Efficient sequence comparison, part 3

Benedict Paten (bpaten@ucsc.edu)

Acknowledgements

These slides are mostly from myself, Adam Novak (PhD and engineer in the Genomics Institute) and the internet

Adam Novak

Overview/objectives of all four parts

Part 1:

1. The essential genomics background (Recap)
2. Strings (Recap)
3. Exact string comparison and Boyer Moore

|P|log(n)

Part 2:

Substring indexes 1 - sub-linear gapless string search

Part 3:

4. Substring indexes 2 - compressed sub-linear gapless string search

Borrows-Wheeler-Transform

Part 4:

5. Gapped sequence comparison and dynamic programming methods
6. MinHash, Minimizers and sketch methods - constant time string search

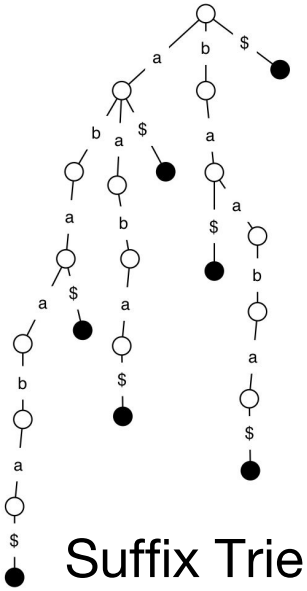
- 1. Section synopsis:
  - Substring indexes 2 - compressed sub-linear gapless string search*:*
    - The BWT
    - Compression with the BWT
    - Backward search: The BWT with FM Index

  - Reading: Ferragina and Manzini, Opportunistic Data Structures with Applications
    https://people.unipmn.it/manzini/papers/focs00draft.pdf

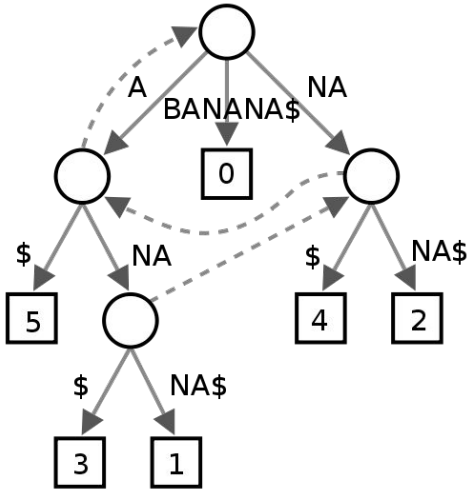
The evolution of suffix based search..

At the end of the last lecture we saw how more and more memory efficient data-structures could enable O(|P|) search

We'll now show this can be pushed further with the FM-index



Suffix Trie



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

BWT/FM Index

Computer scientists always want to do better!

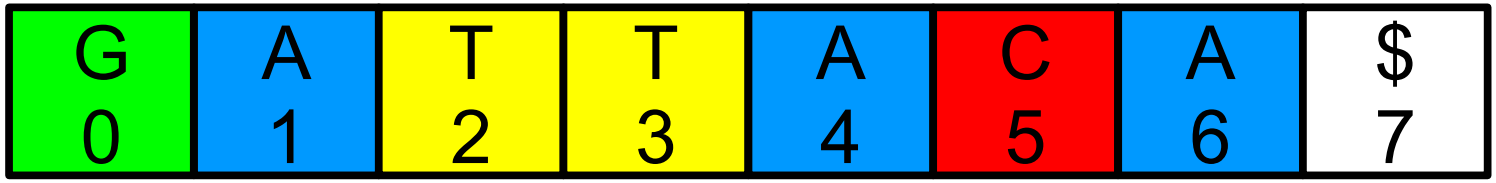
An(other) Entropy-Bounded Compressed Suffix Tree

Johannes Fischer[1], Veli Mäkinen[2] *, and Gonzalo Navarro[1] **

[1] Dept. of Computer Science, Univ. of Chile. {jfischer|gnavarro}@dcc.uchile.cl
[2] Dept. of Computer Science, Univ. of Helsinki, Finland. vmakinen@cs.helsinki.fi

there is a maximum amount of complexity that we can store using the idea of entropy and the amount of bits of information that they each require

Abstract. Suffix trees are among the most important data structures in stringology, with myriads of applications. Their main problem is space usage, which has triggered much research striving for compressed representations that are still functional. We present a novel compressed suffix tree. Compared to the existing ones, ours is the first achieving at the same time sublogarithmic complexity for the operations, and space usage which goes to zero as the entropy of the text does. Our development contains several novel ideas, such as compressing the longest common prefix information, and totally getting rid of the suffix tree topology, expressing all the suffix tree operations using range minimum queries and a new primitive called next/previous smaller value in a sequence.



String S

Rotate

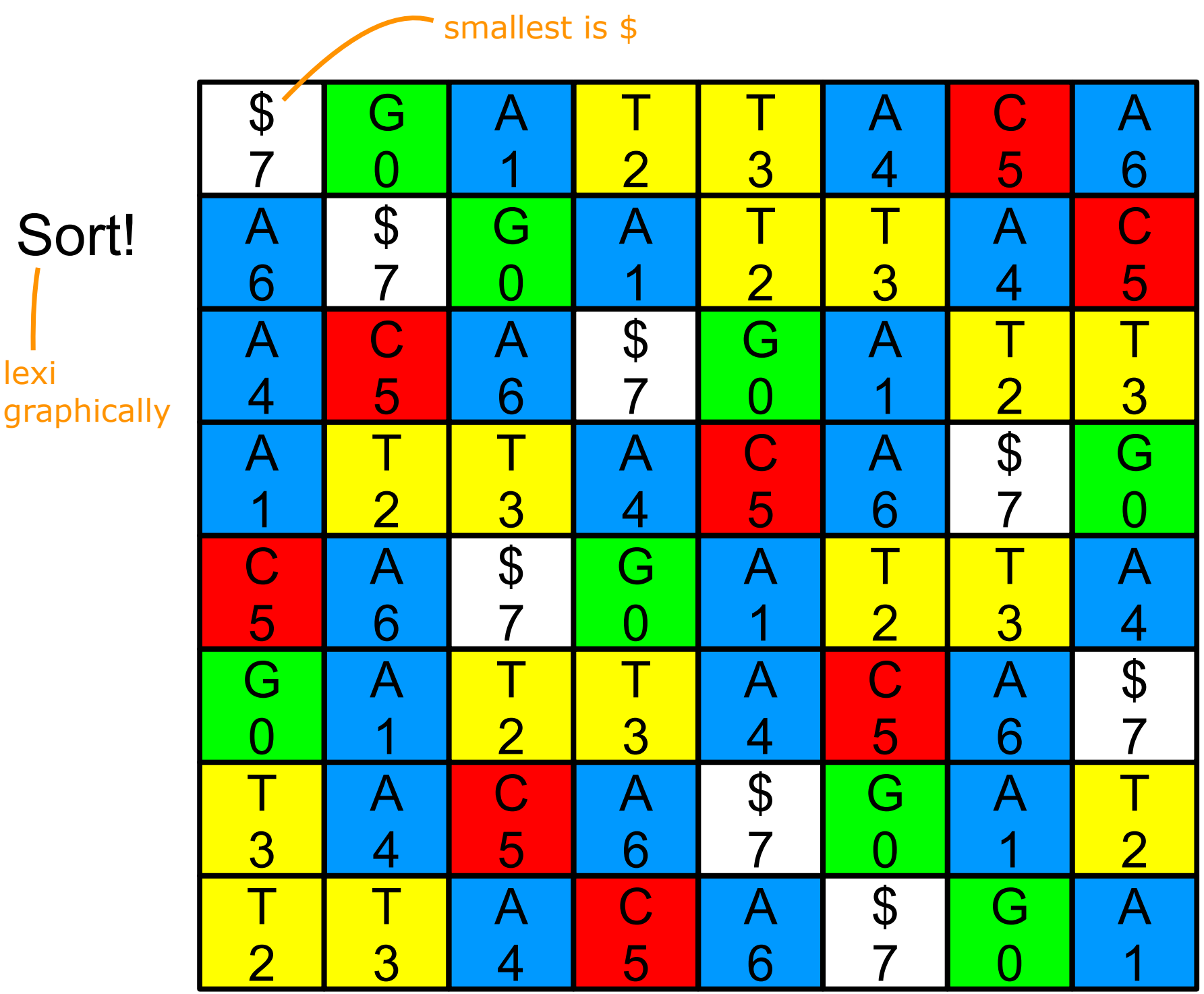
cyclic rotations of original string

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

Rotate

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

Like writing out a DNA string *S* around a cylinder |*S$*| times and then rotating each successively by one more character

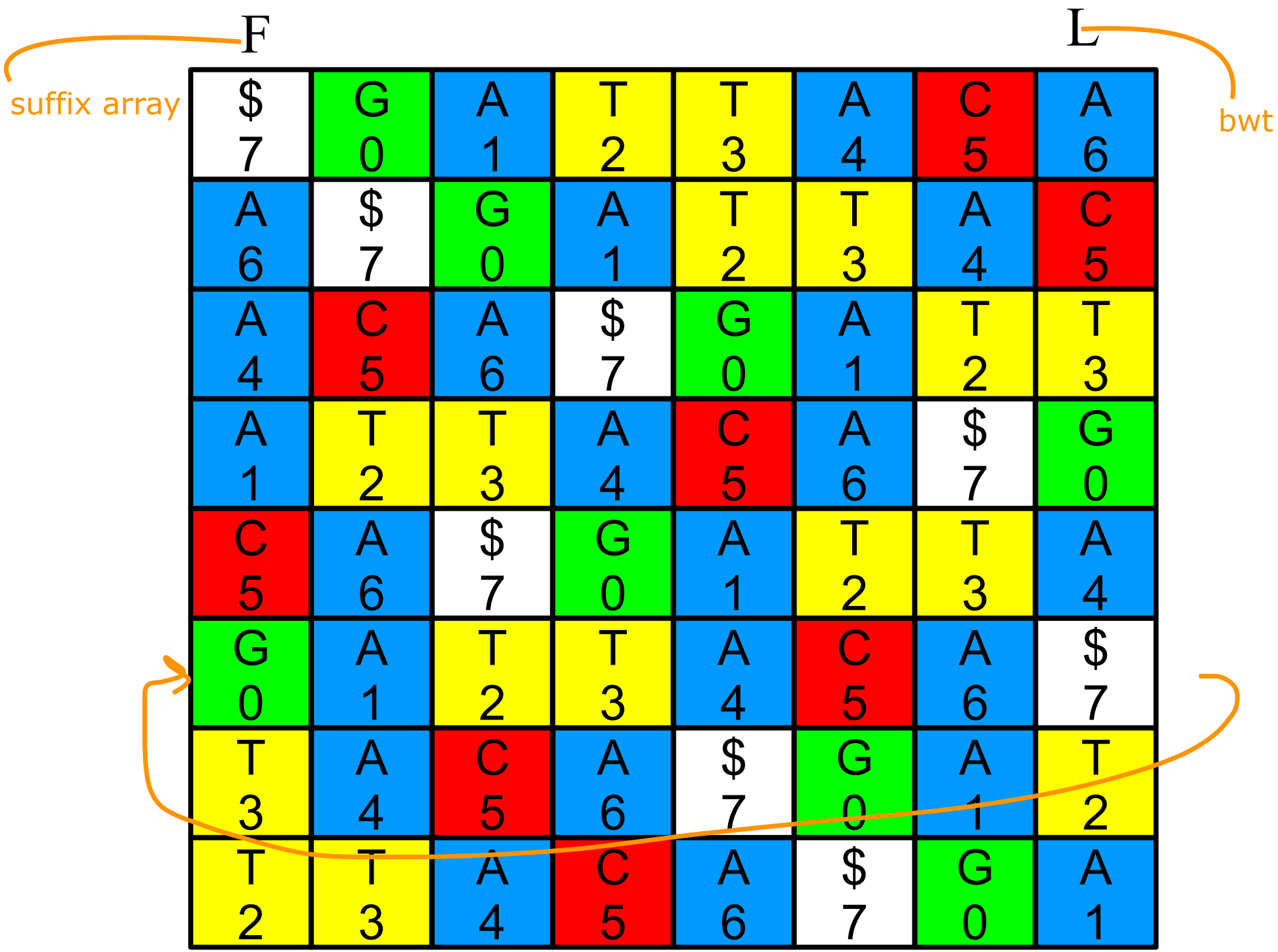

smallest is $
Sort!
lexi
graphically
$ 7 | G 0 | A 1 | T 2 | T 3 | A 4 | C 5 | A 6
A 6 | $ 7 | G 0 | A 1 | T 2 | T 3 | A 4 | C 5
A 4 | C 5 | A 6 | $ 7 | G 0 | A 1 | T 2 | T 3
A 1 | T 2 | T 3 | A 4 | C 5 | A 6 | $ 7 | G 0
C 5 | A 6 | $ 7 | G 0 | A 1 | T 2 | T 3 | A 4
G 0 | A 1 | T 2 | T 3 | A 4 | C 5 | A 6 | $ 7
T 3 | A 4 | C 5 | A 6 | $ 7 | G 0 | A 1 | T 2
T 2 | T 3 | A 4 | C 5 | A 6 | $ 7 | G 0 | A 1

Column of integers is suffix array


$ 7
G 0
A 1
T 2
T 3
A 4
C 5
A
A 6
$ 7
G 0
A 1
T 2
T 3
A 4
C
A 4
C 5
A 6
$ 7
G 0
A 1
T 2
T
A 1
T 2
T 3
A 4
C 5
A 6
$ 7
G
C 5
A 6
$ 7
G 0
A 1
T 2
T 3
A
G 0
A 1
T 2
T 3
A 4
C 5
A 6
$
T 3
A 4
C 5
A 6
$ 7
G 0
A 1
T
T 2
T 3
A 4
C 5
A 6
$ 7
G 0
A
BWT

It's All Connected

- BWT: Sort all rotations, take last character
- Suffix Array: Sort all suffixes, take character index
- The BWT character is therefore the character before (in the cyclic permutation sense) the suffix array index.


F
L
suffix array
bwt
$ 7
G 0
A 1
T 2
T 3
A 4
C 5
A 6
A 6
$ 7
G 0
A 1
T 2
T 3
A 4
C 5
A 4
C 5
A 6
$ 7
G 0
A 1
T 2
T 3
A 1
T 2
T 3
A 4
C 5
A 6
$ 7
G 0
C 5
A 6
$ 7
G 0
A 1
T 2
T 3
A 4
G 0
A 1
T 2
T 3
A 4
C 5
A 6
$ 7
T 3
A 4
C 5
A 6
$ 7
G 0
A 1
T 2
T 2
T 3
A 4
C 5
A 6
$ 7
G 0
A 1

BWT Compression

- DNA sequences have much lower entropy than entirely random sequences. — can be modeled using hid markov models
- This is reflected in the conditional probability of seeing a particular base at a point in a sequence given the previous few characters in the sequence - some characters are much more likely than others. ← HMM
- Clearly the first column of the cyclic rotations of S has very low entropy - much lower than S.
- Given the dependencies between adjacent characters, it follows that BWT itself generally has lower entropy than S.
- Using run-length encoding/move-to-front encoding the BWT can therefore be compressed better than S. This is how bzip works.

suffix array

String

— compression in DNA strings = 3x more

uses BWT to encode into zipped compressed format

Exercise 1 (10 mins)

Build the BWT.

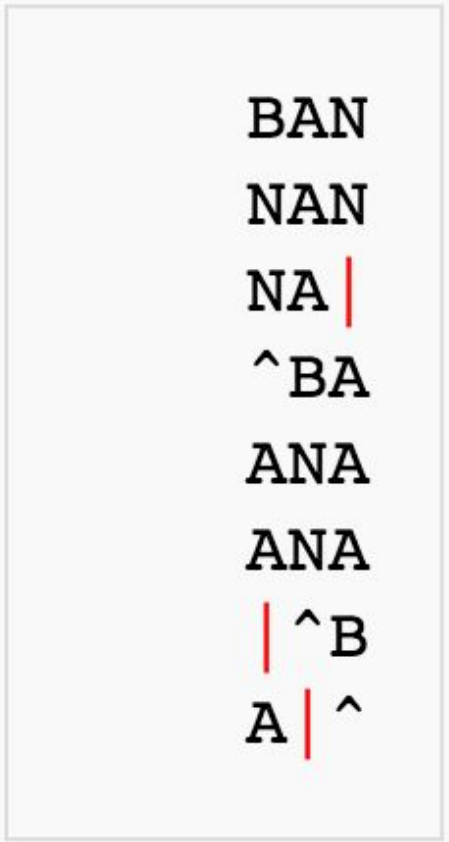
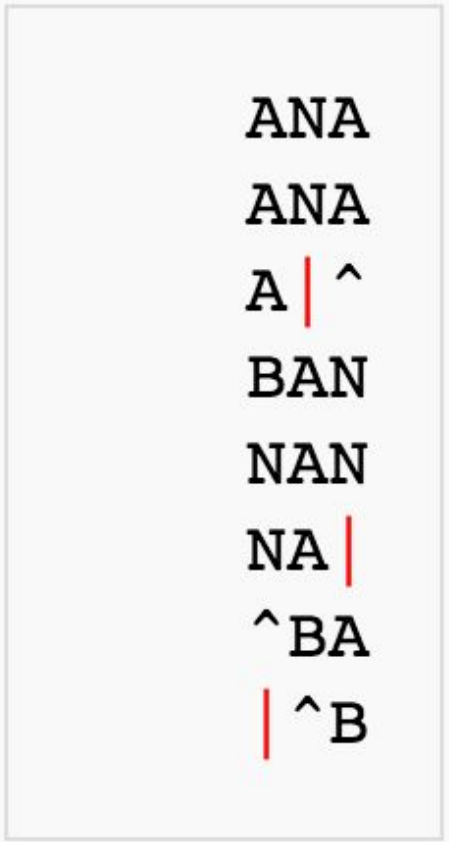
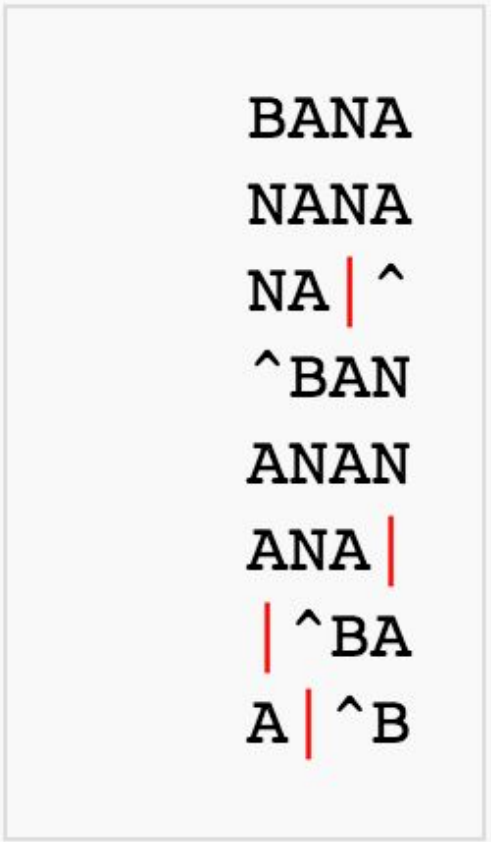
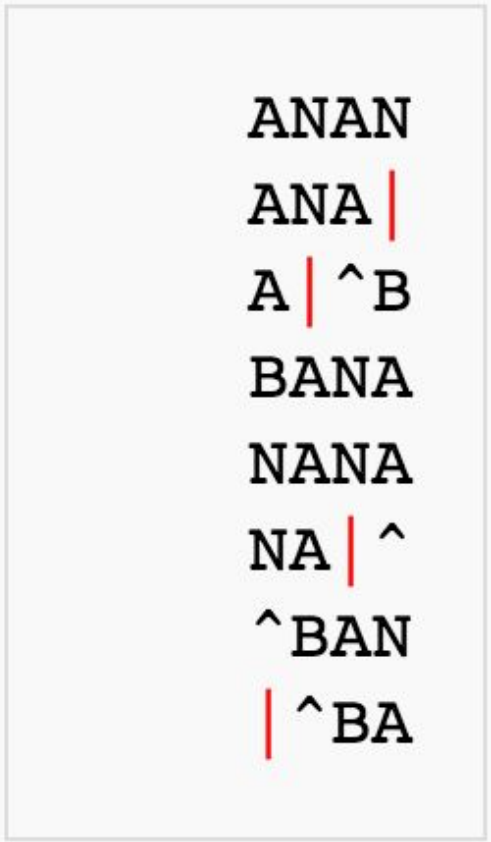
See Problem 1 in: http://bit.ly/2R7bXYj

BWT Compression

- The BWT is interesting only because it is reversible - i.e. given the BWT of S you can get back S! So the compression is lossless.

Inverse Transformation			
Input			
BNN^AA|A			
Add 1	**Sort 1**	**Add 2**	**Sort 2**
B N N ^ A A | A	A A A B N N ^ |	BA NA NA ^B AN AN |^ A|	AN AN A| BA NA NA ^B |^

http://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform

Add 3	Sort 3	Add 4	Sort 4
			

Add 7	Sort 7	Add 8	Sort 8
BANANA| NANA|^B NA|^BAN ^BANANA ANANA|^ ANA|^BA |^BANAN A|^BANA	ANANA|^ ANA|^BA A|^BANA BANANA| NANA|^B NA|^BAN ^BANANA |^BANAN	BANANA|^ NANA|^BA NA|^BANA ^BANANA| ANANA|^B ANA|^BAN |^BANANA A|^BANAN	ANANA|^B ANA|^BAN A|^BANAN BANANA|^ NANA|^BA NA|^BANA ^BANANA| |^BANANA
Output			
^BANANA|			

Exercise 2 (20 mins)

Reverse the BWT.

See Problem 2

The FM-index : Compressed BWT Search

- With suffix trees and suffix arrays we have *O(|P|)* time search for instances of a pattern string *P* in a collection of strings *T*
- But, suffix trees and suffix arrays require 5 to 15 bytes of memory per character in *T*
- What if we could search the BWT?
  - We can! it's called the FM-index

The FM-index : Compressed BWT Search

F		L
$	abracadabr	a
a	$abracadab	r
a	bra$abraca	d
a	bracadabra	$
a	cadabra$ab	r
a	dabra$abra	c
b	ra$abracad	a
b	racadabra$	a
c	adabra$abr	a
d	abra$abrac	a
r	a$abracada	b
r	acadabra$a	b

The FM-index : Compressed BWT Search

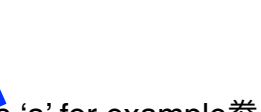
- FM-index (FM = “Full text Minute space”, but also “Ferragina and Manzini”):
  - Conceptually is F and L arrays
  - L is the BWT

F		L
$	abracadabr	a
a	$abracadab	r
a	bra$abraca	d
a	bracadabra	$
a	cadabra$ab	r
a	dabra$abra	c
b	ra$abracad	a
b	racadabra$	a
c	adabra$abr	a
d	abra$abrac	a
r	a$abracada	b
r	acadabra$a	b

= BWT

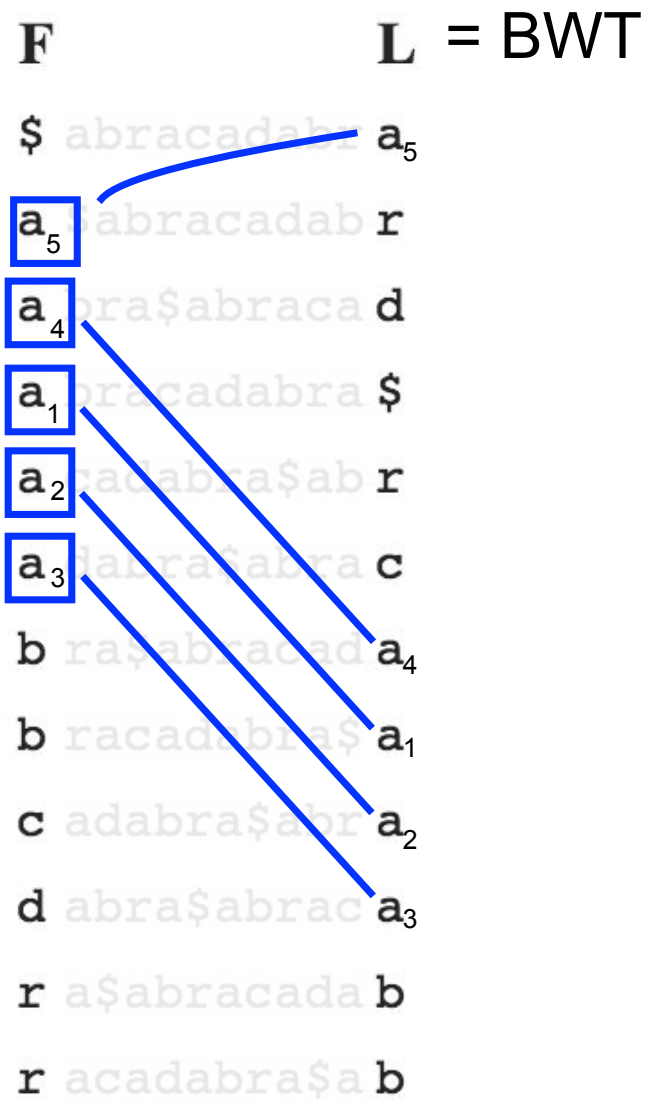
The FM-index : Compressed BWT Search

- Let the *character rank* of an index in *S$* be the number of occurrences of *S[i]* in the prefix *S[1 … i]*
  - *e.g. S$ = "$a_1bra_2ca_3da_4bra_5$"*
    - *therefore for 'a' characters:*
      - S_1 character rank is *1*
      - S_4 character rank is *2*
      - S_6 character rank is *3, etc.*



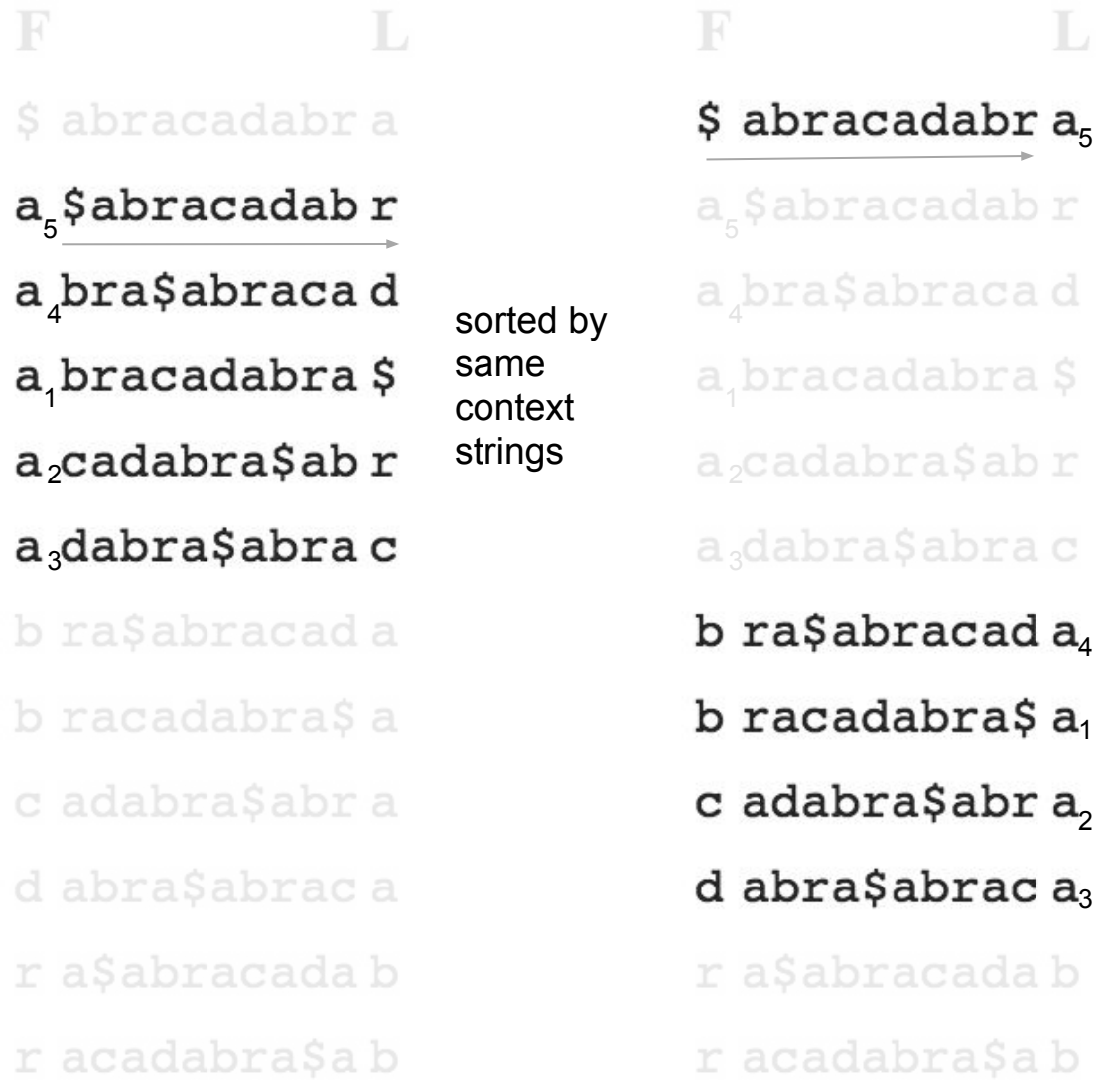
The FM-index : Compressed BWT Search

- We can label the indices in *F* and *L* with their character ranks in *S*
  - *Shown are the character ranks for 'a' instances*
- *Key observation:*
  - *The character ranks have the same order in both F and L*
  - *Why?*



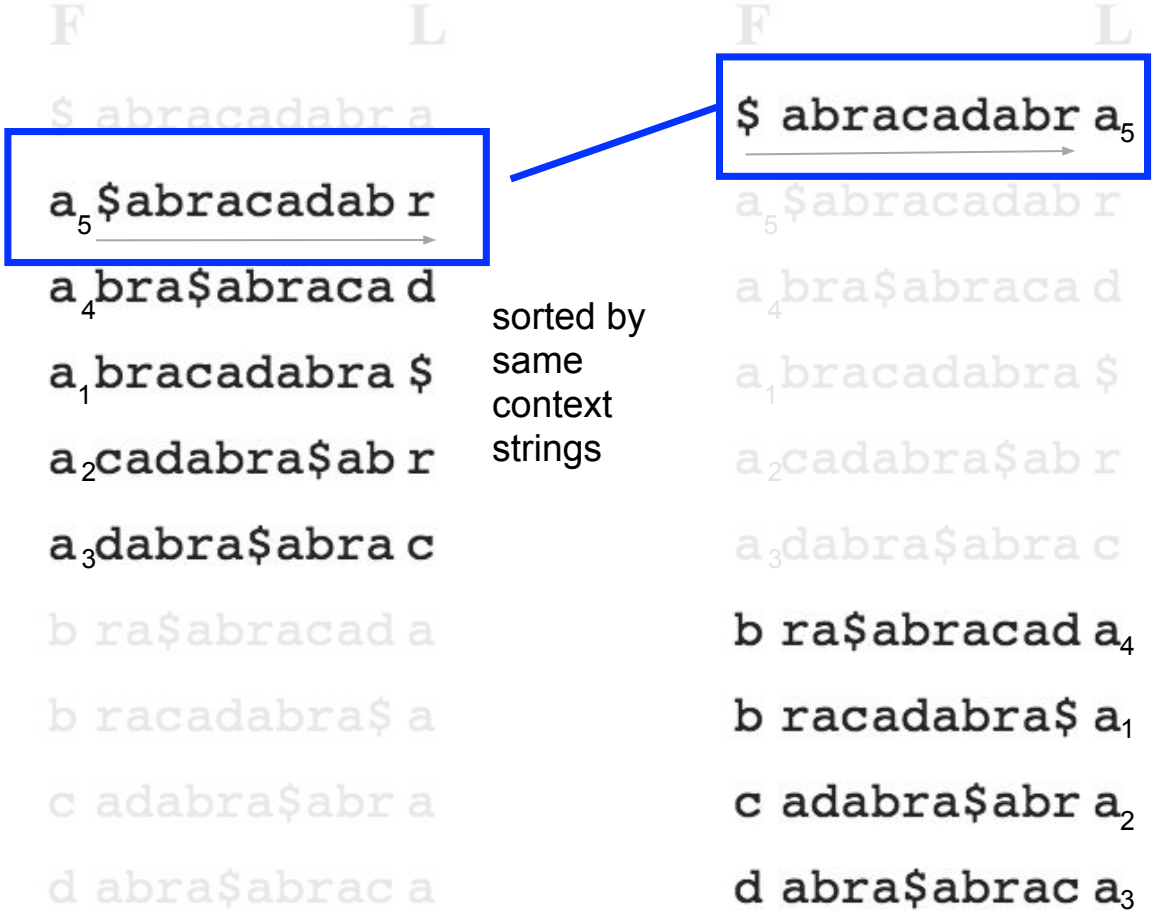
The FM-index : Compressed BWT Search

- We can label the indices in *F* and *L* with their character ranks in *S*
  - *Shown are the character ranks for 'a' instances*
- *Key observation:*
  - *The character ranks have the same order in both F and L*
  - *Why?*



The FM-index : Compressed BWT Search

- We can label the indices in *F* and *L* with their character ranks in *S*
  - *Shown are the character ranks for 'a' instances*
- *Key observation:*
  - *The character ranks have the same order in both F and L*
  - *Why?*
    - **Rephrased: the relative ordering of elements of the same character in |S| is the same in both F and L**



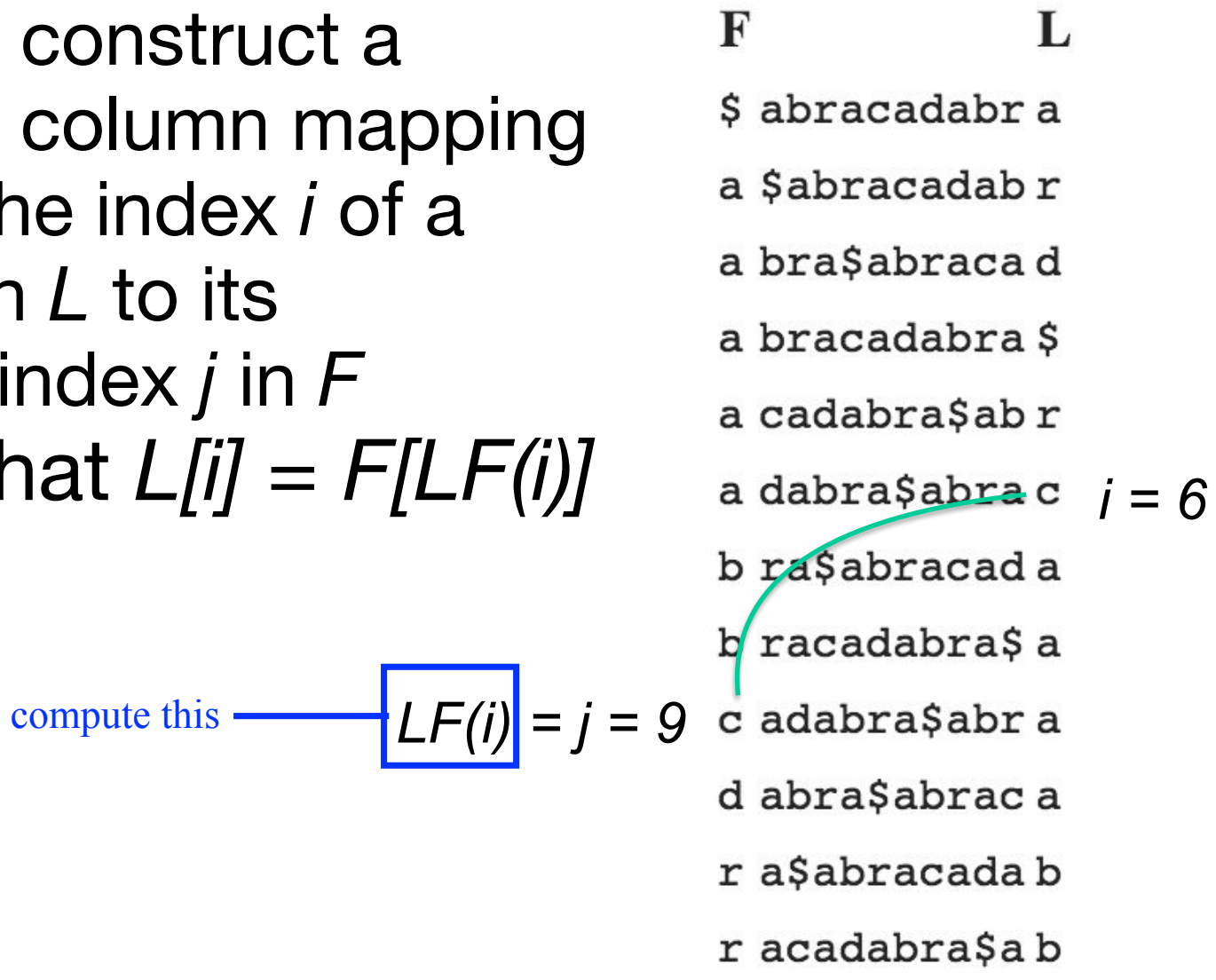
The FM-index : Compressed BWT Search

- Given key observation, it is possible to construct a last-to-first column mapping *LF(i)* from the index *i* of a character in *L* to its equivalent index *j* in *F*
  - i.e. so that *L[i] = F[LF(i)]*

F		L
$	abracadabr	a
a	$abracadab	r
a	bra$abraca	d
a	bracadabra	$
a	cadabra$ab	r
a	dabra$abra	c
b	ra$abracad	a
b	racadabra$	a
c	adabra$abr	a
d	abra$abrac	a
r	a$abracada	b
r	acadabra$a	b

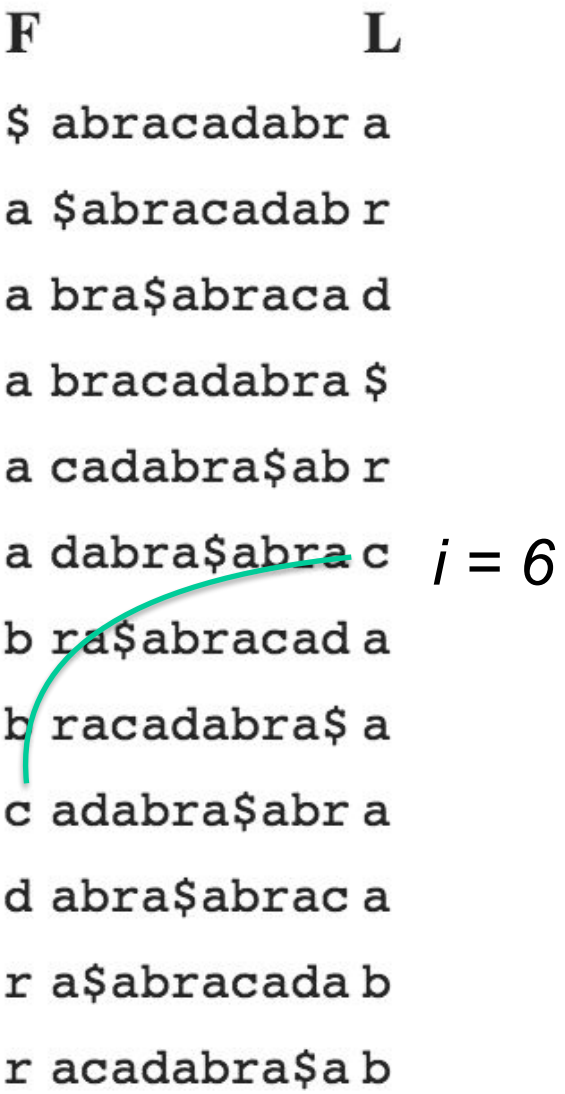
The FM-index : Compressed BWT Search

- Given key observation, it is possible to construct a last-to-first column mapping *LF(i)* from the index *i* of a character in *L* to its equivalent index *j* in *F*
  - i.e. so that *L[i] = F[LF(i)]*



The FM-index : Compressed BWT Search

- Given key observation, it is possible to construct a last-to-first column mapping *LF(i)* from the index *i* of a character in *L* to its equivalent index *j* in *F*
  - i.e. so that $L[i] = F[LF(i)]$
- We will use this for searching substrings



The FM-index : Compressed BWT Search

- $LF(i) = C[L[i]] + Occ(L[i], i)$

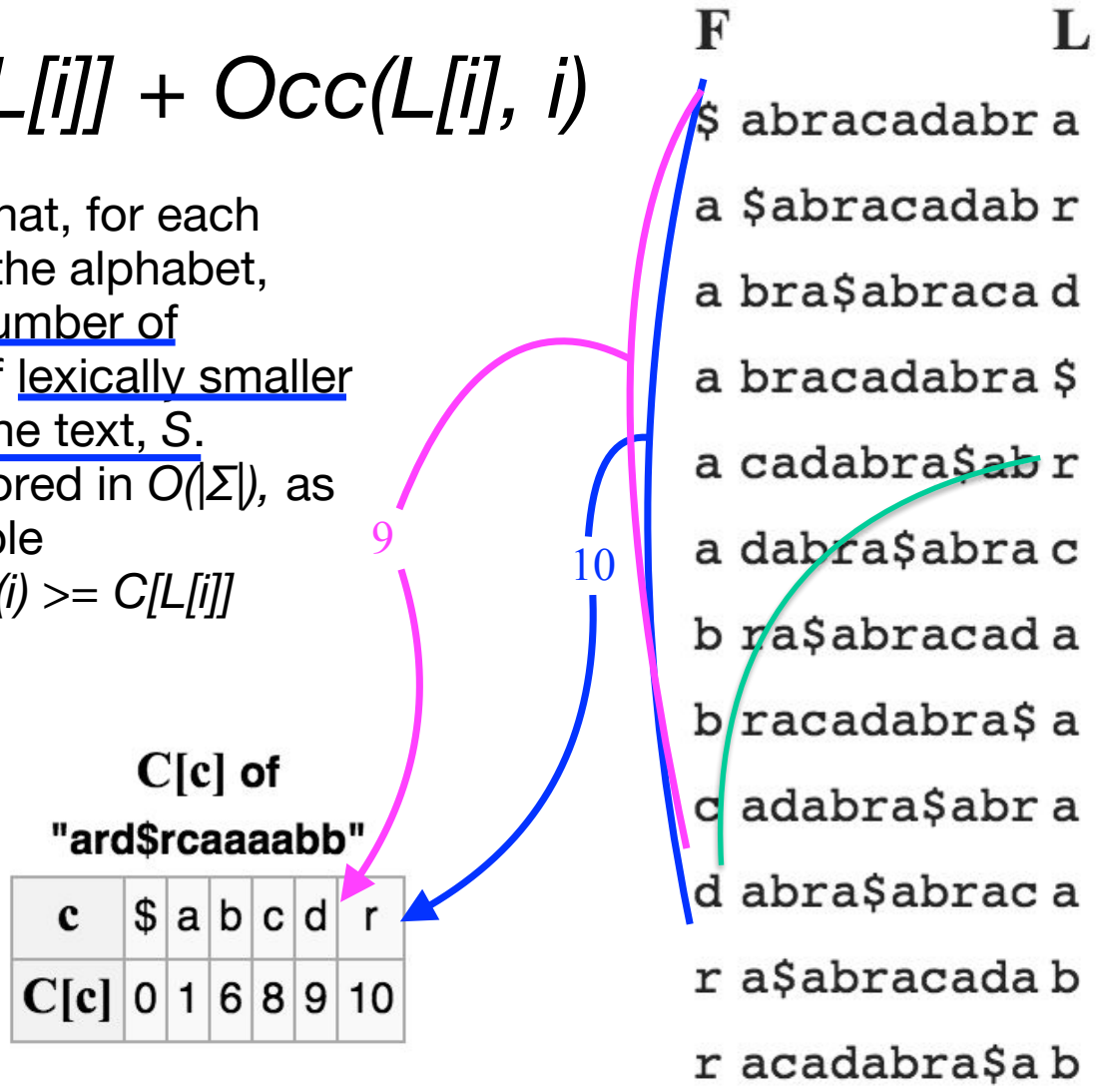
smaller chars

character ranks

F	L
$	abracadabra
a	$abracadabr
a	bra$abracad
a	bracadabra$
a	cadabra$abr
a	dabra$abrac
b	ra$abracada
b	racadabra$a
c	adabra$abra
d	abra$abrac
r	a$abracadab
r	acadabra$ab

The FM-index : Compressed BWT Search

- *LF(i) = C[L[i]] + Occ(L[i], i)*
  - C[c] is a map that, for each character *c* in the alphabet, contains the number of occurrences of lexically smaller characters in the text, *S*.
    - Can be stored in *O(|Σ|)*, as lookup table
    - Clearly *LF(i) >= C[L[i]]*



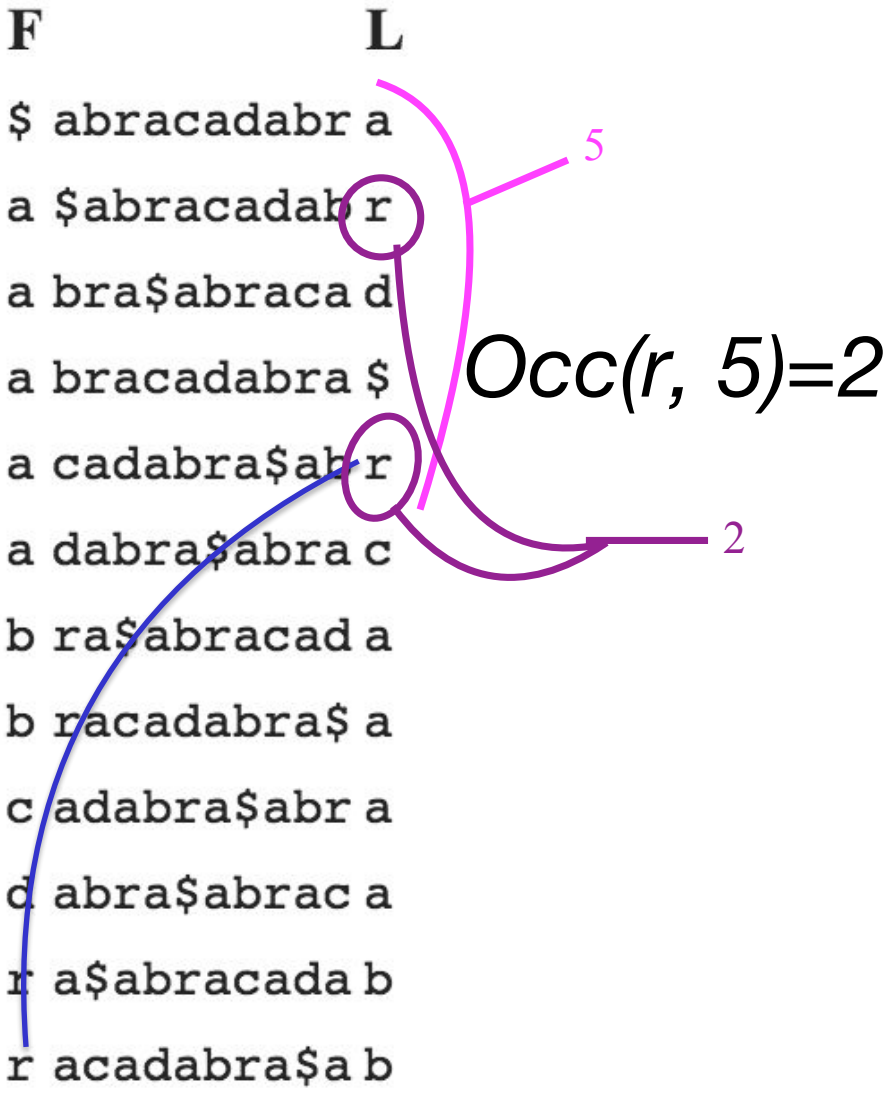
C[c] of "ard$rcaaaabb"

c	$	a	b	c	d	r
C[c]	0	1	6	8	9	10

The FM-index : Compressed BWT Search

- *LF(i) = C[L[i]] + Occ(L[i], i)*
  - *Occ(c, k)* is the number of occurrences of character *c* in the prefix *L[1..k]*, i.e. character rank
    - **We could calculate this in O(|S|) time by counting occurrences in L[1..k]**

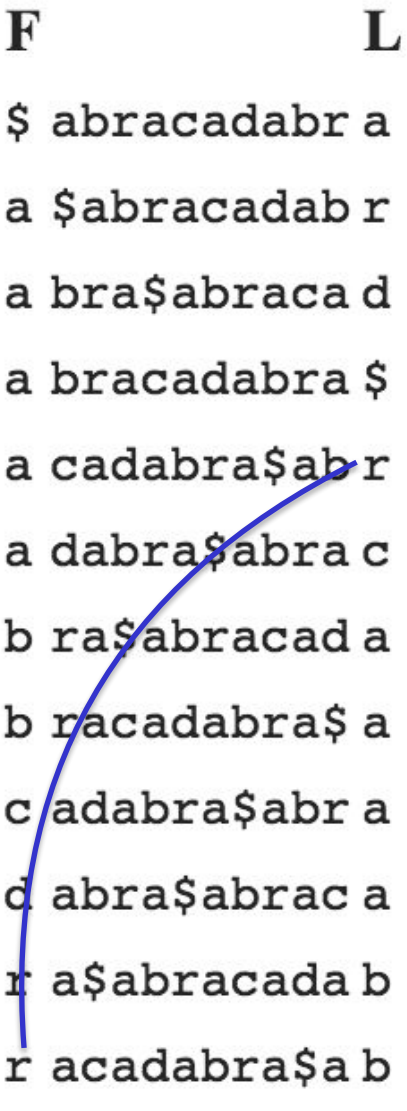
LF[5] = C(r) + Occ(r, 5) = 12



The FM-index : Compressed BWT Search

- *LF(i) = C[L[i]] + Occ(L[i], i)*
  - *Occ(c, k)* is the number of occurrences of character *c* in the prefix *L[1..k]*, i.e. character rank
    - **Key result: Ferragina and Manzini showed that it is possible to compute *Occ(c, k)* in constant time given *L*.**
    - How?

LF[5] = C(r) + Occ(r, 5) = 12



Occ(r, 5)=2

The FM-index : Compressed BWT Search

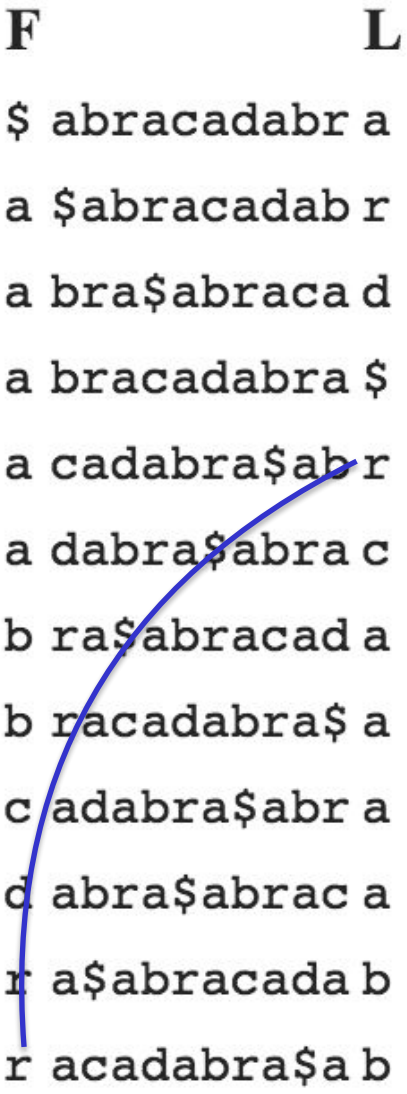
- *LF(i) = C[L[i]] + Occ(L[i], i)*

Occ(c, k) can be stored as a lookup table...

Occ(c, k) of "ard$rcaaaabb"

	a	r	d	$	r	c	a	a	a	a	b	b
	1	2	3	4	5	6	7	8	9	10	11	12
$	0	0	0	1	1	1	1	1	1	1	1	1
a	1	1	1	1	1	1	2	3	4	5	5	5
b	0	0	0	0	0	0	0	0	0	0	1	2
c	0	0	0	0	0	1	1	1	1	1	1	1
d	0	0	1	1	1	1	1	1	1	1	1	1
r	0	1	1	1	2	2	2	2	2	2	2	2

But this requires |S| * Σ integers - too big!



Occ(r, 5)=2

The FM-index : Compressed BWT Search

- *LF(i) = C[L[i]] + Occ(L[i], i)*

Occ(c, k) can be stored as a lookup table...

Occ(c, k) of "ard$rcaaaabb"

	a	r	d	$	r	c	a	a	a	a	b	b
	1	2	3	4	5	6	7	8	9	10	11	12
$	0	0	0	1	1	1	1	1	1	1	1	1
a	1	1	1	1	1	1	2	3	4	5	5	5
b	0	0	0	0	0	0	0	0	0	0	1	2
c	0	0	0	0	0	1	1	1	1	1	1	1
d	0	0	1	1	1	1	1	1	1	1	1	1
r	0	1	1	1	2	2	2	2	2	2	2	2

Solution: store only every b entries and scan.. assume entries evenly spaced, lookups are O(1) time

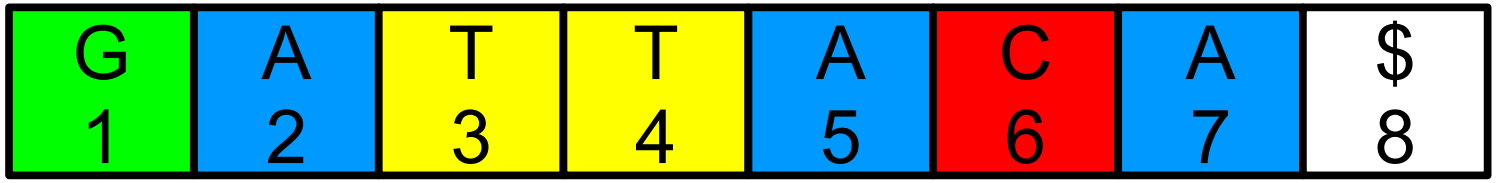
F	L
$ abracadabr	a
a $abracadab	r
a bra$abraca	d
a bracadabra	$
a cadabra$ab	r
a dabra$abra	c
b ra$abracad	a
b racadabra$	a
c adabra$abr	a
d abra$abrac	a
r a$abracada	b
r acadabra$a	b

Occ(r, 5)=2

Exercise 3 (10 mins)

Complete the FM index class

See Problem 3



String S

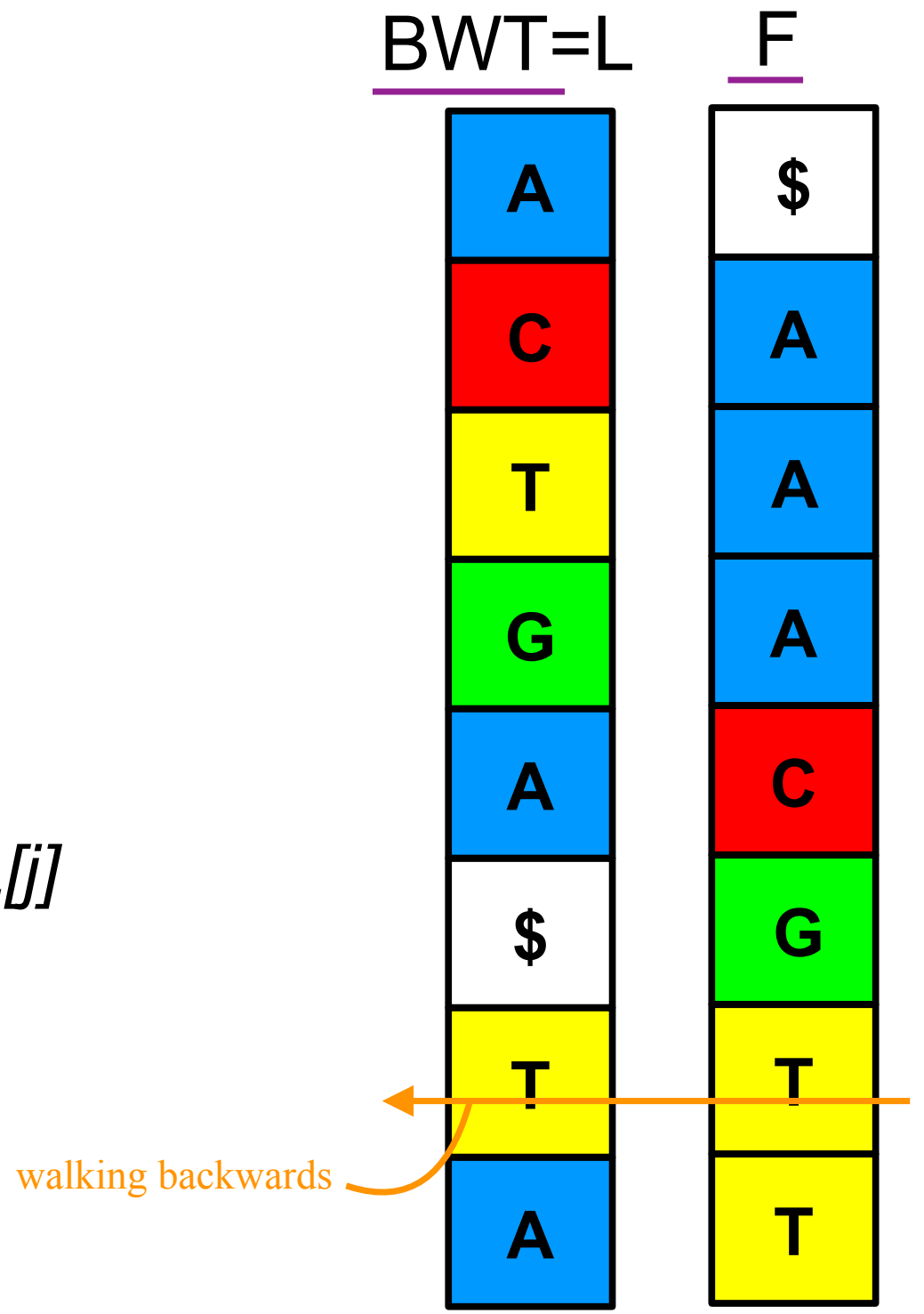
We will search for a pattern *P* = *"ATTA"* in the FM-index for *S* ***backwards***, in contrast to the forward search of the suffix trie/suffix tree/suffix array

Small technical note: we're indexing from 1 instead of 0 for the following, unlike when we used this example for the suffix array

Backward Search

Initial search string: ε
Range: 1-8

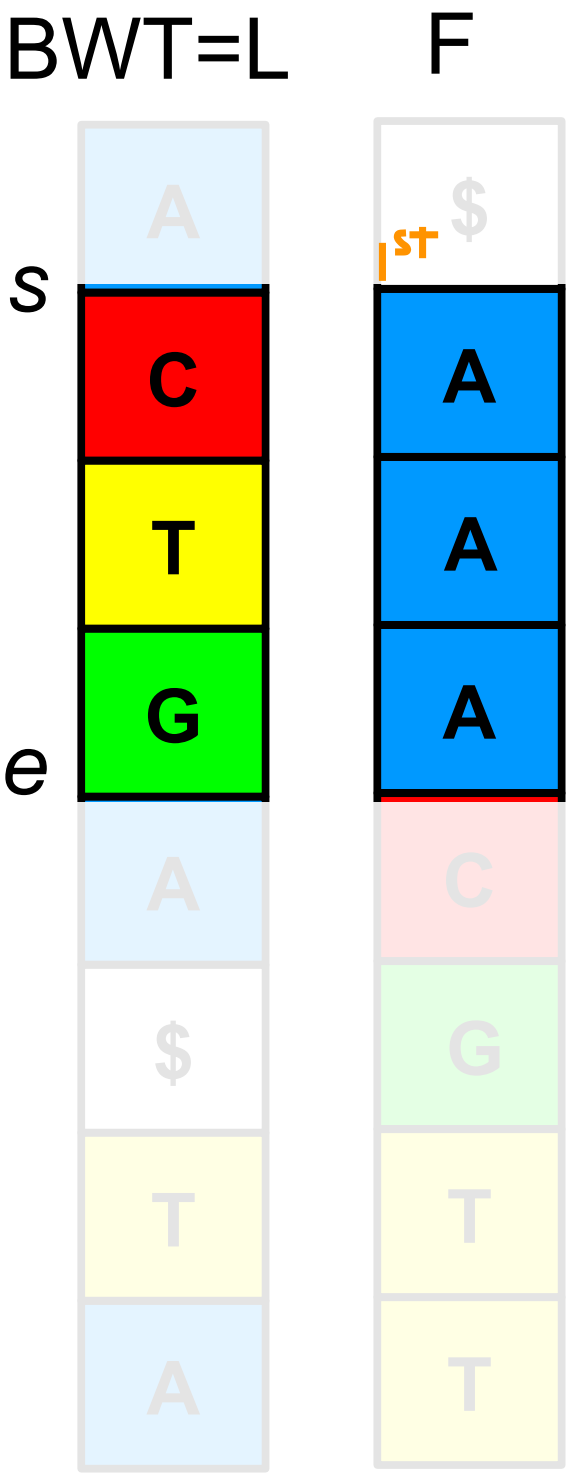
- Note: I'm putting the BWT (L) before F, because for each index *1* <= *j* <= |S|, *L[j]* represents the character that precedes *F[j]*.



Backward Search

Search string: A
Range: 2-4

- The initial range is set to *[s=C[A]+1, .., e=C[A+1]] = [s=2, .., e=4]*.
- This range represents the suffixes beginning with *A* in *F*.



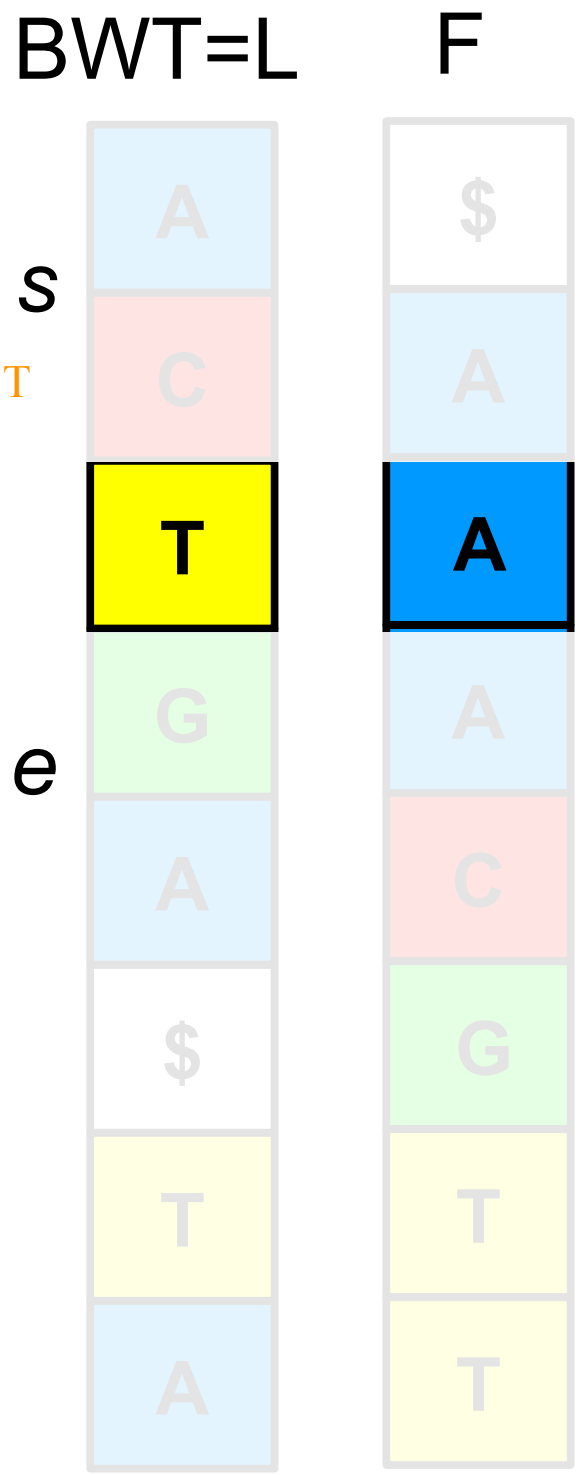
Backward Search

Search string: TA
Range: ?-?

look for all of the As prefixed by a T

- The next character in *P* back-to-front is: T.
- The highlighted sub-range of [s, …, e] represents the interval in F of suffixes beginning with an A and preceded by T.
- Using LF mapping want to shift forward by one column to the interval of suffixes beginning with T and followed by an A, this is equivalent to...

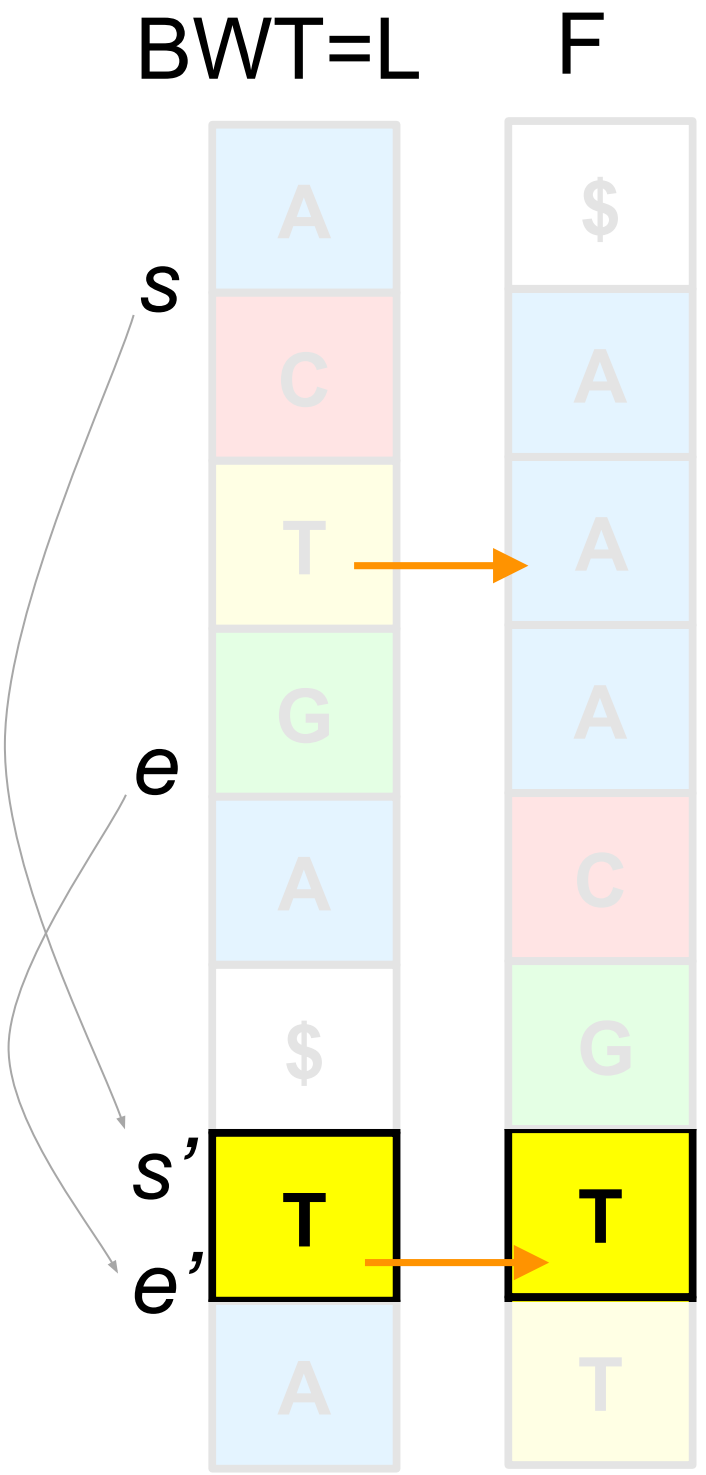
last to first



Backward Search

Search string: TA
Range: 7-7

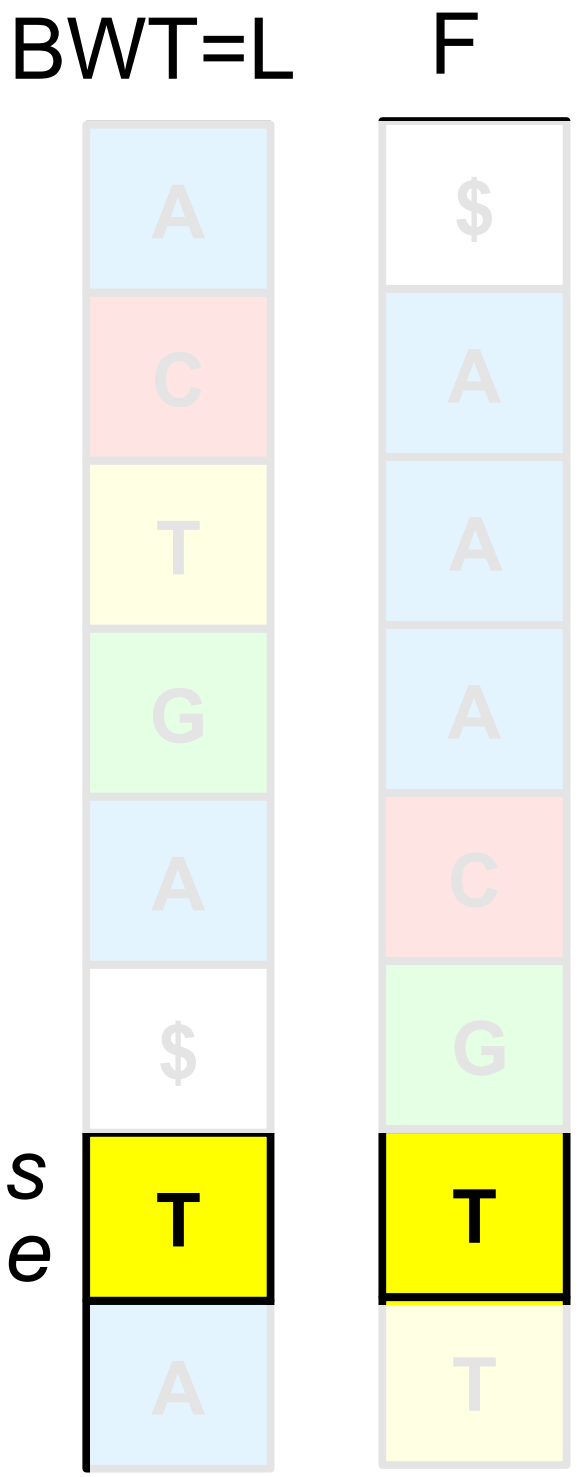
- The next character in *P* back-to-front is: T.
- The new range is:
  - s’ = C[T] + Occ(T, s-1) + 1
  - e’ = C[T] + Occ(T, e)]
  - [s’ = 6 + 0 + 1, …, e’ = 6 + 1]
  - [s’=7, … , e’=7]
- This range over F is all the suffixes beginning with TA.



Backward Search

Search string: TTA
Range: ?-?

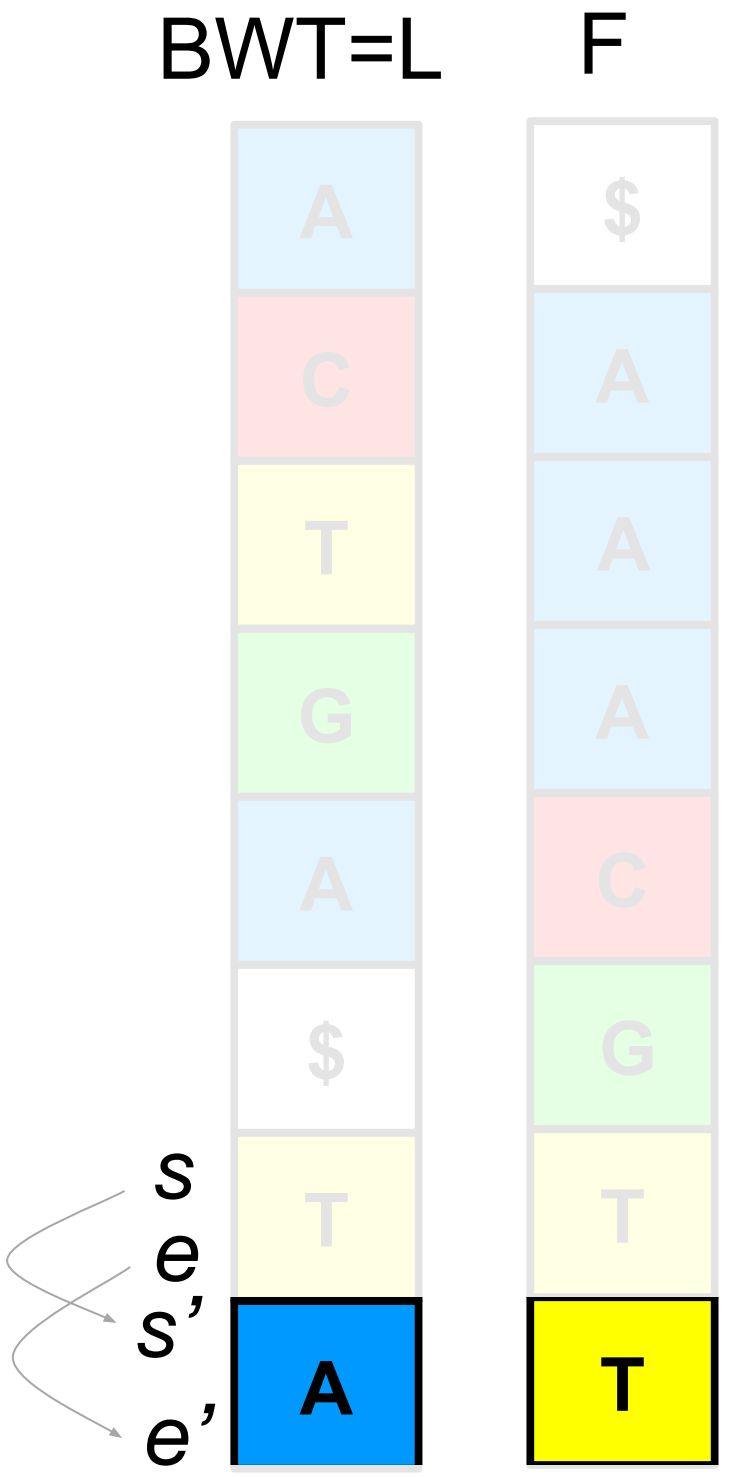
- The next character in *P* back-to-front is (again): T.
- The highlighted sub-range of [s, …, e] represents the interval in F of suffixes beginning with **TA** and preceded by **T**.
- Using LF mapping want to shift forward by one column to the interval of suffixes beginning with **TTA**, this is again equivalent to...



Backward Search

Search string: TTA
Range: 8-8

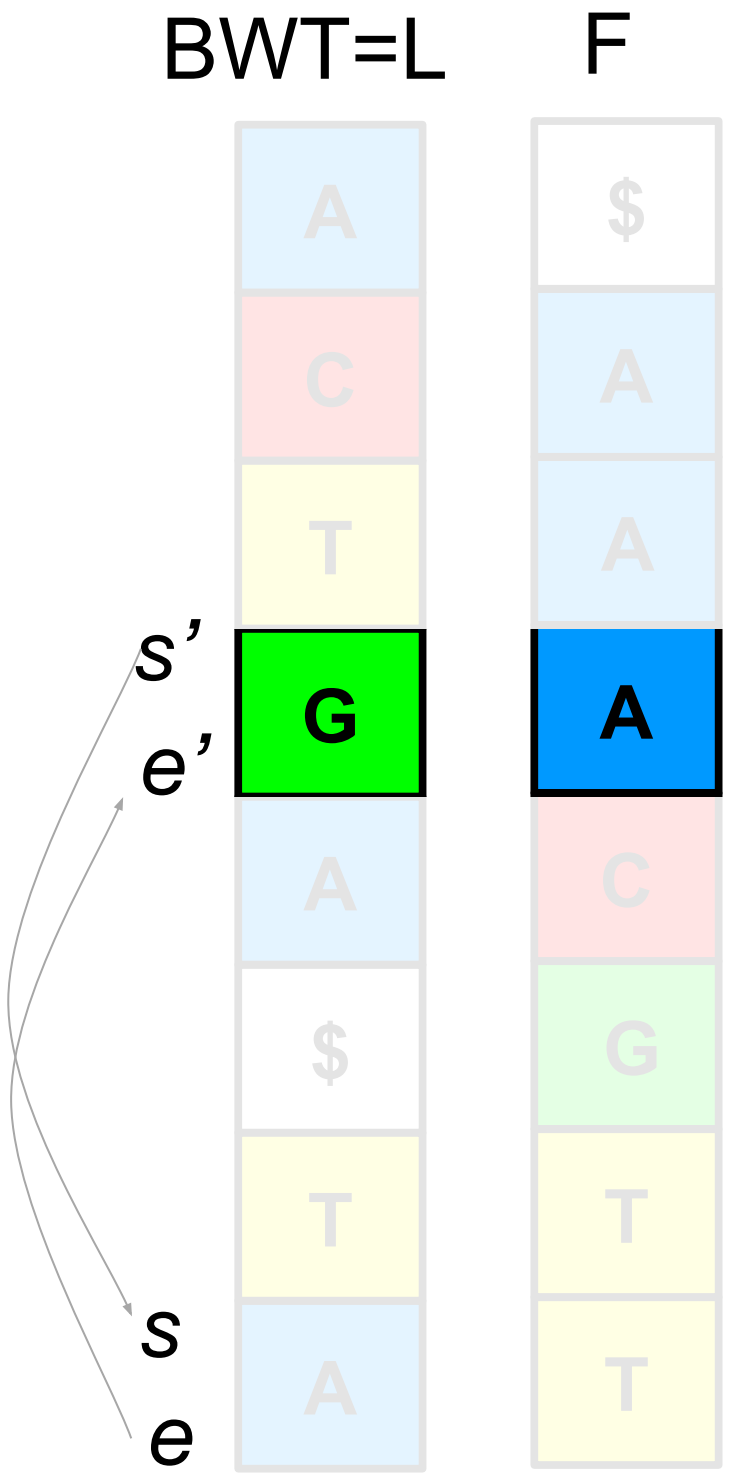
- The next character in *P* back-to-front is (again): T.
- The new range is:
  - s’ = C[T] + Occ(T, s-1) + 1
  - e’ = C[T] + Occ(T, e)]
  - [s’ = 6 + 1 + 1, …, e’ = 6 + 2]
  - [s’=8, … , e’=8]
- This range over F is all the suffixes beginning with TTA.



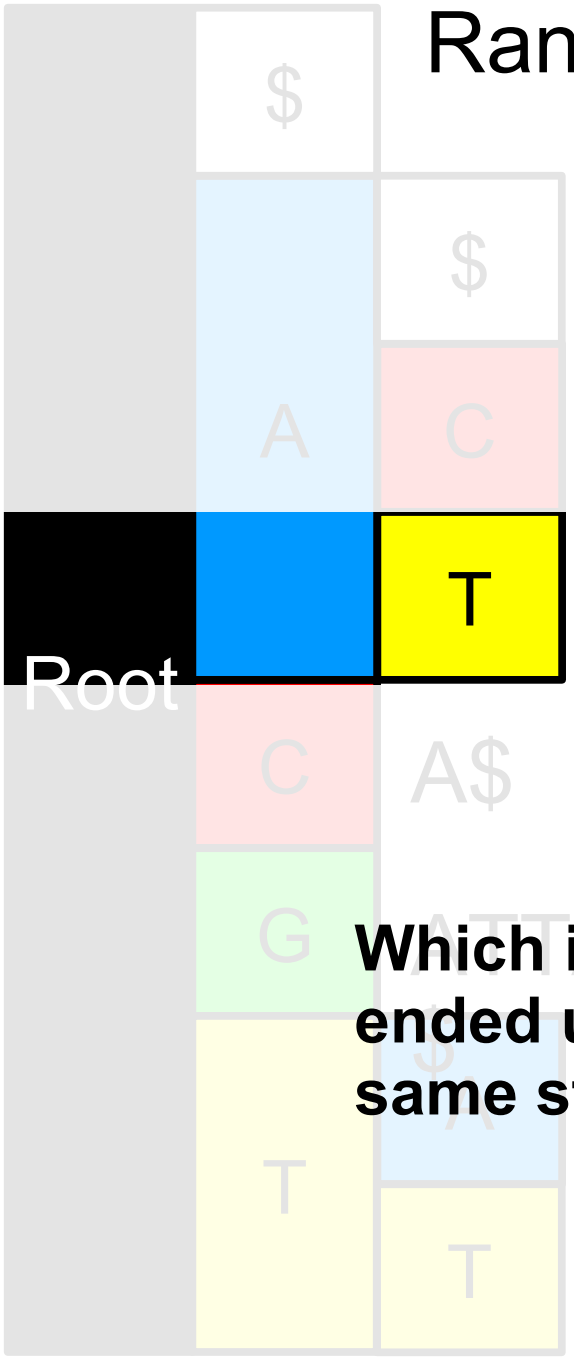
Backward Search

Search string: ATTA
Range: 4-4

- The last character in *P* is: A.
- The new range is:
  - s’ = C[T] + Occ(T, s-1) + 1
  - e’ = C[T] + Occ(T, e)]
  - [s’ = 1 + 2 + 1, …, e’ = 1 + 3]
  - [s’=4, … , e’=4]
- This range over F is all the suffixes beginning with **ATTA**.

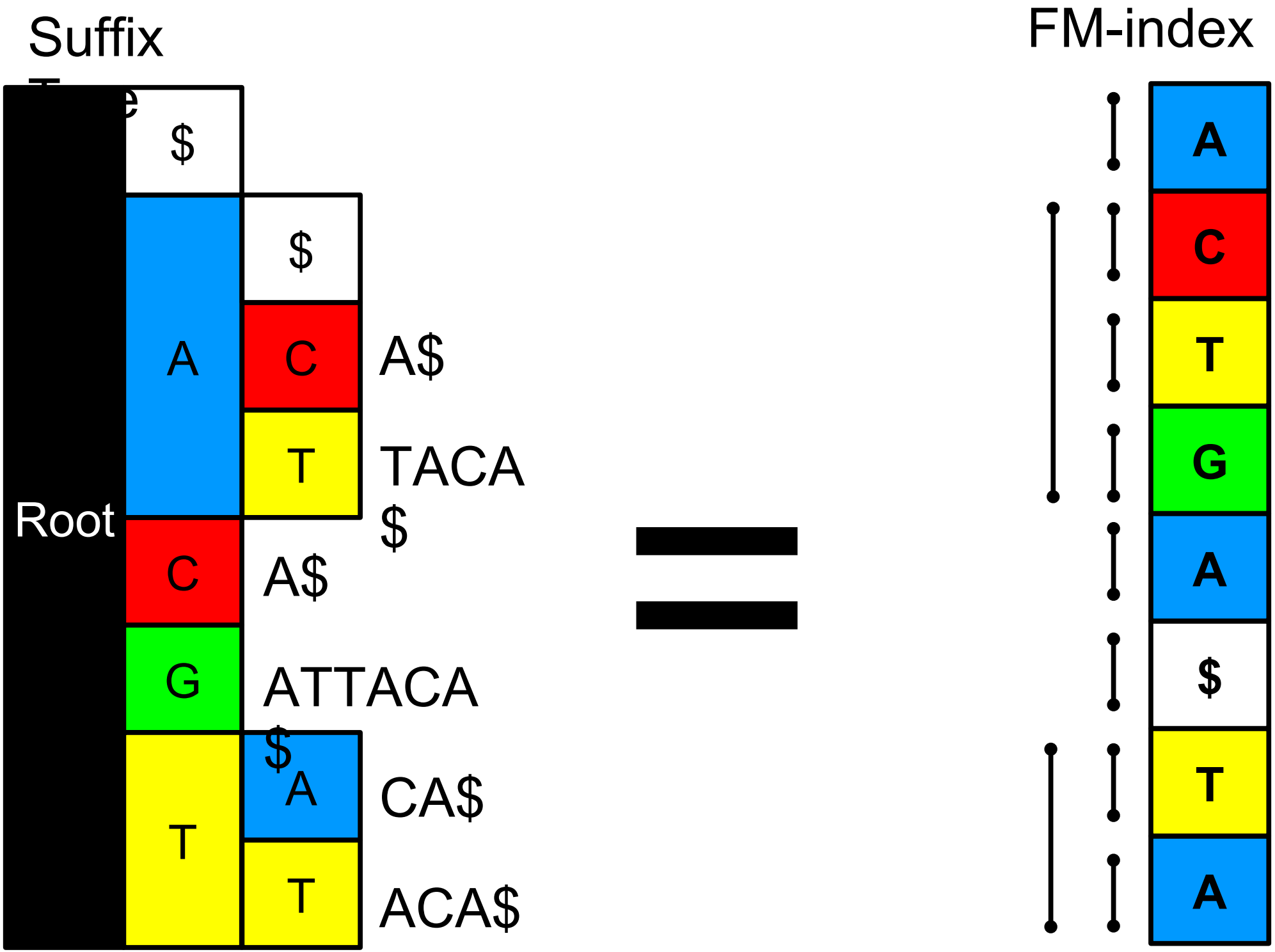


Search string: ATTA
Range: 3-3



Which is the same place we ended up when searching the same string by forward search.

F	BWT
$	A
A	C
A	T
A	**G**
C	A
G	$
T	T
T	A


Suffix
FM-index
Root
$
A
C
T
G
A$
TACA$
A$
ATTACA$
CA$
ACA$

Exercise 4 (20 mins)

Implement simple backward search to invert bwt

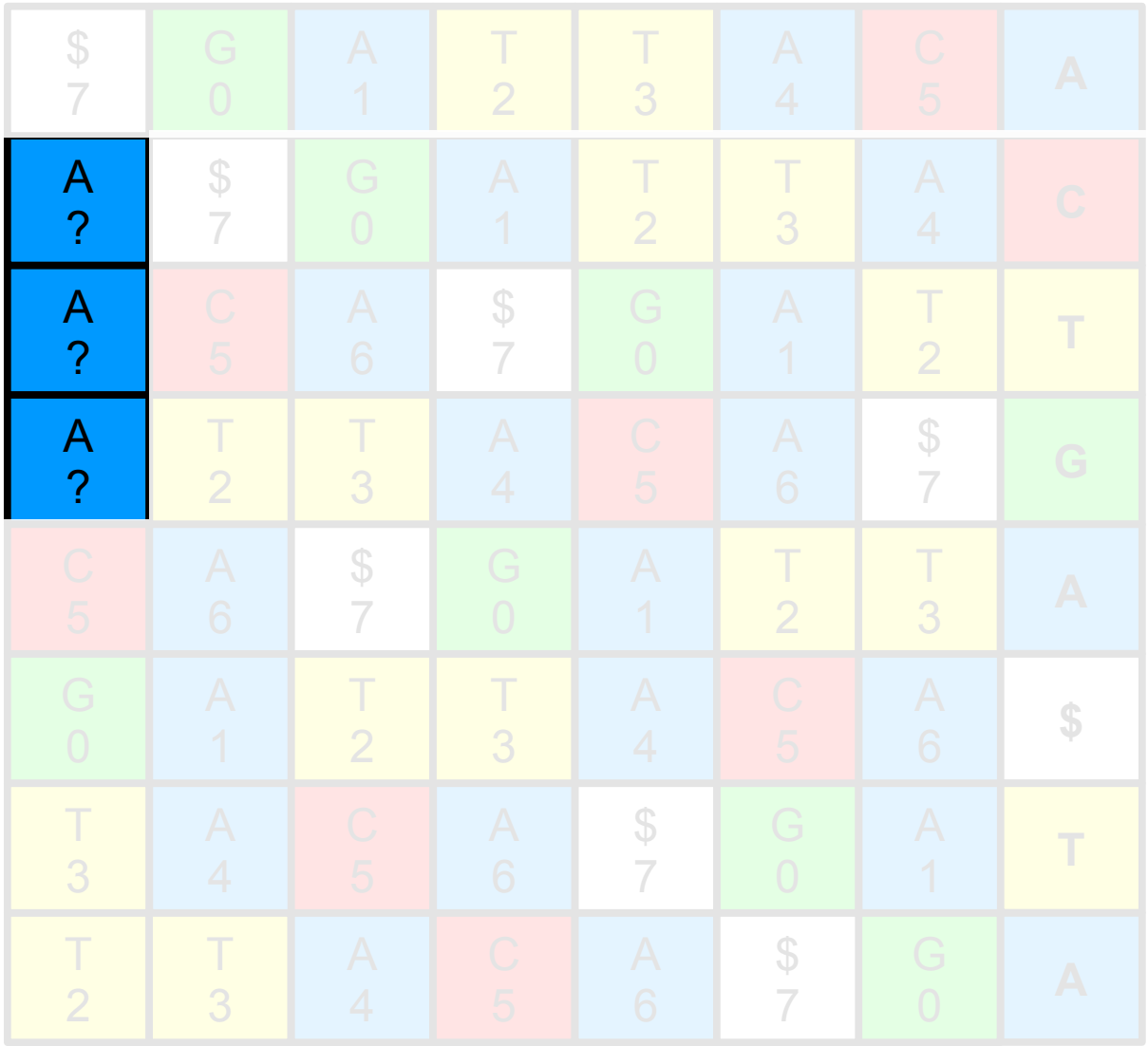
— keep looking at preceeding character to create the original string

See Problem 4

(as extra credit, you are welcome to implement full backward search for a query string)

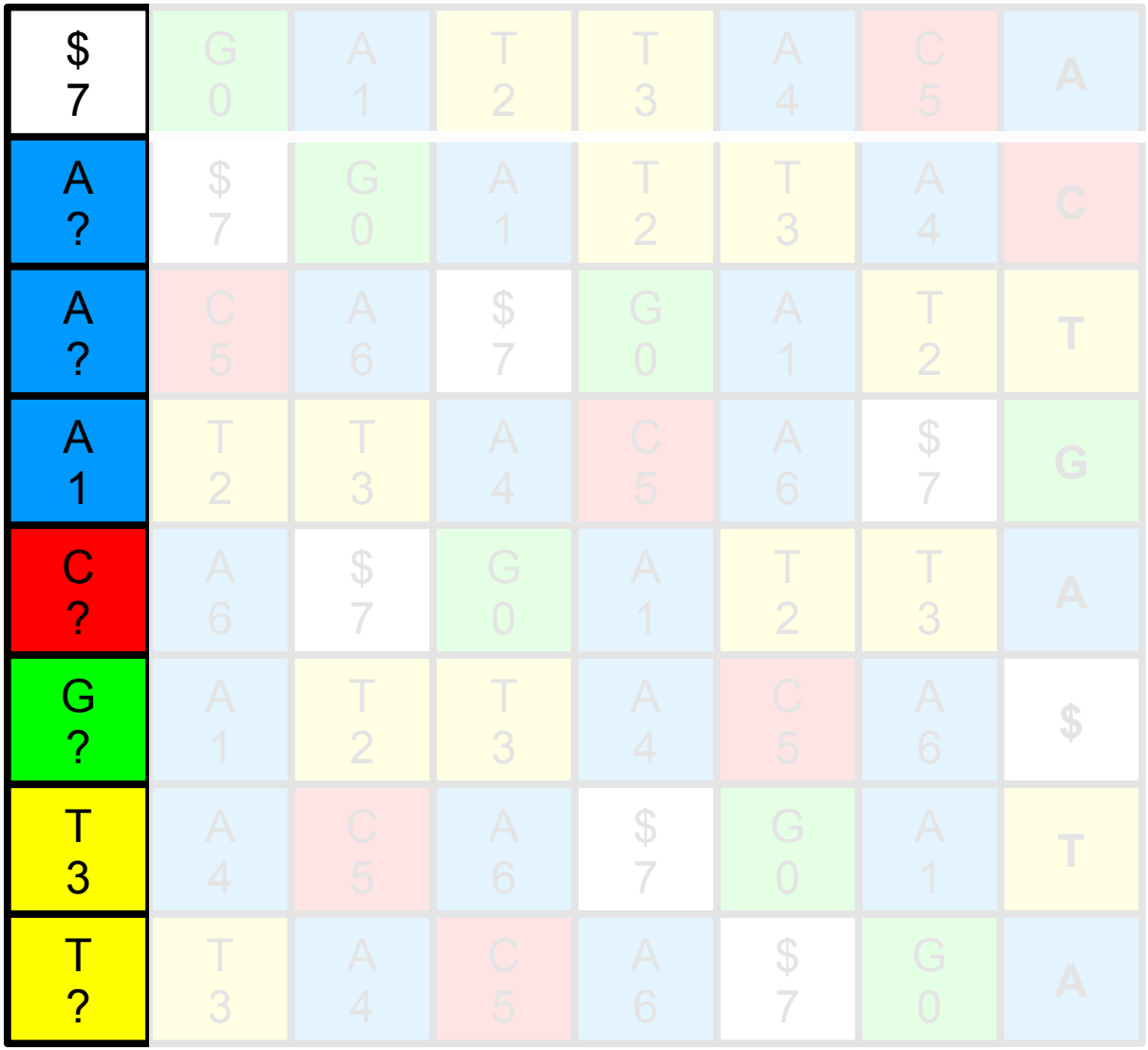
FM-index and locate

- Backward search identifies an interval of *F* whose suffixes start with a *P* pattern prefix.
- However, without the suffix array, we don't know the indices of these suffixes - we can't "locate" instances



FM-index and locate

- Solution: as with Occ(), store a fraction of rows of the suffix array
- When we need to locate, use repeated LF mappings to find a row with a suffix array value. Gives O(1) locates!



The FM-index : Space Considerations

FM-index = BWT + C[] + Occ()

BWT = 2 bits/base

C[] = |Σ| characters

Occ() = |Σ| * 1/b, where b is the sampling rate

Suffix array sample = 1/a integers, where a is the sampling rate

For b = 128, a = 16 we get complete index in around 1.5GB for human genome (# from Ben Langmead)

Substring indexes extend to graphs!

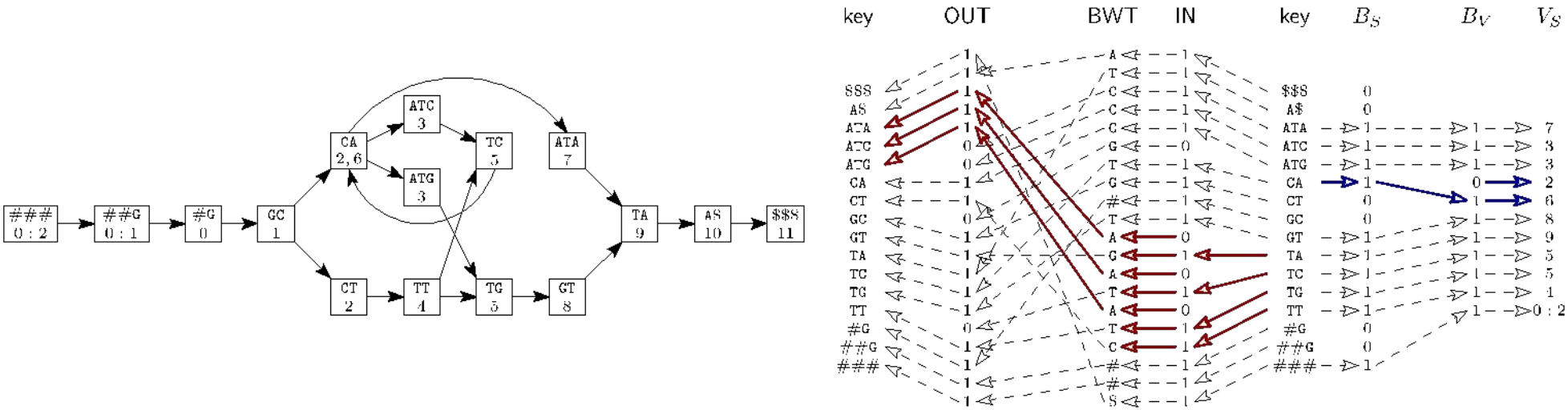


Figure 3: Left: An order-3 pruned de Bruijn graph G'' 3-equivalent to the de Bruijn graph in Figure 2. Right: GCSA for graph G''. Leftward arrows illustrate backward searching, with the red arrows showing it from `T` to `AT`. Rightward arrows mark the samples belonging to each node, with the blue ones showing them for node `CAT`.

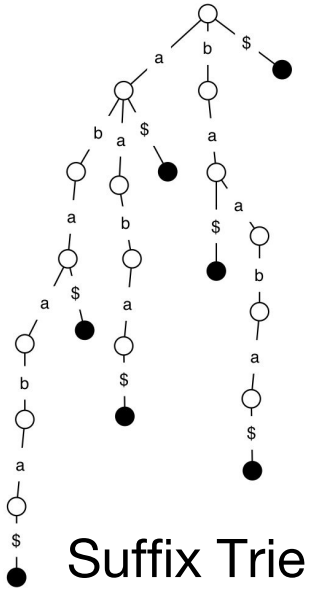
https://arxiv.org/abs/1604.06605 -

Jouni Siren

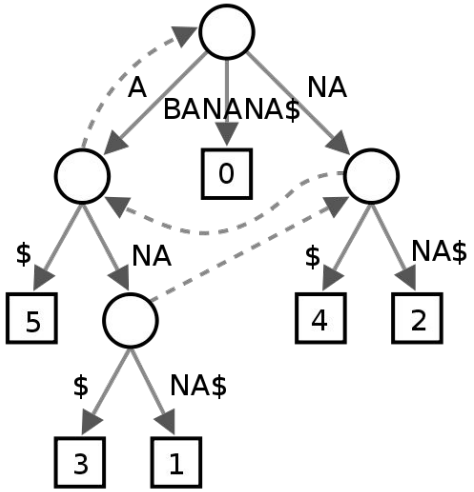
FM-index conclusion

What have we gained?

- O(|P|) pattern search, eliminating the log(|S|) term of the suffix array
- In practise, need about ½ byte per character for FM-index, contrast to 5-12 for suffix array and suffix approaches
- The FM-index is used by todays tools like bowtie and bwa-mem
- Indexing more complex sequence ensembles (graphs), is the new frontier



Suffix Trie



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

BWT/FM Index

smaller!

Thanks! Don’t forget to submit your completed notebook through canvas before the next lecture

The first homework is now issued and can be found here: http://bit.ly/2s9FW2E