

The Burrows-Wheeler Transform as a compressed index

Giovanni Manzini

Off-line pattern searching

We have a collection of documents (think of project Gutenberg or Wikipedia) and we want to search for information (substrings) in it.

Since the collection is known in advance, we speed-up the search building an index for the collection

The Suffix Array

The Suffix Array is the simplest indexing data structure supporting fast pattern searching

Consider for example

$T =$ `swiss·miss·missing`

The Suffix Array

swiss·miss·missing

We consider all the
suffixes of the input text

1	swiss·miss·missing
2	wiss·miss·missing
3	iss·miss·missing
4	ss·miss·missing
5	s·miss·missing
6	·miss·missing
7	miss·missing
8	iss·missing
9	ss·missing
10	s·missing
11	·missing
12	missing
13	issing
14	ssing
15	sing
16	ing
17	ng
18	g

The Suffix Array

swiss·miss·missing

We consider all the
suffixes of the input text

... and we sort them
in lexicographic order

6	·miss·missing
11	·missing
18	g
16	ing
3	iss·miss·missing
8	iss·missing
13	issing
7	miss·missing
12	missing
17	ng
5	s·miss·missing
10	s·missing
15	sing
4	ss·miss·missing
9	ss·missing
14	ssing
1	swiss·miss·missing
2	wiss·miss·missing

The Suffix Array

swiss·miss·missing

Using binary search
we find that miss
appears starting at
positions 7 and 12

6	·miss·missing
11	·missing
18	g
16	ing
3	iss·miss·missing
8	iss·missing
13	issing
7	miss ·missing
12	miss ing
17	ng
5	s·miss·missing
10	s·missing
15	sing
4	ss·miss·missing
9	ss·missing
14	ssing
1	swiss·miss·missing
2	wiss·miss·missing

Using the suffix array we can find all the occurrences of any pattern P in T using binary search in $O(|P| \log |T|)$ time.

Using also the LCP array we can reduce search time to $O(|P| + \log |T|)$.

The Suffix Array

swiss·miss·missing

To represent the Suffix
Array we use $|T|$ integers
in the range $1 \dots |T|$
 $\Rightarrow |T| \log |T|$ bits

6	·miss·missing
11	·missing
18	g
16	ing
3	iss·miss·missing
8	iss·missing
13	issing
7	miss·missing
12	missing
17	ng
5	s·miss·missing
10	s·missing
15	sing
4	ss·miss·missing
9	ss·missing
14	ssing
1	swiss·miss·missing
2	wiss·miss·missing

In typical implementations we use a 4 byte integer for each suffix array entry and a byte for each text character

Better than the Suffix Tree but
still a lot of space!

Example

100 Megabytes of email messages

400 Megabytes for the suffix array

500 Mbytes overall!

Another example:

Human Chromosome 2

240 Million bases (A,C,G,T)

960 Megabytes for the suffix array

1.2 Gigabytes overall!

What about compression?

Using standard tools, like gzip or bzip2,
we can usually save a lot of space.

Examples:

100Mb email messages → 25Mb

240M DNA bases → 60Mb

Is this gap unavoidable?

Compression
(small storage)

Email messages
25Mb

Human Chromosome
60Mb

Suffix Array
(fast retrieval)

Email messages
500Mbytes

Human Chromosome
1.2 Gb

Burrows-Wheeler compression

The Burrows-Wheeler (or Block-Sorting) compression algorithm is based on the new concept of transforming the text in order to make it easier to compress.

The transformation is now called Burrows-Wheeler Transform and is strictly related to the Suffix Array.

Burrows-Wheeler Transform

swiss·miss·missing

To compute the **BWT** we go on transforming each suffix into a cyclic shift of **T**.

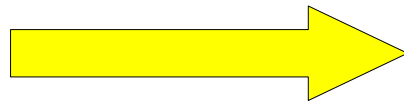
The last column of the resulting matrix is the **BWT**.

6	·miss·missing	swiss	s
11	·missing	swiss·miss	s
18	g	swiss·miss·missi	n
16	ing	swiss·miss·miss	s
3	iss·miss·missing	s	w
8	iss·missing	swiss·	m
13	issing	swiss·miss·	m
7	miss·missing	swiss	·
12	missing	swiss·miss	·
17	ng	swiss·miss·miss	i
5	s·miss·missing	swi	s
10	s·missing	swiss·mi	s
15	sing	swiss·miss·mi	s
4	ss·miss·missing	sw	i
9	ss·missing	swiss·m	i
14	ssing	swiss·miss·m	i
1	swiss·miss·missin	g	
2	wiss·miss·missing	s	

Burrows-Wheeler Transform

swiss·miss·missing

We discard the Suffix Array
but we keep track of the row
containing the original string

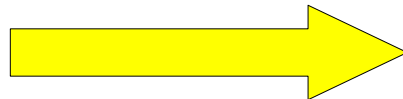


6	·miss·missing	swiss	s
11	·missing	swiss·miss	s
18	g	swiss·miss·missi	n
16	ing	swiss·miss·miss	s
3	iss·miss·missing	s	w
8	iss·missing	swiss·	m
13	issing	swiss·miss·	m
7	miss·missing	swiss	·
12	missing	swiss·miss	·
17	ng	swiss·miss·miss	i
5	s·miss·missing	swi	s
10	s·missing	swiss·mi	s
15	sing	swiss·miss·mi	s
4	ss·miss·missing	sw	i
9	ss·missing	swiss·m	i
14	ssing	swiss·miss·m	i
1	swiss·miss·missin	g	
2	wiss·miss·missing	s	

Burrows-Wheeler Transform

BWT **swiss·miss·missing**
=
ssnswmm··issiiigs

Important: each column
is a permutation of **T**.
The first column contains
the sorted characters



·miss·missing	swiss	s
·missing	swiss·mis	s
gswiss·miss·missi	n	
ing	swiss·miss·mis	s
iss·miss·missing	s	w
iss·missing	swiss·	m
issing	swiss·miss·	m
miss·missing	swiss	·
missing	swiss·miss	·
ng	swiss·miss·miss	i
s·miss·missing	swi	s
s·missing	swiss·mi	s
sing	swiss·miss·mi	s
ss·miss·missing	sw	i
ss·missing	swiss·m	i
ssing	swiss·miss·m	i
swiss·miss·missin	g	
wiss·miss·missing	s	




swiss · miss · miss**ing**

in F **s** is above **s** because
 ing ≤ wiss · · ·

in L **s** is in the row prefixed
 by **ing** hence is above **s**

F		L
·	miss · missing swiss	s
·	missing swiss · mis	s
g	swiss · miss · missi	n
i	ng swiss · miss · mis	s
i	ss · miss · missing s	w
i	ss · missing swiss ·	m
i	ssing swiss · miss ·	m
m	iss · missing swiss	·
m	issing swiss · miss	·
n	g swiss · miss · miss	i
s	· miss · missing swi	s
s	· missing swiss · mi	s
s	ing swiss · miss · mi	s
s	s · miss · missing sw	i
s	s · missing swiss · m	i
s	sing swiss · miss · m	i
s	wiss · miss · missin	g
w	iss · miss · missing	s

We can map each character in L to its image in F

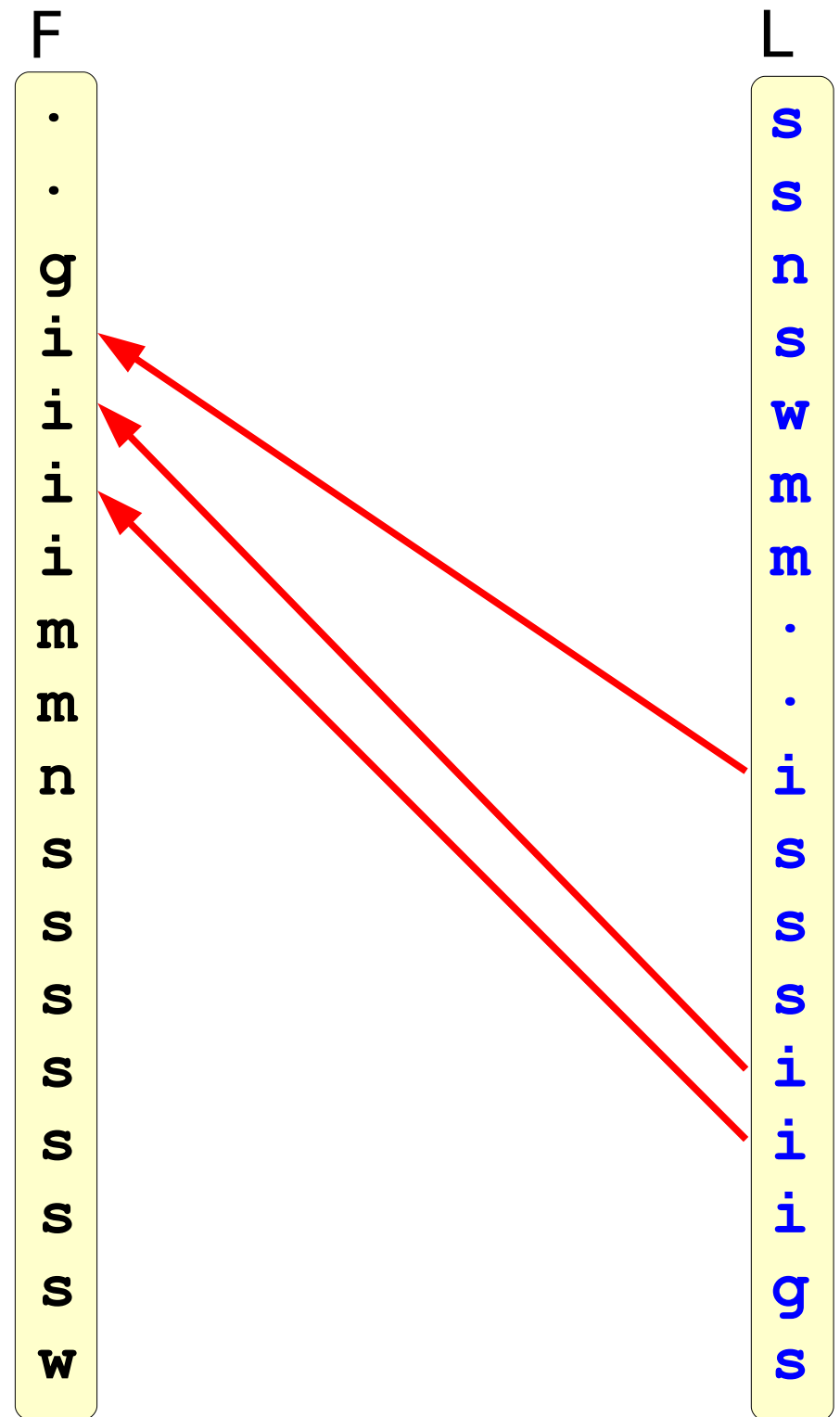
F		L
·	miss·missing	swiss
·	missing	swiss·mis
g	swiss·miss·missi	n
i	ngswiss·miss·mis	s
i	ss·miss·missing	w
i	ss·missing	swiss·
i	ssing	swiss·miss·
m	iss·missing	swiss
m	issing	swiss·miss
n	gswiss·miss·miss	i
s	·miss·missing	swi
s	·missing	swiss·mi
s	ing	swiss·miss·mi
s	s·miss·missing	sw
s	s·missing	swiss·m
s	sing	swiss·miss·m
s	wiss·miss·missin	g
w	iss·miss·missing	s

We can map each character in L to its image in F

F		L
·	miss·missing swiss	s
·	missing swiss ·mis	s
g	swiss ·miss·missi	n
i	ngswiss ·miss·mis	s
i	ss ·miss·missing s	w
i	ss ·missing swiss ·	m
i	ssingbwiss ·miss·	m
m	iss·missing swiss	·
m	issing swiss ·miss	·
n	gswiss ·miss·miss	i
s	·miss·missing swi	s
s	·missing swiss ·mi	s
s	ing swiss ·miss·mi	s
s	s·miss·missing sw	i
s	s·missing swiss ·m	i
s	sing swiss ·miss·m	i
s	wiss·miss·missin	g
w	iss·miss·missing	s

We can map each
character in L to
its image in F

... even if we only
have L and F ...



Summing up

The relative order of the occurrences of any given character in **F** and **L** is the same

We can easily build a map telling us where each character in **L** is in **F**

We call this the **LF map** and is crucial to recover **T** given **L**

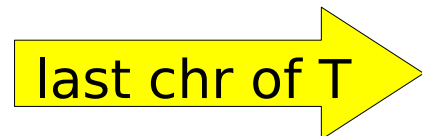
Using the LF map we
retrieve **T** right-to-left

T =

g

F
·
·
g
i
i
i
i
i
m
m
n
s
s
s
s
s
s
s
w

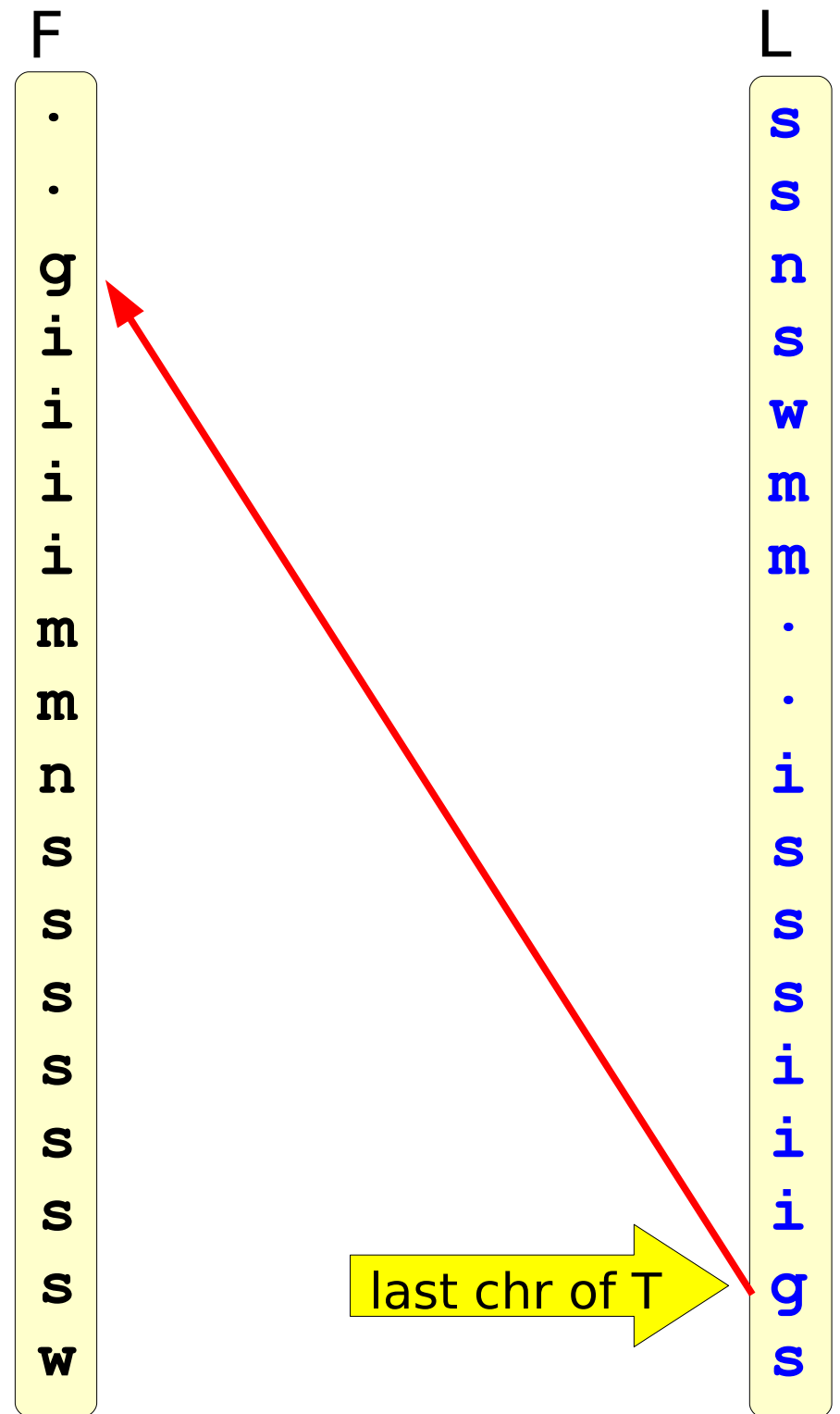
L
s
s
n
s
w
m
m
·
·
i
s
s
s
i
i
i
g
s



Using the LF map we
retrieve **T** right-to-left

T =

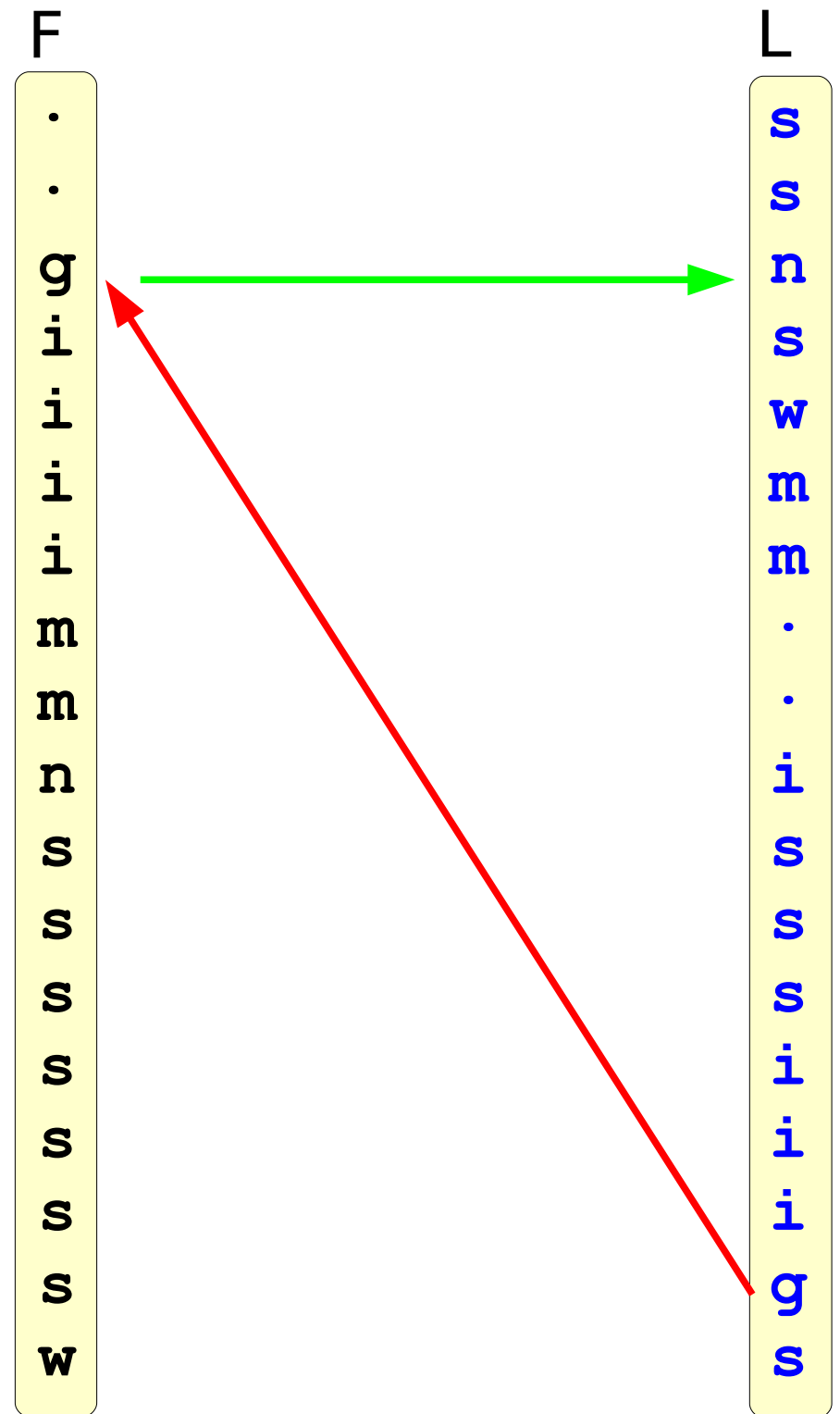
g



Using the LF map we
retrieve **T** right-to-left

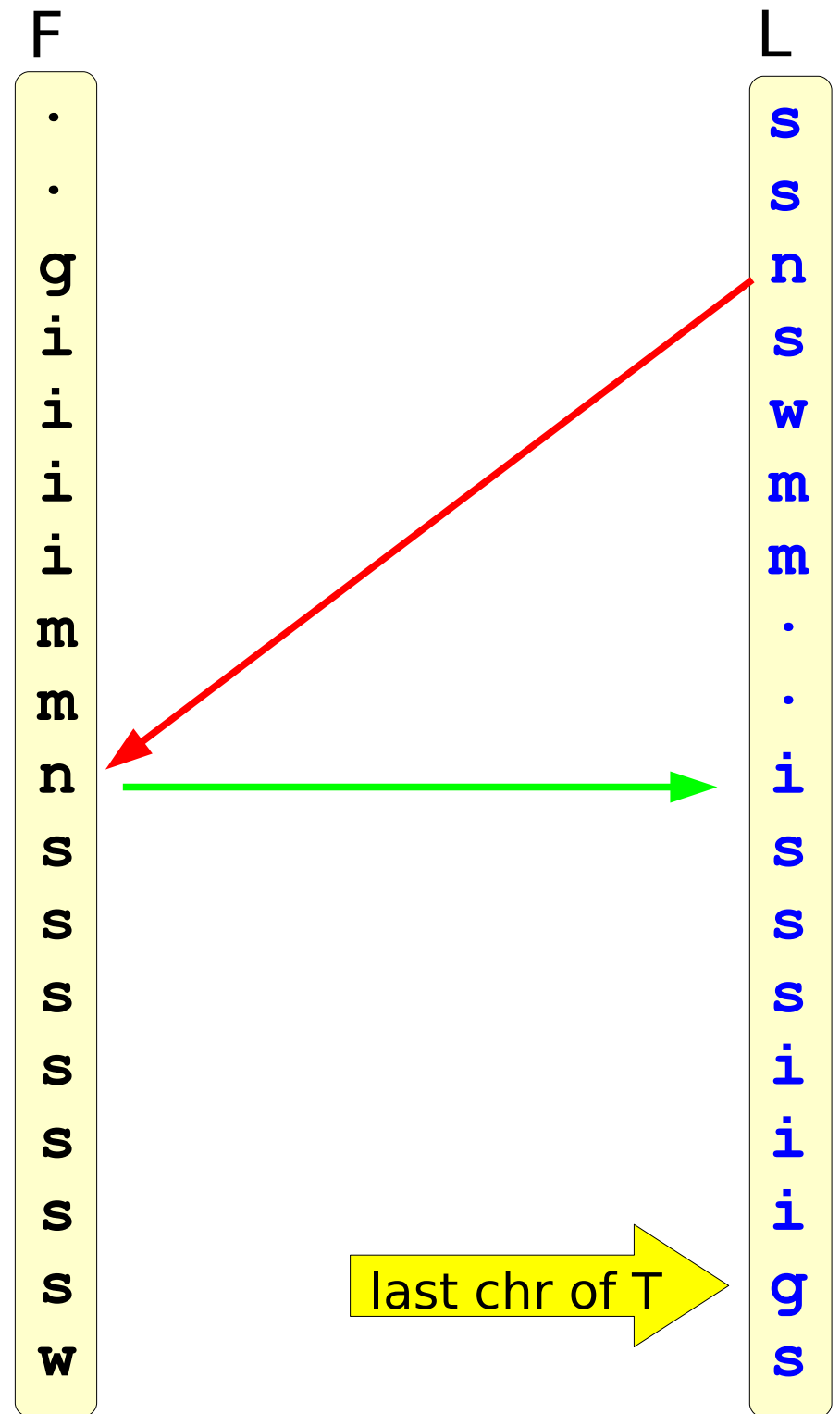
T =

ng



Using the LF map we
retrieve T right-to-left

T = ing
...and so on



The BWT was introduced for data compression: column L is a permutation of the original text which is locally homogeneous: equal chars are usually grouped together

The bzip2 compressor is based on the **BWT!**.

final char (L)	sorted rotations
a	n to decompress. It achieves compression
o	n to perform only comparisons to a depth
o	n transformation} This section describes
o	n transformation} We use the example and
o	n treats the right-hand side as the most
a	n tree for each 16 kbyte input block, enc
a	n tree in the output stream, then encodes
i	n turn, set \$L[1]\$ to be the
i	n turn, set \$R[1]\$ to the
o	n unusual data. Like the algorithm of Man
a	n use a single set of probabilities table
e	n using the positions of the suffixes in
i	n value at a given point in the vector \$R
e	n we present modifications that improve t
e	n when the block size is quite large. Ho
i	n which codes that have not been seen in
i	n with \$ch\$ appear in the {\em same order
i	n with \$ch\$. In our exam
o	n with Huffman or arithmetic coding. Bri
o	n with figures given by Bell~\cite{bell}.

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