Burrows-Wheeler Transform

Methods in Bioinformatics (TI)

UNIVERZITA
KOMENSKÉHO
V BRATISLAVE

Luca Denti

Strings

- **Alphabet Σ**

- **String S**

String: AAGTGCTCAAAGCTAAGCTCCAT

Prefix:

Suffix:

Substring:

Strings

- **Alphabet Σ**: set of *characters* (e.g., Σ={A,C,G,T})

- **String S**: sequence of n=|S| characters drawn from Σ, i.e., S[i]∈Σ for 0≤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.

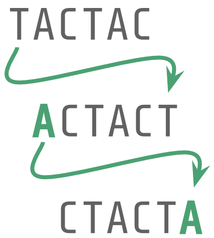
TACTAC

String Rotations

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

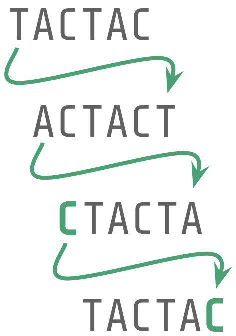
String Rotations

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



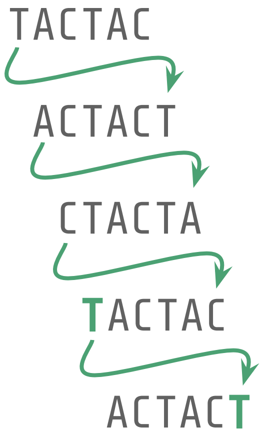
String Rotations

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.

TACTAC

ACTACT

CTACTA

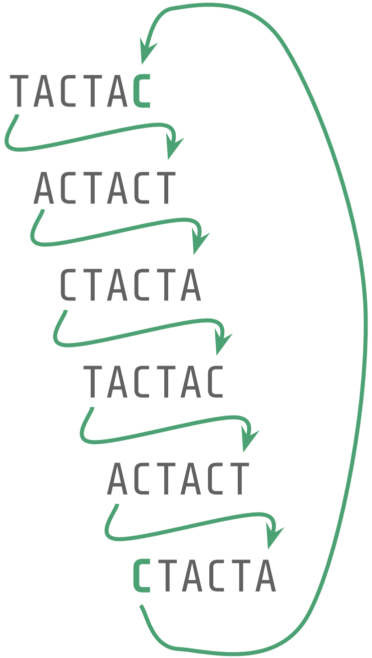
TACTAC

ACTACT

CTACT**A**

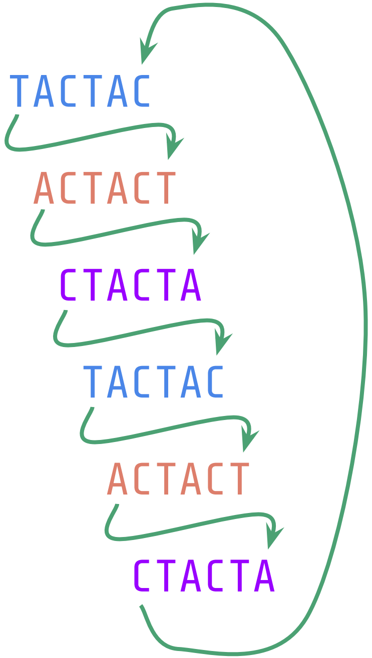
String Rotations

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 **$**:
    - $\$ \notin \Sigma$
    - $\$ < c \ \forall c \in \Sigma$
2. Append **$** to the string

Special character $

1. Define a new symbol **$**:
   - $\$ \notin \Sigma$
   - $\$ < c \ \forall c \in \Sigma$
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 **$**:
   - $\$ \notin \Sigma$
   - $\$ < c \ \forall c \in \Sigma$
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 M, Wheeler DJ: A block sorting lossless data compression algorithm.
Digital Equipment Corporation, Palo Alto, CA 1994, Technical Report 124; 1994

Burrows-Wheeler Transform (BWT)

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

(i) (ii) (iii)

Burrows M, Wheeler DJ: A block sorting lossless data compression algorithm.
Digital Equipment Corporation, Palo Alto, CA 1994, Technical Report 124; 1994

Burrows-Wheeler Transform (BWT)

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

(i) *(ii)* *(iii)*

TACTAC$

Burrows M, Wheeler DJ: A block sorting lossless data compression algorithm.
Digital Equipment Corporation, Palo Alto, CA 1994, Technical Report 124; 1994

Burrows-Wheeler Transform (BWT)

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

(i) *(ii)* *(iii)*

```
                 TACTAC$
                 ACTAC$T
                 CTAC$TA
TACTAC$ --Rotate--> TAC$TAC
                 AC$TACT
                 C$TACTA
                 $TACTAC
```

Burrows M, Wheeler DJ: A block sorting lossless data compression algorithm. Digital Equipment Corporation, Palo Alto, CA 1994, Technical Report 124; 1994

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

—Sort→

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

Burrows-Wheeler Matrix (BWM)

Burrows M, Wheeler DJ: A block sorting lossless data compression algorithm. Digital Equipment Corporation, Palo Alto, CA 1994, Technical Report 124; 1994

Burrows-Wheeler Transform (BWT)

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

(i) (ii) (iii)

Input		Rotations		Sorted (BWM)		BWT
		TACTAC\$		\$TACTAC		
		ACTAC\$T		AC\$TACT		
		CTAC\$TA		ACTAC\$T		
TACTAC\$	Rotate →	TAC\$TAC	Sort →	C\$TACTA	Get last →	CTTAAC\$
		AC\$TACT		CTAC\$TA		*(BWT)*
		C\$TACTA		TAC\$TAC		
		\$TACTAC		TACTAC\$		

Burrows-Wheeler Matrix
(BWM)

Burrows M, Wheeler DJ: A block sorting lossless data compression algorithm.
Digital Equipment Corporation, Palo Alto, CA 1994, Technical Report 124; 1994

Permutation

BWT permutes characters according to their right contexts

```
$TACTA C
AC$TAC T
ACTAC$ T
C$TACT A
CTAC$T A
TAC$TA C
TACTAC $
```

First BWT (Last)

```
C $TACTA
T AC$TAC
T ACTAC$
A C$TACT
A CTAC$T
C TAC$TA
$ TACTAC
```

right context

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**GGGGGGG** TGTTCAGGGTCCCCGACTCCTCCCCGCCCCTTG**CCCCCC** GGGGTCGGCA
CTAT**GGGGGGGGGG** TAGGGAAATAAT**AAAAAA** T$TAATATACATCACCACCCCAAACAAA
CCC**AAAAAAAAAAAAAA** TACAACCGAACGAGCAAC**AAAAAAAA**

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

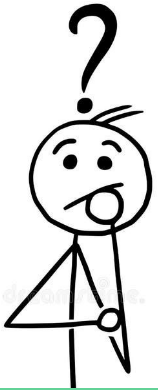
Ultrafast and memory-efficient alignment of short DNA sequences to the human genome
B Langmead, C Trapnell, M Pop, SL Salzberg - Genome biology, 2009 - Springer
… For the human genome, Burrows-Wheeler indexing allows **Bowtie** to **align** more than 25 million reads per CPU hour with a memory footprint of approximately 1.3 gigabytes. …
☆ Salva 99 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 99 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 99 Cita Citato da 50265 Articoli correlati Tutte e 21 le versioni

Reversible

$$S \Rightarrow BWT(S) \Rightarrow S$$



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)*

$B_0 A_0 N_0 A_1 N_1 A_2 \$_0$

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)*

$$
B_0A_0N_0A_1N_1A_2\$_0 \longrightarrow
\begin{array}{ccccccc}
\$_0 & B_0 & A_0 & N_0 & A_1 & N_1 & A_2 \\
A_2 & \$_0 & B_0 & A_0 & N_0 & A_1 & N_1 \\
A_1 & N_1 & A_2 & \$_0 & B_0 & A_0 & N_0 \\
A_0 & N_0 & A_1 & N_1 & A_2 & \$_0 & B_0 \\
B_0 & A_0 & N_0 & A_1 & N_1 & A_2 & \$_0 \\
N_1 & A_2 & \$_0 & B_0 & A_0 & N_0 & A_1 \\
N_0 & A_1 & N_1 & A_2 & \$_0 & B_0 & A_0
\end{array}
$$

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)*

F	L
$\$_0$	A_2
A_2	N_1
A_1	N_0
A_0	B_0
B_0	$\$_0$
N_1	A_1
N_0	A_0

We do not need the entire matrix

LF-Mapping

Let's look at F:

F	L
$\$_0$	A_2
A_2	N_1
A_1	N_0
A_0	B_0
B_0	$\$_0$
N_1	A_1
N_0	A_0

LF-Mapping

Let’s look at F:

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

$\$_0$	A_2
A_2	N_1
A_1	N_0
A_0	B_0
B_0	$\$_0$
N_1	A_1
N_0	A_0
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:

F	L
$\$_0$	A_2
A_2	N_1
A_1	N_0
A_0	B_0
B_0	$\$_0$
N_1	A_1
N_0	A_0

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

F	*L*
	A_2
A_2	
A_1	
A_0	
	A_1
	A_0

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

LF-Mapping

Let's look at F:

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

N_1
N_0

Let's look at F and L:

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

N_1
N_0

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)

F	L
$\$_0$	A_2
A_2	N_1
A_1	N_0
A_0	B_0
B_0	$\$_0$
N_1	A_1
N_0	A_0

Why does LF-Mapping hold?

$$
\begin{array}{ccccccc}
\$_0 & B & A & N & A & N & A_2 \\
A_2 & \$ & B & A & N & A & N_1 \\
A_1 & N & A & \$ & B & A & N_0 \\
A_0 & N & A & N & A & \$ & B_0 \\
B_0 & A & N & A & N & A & \$_0 \\
N_1 & A & \$ & B & A & N & A_1 \\
N_0 & A & N & A & \$ & B & A_0
\end{array}
\qquad\qquad
\begin{array}{ccccccc}
\$_0 & B & A & N & A & N & A_2 \\
A_2 & \$ & B & A & N & A & N_1 \\
A_1 & N & A & \$ & B & A & N_0 \\
A_0 & N & A & N & A & \$ & B_0 \\
B_0 & A & N & A & N & A & \$_0 \\
N_1 & A & \$ & B & A & N & A_1 \\
N_0 & A & N & A & \$ & B & A_0
\end{array}
$$

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

$\$_0$	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?

$\$_0$	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?

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?

$$
\begin{array}{ccccccc}
\$_0 & B & A & N & A & N & A_2 \\
A_2 & \$ & B & A & N & A & N_1 \\
A_1 & N & A & \$ & B & A & N_0 \\
A_0 & N & A & N & A & \$ & B_0 \\
B_0 & A & N & A & N & A & \$_0 \\
N_1 & A & \$ & B & A & N & A_1 \\
N_0 & A & N & A & \$ & B & A_0
\end{array}
$$

They are sorted by their right context

$$
\begin{array}{ccccccc}
\$_0 & B & A & N & A & N & A_2 \\
A_2 & \$ & B & A & N & A & N_1 \\
A_1 & N & A & \$ & B & A & N_0 \\
A_0 & N & A & N & A & \$ & B_0 \\
B_0 & A & N & A & N & A & \$_0 \\
N_1 & A & \$ & B & A & N & A_1 \\
N_0 & A & N & A & \$ & B & A_0
\end{array}
$$

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?

$\$_0$	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

Why does LF-Mapping hold?

Why are these As in this relative order?

$$
\begin{array}{ccccccc}
\$_0 & B & A & N & A & N & A_2 \\
A_2 & \boxed{\$ \; B \; A \; N \; A \; N_1} \\
A_1 & \boxed{N \; A \; \$ \; B \; A \; N_0} \\
A_0 & \boxed{N \; A \; N \; A \; \$ \; B_0} \\
B_0 & A & N & A & N & A & \$_0 \\
N_1 & A & \$ & B & A & N & A_1 \\
N_0 & A & N & A & \$ & B & A_0
\end{array}
$$

They are sorted by their right context

$$
\begin{array}{cccccccc}
A_2 & \boxed{\$_0 \; B \; A \; N \; A \; N} & A_2 \\
N_1 & A_2 & \$ & B & A & N & A & N_1 \\
N_0 & A_1 & N & A & \$ & B & A & N_0 \\
B_0 & A_0 & N & A & N & A & \$ & B_0 \\
\$_0 & B_0 & A & N & A & N & A & \$_0 \\
A_1 & \boxed{N_1 \; A \; \$ \; B \; A \; N} & A_1 \\
A_0 & \boxed{N_0 \; A \; N \; A \; \$ \; B} & A_0
\end{array}
$$

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

F	L=BWT
$\$_0$	A_2
A_2	N_1
A_1	N_0
A_0	B_0
B_0	$\$_0$
N_1	A_1
N_0	A_0

Reversing a BWT

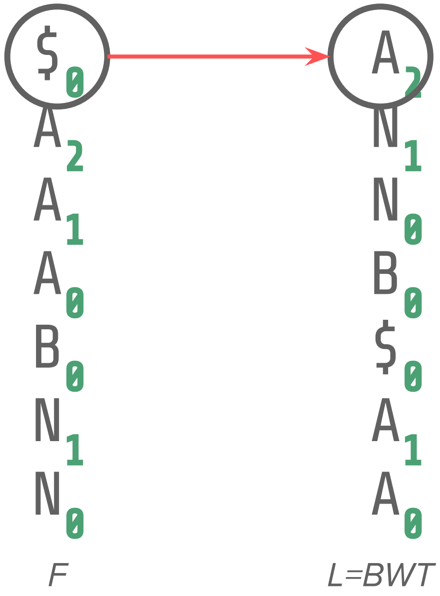
1. Start from first row *($ in F, by construction)*

F	L=BWT
$\$_0$	A_2
A_2	N_1
A_1	N_0
A_0	B_0
B_0	$\$_0$
N_1	A_1
N_0	A_0

$\$_0$

Reversing a BWT

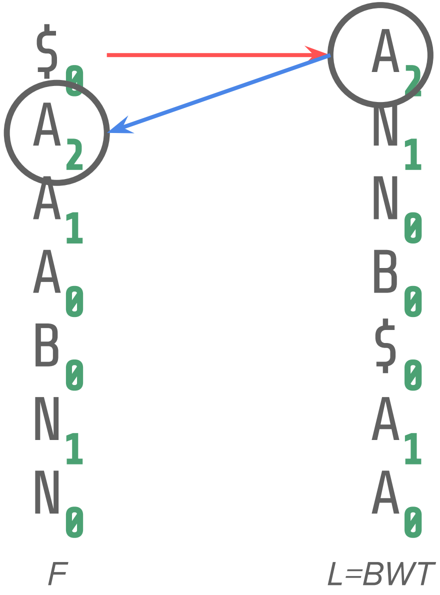
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*



$A_2\$_0$

Reversing a BWT

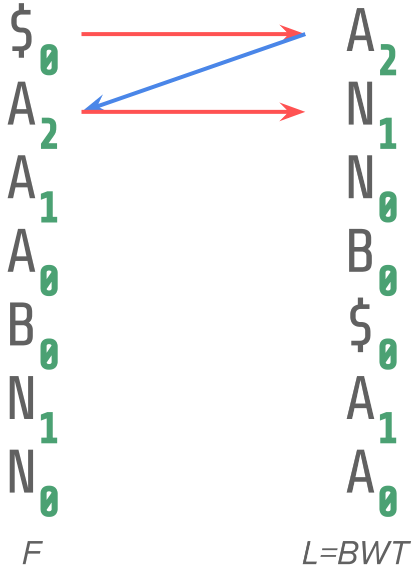
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*
3. **Jump** to F using LF-Mapping *(same rank)*



$A_2 \$_0$

Reversing a BWT

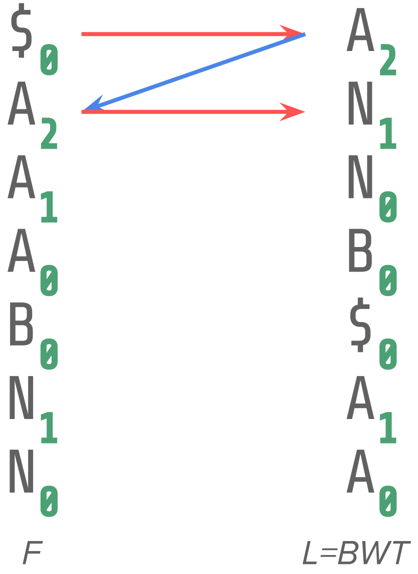
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*
3. **Jump** to F using LF-Mapping *(same rank)*
4. **Move** to L, it contains the preceding character



$N_1A_2\$_0$

Reversing a BWT

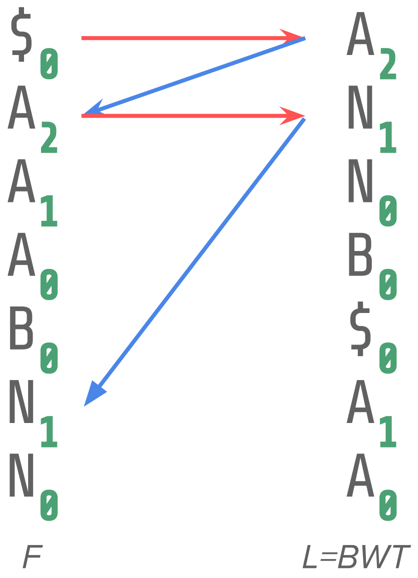
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*
3. **Jump** to F using LF-Mapping *(same rank)*
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching $ in L



$N_1 A_2 \$_0$

Reversing a BWT

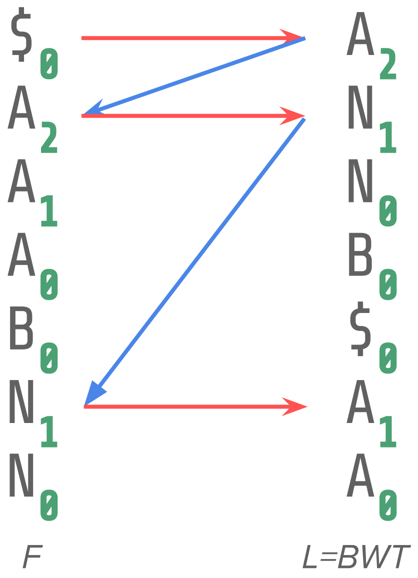
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*
3. **Jump** to F using LF-Mapping *(same rank)*
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching $ in L



$N_1 A_2 \$_0$

Reversing a BWT

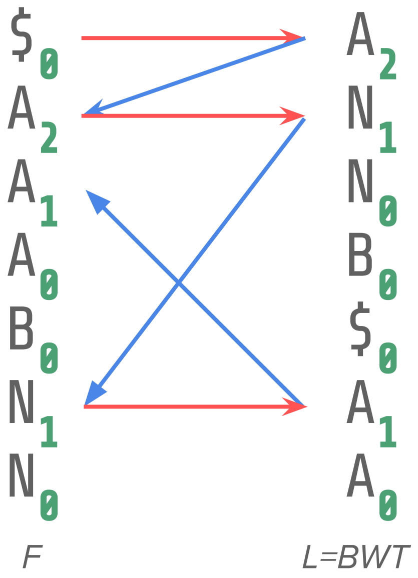
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*
3. **Jump** to F using LF-Mapping *(same rank)*
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching $ in L



$A_1 N_1 A_2 \$_0$

Reversing a BWT

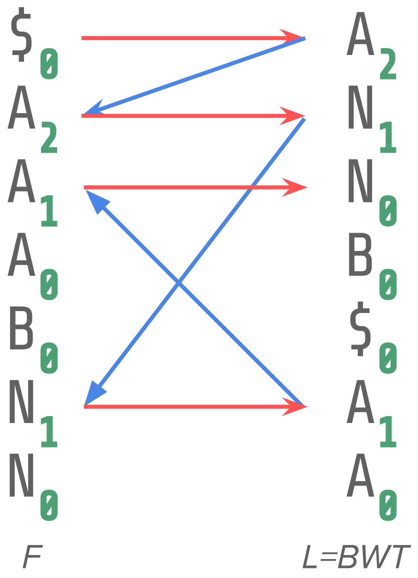
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*
3. **Jump** to F using LF-Mapping *(same rank)*
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching $ in L



$A_1N_1A_2\$_0$

Reversing a BWT

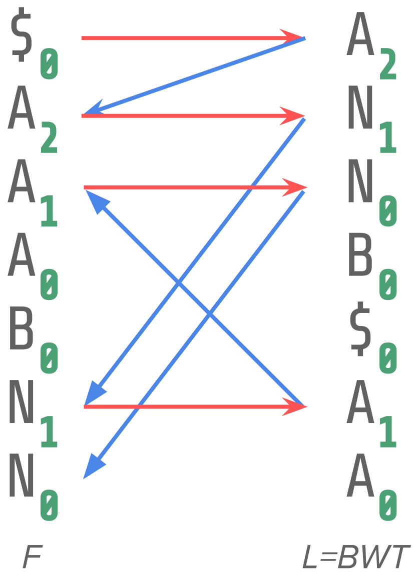
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*
3. **Jump** to F using LF-Mapping *(same rank)*
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching $ in L



N$_0$A$_1$N$_1$A$_2$$$_0$

Reversing a BWT

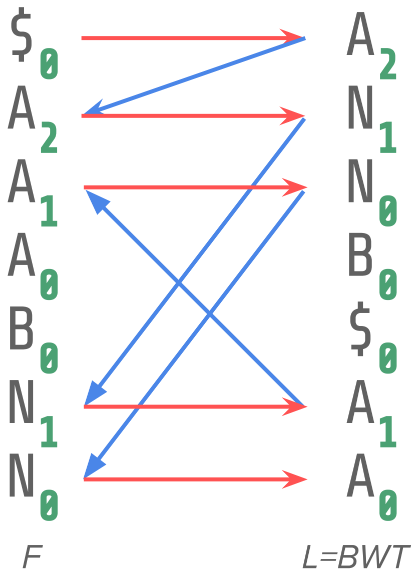
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*
3. **Jump** to F using LF-Mapping *(same rank)*
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching $ in L



$N_0A_1N_1A_2\$_0$

Reversing a BWT

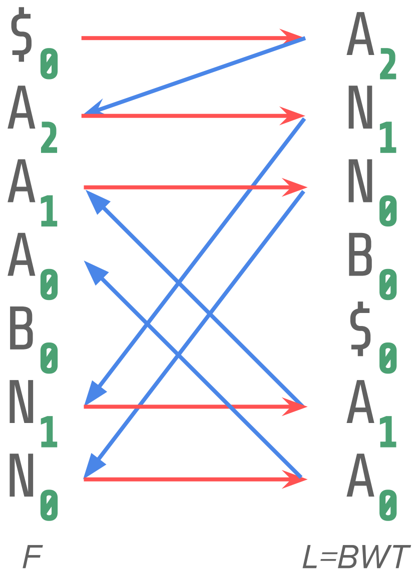
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*
3. **Jump** to F using LF-Mapping *(same rank)*
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching $ in L



$A_0N_0A_1N_1A_2\$_0$

Reversing a BWT

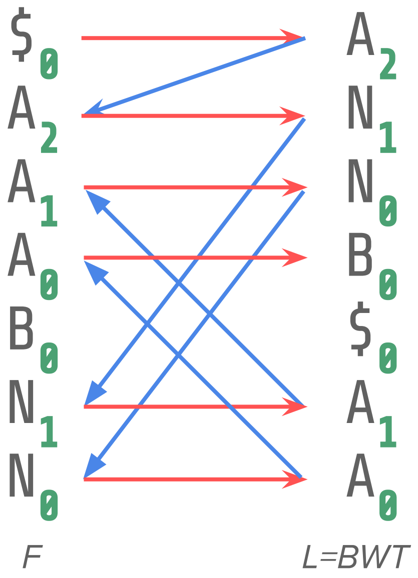
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*
3. **Jump** to F using LF-Mapping *(same rank)*
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching $ in L



$A_0N_0A_1N_1A_2\$_0$

Reversing a BWT

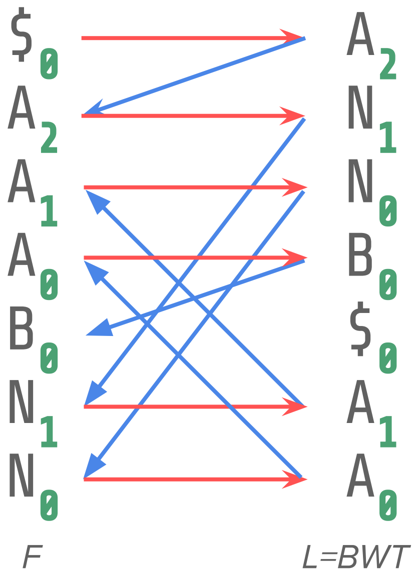
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*
3. **Jump** to F using LF-Mapping *(same rank)*
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching $ in L



$B_0A_0N_0A_1N_1A_2\$_0$

Reversing a BWT

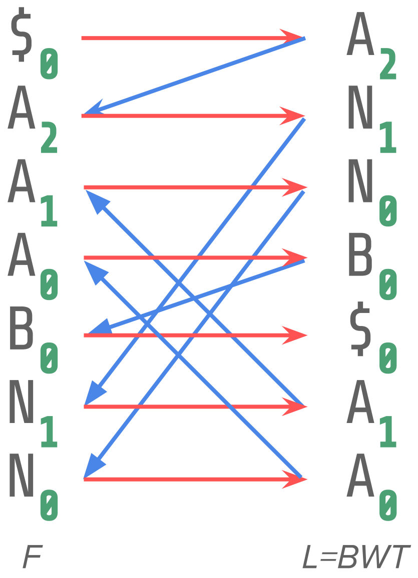
1. Start from first row *($ in F, by construction)*
2. **Move** to L, it contains the character preceding $ *(by construction)*
3. **Jump** to F using LF-Mapping *(same rank)*
4. **Move** to L, it contains the preceding character
5. Repeat 3-4 until reaching $ in L



$B_0A_0N_0A_1N_1A_2\$_0$

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



$B_0 A_0 N_0 A_1 N_1 A_2 \$_0$

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

$|T| = n$
$|P| = m$

Naive solution:

Advanced algorithms:

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

**locating requires a more complete analysis*

Solutions

$|T| = n$
$|P| = m$

Naive solution:

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

Advanced algorithms:

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

**locating requires a more complete analysis*

Solutions

$|T| = n$
$|P| = m$

Naive solution:

- 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)*

**locating requires a more complete analysis*

Solutions

$|T| = n$
$|P| = m$

Naive solution:

- 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	$
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
	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

		F		L=BWT
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

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
	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:
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:

	F						L=BWT
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

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
	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	$\$_0$						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	$\$_0$						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	$\$_0$						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]

⬇️

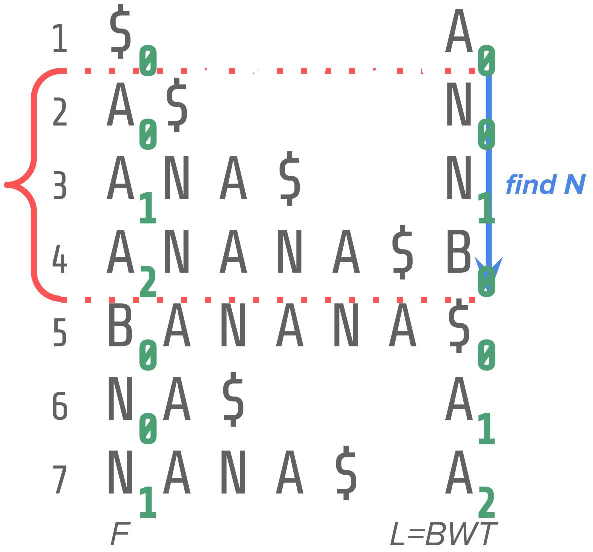
NA-interval = [6,7]

?

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)***



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]

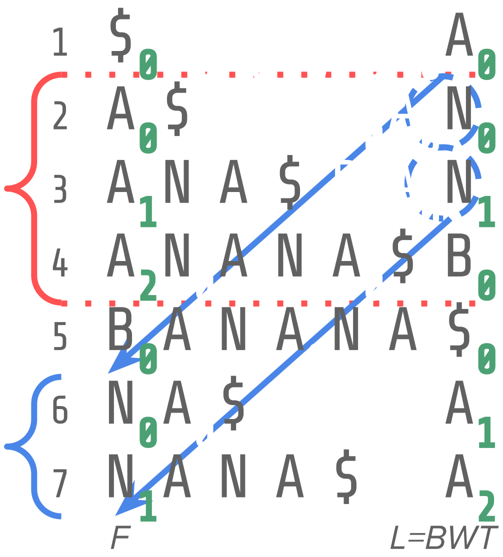
?

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]

?

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 |P| backward extensions

Pattern Matching with the BWT - Backward search

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

P = BANA

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*

Pattern Matching with the BWT - Backward search

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

*P = BAN**A***

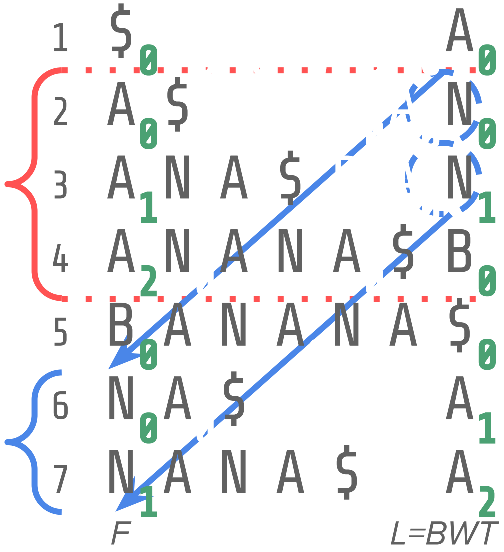
	F						L=BWT
1	$\$_0$						A_0
2	A_0	$					N_0
3	A_1	N	A	$			N_1
4	A_2	N	A	N	A	$	B_0
5	B_0	A	N	A	N	A	$\$_0$
6	N_0	A	$				A_1
7	N_1	A	N	A	$		A_2

A-interval = [2,4]

Pattern Matching with the BWT - Backward search

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

*P = BA**NA***



A-interval = [2,4] ➡ NA-interval = [6,7]

Pattern Matching with the BWT - Backward search

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

*P = B**ANA***

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 - Backward search

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

*P = B**ANA***

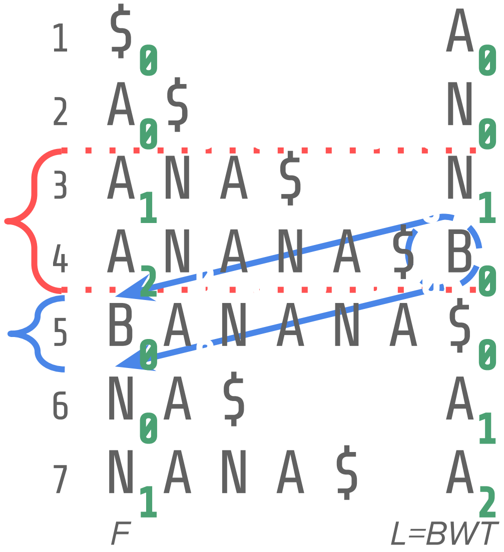
	F						L=BWT
1	$\$_0$						A_0
2	A_0	\$					N_0
3	A_1	N	A	\$			N_1
4	A_2	N	A	N	A	\$	B_0
5	B_0	A	N	A	N	A	$\$_0$
6	N_0	A	\$				A_1
7	N_1	A	N	A	\$		A_2

A-interval = [2,4] ➡ NA-interval = [6,7] ➡ ANA-interval [3,4]

Pattern Matching with the BWT - Backward search

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

P = **BANA**



A-interval = [2,4] ➡ NA-interval = [6,7] ➡ ANA-interval [3,4] ➡ BANA-interval [5,5]

Backward search - Complexity

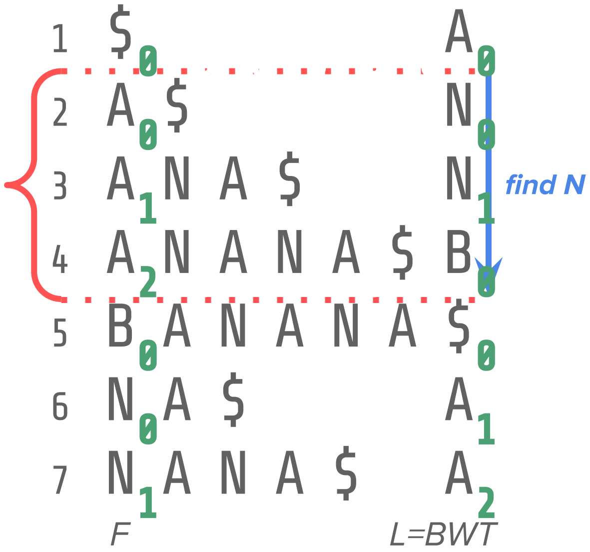
Backward search - Complexity

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

Backward search - Complexity

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

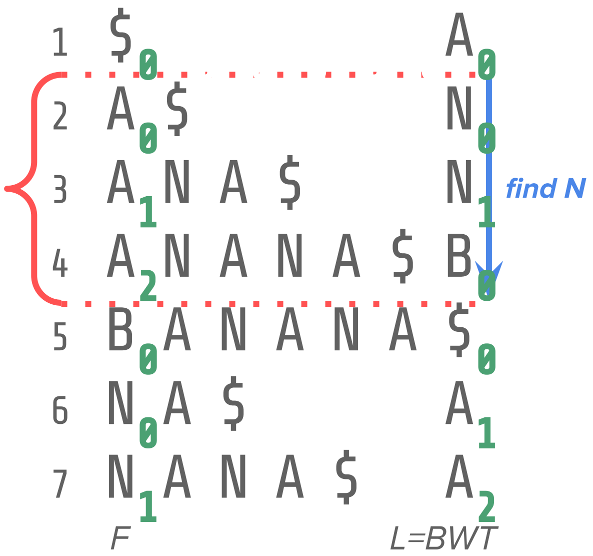
but this is true if we do not need to iterate over each interval to find the character we are interested in!



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!

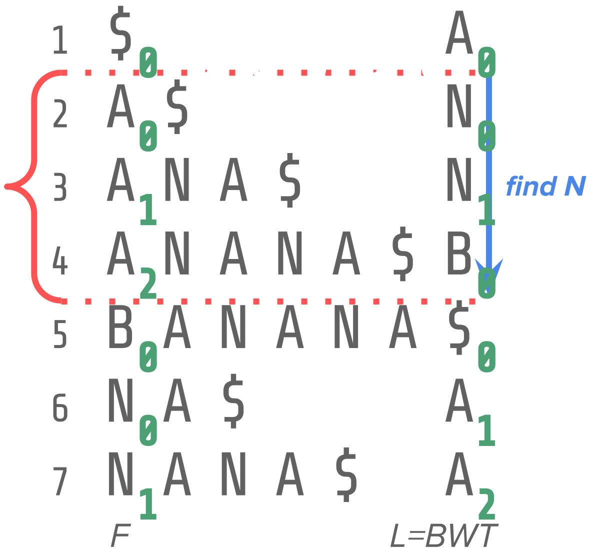


Can we do this in O(1)?

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)*

Ferragina et Manzini. Opportunistic data structures with applications. Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on. IEEE, 2000.

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∈{$} U Σ*

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*

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∈{$} U Σ*

C = [ ]

How many symbols we have smaller than $?

	F						L=BWT
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

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∈{$} U Σ*

*C = [****0,****   ]*

How many symbols we have smaller than $?

	F						L=BWT
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

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∈{$} U Σ*

C = [0,   ]

How many symbols we have smaller than 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*

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∈{$} U Σ*

C = [0, 1,]

How many symbols we have smaller than 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*

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∈{$} U Σ*

C = [0, 1,   ]

How many symbols we have smaller than B?

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*

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∈{$} U Σ*

C = [0, 1, ***4****,   ]*

How many symbols we have smaller than B?

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*

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∈{$} U Σ*

C = [0, 1, 4,   ]

How many symbols we have smaller than N?

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*

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∈{$} U Σ*

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

How many symbols we have smaller than N?

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*

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∈{$} U Σ*

C = [0, 1, 4, 5]

	F						L=BWT
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

Rank matrix Occ

Rank Matrix Occ

Occ is a matrix |Σ|⨉|T| that stores for each position i on BWT(T) and for each character c∈Σ, the counts the occurrences of c in the first i elements of 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 BWT(T) and for each character $c \in \Sigma$, the counts the occurrences of c in the first i elements of 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 cQ-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 Extenstion

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

	F						L=BWT
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

	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 Extenstion

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

	F						L=BWT
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

	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 Extenstion

```
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]
```

	F		L=BWT
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

	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 Extenstion

```
def backwardExtend (c, [i, j]):
    i = C[c] + Occ(c, i - 1) + 1     = 4 + 1 + 1   = 6
    j = C[c] + Occ(c, j)             = 4 + 1       = 5
    return [i, j]
```

	F						L=BWT
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

	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 Extenstion

```
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]
```

	F						L=BWT
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

	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

<u>Not covered here:</u> *how to locate occurrences? Use Suffix Array (although quite expensive, O(nlog(n))*

Pattern matching with the FM-Index - Complexity

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

Space: O(n*|Σ|) - *Occ matrix*

...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  $CGCGCGCGCAGACCAGTTTC
 2  ACCAGTTTC$CGCGCGCGCAG
 3  AGACCAGTTTC$CGCGCGCGC
 4  AGTTTC$CGCGCGCGCAGACC
 5  C$CGCGCGCGCAGACCAGTTT
 6  CAGACCAGTTTC$CGCGCGCG
 7  CAGTTTC$CGCGCGCGCAGAC
 8  CCAGTTTC$CGCGCGCGCAGA
 9  CGCAGACCAGTTTC$CGCGCG
10  CGCGCAGACCAGTTTC$CGCG
11  CGCGCGCAGACCAGTTTC$CG
12  CGCGCGCGCAGACCAGTTTC$
13  GACCAGTTTC$CGCGCGCGCA
14  GCAGACCAGTTTC$CGCGCGC
15  GCGCAGACCAGTTTC$CGCGC
16  GCGCGCAGACCAGTTTC$CGC
17  GCGCGCGCAGACCAGTTTC$C
18  GTTTC$CGCGCGCGCAGACCA
19  TC$CGCGCGCGCAGACCAGTT
20  TTC$CGCGCGCGCAGACCAGT
21  TTTC$CGCGCGCGCAGACCAG
```

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
```