UNIVERSITÄT
DES
SAARLANDES

ZBI ZENTRUM FÜR BIOINFORMATIK

The Burrows-Wheeler Transform (BWT)

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2021

The Burrows-Wheeler Transform

Burrows and Wheeler, A block sorting lossless data compression algorithm, 1994

Applications of the BWT

Motivation

Modern DNA sequencers (Illumina NovaSeq 6000) produce more than 3Tbp per day.

- Compression of arbitrary text: bzip2
- Compression of sequenced read data sets (FASTQ files)
- At the core of popular read mappers (BWA, Bowtie)
- Overlap alignment in genome assembly (Simpson & Durbin, 2010)

Bottom line: BWT is essential for many string processing applications.

Definition via Suffix Array

Burrows-Wheeler Transform (BWT)

For a string $s\$$ of length n with unique sentinel and suffix array $\texttt{pos}$, the transformed string $\texttt{bwt}[0,\ldots,n-1]$ is defined by

$$\texttt{bwt}[i] := \begin{cases} \$ & \text{if } \texttt{pos}[i] = 0, \\ S[\texttt{pos}[i]-1] & \text{if } \texttt{pos}[i] \neq 0. \end{cases}$$

In other words: To construct the BWT...

... take each character before the lexicographically sorted suffixes

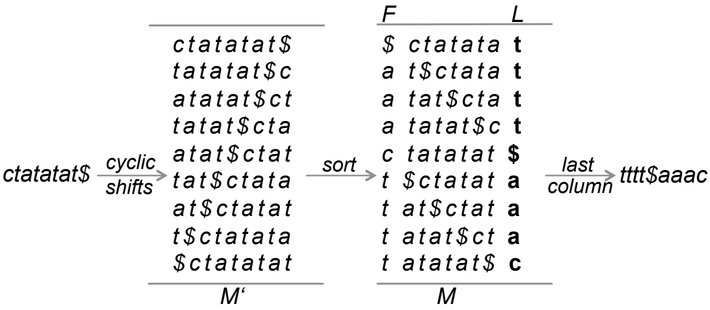
Note

The BWT is a function that (bijectively) **maps strings** (with unique sentinel) emphto strings of the same length.

BWT via Matrix of Cyclic Shifts

The encoding of `bwt` from S runs in 3 steps

1. Use conceptual matrix M' whose rows are cyclic shifts of S.
2. Compute the matrix M by sorting the rows of M' lexicographically.
3. Output the last column L of M.



ctatatat$ → cyclic shifts → M' → sort → M → last column → tttt$aaac

M'
ctatatat$
tatatat$c
atatat$ct
tatat$cta
atat$ctat
tat$ctata
at$ctatat
t$ctatata
$ctatatat

F		L
$	ctatata	**t**
a	t$ctata	**t**
a	tat$cta	**t**
a	tatat$c	**t**
c	tatatat	**$**
t	$ctatat	**a**
t	at$ctat	**a**
t	atat$ct	**a**
t	atatat$	**c**

M

BWT from `pos`

Create the BWT transform of a string `s` in $O(n)$ time

```
def compute_bwt(s, pos):
  return ''.join(s[p-1] for p in pos)  # s[-1] is s[len(s)-1]
```

BWT from pos

Create the BWT transform of a string `s` in $O(n)$ time

```python
def compute_bwt(s, pos):
  return ''.join(s[p-1] for p in pos)  # s[-1] is s[len(s)-1]
```

Note

Constructing the SA first is **expensive** in terms of space;
defeats the space advantage of BWT over SA.
Better: direct construction of BWT (not discussed here).

BWT Decoding?

So far

For a given string s, compute the `bwt`(s).

Question

Given `bwt`(s), how to recover the original string s ?

Last-to-First (LF) Mapping

Text: `ctatatat$`, **BWT:** `tttt$aaac`

F		*L*
$	*ctatata*	**t**
a	*t$ctata*	**t**
a	*tat$cta*	**t**
a	*tatat$c*	**t**
c	*tatatat*	**$**
t	*$ctatat*	**a**
t	*at$ctat*	**a**
t	*atat$ct*	**a**
t	*atatat$*	**c**

M

i	0	1	2	3	4	5	6	7	8
L[i]	t	t	t	t	$	a	a	a	c
LF(i)	5	6	7	8	0	1	2	3	4
F[i]	$	a	a	a	c	t	t	t	t

Definition of $LF : \{0, \ldots, n\} \to \{0, \ldots, n\}$

If $L[i] = c$ is the k-th occurrence of character c in L (i.e., in the BWT), then $LF(i) = j$ is the index j such that $F[j]$ is the k-th occurrence of c in F (sorted).

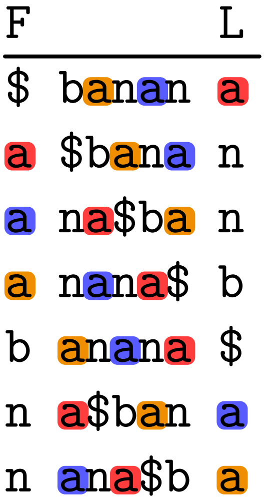
Why Is the LF Mapping Useful?

F		L
`$`	`banan`	`a`
`a`	`$bana`	`n`
`a`	`na$ba`	`n`
`a`	`nana$`	`b`
`b`	`anana`	`$`
`n`	`a$ban`	`a`
`n`	`ana$b`	`a`

LF mapping

If $L[i] = c$ is the k-th occurrence of c in L, then $LF(i) = j$ is the index j such that $F[j]$ is the k-th occurrence of c in F.

Why Is the LF Mapping Useful?

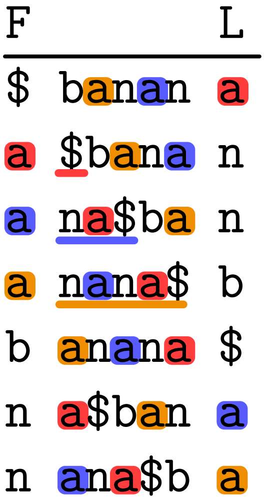


F		L
$	banan	a
a	$bana	n
a	na$ba	n
a	nana$	b
b	anana	$
n	a$ban	a
n	ana$b	a

LF mapping

If $L[i] = c$ is the k-th occurrence of c in L, then $LF(i) = j$ is the index j such that $F[j]$ is the k-th occurrence of c in F.

Why Is the LF Mapping Useful?

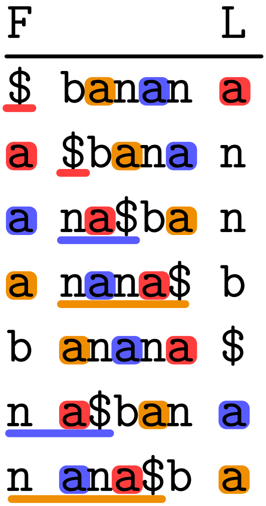


F		L
$	banan	a
a	$bana	n
a	na$ba	n
a	nana$	b
b	anana	$
n	a$ban	a
n	ana$b	a

LF mapping

If $L[i] = c$ is the k-th occurrence of c in L, then $LF(i) = j$ is the index j such that $F[j]$ is the k-th occurrence of c in F.

Why Is the LF Mapping Useful?



F		L
$	banan	a
a	$bana	n
a	na$ba	n
a	nana$	b
b	anana	$
n	a$ban	a
n	ana$b	a

LF mapping

If $L[i] = c$ is the k-th occurrence of c in L, then $LF(i) = j$ is the index j such that $F[j]$ is the k-th occurrence of c in F.

Code and Example: BWT Decoding

```python
def decode_bwt(bwt, LF):
    n, r = len(bwt), 0
    s = ['$']
    while len(s) < n:
        s.append(bwt[r])  # build reverse
        r = LF[r]
    return ''.join(s[::-1])  # reverse
```

F		L
$	banan	a
a	$bana	n
a	na$ba	n
a	nana$	b
b	anana	$
n	a$ban	a
n	ana$b	a

BWT Decoding

In order to construct the LF-mapping we need the `C` array, similar to the buckets for suffix array construction:

Definition

For an alphabet Σ let $\texttt{C}[c]$, $c \in \Sigma$, be the number of occurrences of all characters c', $c' < c$, in S.

BWT Decoding

In order to construct the LF-mapping we need the `C` array, similar to the buckets for suffix array construction:

Definition

For an alphabet Σ let $\texttt{C}[c]$, $c \in \Sigma$, be the number of occurrences of all characters c', $c' < c$, in S.

For the BWT string `ttt$aaac`:

C [$]	C [a]	C [c]	C [t]
0	1	4	5

BWT Decoding

In order to construct the LF-mapping we need the `C` array, similar to the buckets for suffix array construction:

Definition

For an alphabet Σ let $\mathtt{C}[c]$, $c \in \Sigma$, be the number of occurrences of all characters c', $c' < c$, in S.

For the BWT string `ttt$aaac`:

C [$]	C [a]	C [c]	C [t]
0	1	4	5

Observation:

First occurrence of c in F is at index $\mathtt{C}[c]$
k-th occurrence of c in F is at index $\mathtt{C}[c] + k - 1$

Compute LF

Compute the *LF*-array from BWT and the `C` array in $O(n)$ time

```
def compute_LF(bwt: str, C: dict|Counter):
  LF = []
  for a in bwt:
    LF.append(C[a])
    C[a] += 1
  return LF
```

Compute LF

Compute the *LF*-array from BWT and the `C` array in $O(n)$ time

```
def compute_LF(bwt: str, C: dict|Counter):
  LF = []
  for a in bwt:
    LF.append(C[a])
    C[a] += 1
  return LF
```

Problem: LF-mapping uses $O(n)$ space, as the suffix array does.

Compute LF

Compute the *LF*-array from BWT and the `C` array in $O(n)$ time

```
def compute_LF(bwt: str, C: dict|Counter):
  LF = []
  for a in bwt:
    LF.append(C[a])
    C[a] += 1
  return LF
```

Problem: LF-mapping uses $O(n)$ space, as the suffix array does.
Solution: Do not store LF directly, but only `C` and a succinct data structure that supports $O(1)$ time queries about the number of occurrences of any letter a in $\texttt{bwt}[\ldots r]$, conceptually a table $\texttt{Occ}(a, r)$.

FM-index:
Backward Search with the BWT

Ferragina and Manzini, Opportunistic Data Structures with Applications 2000

Compressed full-text indexes

So far we have been **forward** searching characters from the pattern P.
A **FM-index** is a compressed full text index that supports **backward search**.

For a pattern of length m:

- forward search: $P[0]$, $P[0,1]$, …, $P[0 \dots m-1]$
- backward search: $P[m-1]$, $P[m-2, m-1]$, …, $P[0 \dots m-1]$

Compressed full-text indexes

So far we have been **forward** searching characters from the pattern P.
A **FM-index** is a compressed full text index that supports **backward search**.

For a pattern of length m:

- forward search: $P[0]$, $P[0,1]$, …, $P[0 \ldots m-1]$
- backward search: $P[m-1]$, $P[m-2, m-1]$, …, $P[0 \ldots m-1]$

Definition

Given a `bwt` string of length n on the alphabet Σ,
let `Occ(c,i)` return the number of occurrences of $c \in \Sigma$ in the prefix $\texttt{bwt}[0 \ldots i]$.

Occ Table Example; Relation of *LF* to `C` and `Occ`

s = `ctatatat$` and `bwt`=`tttt$aaac`:

C [$]	C [a]	C [c]	C [t]
0	1	4	5

	t	**t**	**t**	**t**	**$**	**a**	**a**	**a**	**c**
	0	1	2	3	4	5	6	7	8
Occ [$]	0	0	0	0	1	1	1	1	1
Occ [a]	0	0	0	0	0	1	2	3	3
Occ [c]	0	0	0	0	0	0	0	0	1
Occ [t]	1	2	3	4	4	4	4	4	4

`Occ`

`Occ`(c, i) returns the number of occurrences of $c \in \Sigma$ in the prefix `bwt`$[0 \ldots i]$.

Observations

index	*bwt*	*S[pos[i]]*
0	**t**	*$*
1	**t**	*at$*
2	**t**	*atat$*
3	**t**	*atatat$*
4	**$**	*ctatatat$*
→ *5*	**a**	*t$*
6	**a**	*tat$*
7	**a**	*tatat$*
→ *8*	**c**	*tatatat$*

- Assume we are looking for pattern $P = \texttt{at}$.
  We have already found (via backward search)
  the interval for all suffixes that start with `t`: $[5, 8]$.

Observations

index	*bwt*	*S[pos[i]]*
0	**t**	*$*
1	**t**	*at$*
2	**t**	*atat$*
3	**t**	*atatat$*
4	**$**	*ctatatat$*
→ *5*	**a**	*t$*
6	**a**	*tat$*
7	**a**	*tatat$*
→ *8*	**c**	*tatatat$*

- Assume we are looking for pattern $P = \texttt{at}$.
  We have already found (via backward search)
  the interval for all suffixes that start with `t`: $[5, 8]$.
- Suffixes starting with `at` have to come from
  $S[pos[5] - 1], \ldots, S[pos[8] - 1]$.

Observations

index	*bwt*	*S[pos[i]]*
0	**t**	*$*
1	**t**	*at$*
2	**t**	*atat$*
3	**t**	*atatat$*
4	**$**	*ctatatat$*
→ *5*	**a**	*t$*
6	**a**	*tat$*
7	**a**	*tatat$*
→ *8*	**c**	*tatatat$*

- Assume we are looking for pattern $P = \texttt{at}$.
  We have already found (via backward search)
  the interval for all suffixes that start with `t`: $[5, 8]$.
- Suffixes starting with `at` have to come from
  $S[pos[5] - 1], \ldots, S[pos[8] - 1]$.
- These candidates can be found in the suffix array
  at ranks LF(5), . . ., LF(8),
  because $S[pos[r] - 1]$ starts with `at`,
  iff $\texttt{bwt}[r] = a$ and r belongs to the `t`-interval.

Observations continued

index	*bwt*	*S[pos[i]]*
0	**t**	*$*
1	**t**	*at$*
2	**t**	*atat$*
3	**t**	*atatat$*
4	**$**	*ctatatat$*
→ *5*	**a**	*t$*
6	**a**	*tat$*
7	**a**	*tatat$*
→ *8*	**c**	*tatatat$*

- In fact we only need the **first index** p and the **last index** q such that $5 \leq p \leq q \leq 8$ and $\texttt{bwt}[p]{=}\texttt{bwt}[q] = a$ (because of lexicographic sorting)

Observations continued

index	*bwt*	*S[pos[i]]*
0	**t**	*$*
1	**t**	*at$*
2	**t**	*atat$*
3	**t**	*atatat$*
4	**$**	*ctatatat$*
→ *5*	**a**	*t$*
6	**a**	*tat$*
7	**a**	*tatat$*
→ *8*	**c**	*tatatat$*

- In fact we only need the **first index** p and the **last index** q such that $5 \leq p \leq q \leq 8$ and $\texttt{bwt}[p]{=}\texttt{bwt}[q] = a$ (because of lexicographic sorting)
- So we have $p = 5$ and $q = 7$, and the `at`-interval can be found via $LF(5) = 1$ and $LF(7) = 3$.
- But how can we find p and q efficiently?

Observations (conclusion)

index	*bwt*	*S[pos[i]]*
0	**t**	*\$*
1	**t**	*at\$*
2	**t**	*atat\$*
3	**t**	*atatat\$*
4	**\$**	*ctatatat\$*
→ *5*	**a**	*t\$*
6	**a**	*tat\$*
7	**a**	*tatat\$*
→ *8*	**c**	*tatatat\$*

More generally

- Assume we have interval $[i, \ldots, j]$ for suffix μ, find the interval $[i', \ldots, j']$ for suffix $c\mu$.
- If $c\mu$ exists in the text, then $[i', \ldots, j']$ is non-empty.
- We need the smallest p and largest q such that $i \leq p \leq q \leq j$ and $\mathtt{bwt}[p] = \mathtt{bwt}[q] = c$. Then $c\mu$-interval $= [LF(p), \ldots LF(q)]$.
- Obtain i' and j' via `C` and `Occ`:

$$\begin{aligned} i' &= LF(p) &&= \mathtt{C}[c] + \mathtt{Occ}(c, i-1) \\ j' &= LF(q) &&= \mathtt{C}[c] + \mathtt{Occ}(c, j) - 1 \end{aligned}$$

Backward Search (one step)

Given interval $[i, \ldots, j]$ for suffix μ, find the interval $[i', \ldots, j']$ for suffix $c\mu$:

```
def backward_step(c, i, j, occ, C):
    i = C[c] + occ.get(c, i-1)
    j = C[c] + occ.get(c, j-1)
    return (i,j) if i < j else None
```

Notes

- `i` and `j` specify a "pythonic" interval: `j` is **not** included (different indexing!)
- `occ` is an object implementing the `Occ` **table**.
- For `i` $<$ 0 we define `occ.get(c,i)` $:=$ 0.

Backward Search (full)

```python
def backward_search(fm, P):
    interval = (0, fm.n)
    for k in range(len(P)-1, -1, -1):
        c = P[k]
        i, j = interval
        interval = backward_step(c, *interval, fm.occ, fm.C)
        if interval is None: break
        #print(P[k:], interval)  # debug
    return interval
```

Note

Object `fm` encapsulates the ingredients of an **FM index**:
text, length *n*, bwt, C table, and Occ table.

Backward search example ($P =$ `ata`)

Search P= ata
Step 1
i=0 j=n-1=8 k=2 c=a
i=C[a]+Occ(a,0-1)=1+0=1
j=C[a]+Occ(a,8)-1=1+3-1=3

index	*bwt*	*S[pos[i]]*
→ *0*	**t**	*$*
1	**t**	*at$*
2	**t**	*atat$*
3	**t**	*atatat$*
4	**$**	*ctatatat$*
5	**a**	*t$*
6	**a**	*tat$*
7	**a**	*tatat$*
→ *8*	**c**	*tatatat$*

C [$]	**C [a]**	**C [c]**	**C [t]**
0	1	4	5

	t	**t**	**t**	**t**	**$**	**a**	**a**	**a**	**c**
	0	1	2	3	4	5	6	7	8
Occ [$]	0	0	0	0	1	1	1	1	1
Occ [a]	0	0	0	0	0	1	2	3	3
Occ [c]	0	0	0	0	0	0	0	0	1
Occ [t]	1	2	3	4	4	4	4	4	4

Backward search example ($P =$ `ata`)

Search P= ata
Step 2
i=1 j=3 k=1 c=t
i=C[t]+Occ(t,1-1)=5+1=6
j=C[t]+Occ(t,3)-1=5+4-1=8

index	*bwt*	*S[pos[i]]*
0	**t**	*$*
→ *1*	**t**	*at$*
2	**t**	*atat$*
→ *3*	**t**	*atatat$*
4	**$**	*ctatatat$*
5	**a**	*t$*
6	**a**	*tat$*
7	**a**	*tatat$*
8	**c**	*tatatat$*

C [$]	**C [a]**	**C [c]**	**C [t]**
0	1	4	5

	t	**t**	**t**	**t**	**$**	**a**	**a**	**a**	**c**
	0	1	2	3	4	5	6	7	8
Occ [$]	0	0	0	0	1	1	1	1	1
Occ [a]	0	0	0	0	0	1	2	3	3
Occ [c]	0	0	0	0	0	0	0	0	1
Occ [t]	1	2	3	4	4	4	4	4	4

Backward search example ($P =$ `ata`)

Search P= ata
Step 3
i=6   j=8   k=0   c=a
i=C[a]+Occ(a,6-1)=1+1=2
j=C[a]+Occ(a,8)-1=1+3-1=3

C [$]	C [a]	C [c]	C [t]
0	1	4	5

index	*bwt*	*S[pos[i]]*
0	**t**	*$*
1	**t**	*at$*
2	**t**	*atat$*
3	**t**	*atatat$*
4	**$**	*ctatatat$*
5	**a**	*t$*
→ *6*	**a**	*tat$*
7	**a**	*tatat$*
→ *8*	**c**	*tatatat$*

	t	t	t	t	$	a	a	a	c
	0	1	2	3	4	5	6	7	8
Occ [$]	0	0	0	0	1	1	1	1	1
Occ [a]	0	0	0	0	0	1	2	3	3
Occ [c]	0	0	0	0	0	0	0	0	1
Occ [t]	1	2	3	4	4	4	4	4	4

Backward search example ($P =$ `ata`)

Search P= ata
Done
i=2   j=3   k=-1
because i <= j we found
the valid interval [2,3] with ata

index	*bwt*	S[*pos[i]*]
0	**t**	*$*
1	**t**	*at$*
→ *2*	**t**	*atat$*
→ *3*	**t**	*atatat$*
4	**$**	*ctatatat$*
5	**a**	*t$*
6	**a**	*tat$*
7	**a**	*tatat$*
8	**c**	*tatatat$*

C [$]	**C [a]**	**C [c]**	**C [t]**
0	1	4	5

	t	**t**	**t**	**t**	**$**	**a**	**a**	**a**	**c**
	0	1	2	3	4	5	6	7	8
Occ [$]	0	0	0	0	1	1	1	1	1
Occ [a]	0	0	0	0	0	1	2	3	3
Occ [c]	0	0	0	0	0	0	0	0	1
Occ [t]	1	2	3	4	4	4	4	4	4

Backward search example ($P =$ `tt`)

Search P= tt
Step 1
i=0 j=n-1=8 k=1 c=t
i=C[t]+Occ(t,0-1)=5+0=5
j=C[t]+Occ(t,8)-1=5+4-1=8

	index	*bwt*	*S[pos[i]]*
→	*0*	**t**	*$*
	1	**t**	*at$*
	2	**t**	*atat$*
	3	**t**	*atatat$*
	4	**$**	*ctatatat$*
	5	**a**	*t$*
	6	**a**	*tat$*
	7	**a**	*tatat$*
→	*8*	**c**	*tatatat$*

C [$]	**C [a]**	**C [c]**	**C [t]**
0	1	4	5

	t	**t**	**t**	**t**	**$**	**a**	**a**	**a**	**c**
	0	1	2	3	4	5	6	7	8
Occ [$]	0	0	0	0	1	1	1	1	1
Occ [a]	0	0	0	0	0	1	2	3	3
Occ [c]	0	0	0	0	0	0	0	0	1
Occ [t]	1	2	3	4	4	4	4	4	4

Backward search example ($P = \texttt{tt}$)

Search P= tt
Step 2
i=5 j=8 k=0   c=t
i=C[t]+Occ(t,5-1)=5+4=9
j=C[t]+Occ(t,8)-1=5+4-1=8

index	*bwt*	S[*pos[i]*]
0	**t**	*$*
1	**t**	*at$*
2	**t**	*atat$*
3	**t**	*atatat$*
4	**$**	*ctatatat$*
→*5*	**a**	*t$*
6	**a**	*tat$*
7	**a**	*tatat$*
→*8*	**c**	*tatatat$*

C [$]	C [a]	C [c]	C [t]
0	1	4	5

	t	**t**	**t**	**t**	**$**	**a**	**a**	**a**	**c**
	0	1	2	3	4	5	6	7	8
Occ [$]	0	0	0	0	1	1	1	1	1
Occ [a]	0	0	0	0	0	1	2	3	3
Occ [c]	0	0	0	0	0	0	0	0	1
Occ [t]	1	2	3	4	4	4	4	4	4

Backward search example ($P = \texttt{tt}$)

Search P= tt
Done
i=9 j=8 k=-1
i > j no interval found

index	*bwt*	*S[pos[i]]*
0	**t**	*$*
1	**t**	*at$*
2	**t**	*atat$*
3	**t**	*atatat$*
4	**$**	*ctatatat$*
5	**a**	*t$*
6	**a**	*tat$*
7	**a**	*tatat$*
8	**c**	*tatatat$*

C [$]	C [a]	C [c]	C [t]
0	1	4	5

	t	**t**	**t**	**t**	**$**	**a**	**a**	**a**	**c**
	0	1	2	3	4	5	6	7	8
Occ [$]	0	0	0	0	1	1	1	1	1
Occ [a]	0	0	0	0	0	1	2	3	3
Occ [c]	0	0	0	0	0	0	0	0	1
Occ [t]	1	2	3	4	4	4	4	4	4

Summary

- Burrows-Wheeler Transform (BWT)
  - Encoding: $s \rightarrow \texttt{bwt}(s)$,
  - The LF mapping for the `bwt`
  - Decoding: $\texttt{bwt}(s) \rightarrow s$, using LF mapping.
  - Fundamental property: The k-th occurrence of c in the BWT corresponds to the k-th occurrence of c in the first letters of the sorted suffixes.
- Pattern search with the FM-index
  - Replacing the LF mapping with `C` and `Occ`
  - Backward Search in the `bwt` with `C` and `Occ`
  - Next lecture: Compression of the Occ table

Possible Exam Questions

- Define the BWT.
- What is the relation of the BWT to the suffix array of the same string?
- Compute the BWT for a given string.
- Compute the original string from a given BWT.
- Define the Last-to-First (LF) mapping.
- Why is it useful?
- How can the LF-mapping be substituted by `C` and `Occ`?
- What is an FM-index?
- Explain backward pattern search with the FM-index.