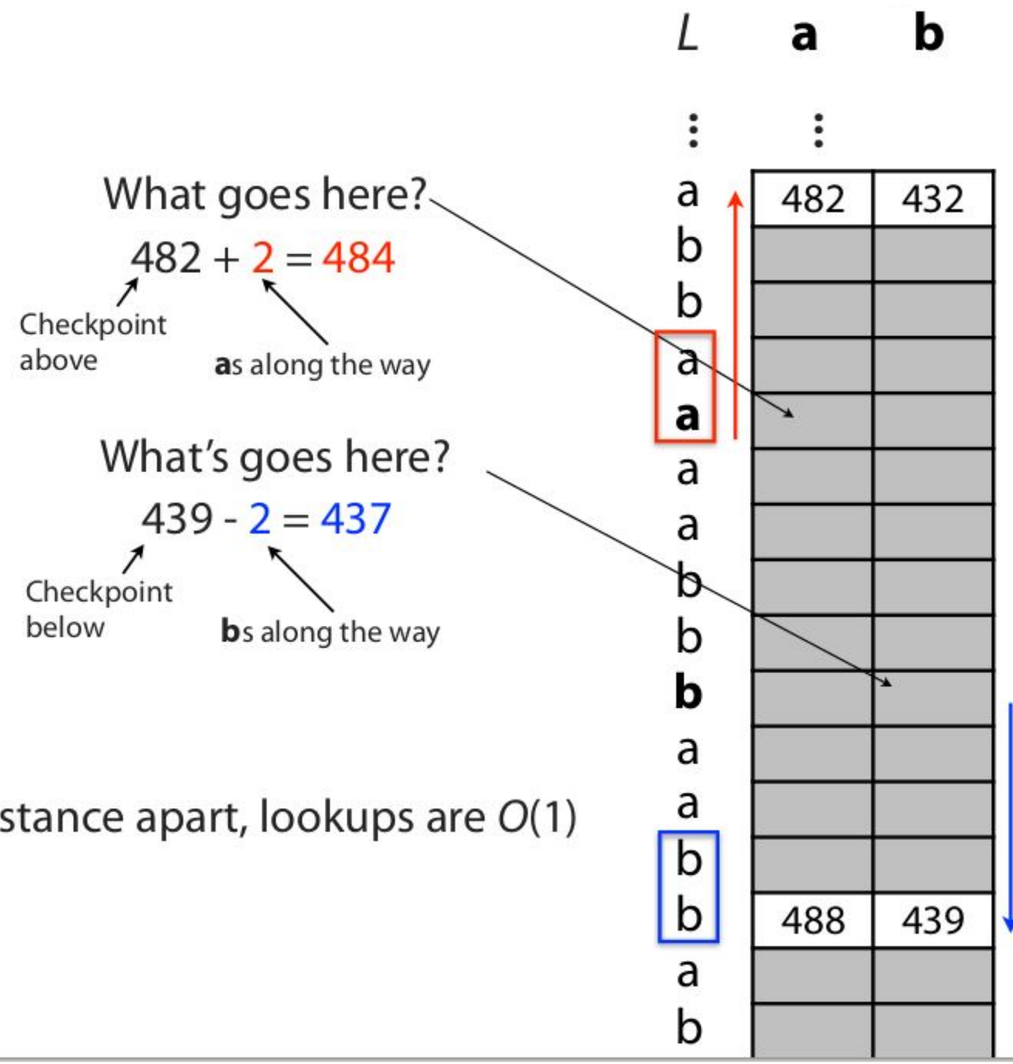
Pre-calculate only some rows; for example, every 5th row

$ a b a a b a
a $ a b a a b
a a b a $ a b
a b a $ a b a
a b a a b a $
b a $ a b a a
b a a b a $ a

F	L	a	b	
$	a	1	0	successful lookup
a	b			
a	b			
a	a			
a	$			failed lookup; but there is one close
b	a	3	2	
b	a			

Slide from Ben Langmead

Sparse Occ table

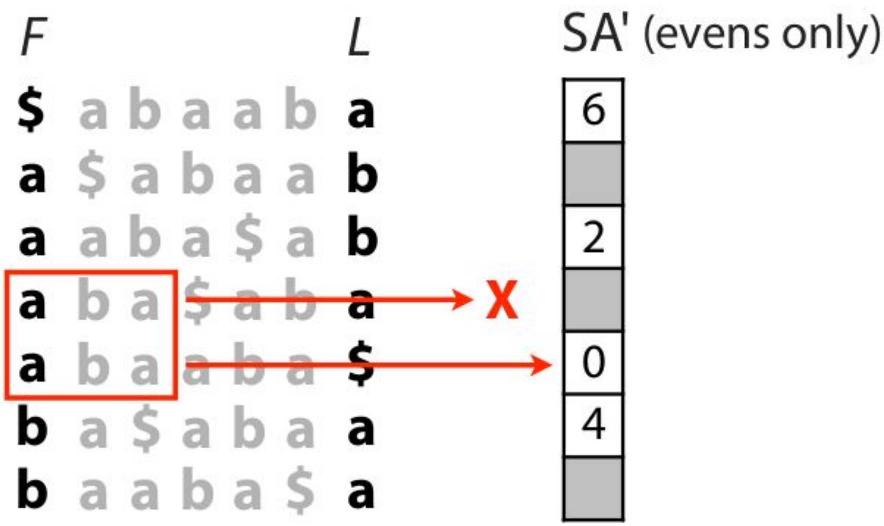


If checkpoints are $O(1)$ distance apart, lookups are $O(1)$

Slide from Ben Langmead

Sparse Suffix Array

Idea: store some suffix array elements, but not all



Lookup for row 4 succeeds

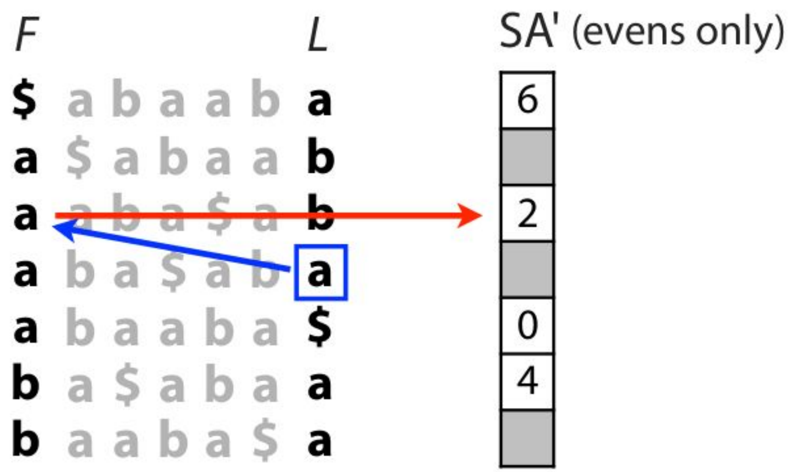
Lookup for row 3 fails - SA entry was discarded

Slide from Ben Langmead

Sparse Suffix Array

LF Mapping tells us that "**a**" at the end of row 3 corresponds to...

... "**a**" at the beginning of row 2



Row 2 of suffix array = 2

Missing value in row 3 = 2 (row 2's SA val) + 1 (# steps to row 2) = **3**

If saved SA values are O(1) positions apart in *T*, resolving offset is O(1) time

Slide from Ben Langmead

FM Index

a: fraction of Suffix Array rows we keep
b: fraction of Occ rows we keep
Components of FM Index:

First column (F):	~ | ∑ | integers
Last column (L):	m characters
SA sample:	m · a integers, a is fraction of SA elements kept
Checkpoints:	m · | ∑ | · b integers, b is fraction of occ kept

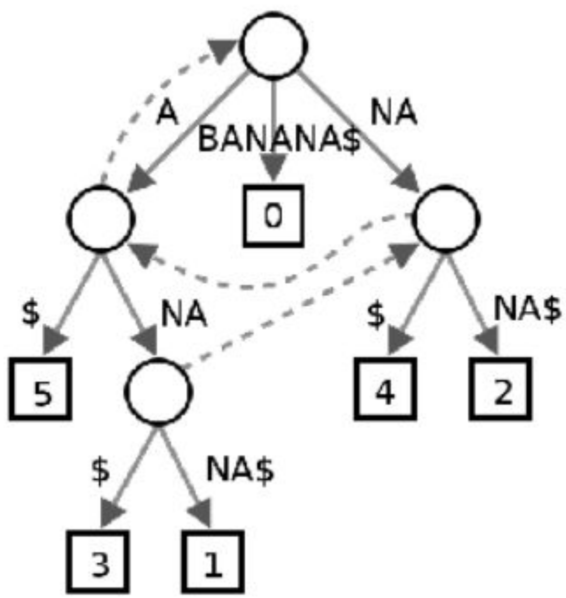
For DNA alphabet (2 bits / nt), T = human genome, a = 1/32, b = 1/128 :

First column (F):	16 bytes
Last column (L):	2 bits * 3 billion chars = 750 MB
SA sample:	3 billion chars * 4 bytes / 32 = ~ 400 MB
Checkpoints:	3 billion * 4 alphabet chars * 4 bytes / 128 = ~ 400 MB

Total ≈ 1.5 GB
~0.5 bytes per input char

Slide from Ben Langmead

FM Index 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

Slide from Ben Langmead

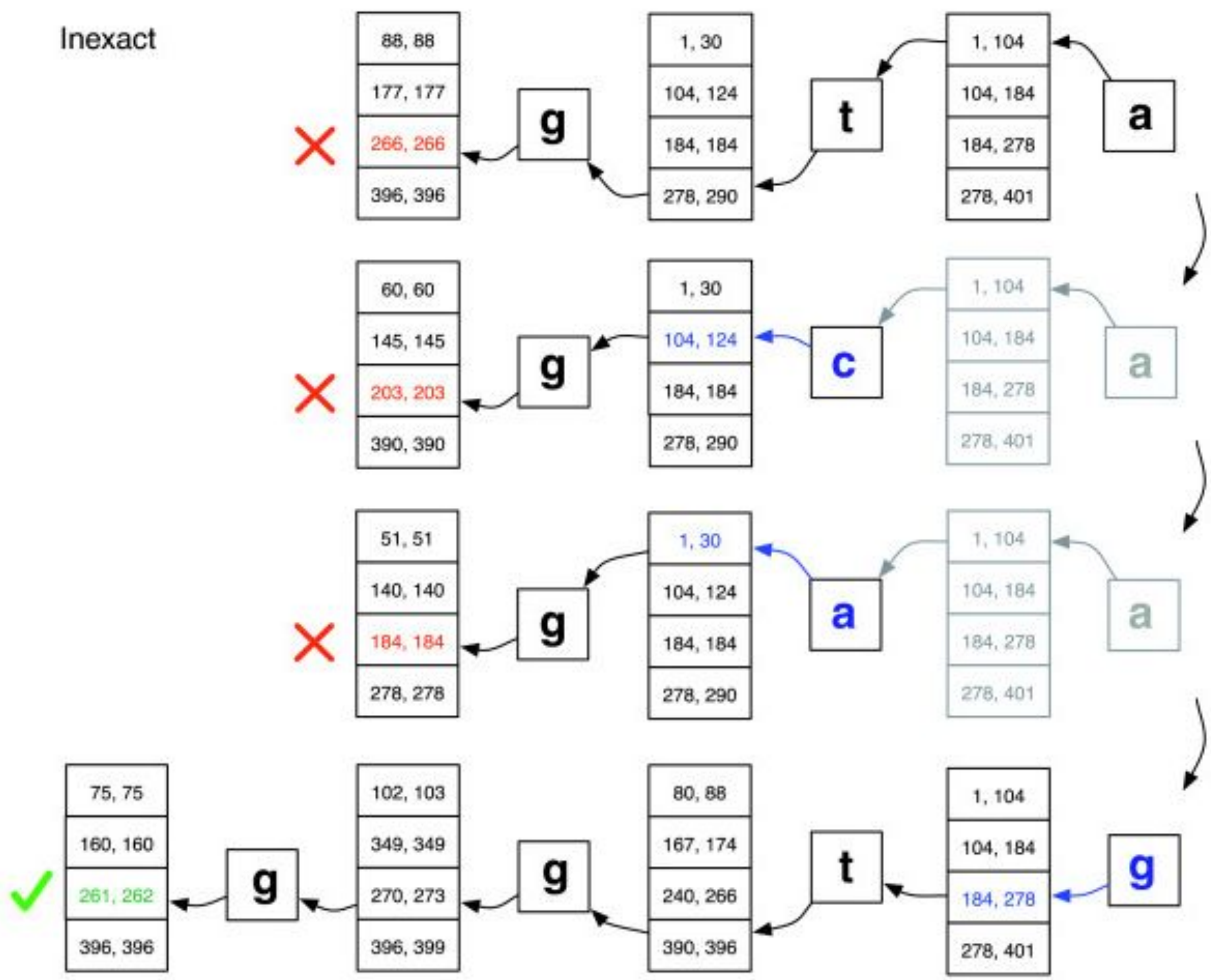
Index bounds

	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

Slide from Ben Langmead

Inexact match



Videos

- BWT
  - https://www.youtube.com/watch?v=4n7NPk5lwbI
- FM-index
  - https://www.youtube.com/watch?v=kvVGj5V65io