

Burrows-Wheeler Transform

Methods in Bioinformatics (TI)

Strings

- Alphabet Σ
- String S

String: AAGTGCTCAAAGCTAAGCTCCAT

Prefix:

Suffix:

Substring:

Strings

- **Alphabet Σ :** set of *characters* (e.g., $\Sigma=\{A,C,G,T\}$)
- **String S :** sequence of $n=|S|$ characters drawn from Σ , i.e., $S[i]\in\Sigma$ for $0\leq i<n$

String: AAGTGCTCAAAGCTAAGCTCCAT

Prefix: AAGTGC

Suffix: CAT

Substring: AAAGC

String Ordering

Lexicographic/alphabetical order

animal < house < ta < tac < zoo

*When no character breaks the tie (e.g., one string is prefix of the other),
shorter comes first.*

String Rotations

String rotation refers to the process of moving characters in a string from one end to the other while maintaining their order.

TACTAC

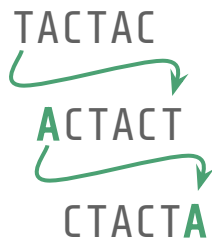
String Rotations

String rotation refers to the process of moving characters in a string from one end to the other while maintaining their order.

TACTAC
ACTACT

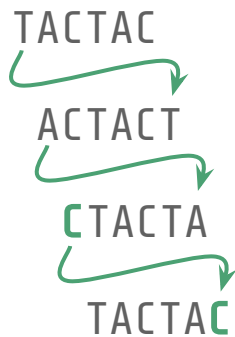
String Rotations

String rotation refers to the process of moving characters in a string from one end to the other while maintaining their order.



String Rotations

String rotation refers to the process of moving characters in a string from one end to the other while maintaining their order.



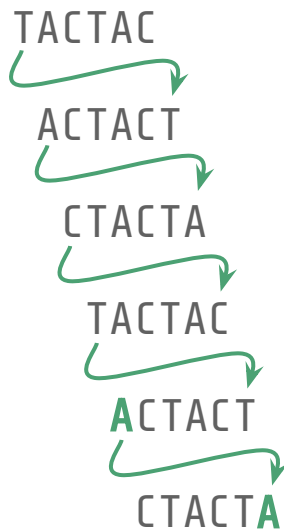
String Rotations

String rotation refers to the process of moving characters in a string from one end to the other while maintaining their order.



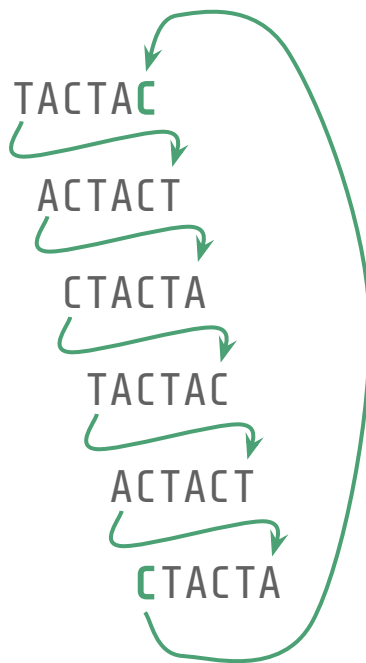
String Rotations

String rotation refers to the process of moving characters in a string from one end to the other while maintaining their order.



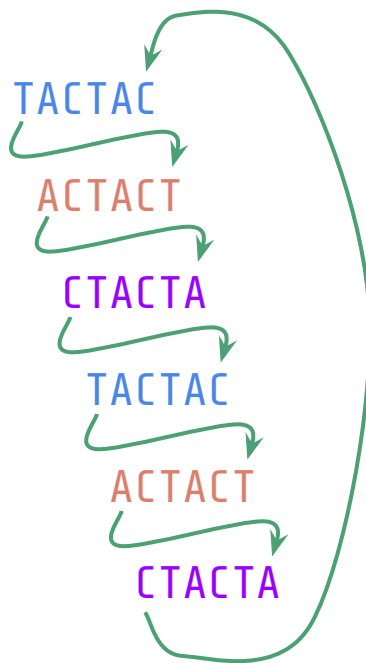
String Rotations

String rotation refers to the process of moving characters in a string from one end to the other while maintaining their order.



String Rotations

String rotation refers to the process of moving characters in a string from one end to the other while maintaining their order.



Special character \$

1. Define a new symbol $\$$:
 - $\$ \notin \Sigma$
 - $\$ < c \quad \forall c \in \Sigma$
2. Append $\$$ to the string

Special character \$

1. Define a new symbol **\$**:
 - $\$ \notin \Sigma$
 - $\$ < c \quad \forall c \in \Sigma$
2. Append **\$** to the string

TACTAC\$
ACTAC\$
CTAC\$
TAC\$
AC\$
AC\$
C\$
\$

enforces order on suffixes
(no suffix is a prefix of any other suffix)

Special character \$

1. Define a new symbol **\$**:
 - $\$ \notin \Sigma$
 - $\$ < c \quad \forall c \in \Sigma$
2. Append **\$** to the string

TACTAC\$
ACTAC\$
CTAC\$
TAC\$
AC\$
AC\$
C\$
\$

enforces order on suffixes
(no suffix is a prefix of any other suffix)

TACTAC\$
ACTAC\$T
CTAC\$TA
TAC\$TAC
AC\$TACT
C\$TACTA
\$TACTAC

makes all rotations different

Burrows-Wheeler Transform (BWT)

*Reversible permutation of the characters of a string, introduced for **compression***

Burrows-Wheeler Transform (BWT)

Reversible permutation** of the characters of a string, introduced for **compression

(i)

(ii)

(iii)

Burrows-Wheeler Transform (BWT)

Reversible permutation** of the characters of a string, introduced for **compression

(i)

(ii)

(iii)

TACTAC\$

Burrows-Wheeler Transform (BWT)

Reversible permutation** of the characters of a string, introduced for **compression

(i)

(ii)

(iii)

TACTAC\$

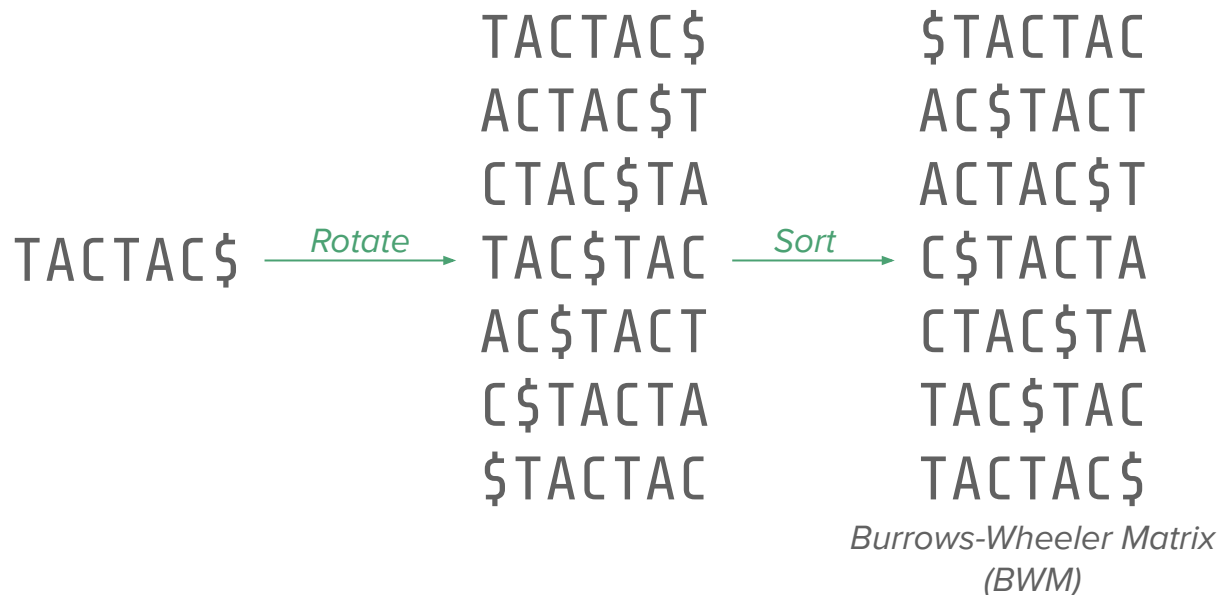
→ Rotate →

TACTAC\$
ACTAC\$T
CTAC\$TA
TAC\$TAC
AC\$TACT
C\$TACTA
\$TACTAC

Burrows-Wheeler Transform (BWT)

Reversible permutation of the characters of a string, introduced for **compression**

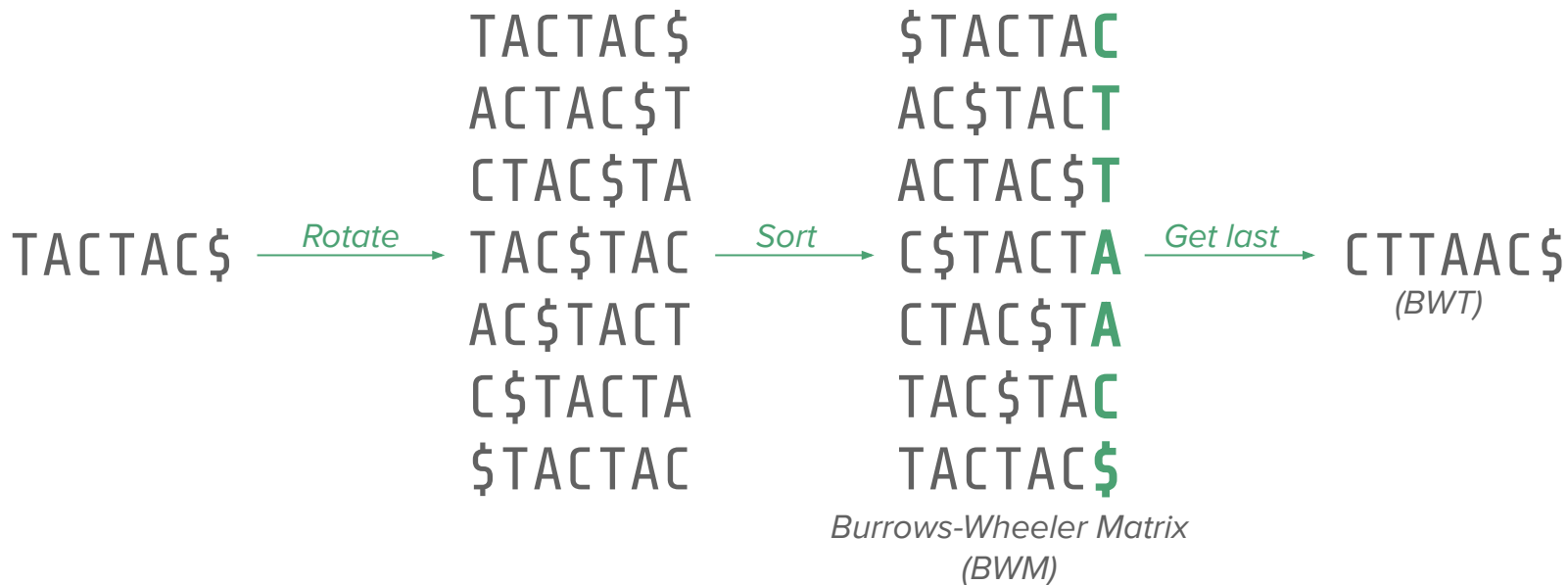
(i) (ii) (iii)



Burrows-Wheeler Transform (BWT)

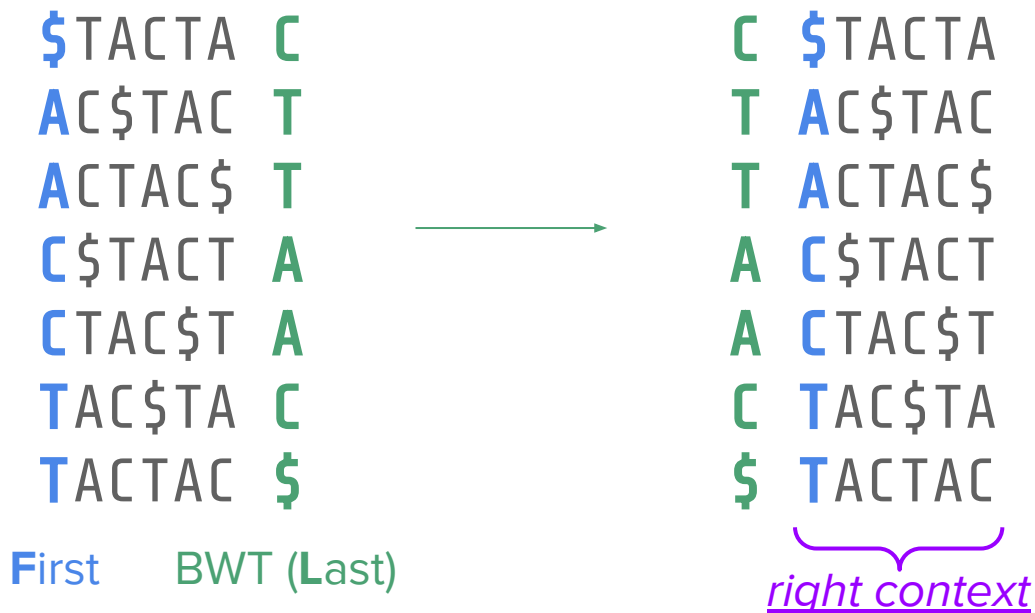
Reversible permutation of the characters of a string, introduced for **compression**

(i) (ii) (iii)



Permutation

BWT permutes characters according to their right contexts



Compression

BWT facilitates compression (it does not compress the input string)

- *it tends to cluster identical characters together*
- *it combines repeated patterns into larger contiguous blocks*
- *it makes the string easier to compress (e.g., run-length encoding)*

Compression

BWT facilitates compression (it does not compress the input string) - **bzip**

- *it tends to cluster identical characters together*
- *it combines repeated patterns into larger contiguous blocks*
- *it makes the string easier to compress (e.g., run-length encoding)*

CGATGCATCGTAGCAGCATCGATGACCAGAGCATCGACGACGAGCAGACCACAGCAGCAGTACTCAA
GCAGCAGCATCAGCACGACCAGATCTAGCAGCAGTATAGAGAGAGACGACGATCTCATCAGCAGCAT
AAGACGACGACGACTACTACATCGACAGATATAG\$



GTCGCTGGGC **GGGGGG** TGTTCAGGGTCCCCGACTCCTCCCCGCCCTTG **CCCCC** GGGGTGGCA
CTAT **GGGGGGGGGG** TAGGGAAATAAT **AAAAAA** T\$TAATATACATCACCACCCCAAACAAA
CCC **AAAAAAAAAAAAAAAA** TACAACCGAACGAGCAAC **AAAAAAA**

14,A
(9 bytes vs 14 bytes)*

**Very rough approximation, implementation-dependent,
no encoding (2bit/packed)*

BWT in Bioinformatics

- especially convenient for short reads
 - *millions of string searches in a long string*
- query complexity depends on read size
(*not on genome size*)
- “construction” is a 1 time expense

Ultrafast and memory-efficient **alignment** of short DNA sequences to the human genome

[B Langmead](#), [C Trapnell](#), [M Pop](#), [SL Salzberg](#) - Genome biology, 2009 - Springer

... For the human genome, Burrows-Wheeler indexing allows **Bowtie** to **align** more than 25 million reads per CPU hour with a memory footprint of approximately 1.3 gigabytes. ...

☆ Salva 77 Cita Citato da 25169 Articoli correlati Tutte e 29 le versioni

Fast and accurate short read **alignment** with Burrows–Wheeler transform

[H Li](#), [R Durbin](#) - bioinformatics, 2009 - academic.oup.com

... , and present the algorithm for inexact matching which is implemented in **BWA**. We evaluate the performance of **BWA** on simulated data by comparing the **BWA alignment** with the true ...

☆ Salva 77 Cita Citato da 52831 Articoli correlati Tutte e 29 le versioni

STAR: ultrafast universal RNA-seq aligner

[A Dobin](#), [CA Davis](#), [F Schlesinger](#), [J Drenkow](#)... - ..., 2013 - academic.oup.com

... **Alignment** to a Reference (**STAR**) software based on a previously undescribed RNA-seq **alignment** ... **STAR** outperforms other aligners by a factor of >50 in mapping speed, aligning to the ...

☆ Salva 77 Cita Citato da 50265 Articoli correlati Tutte e 21 le versioni

Reversible

$$S \rightarrow \text{BWT}(S) \rightarrow S$$



LF-Mapping (Last-to-First)

Property of BWT that allows to reconstruct the original string from the BWT, starting from its end and going backward

LF-Mapping

BWT(BANANA)

?

LF-Mapping

BWT(BANANA)

\$	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

ANNB\$AA

LF-Mapping

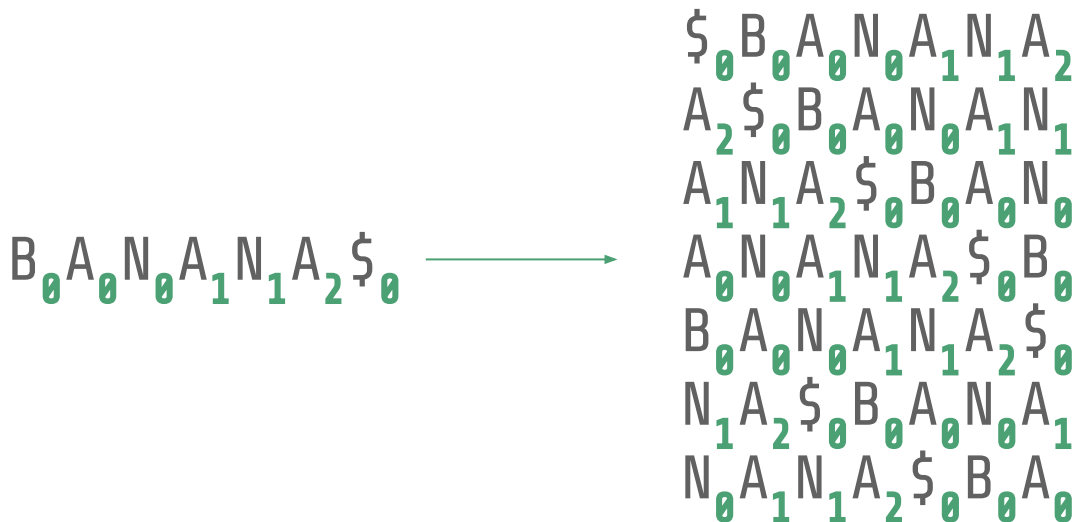
Let's give each character, its **rank** (*number of occurrences up to its position*)

B₀ A₀ N₀ A₁ N₁ A₂ \$₀

Ranks are not explicitly stored, we just use them to simplify the exposition

LF-Mapping

Let's give each character, its **rank** (*number of occurrences up to its position*)



Ranks are not explicitly stored, we just use them to simplify the exposition

LF-Mapping

Let's give each character, its **rank** (*number of occurrences up to its position*)

\$₀
A₂
A₁
A₀
B₀
N₁
N₀
F

A₂
N₁
N₀
B₀
\$₀
A₁
A₀
L

We do not need the entire matrix

LF-Mapping

Let's look at F:

\$₀
A₂
A₁
A₀
B₀
N₁
N₀
F

A₂
N₁
N₀
B₀
\$₀
A₁
A₀
L

LF-Mapping

Let's look at F:

- sorted column
- predictable column *(as long as we know how many times each character occur)*

\$₀
A₂
A₁
A₀
B₀
N₁
N₀
F

A₂
N₁
N₀
B₀
\$₀
A₁
A₀
L

LF-Mapping

Let's look at F:

- sorted column
- predictable column *(as long as we know how many times each character occur)*

Let's look at F and L:

\$₀
A₂
A₁
A₀
B₀
N₁
N₀
F

A₂
N₁
N₀
B₀
\$₀
A₁
A₀
L

LF-Mapping

Let's look at F:

- sorted column
- predictable column *(as long as we know how many times each character occur)*

A₂
A₁
A₀

Let's look at F and L:

- As occur in the same order

A₁
A₀
F
L

LF-Mapping

Let's look at F:

- sorted column
- predictable column *(as long as we know how many times each character occur)*

Let's look at F and L:

- As occur in the same order
- Same for Bs

B₀

B₀

F

L

LF-Mapping

Let's look at F:

- sorted column
- predictable column *(as long as we know how many times each character occur)*

N₁
N₀

Let's look at F and L:

- As occur in the same order
- Same for Bs
- Same for Ns

N₁
N₀
F

L

LF-Mapping

More generally,

the i^{th} occurrence of a character in L and the i^{th} occurrence of a character in F , correspond to the same occurrence in the original string (i.e., they have the same rank)

\$₀
A₂
A₁
A₀
B₀
N₁
N₀
 F

A₂
N₁
N₀
B₀
\$₀
A₁
A₀
 L

Why does LF-Mapping hold?

\$ ₀	B	A	N	A	N	A ₂
A ₂	\$	B	A	N	A	N ₁
A ₁	N	A	\$	B	A	N ₀
A ₀	N	A	N	A	\$	B ₀
B ₀	A	N	A	N	A	\$ ₀
N ₁	A	\$	B	A	N	A ₁
N ₀	A	N	A	\$	B	A ₀

\$ ₀	B	A	N	A	N	A ₂
A ₂	\$	B	A	N	A	N ₁
A ₁	N	A	\$	B	A	N ₀
A ₀	N	A	N	A	\$	B ₀
B ₀	A	N	A	N	A	\$ ₀
N ₁	A	\$	B	A	N	A ₁
N ₀	A	N	A	\$	B	A ₀

Why does LF-Mapping hold?

*Why are these As
in this relative
order?*

\$	B	A	N	A	N	A	2
A	\$	B	A	N	A	N	1
A	N	A	\$	B	A	N	0
A	N	A	N	A	\$	B	0
B	A	N	A	N	A	\$	0
N	A	\$	B	A	N	A	1
N	A	N	A	\$	B	A	0

\$	B	A	N	A	N	A	2
A	\$	B	A	N	A	N	1
A	N	A	\$	B	A	N	0
A	N	A	N	A	\$	B	0
B	A	N	A	N	A	\$	0
N	A	\$	B	A	N	A	1
N	A	N	A	\$	B	A	0

Why does LF-Mapping hold?

*Why are these As
in this relative
order?*

\$ ₀	B	A	N	A	N	A ₂
A ₂	\$	B	A	N	A	N ₁
A ₁	N	A	\$	B	A	N ₀
A ₀	N	A	N	A	\$	B ₀
B ₀	A	N	A	N	A	\$ ₀
N ₁	A	\$	B	A	N	A ₁
N ₀	A	N	A	\$	B	A ₀

*They are sorted by
their right context*

\$ ₀	B	A	N	A	N	A ₂
A ₂	\$	B	A	N	A	N ₁
A ₁	N	A	\$	B	A	N ₀
A ₀	N	A	N	A	\$	B ₀
B ₀	A	N	A	N	A	\$ ₀
N ₁	A	\$	B	A	N	A ₁
N ₀	A	N	A	\$	B	A ₀

Why does LF-Mapping hold?

*Why are these As
in this relative
order?*

\$ ₀	B	A	N	A	N	A ₂
A ₂	\$	B	A	N	A	N ₁
A ₁	N	A	\$	B	A	N ₀
A ₀	N	A	N	A	\$	B ₀
B ₀	A	N	A	N	A	\$ ₀
N ₁	A	\$	B	A	N	A ₁
N ₀	A	N	A	\$	B	A ₀

*They are sorted by
their right context*

\$ ₀	B	A	N	A	N	A ₂
A ₂	\$	B	A	N	A	N ₁
A ₁	N	A	\$	B	A	N ₀
A ₀	N	A	N	A	\$	B ₀
B ₀	A	N	A	N	A	\$ ₀
N ₁	A	\$	B	A	N	A ₁
N ₀	A	N	A	\$	B	A ₀

*Why are these As
in this relative
order?*

Why does LF-Mapping hold?

*Why are these As
in this relative
order?*

\$	B	A	N	A	N	A	2
A	\$	B	A	N	A	N	1
A	N	A	\$	B	A	N	0
A	N	A	N	A	\$	B	0
B	A	N	A	N	A	\$	0
N	A	\$	B	A	N	A	1
N	A	N	A	\$	B	A	0

*They are sorted by
their right context*

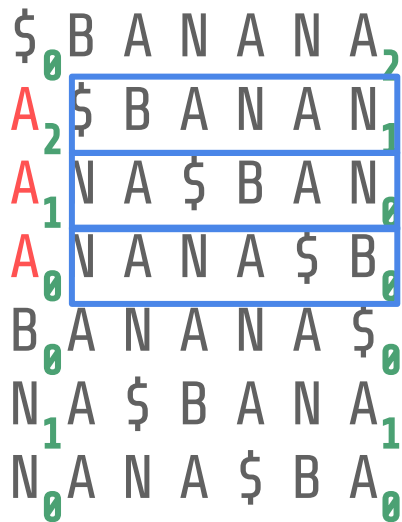
\$	B	A	N	A	N	A	2
A	\$	B	A	N	A	N	1
A	N	A	\$	B	A	N	0
A	N	A	N	A	\$	B	0
B	A	N	A	N	A	\$	0
N	A	\$	B	A	N	A	1
N	A	N	A	\$	B	A	0

*Why are these As
in this relative
order?*

They are sorted by their left context

Why does LF-Mapping hold?

*Why are these As
in this relative
order?*



*They are sorted by
their right context*



*Why are these As
in this relative
order?*

*They are sorted by their left context,
that by construction (rotations) it's
their right context*

Why does LF-Mapping hold?

*Why are these As
in this relative
order?*

\$ ₀	B	A	N	A	N	A ₂
A ₂	\$	B	A	N	A	N ₁
A ₁	N	A	\$	B	A	N ₀
A ₀	N	A	N	A	\$	B ₀
B ₀	A	N	A	N	A	\$ ₀
N ₁	A	\$	B	A	N	A ₁
N ₀	A	N	A	\$	B	A ₀

*They are sorted by
their right context*

A ₂	\$ ₀	B	A	N	A	N	A ₂
N ₁	A ₂	\$	B	A	N	A	N ₁
N ₀	A ₁	N	A	\$	B	A	N ₀
B ₀	A ₀	N	A	N	A	\$	B ₀
\$ ₀	B ₀	A	N	A	N	A	\$ ₀
A ₁	N ₁	A	\$	B	A	N	A ₁
A ₀	N ₀	A	N	A	\$	B	A ₀

*Why are these As
in this relative
order?*

*They are sorted by their left context,
that by construction (rotations) it's
their right context*

Both columns are sorted following the same principle, therefore are in the same order

Reversing a BWT

These are just arrays

\$₀
A₂
A₁
A₀
B₀
N₁
N₀

F

A₂
N₁
N₀
B₀
\$₀
A₁
A₀

L=BWT

Reversing a BWT

1. Start from first row ($\$$ in F , by construction)

$\$$ ₀

A₂

A₁

A₀

B₀

N₁

N₀

F

A₂

N₁

N₀

B₀

$\$$ ₀

A₁

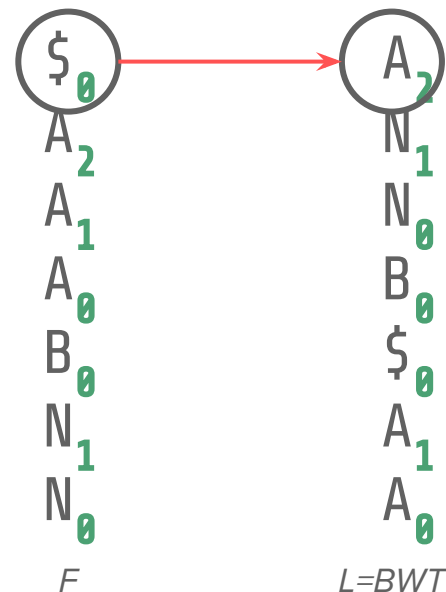
A₀

$L=BWT$

$\$$ ₀

Reversing a BWT

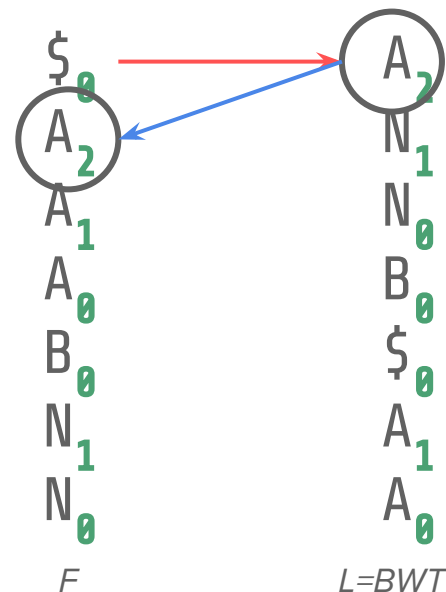
1. Start from first row (\$ in F , by construction)
2. **Move** to L , it contains the character preceding \$ (by construction)



$A2 \$0$

Reversing a BWT

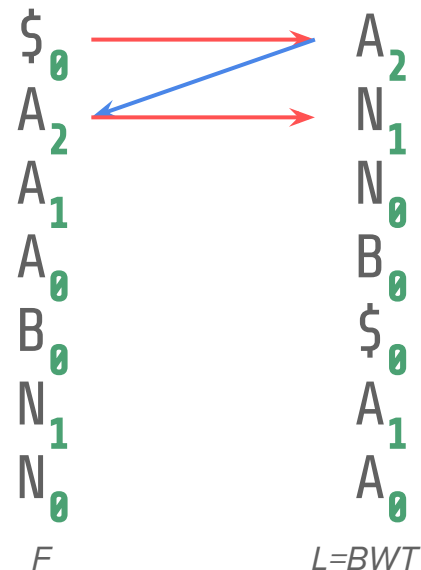
1. Start from first row (\$ in F, by construction)
2. **Move** to L, it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)



A₂\$₀

Reversing a BWT

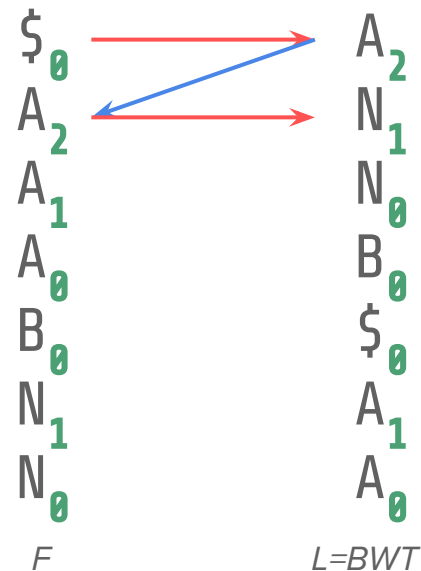
1. Start from first row (\$ in F , by construction)
2. **Move** to L , it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)
4. **Move** to L , it contains the preceding character



N₁ A₂ \$₀

Reversing a BWT

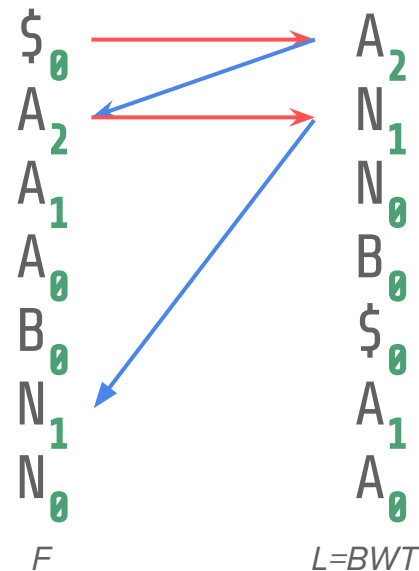
1. Start from first row (\$ in F, by construction)
2. **Move** to L, it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching \$ in L



N₁ A₂ \$₀

Reversing a BWT

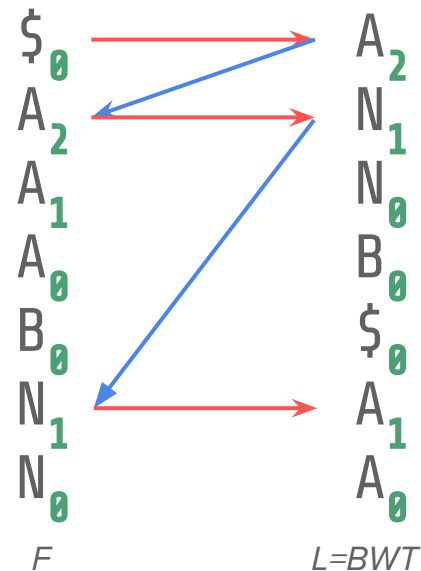
1. Start from first row (\$ in F, by construction)
2. **Move** to L, it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching \$ in L



N₁ A₂ \$₀

Reversing a BWT

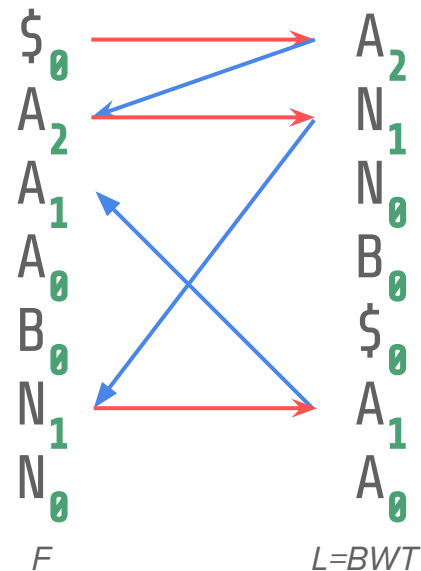
1. Start from first row (\$ in F, by construction)
2. **Move** to L, it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching \$ in L



A₁ N₁ A₂ \$₀

Reversing a BWT

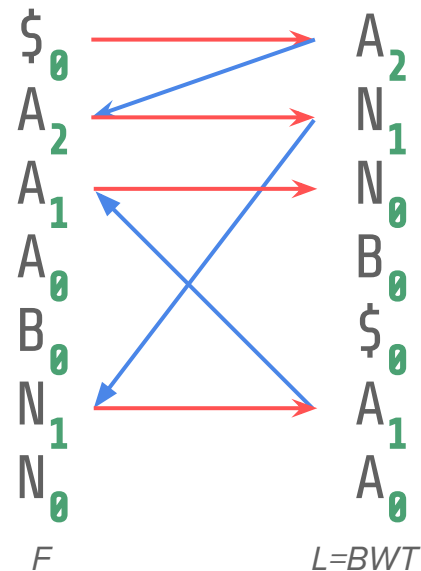
1. Start from first row (\$ in F, by construction)
2. **Move** to L, it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching \$ in L



A₁ N₁ A₂ \$₀

Reversing a BWT

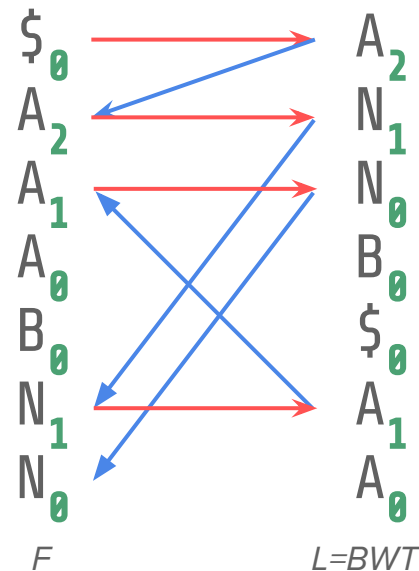
1. Start from first row (\$ in F, by construction)
2. **Move** to L, it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching \$ in L



N₀ A₁ N₁ A₂ \$₀

Reversing a BWT

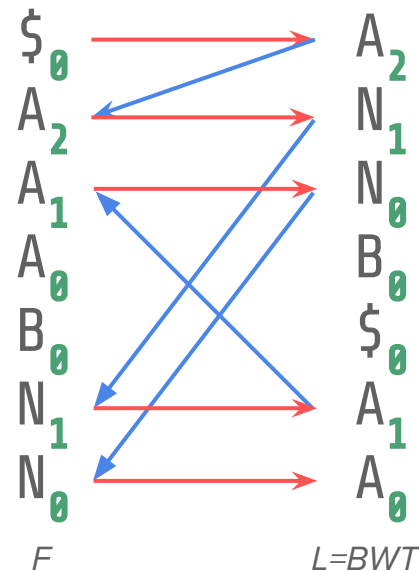
1. Start from first row (\$ in F, by construction)
2. **Move** to L, it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching \$ in L



N₀ A₁ N₁ A₂ \$₀

Reversing a BWT

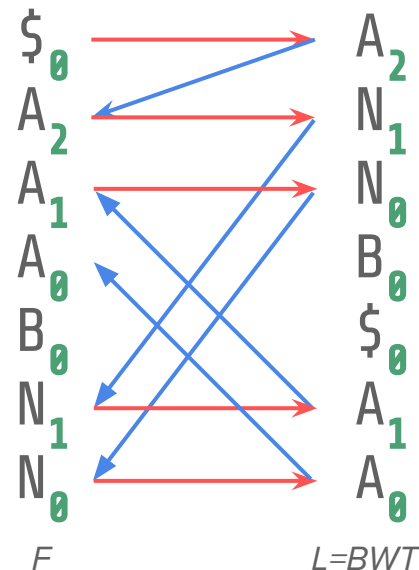
1. Start from first row (\$ in F, by construction)
2. **Move** to L, it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching \$ in L



A₀ N₀ A₁ N₁ A₂ \$₀

Reversing a BWT

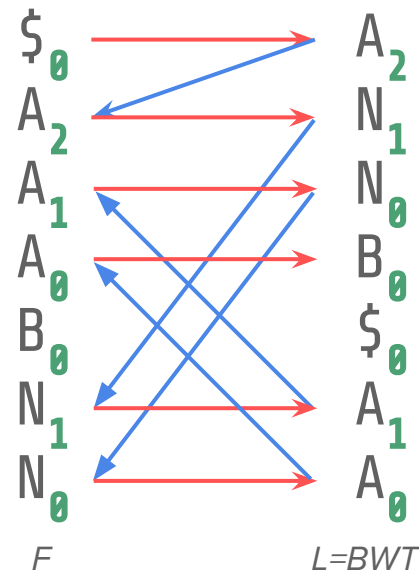
1. Start from first row (\$ in F, by construction)
2. **Move** to L, it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching \$ in L



A₀ N₀ A₁ N₁ A₂ \$₀

Reversing a BWT

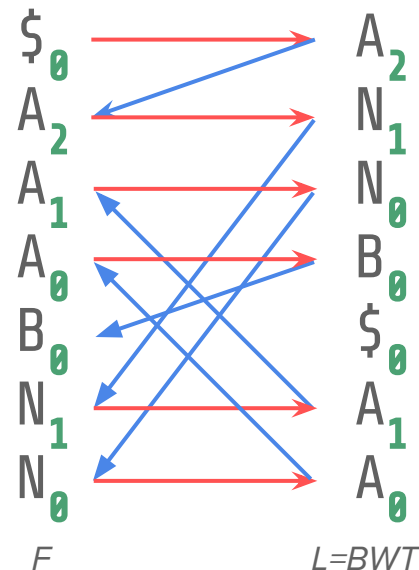
1. Start from first row (\$ in F, by construction)
2. **Move** to L, it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching \$ in L



B₀ A₀ N₀ A₁ N₁ A₂ \$₀

Reversing a BWT

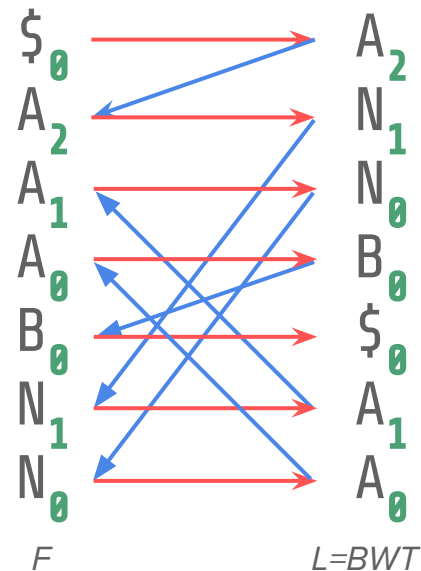
1. Start from first row (\$ in F, by construction)
2. **Move** to L, it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching \$ in L



B₀ A₀ N₀ A₁ N₁ A₂ \$₀

Reversing a BWT

1. Start from first row (\$ in F, by construction)
2. **Move** to L, it contains the character preceding \$ (by construction)
3. **Jump** to F using LF-Mapping (same rank)
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching \$ in L



B₀ A₀ N₀ A₁ N₁ A₂ \$₀

Pattern Matching

Pattern Matching

Given a text T and a pattern P , find P in T

Pattern Matching

Given a text T and a pattern P , find P in T

Three queries:

Pattern Matching

Given a text T and a pattern P , find P in T

Three queries:

- **exist:** *does P occur in T ?* *Yes/no*

Pattern Matching

Given a text T and a pattern P , find P in T

Three queries:

- **exist:** *does P occur in T ?* Yes/no
- **count:** *how many times does P occur in T ?* 3

Pattern Matching

Given a text T and a pattern P , find P in T

Three queries:

- **exist:** *does P occur in T ?* *Yes/no*
- **count:** *how many times does P occur in T ?* *3*
- **locate:** *where does P occur in T ?* *Positions 2 and 5*

Solutions

Naive solution:

$$|T| = n$$

$$|P| = m$$

Advanced algorithms:

Index-based algorithms *(very useful in bioinformatics)*

Solutions

Naive solution:

$$|T| = n$$

$$|P| = m$$

- check for P at every position in T $O(n*m)$

Advanced algorithms:

Index-based algorithms (*very useful in bioinformatics*)

Solutions

Naive solution:

$$|T| = n$$

$$|P| = m$$

- check for P at every position in T $O(n*m)$

Advanced algorithms:

- Knuth-Morris-Pratt $O(n + m)$
- Boyer-Moore $O(n/m)$ on average, $O(n*m)$ in worst case
- Rabin-Karp $O(n + m)$ on average, $O(n*m)$ worst case

Index-based algorithms (*very useful in bioinformatics*)

Solutions

Naive solution:

$$|T| = n$$

$$|P| = m$$

- check for P at every position in T $O(n*m)$

Advanced algorithms:

- Knuth-Morris-Pratt $O(n + m)$
- Boyer-Moore $O(n/m)$ on average, $O(n*m)$ in worst case
- Rabin-Karp $O(n + m)$ on average, $O(n*m)$ worst case

Index-based algorithms (*very useful in bioinformatics*)

- FM-Index (BWT-based) $O(n)$ for construction (one time expense), $O(m)$ for matching*

*locating requires a more complete analysis

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$	B	
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A

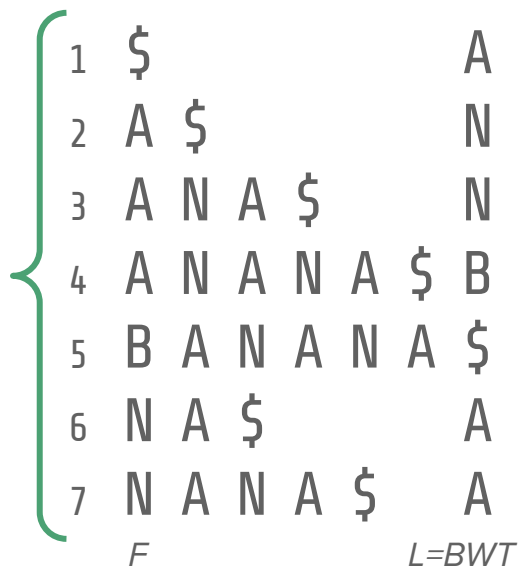
F $L=BWT$

A step back: Suffix Array

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

What are these?



1	\$	A
2	A \$	N
3	A N A \$	N
4	A N A N A \$	B
5	B A N A N A \$	
6	N A \$	A
7	N A N A \$	A

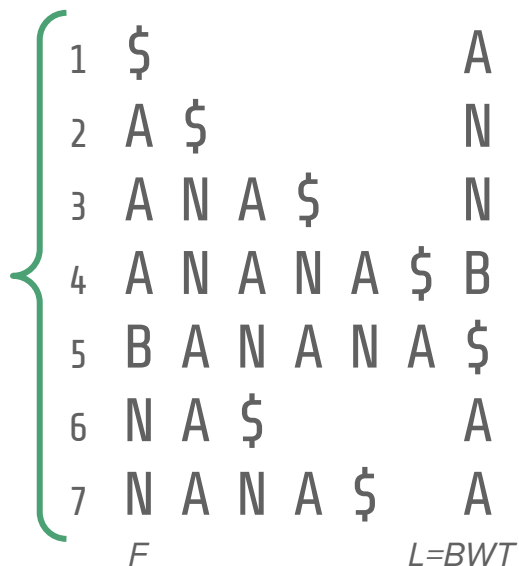
F *L=BWT*

A step back: Suffix Array

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

What are these?
Lexicographically ordered
suffixes



1	\$						A
2	A	\$					N
3	A	N	A	\$			N
4	A	N	A	N	A	\$	B
5	B	A	N	A	N	A	\$
6	N	A	\$				A
7	N	A	N	A	\$		A

F L=BWT

A step back: Suffix Array

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

What are these?
*Lexicographically ordered
suffixes*

1	\$						A
2	A	\$					N
3	A	N	A	\$			N
4	A	N	A	N	A	\$	B
5	B	A	N	A	N	A	\$
6	N	A	\$				A
7	N	A	N	A	\$		A

F *L=BWT*

B	A	N	A	N	A	\$
<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>

A step back: Suffix Array

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

What are these?
Lexicographically ordered
suffixes

6	1	\$					A	
	2	A	\$				N	
	3	A	N	A	\$		N	
	4	A	N	A	N	A	\$	B
	5	B	A	N	A	N	A	\$
	6	N	A	\$				A
	7	N	A	N	A	\$		A

F *L=BWT*

B	A	N	A	N	A	\$
0	1	2	3	4	5	6

A step back: Suffix Array

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

What are these?
Lexicographically ordered
suffixes

6	1	\$						A
5	2	A	\$					N
	3	A	N	A	\$			N
	4	A	N	A	N	A	\$	B
	5	B	A	N	A	N	A	\$
	6	N	A	\$				A
	7	N	A	N	A	\$		A

F L=BWT

B	A	N	A	N	A	\$
0	1	2	3	4	5	6

A step back: Suffix Array

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

What are these?
Lexicographically ordered
suffixes

6	1	\$						A
5	2	A	\$					N
3	3	A	N	A	\$			N
	4	A	N	A	N	A	\$	B
	5	B	A	N	A	N	A	\$
	6	N	A	\$				A
	7	N	A	N	A	\$		A
		F						L=BWT

B	A	N	A	N	A	\$
0	1	2	3	4	5	6

A step back: Suffix Array

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

What are these?
Lexicographically ordered
suffixes



Suffix Array

6	1	\$						A
5	2	A	\$					N
3	3	A	N	A	\$			N
1	4	A	N	A	N	A	\$	B
0	5	B	A	N	A	N	A	\$
4	6	N	A	\$				A
2	7	N	A	N	A	\$		A

F

L=BWT

B	A	N	A	N	A	\$
0	1	2	3	4	5	6

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

B-interval:

BAN-interval:

A-interval:

NA-interval:

1	\$						A
2	A	\$					N
3	A	N	A	\$			N
4	A	N	A	N	A	\$	B
5	B	A	N	A	N	A	\$
6	N	A	\$				A
7	N	A	N	A	\$		A
	<i>F</i>						<i>L=BWT</i>

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

B-interval: [5,5]

BAN-interval:

A-interval:

NA-interval:

1	\$						A
2	A	\$					N
3	A	N	A	\$			N
4	A	N	A	N	A	\$	B
5	B	A	N	A	N	A	\$
6	N	A	\$				A
7	N	A	N	A	\$		A
	F						L=BWT

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

B-interval: [5,5]

BAN-interval: [5,5]

A-interval:

NA-interval:

1	\$						A
2	A	\$					N
3	A	N	A	\$			N
4	A	N	A	N	A	\$	B
5	B	A	N	A	N	A	\$
6	N	A	\$				A
7	N	A	N	A	\$		A
	F						L=BWT

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

B-interval: [5,5]

BAN-interval: [5,5]

A-interval: [2,4]

NA-interval:

1	\$						A
2	A	\$					N
3	A	N	A	\$			N
4	A	N	A	N	A	\$	B
5	B	A	N	A	N	A	\$
6	N	A	\$				A
7	N	A	N	A	\$		A
	F						L=BWT

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

B-interval: [5,5]

BAN-interval: [5,5]

A-interval: [2,4]

NA-interval: [6,7]

1	\$						A
2	A	\$					N
3	A	N	A	\$			N
4	A	N	A	N	A	\$	B
5	B	A	N	A	N	A	\$
6	N	A	\$				A
7	N	A	N	A	\$		A
	F						L=BWT

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

B-interval: [5,5]

BAN-interval: [5,5]

A-interval: [2,4]

NA-interval: [6,7]

1	\$						A
2	A	\$					N
3	A	N	A	\$			N
4	A	N	A	N	A	\$	B
5	B	A	N	A	N	A	\$
6	N	A	\$				A
7	N	A	N	A	\$		A
	<i>F</i>				<i>L=BWT</i>		

A-interval = [2,4]



NA-interval = [6,7]

?

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

1	\$ ₀							A ₀
2	A ₀	\$						N ₀
3	A ₁	N	A	\$				N ₁
4	A ₂	N	A	N	A	\$	B ₀	
5	B ₀	A	N	A	N	A	\$ ₀	
6	N ₀	A	\$					A ₁
7	N ₁	A	N	A	\$			A ₂
	F							L=BWT

A-interval = [2,4]



NA-interval = [6,7]

?

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$		B
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A
	F							L=BWT

A-interval = [2,4]



NA-interval = [6,7]

?

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$		B
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A
	F							L=BWT

Diagram illustrating the BWT matrix. The matrix is a 7x9 grid. The first column is labeled 'F' and the last column is labeled 'L=BWT'. The rows are numbered 1 to 7. A red dashed box highlights the interval [2,4] in the first column, which corresponds to the string 'ANA'. A red bracket on the left side of the matrix indicates the interval [2,4] in the first column. A green bracket on the right side of the matrix indicates the interval [6,7] in the last column. The LF-mapping is shown by a red arrow pointing from the interval [2,4] in the first column to the interval [6,7] in the last column.

A-interval = [2,4]



NA-interval = [6,7]

?

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$	B	
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A

F *L=BWT*

Diagram details: A red bracket on the left groups rows 2, 3, and 4. A red dashed box encloses rows 2 through 4. Green numbers 0, 1, 2 are written below the first column (F) for rows 1, 3, and 4 respectively. A blue arrow labeled "find N" points from row 4 to row 3. Green numbers 0, 1, 2 are written below the last column (L=BWT) for rows 1, 3, and 4 respectively.

A-interval = [2,4]



NA-interval = [6,7]

?

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)

1	\$	0						A	0
2	A	0	\$					N	0
3	A	1	N	A	\$			N	1
4	A	2	N	A	N	A	\$	B	0
5	B	0	A	N	A	N	A	\$	0
6	N	0	A	\$				A	1
7	N	1	A	N	A	\$		A	2

F $L=BWT$

A-interval = [2,4]



NA-interval = [6,7]

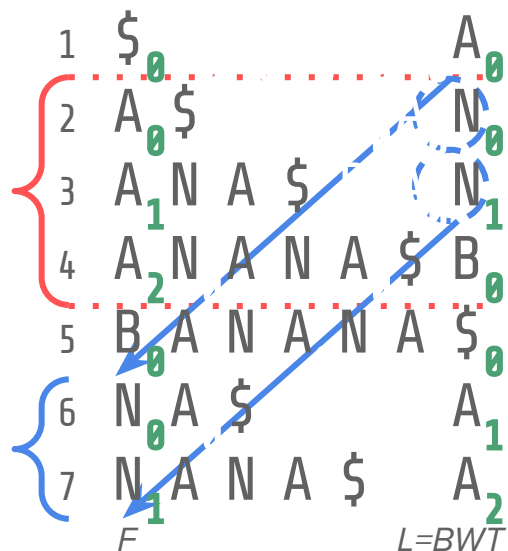
?

same character in L are not always contiguous

Pattern Matching with the BWT

Algorithm is based on:

- **Q-intervals:** intervals on the F column referring to string Q
- **LF-mapping:** how to obtain cQ-interval from Q-interval (*backward extension*)



A-interval = [2,4]



NA-interval = [6,7]

3

*same character in L are not always contiguous
but thanks to LF-mapping, they are on F*

Pattern Matching with the BWT - Backward search

We can search a pattern P via LPI backward extensions

Pattern Matching with the BWT - Backward search

We can search a pattern P via $|P|$ backward extensions

$P = BANA$

1	\$								A
	0								0
2	A	\$							N
	0								0
3	A	N	A	\$					N
	1								1
4	A	N	A	N	A	\$			B
	2								0
5	B	A	N	A	N	A	\$		\$
	0								0
6	N	A	\$						A
	0								1
7	N	A	N	A	\$				A
	1								2

F $L=BWT$

Pattern Matching with the BWT - Backward search

We can search a pattern P via LPI backward extensions

$P = BAN\underline{A}$

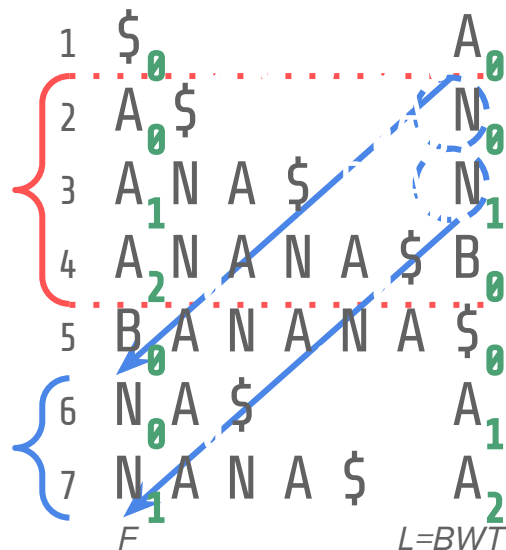
1	\$	₀						A	₀
2	A	₀	\$					N	₀
3	A	₁	N	A	\$			N	₁
4	A	₂	N	A	N	A	\$	B	₀
5	B	₀	A	N	A	N	A	\$	₀
6	N	₀	A	\$				A	₁
7	N	₁	A	N	A	\$		A	₂
	F							$L=BWT$	

$A\text{-interval} = [2,4]$

Pattern Matching with the BWT - Backward search

We can search a pattern P via IPI backward extensions

$P = BA\textcolor{green}{NA}$

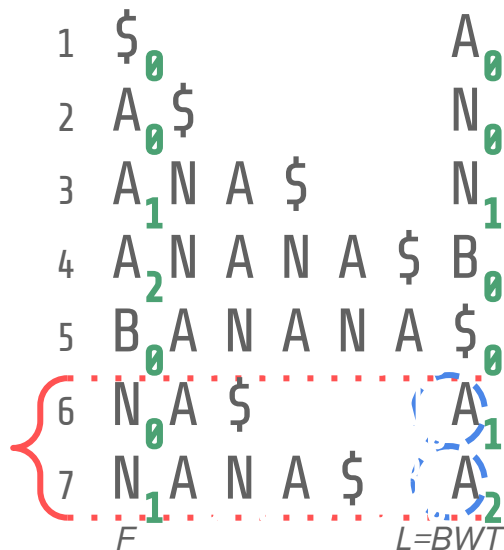


$A\text{-interval} = [2,4] \Rightarrow NA\text{-interval} = [6,7]$

Pattern Matching with the BWT - Backward search

We can search a pattern P via IPI backward extensions

$P = B$ ANA

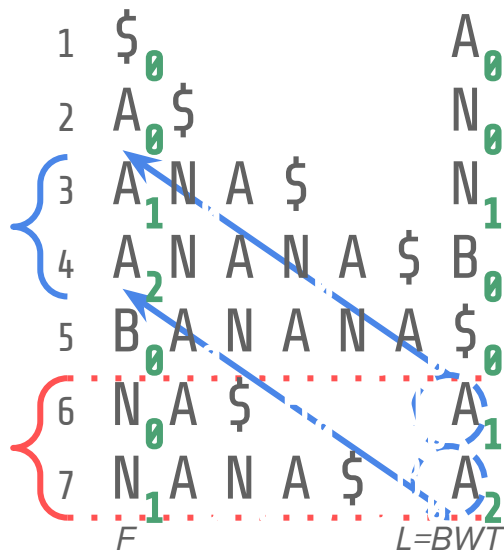


$A\text{-interval} = [2,4] \Rightarrow NA\text{-interval} = [6,7]$

Pattern Matching with the BWT - Backward search

We can search a pattern P via IPI backward extensions

$P = B\text{ANA}$

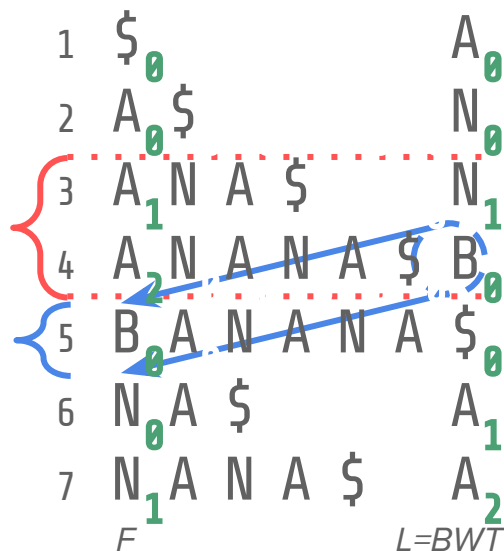


$A\text{-interval} = [2,4] \Rightarrow NA\text{-interval} = [6,7] \Rightarrow ANA\text{-interval} [3,4]$

Pattern Matching with the BWT - Backward search

We can search a pattern P via LPI backward extensions

$P = \underline{BANA}$



$A\text{-interval} = [2,4] \Rightarrow NA\text{-interval} = [6,7] \Rightarrow ANA\text{-interval} [3,4] \Rightarrow BANA\text{-interval} [5,5]$

Backward search - Complexity

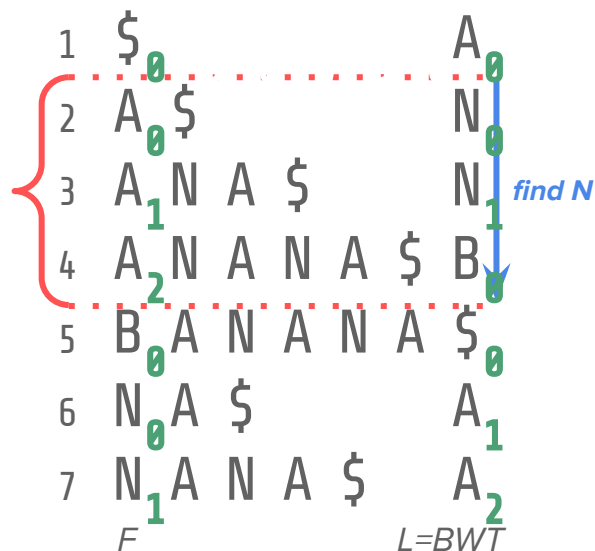
Backward search - Complexity

$O(m)$ where m is the length of pattern P

Backward search - Complexity

$O(m)$ where m is the length of pattern P

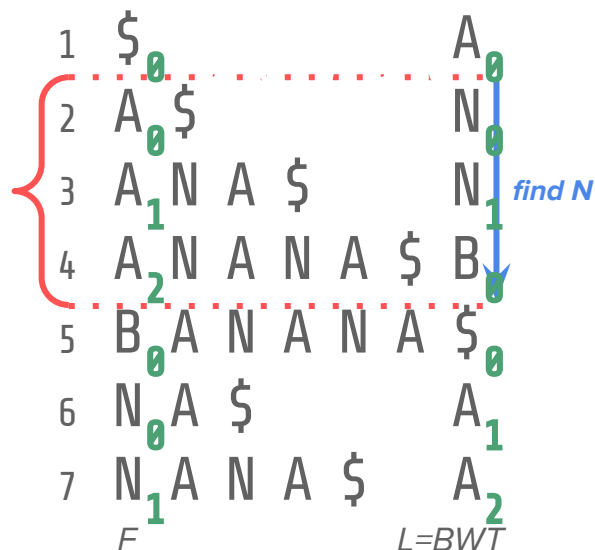
but this is true if we do not need to iterate over each interval to find the character we are interested in!



Backward search - Complexity

$O(m)$ where m is the length of pattern P

but this is true if we do not need to iterate over each interval to find the character we are interested in!

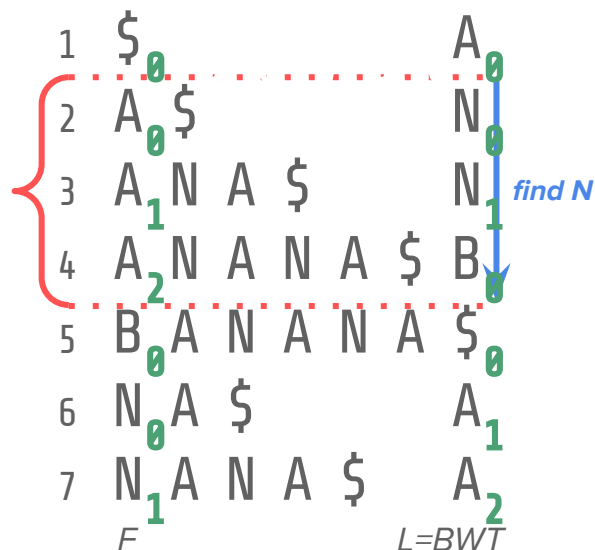


Can we do this in $O(1)$?

Backward search - Complexity

$O(m)$ where m is the length of pattern P

but this is true if we do not need to iterate over each interval to find the character we are interested in!



Can we do this in $O(1)$?
Yes!

FM-Index

- Full-text index combining the BWT with auxiliary data structures
 - efficient indexing
 - efficient querying
 - “store” full input
- **Main idea:** represent F and L in an efficient and compact way
- Potentially very space-efficient (*implementation-dependent*)

Efficient backward extension (FM-Index)

Only things we need for backward extensions/search are F and L columns,
but we can represent them in a more convenient way

Array C with “cumulative
counts” of smaller symbols
for each $c \in \{\$, \} \cup \Sigma$

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$	B	
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A
	F							L=BWT

C follows lexicographic order

Efficient backward extension (FM-Index)

Only things we need for backward extensions/search are F and L columns,
but we can represent them in a more convenient way

Array C with “cumulative counts” of smaller symbols
for each $c \in \{\$, \} \cup \Sigma$

$C = [\quad]$



How many symbols we
have smaller than \$?

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$	B	
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A
	F							L=BWT

C follows lexicographic order

Efficient backward extension (FM-Index)

Only things we need for backward extensions/search are F and L columns,
but we can represent them in a more convenient way

Array C with “cumulative counts” of smaller symbols
for each $c \in \{\$, \} \cup \Sigma$

$C = [0, \quad]$



How many symbols we
have smaller than \$?

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$	B	
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A
	F							L=BWT

C follows lexicographic order

Efficient backward extension (FM-Index)

Only things we need for backward extensions/search are F and L columns,
but we can represent them in a more convenient way

Array C with “cumulative counts” of smaller symbols
for each $c \in \{\$, \} \cup \Sigma$

$C = [0, \quad]$

How many symbols we
have smaller than A?

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$	B	
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A
	F							L=BWT

C follows lexicographic order

Efficient backward extension (FM-Index)

Only things we need for backward extensions/search are F and L columns,
but we can represent them in a more convenient way

Array C with “cumulative counts” of smaller symbols
for each $c \in \{\$, \} \cup \Sigma$

$$C = [0, \mathbf{1}, \dots]$$

How many symbols we
have smaller than A?

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$	B	
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A
	F							L=BWT

C follows lexicographic order

Efficient backward extension (FM-Index)

Only things we need for backward extensions/search are F and L columns,
but we can represent them in a more convenient way

Array C with “cumulative counts” of smaller symbols
for each $c \in \{\$, \} \cup \Sigma$

$C = [0, 1, \dots]$

How many symbols we
have smaller than B?

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$	B	
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A
	F							L=BWT

C follows lexicographic order

Efficient backward extension (FM-Index)

Only things we need for backward extensions/search are F and L columns,
but we can represent them in a more convenient way

Array C with “cumulative counts” of smaller symbols
for each $c \in \{\$, \} \cup \Sigma$

$C = [0, 1, 4, \dots]$

How many symbols we
have smaller than B?

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$	B	
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A
	F							L=BWT

C follows lexicographic order

Efficient backward extension (FM-Index)

Only things we need for backward extensions/search are F and L columns,
but we can represent them in a more convenient way

Array C with “cumulative counts” of smaller symbols
for each $c \in \{\$, \} \cup \Sigma$

$C = [0, 1, 4, \dots]$

How many symbols we
have smaller than N?

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$	B	
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A
	F							L=BWT

C follows lexicographic order

Efficient backward extension (FM-Index)

Only things we need for backward extensions/search are F and L columns,
but we can represent them in a more convenient way

Array C with “cumulative counts” of smaller symbols
for each $c \in \{\$, \} \cup \Sigma$

$C = [0, 1, 4, \mathbf{5}]$

How many symbols we
have smaller than N?

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$	B	
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A
	F							L=BWT

C follows lexicographic order

Efficient backward extension (FM-Index)

Only things we need for backward extensions/search are F and L columns,
but we can represent them in a more convenient way

Array C with “cumulative counts” of smaller symbols
for each $c \in \{\$, \} \cup \Sigma$

$$C = [0, 1, 4, 5]$$

1	\$							A
2	A	\$						N
3	A	N	A	\$				N
4	A	N	A	N	A	\$	B	
5	B	A	N	A	N	A	\$	
6	N	A	\$					A
7	N	A	N	A	\$			A
	F							L=BWT

Rank matrix Occ

Rank Matrix Occ

Occ is a matrix $|\Sigma| \times |T|$ that stores for each position i on $\text{BWT}(T)$ and for each character $c \in \Sigma$, the counts the occurrences of c in the first i elements of $\text{BWT}(T)$

	BWT	\$	A	B	N
1	A				
2	N				
3	N				
4	B				
5	\$				
6	A				
7	A				

Rank Matrix Occ

Occ is a matrix $|\Sigma| \times |T|$ that stores for each position i on $\text{BWT}(T)$ and for each character $c \in \Sigma$, the counts the occurrences of c in the first i elements of $\text{BWT}(T)$

	BWT	\$	A	B	N
1	A	0	1	0	0
2	N	0	1	0	1
3	N	0	1	0	2
4	B	0	1	1	2
5	\$	1	1	1	2
6	A	1	2	1	2
7	A	1	3	1	2

How to backward extend using C and Occ?

Given Q-interval $[i,j]$ and symbol c , return c Q-interval if it exists, empty interval otherwise

```
def backwardExtend (c, [i, j]):  
    i = C[c] + Occ(c, i - 1) + 1  
    j = C[c] + Occ(c, j)  
    return [i, j]
```

Efficient Backward Extension

```
def backwardExtend (c, [i, j]):
    i = C[c] + Occ(c, i - 1) + 1
    j = C[c] + Occ(c, j)
    return [i, j]
```

1	\$ ₀						A ₀
2	A ₀	\$					N ₀
3	A ₁	N	A	\$			N ₁
4	A ₂	N	A	N	A	\$	B ₀
5	B ₀	A	N	A	N	A	\$ ₀
6	N ₀	A	\$				A ₁
7	N ₁	A	N	A	\$		A ₂

F L=BWT

	BWT	\$	A	B	N
1	A	0	1	0	0
2	N	0	1	0	1
3	N	0	1	0	2
4	B	0	1	1	2
5	\$	1	1	1	2
6	A	1	2	1	2
7	A	1	3	1	2
C		0	1	4	5

Efficient Backward Extension

```
def backwardExtend (c, [i, j]):
    i = C[c] + Occ(c, i - 1) + 1
    j = C[c] + Occ(c, j)
    return [i, j]
```

1	\$ ₀						A ₀
2	A ₀	\$					N ₀
3	A ₁	N	A	\$			N ₁
4	A ₂	N	A	N	A	\$	B ₀
5	B ₀	A	N	A	N	A	\$ ₀
6	N ₀	A	\$				A ₁
7	N ₁	A	N	A	\$		A ₂

F L=BWT

	BWT	\$	A	B	N
1	A	0	1	0	0
2	N	0	1	0	1
3	N	0	1	0	2
4	B	0	1	1	2
5	\$	1	1	1	2
6	A	1	2	1	2
7	A	1	3	1	2
C		0	1	4	5

NA-interval = [6,7]



ANA-interval = [3,4] ✓

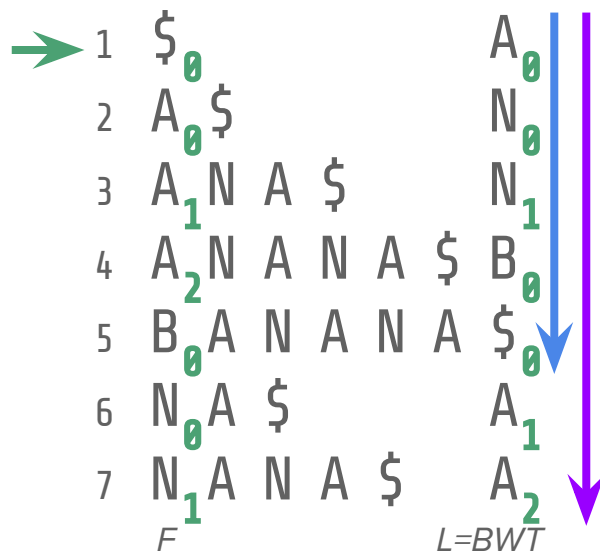
Efficient Backward Extension

```
def backwardExtend (c, [i, j]):
```

$i = C[c] + Occ(c, i - 1) + 1 = 1 + 1 + 1 = 3$

$j = C[c] + Occ(c, j) = 1 + 3 = 4$

return [i, j]



	BWT	\$	A	B	N
1	A	0	1	0	0
2	N	0	1	0	1
3	N	0	1	0	2
4	B	0	1	1	2
5	\$	1	1	1	2
6	A	1	2	1	2
7	A	1	3	1	2
c		0	1	4	5

NA-interval = [6,7]



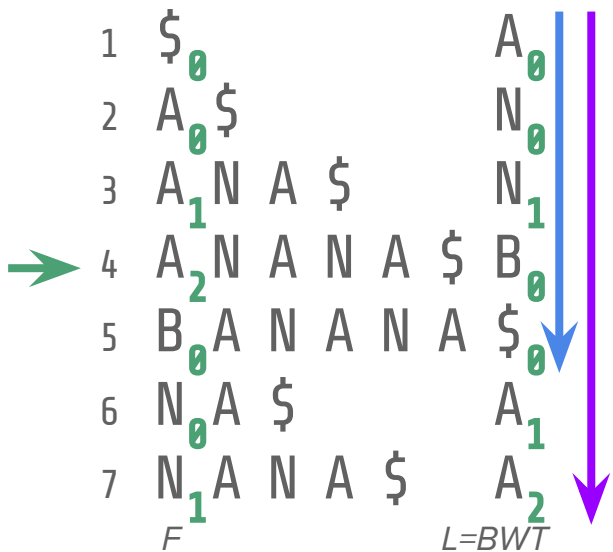
ANA-interval = [3,4] ✓

Efficient Backward Extension

```
def backwardExtend (c, [i, j]):
```

$$i = C[c] + OCC(c, i - 1) + 1 = 4 + 1 + 1 = 6$$
$$j = c[c] + \text{occ}(c, j) = 4 + 1 = 5$$

```
return [i, j]
```



	BWT	\$	A	B	N
1	A	0	1	0	0
2	N	0	1	0	1
3	N	0	1	0	2
4	B	0	1	1	2
5	\$	1	1	1	2
6	A	1	2	1	2
7	A	1	3	1	2
C		0	1	4	5

NA-interval = [6,7]



BNA-interval = [6,5] ✗

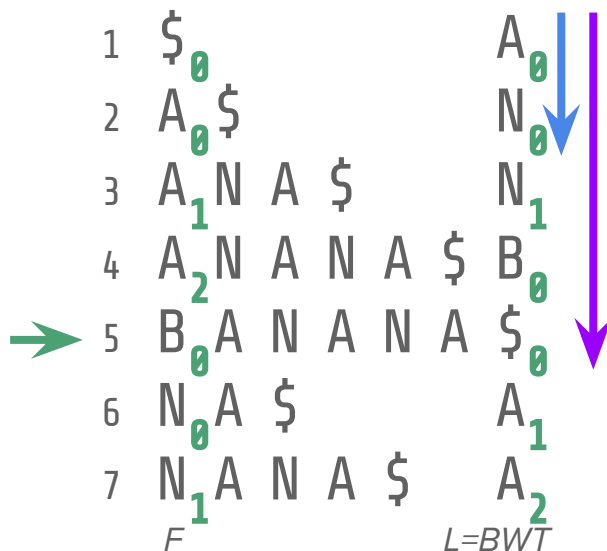
Efficient Backward Extension

```
def backwardExtend (c, [i, j]):
```

```
    i = C[c] + Occ(c, i - 1) + 1    = 5 + 1 + 1 = 7
```

```
    j = C[c] + Occ(c, j)            = 5 + 2     = 7
```

```
    return [i, j]
```



	BWT	\$	A	B	N
1	A	0	1	0	0
2	N	0	1	0	1
3	N	0	1	0	2
4	B	0	1	1	2
5	\$	1	1	1	2
6	A	1	2	1	2
7	A	1	3	1	2
C		0	1	4	5

AN-interval = [3,4]



NAN-interval = [7,7] ✓

Backward Search

Given pattern P , find it in T

```
def backwardSearch (P):  
    p = len(P)-1  
    i,j = C[P[p]], C[P[p]-1] # assuming order  
    while p >= 0 and i >= j:  
        i,j = backwardExtend(P[p], (i,j))  
        p -= 1  
    if p >= 0:  
        print("P not found")  
    else:  
        print(f"P found: {j-i+1} occurrences")
```

Not covered here: *how to locate occurrences?*

Backward Search

Given pattern P, find it in T

```
def backwardSearch (P):  
    p = len(P)-1  
    i,j = C[P[p]], C[P[p]-1] # assuming order  
    while p >= 0 and i >= j:  
        i,j = backwardExtend(P[p], (i,j))  
        p -= 1  
    if p >= 0:  
        print("P not found")  
    else:  
        print(f"P found: {j-i+1} occurrences")
```

6	1	\$						A
5	2	A	\$					N
3	3	A	N	A	\$			N
1	4	A	N	A	N	A	\$	B
0	5	B	A	N	A	N	A	\$
4	6	N	A	\$				A
2	7	N	A	N	A	\$		A

Not covered here: how to locate occurrences? Use Suffix Array (although quite expensive, $O(n\log(n))$)

Pattern matching with the FM-Index - Complexity

Query time: $O(1)$ for backward extension, $O(m)$ for backward search

Space: $O(n \cdot m)$

...but space can be reduced using advanced data structures based on **bit vectors**:

- wavelet tree
- rope

Not covered here: *how to construct BWT/FM-Index*

- $O(n^2 \log(n))$
- Vast literature on $O(n)$ approaches
- Start from Suffix Array, $O(n)$ with larger constants

+ what about approximate matches?

Bigger example

1	\$	C
2	A	G
3	A	C
4	A	C
5	C	T
6	C	G
7	C	C
8	C	A
9	C	G
10	C	G
11	C	G
12	C	\$
13	G	A
14	G	C
15	G	C
16	G	C
17	G	C
18	G	A
19	T	T
20	T	T
21	T	G

Bigger example

```
1  $CGCGCGCGCAGACCAAGTTTC
2  ACCAGTTTTC$CGCGCGCGCAG
3  AGACCAAGTTTTC$CGCGCGCGC
4  AGTTTTC$CGCGCGCGCAGACC
5  C$CGCGCGCGCAGACCAAGTTT
6  CAGACCAAGTTTTC$CGCGCGCG
7  CAGTTTTC$CGCGCGCGCAGAC
8  CCAGTTTTC$CGCGCGCGCAGA
9  CGCAGACCAAGTTTTC$CGCGCG
10 CGCGCAGACCAAGTTTTC$CGCG
11 CGCGCGCAGACCAAGTTTTC$CG
12 CGCGCGCGCAGACCAAGTTTTC$
13 GACCAAGTTTTC$CGCGCGCGCA
14 GCAGACCAAGTTTTC$CGCGCGC
15 GCGCAGACCAAGTTTTC$CGCGC
16 GCGCGCAGACCAAGTTTTC$CGC
17 GCGCGCGCAGACCAAGTTTTC$C
18 GTTTTC$CGCGCGCGCAGACCA
19 TC$CGCGCGCGCAGACCAAGTT
20 TTC$CGCGCGCGCAGACCAAGT
21 TTTTC$CGCGCGCGCAGACCAAG
```


Bigger example

1	\$CGCGCGCGCAGACCAGTTTC	
2	ACCAGTTTC\$	G
3	AGACCAGTTTC\$	C
4	AGTTTC\$	C
5	C\$	T
6	CAGACCAGTTTC\$	G
7	CAGTTTC\$	C
8	CCAGTTTC\$	A
9	CGCAGACCAGTTTC\$	G
10	CGCGCAGACCAGTTTC\$	G
11	CGCGCGCAGACCAGTTTC\$	G
12	CGCGCGCGCAGACCAGTTTC\$	
13	GACCAGTTTC\$	A
14	GCAGACCAGTTTC\$	C
15	GCGCAGACCAGTTTC\$	C
16	GCGCGCAGACCAGTTTC\$	C
17	GCGCGCGCAGACCAGTTTC\$	C
18	GTTTC\$	A
19	TC\$	T
20	TTC\$	T
21	TTTC\$	G