The Burrows-Wheeler Transform

Giovanni Manzini

AD4DA 2025/26

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.

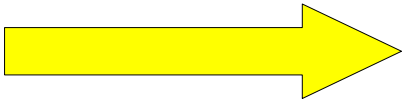
The last column of the resulting matrix is the BWT.

6	·miss·missingswis	s
11	·missingswiss·mis	s
18	gswiss·miss·missi	n
16	ingswiss·miss·mis	s
3	iss·miss·missings	w
8	iss·missingswiss·	m
13	issingswiss·miss·	m
7	miss·missingswiss	·
12	missingswiss·miss	·
17	ngswiss·miss·miss	i
5	s·miss·missingswi	s
10	s·missingswiss·mi	s
15	singswiss·miss·mi	s
4	ss·miss·missingsw	i
9	ss·missingswiss·m	i
14	ssingswiss·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



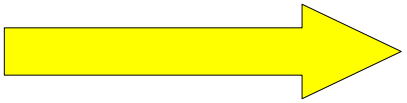
6	·miss·missingswis	s
11	·missingswiss·mis	s
18	gswiss·miss·missi	n
16	ingswiss·miss·mis	s
3	iss·miss·missings	w
8	iss·missingswiss·	m
13	issingswiss·miss·	m
7	miss·missingswiss	·
12	missingswiss·miss	·
17	ngswiss·miss·miss	i
5	s·miss·missingswi	s
10	s·missingswiss·mi	s
15	singswiss·miss·mi	s
4	ss·miss·missingsw	i
9	ss·missingswiss·m	i
14	ssingswiss·miss·m	i
1	**swiss·miss·missin**	**g**
2	wiss·miss·missing	s

Burrows-Wheeler Transform

BWT **swiss·miss·missing**

= **ssnswmm··isssiiigs**

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

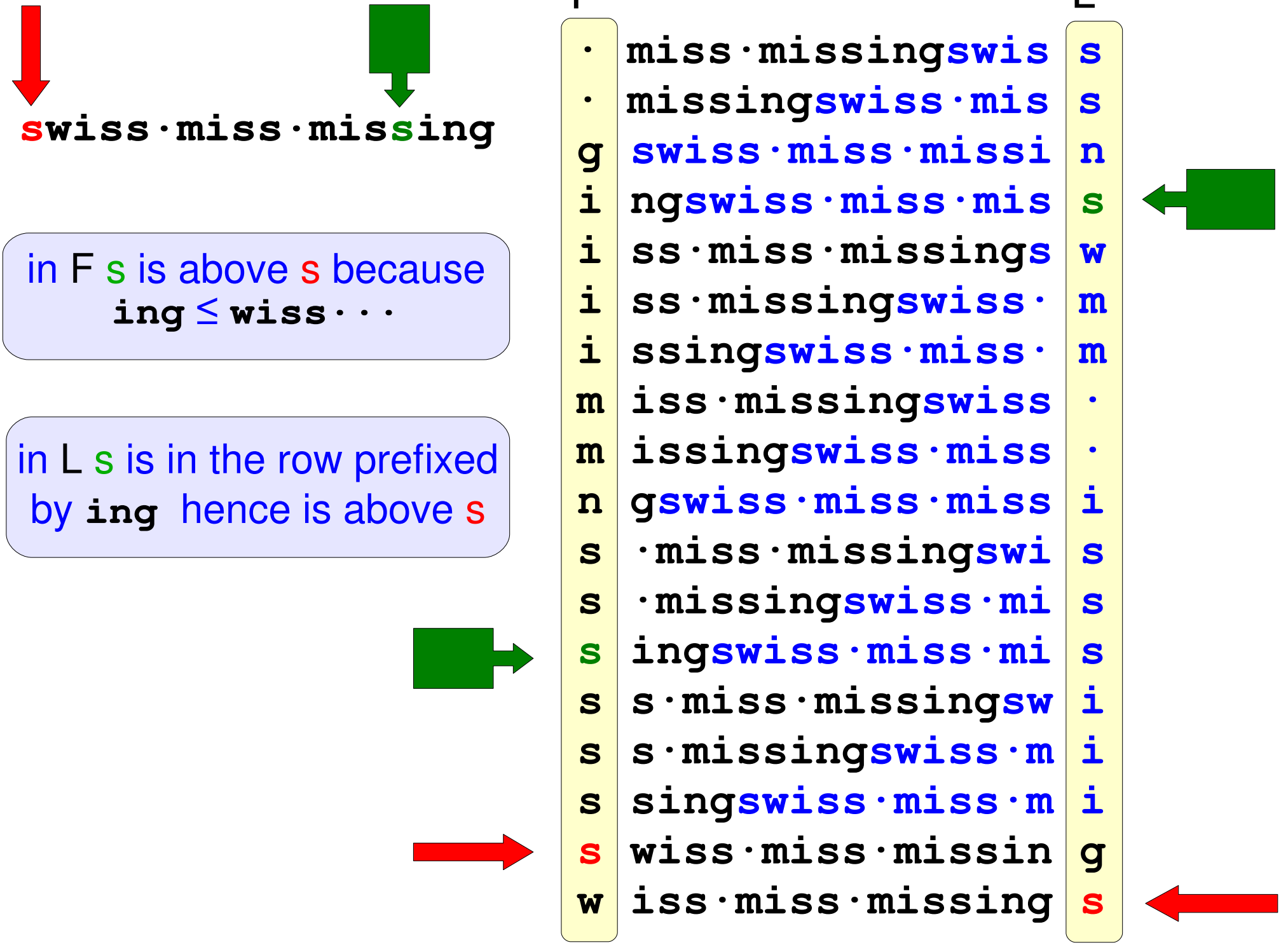


·miss·missingswis	s
·missingswiss·mis	s
gswiss·miss·missi	n
ingswiss·miss·mis	s
iss·miss·missings	w
iss·missingswiss·	m
issingswiss·miss·	m
miss·missingswiss	·
missingswiss·miss	·
ngswiss·miss·miss	i
s·miss·missingswi	s
s·missingswiss·mi	s
singswiss·miss·mi	s
ss·miss·missingsw	i
ss·missingswiss·m	i
ssingswiss·miss·m	i
swiss·miss·missin	**g**
wiss·miss·missing	s

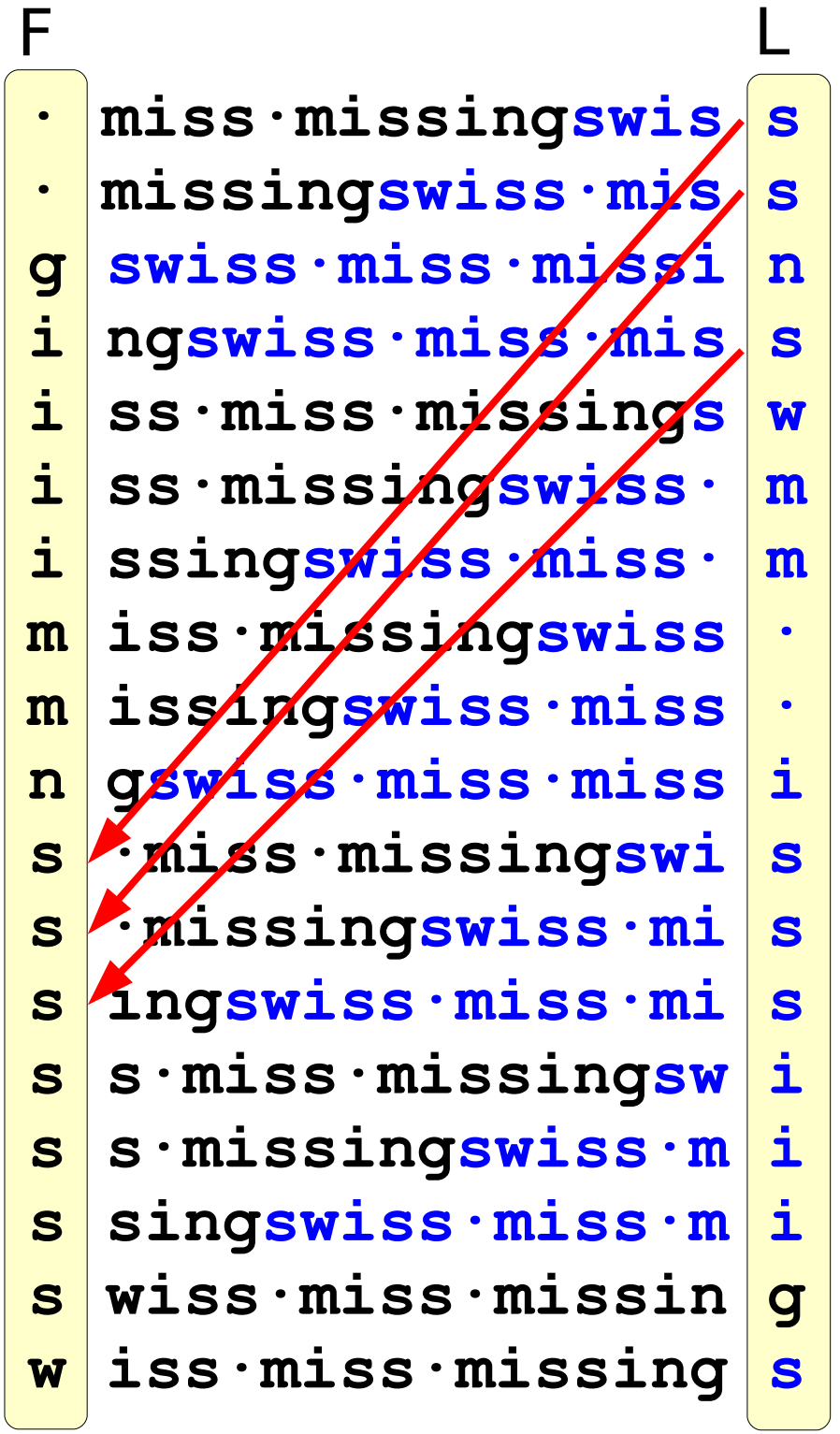
Things to do next

1. Prove that given the transformed text we can retrieve T

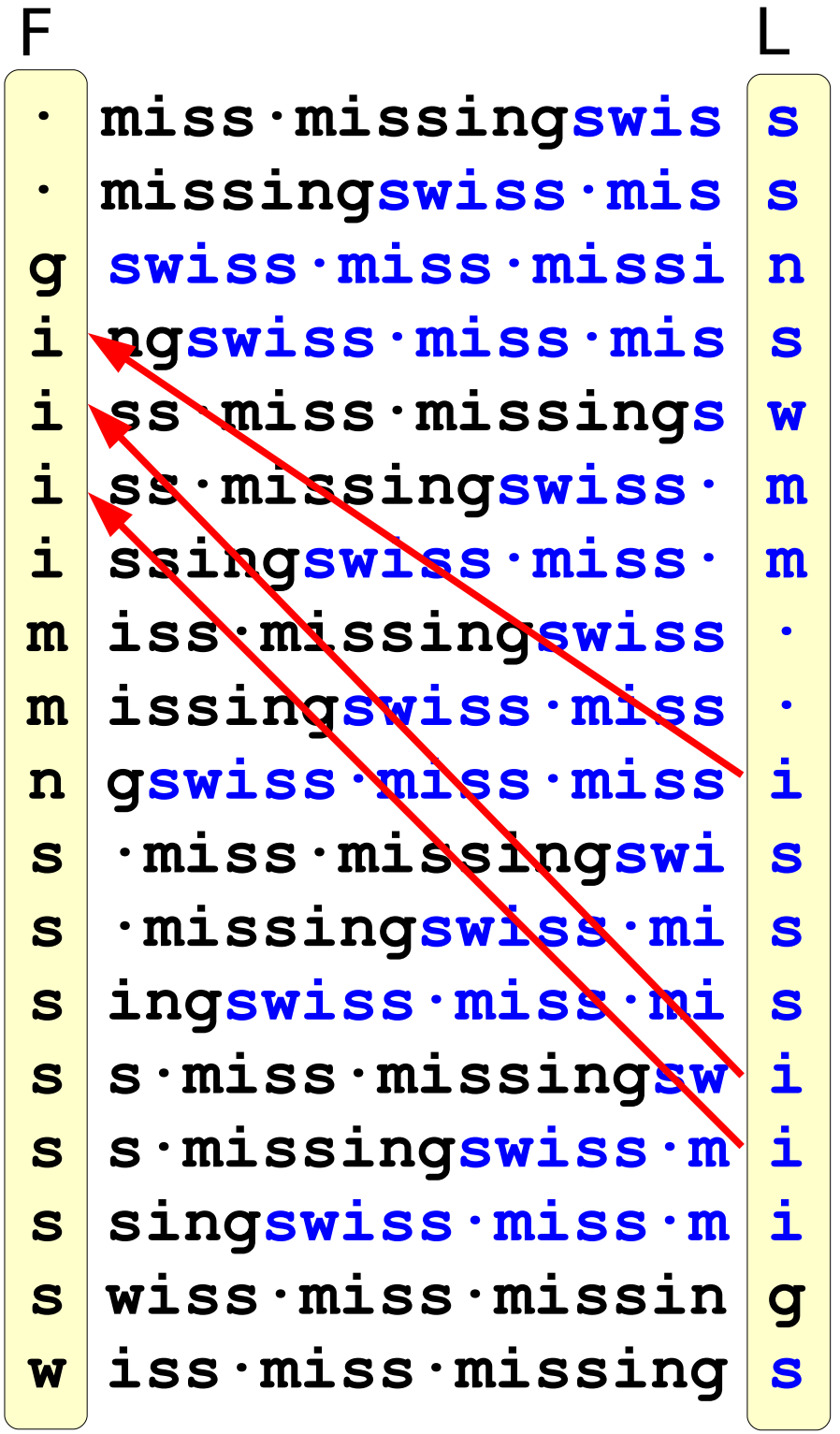
2. Show that the transformed text is easy to compress


swiss·miss·missing
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·missingswis s
· missingswiss·mis s
g swiss·miss·missi n
i ngswiss·miss·mis s
i ss·miss·missings w
i ss·missingswiss· m
i ssingswiss·miss· m
m iss·missingswiss ·
m issingswiss·miss ·
n gswiss·miss·miss i
s ·miss·missingswi s
s ·missingswiss·mi s
s ingswiss·miss·mi s
s s·miss·missingsw i
s s·missingswiss·m i
s singswiss·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

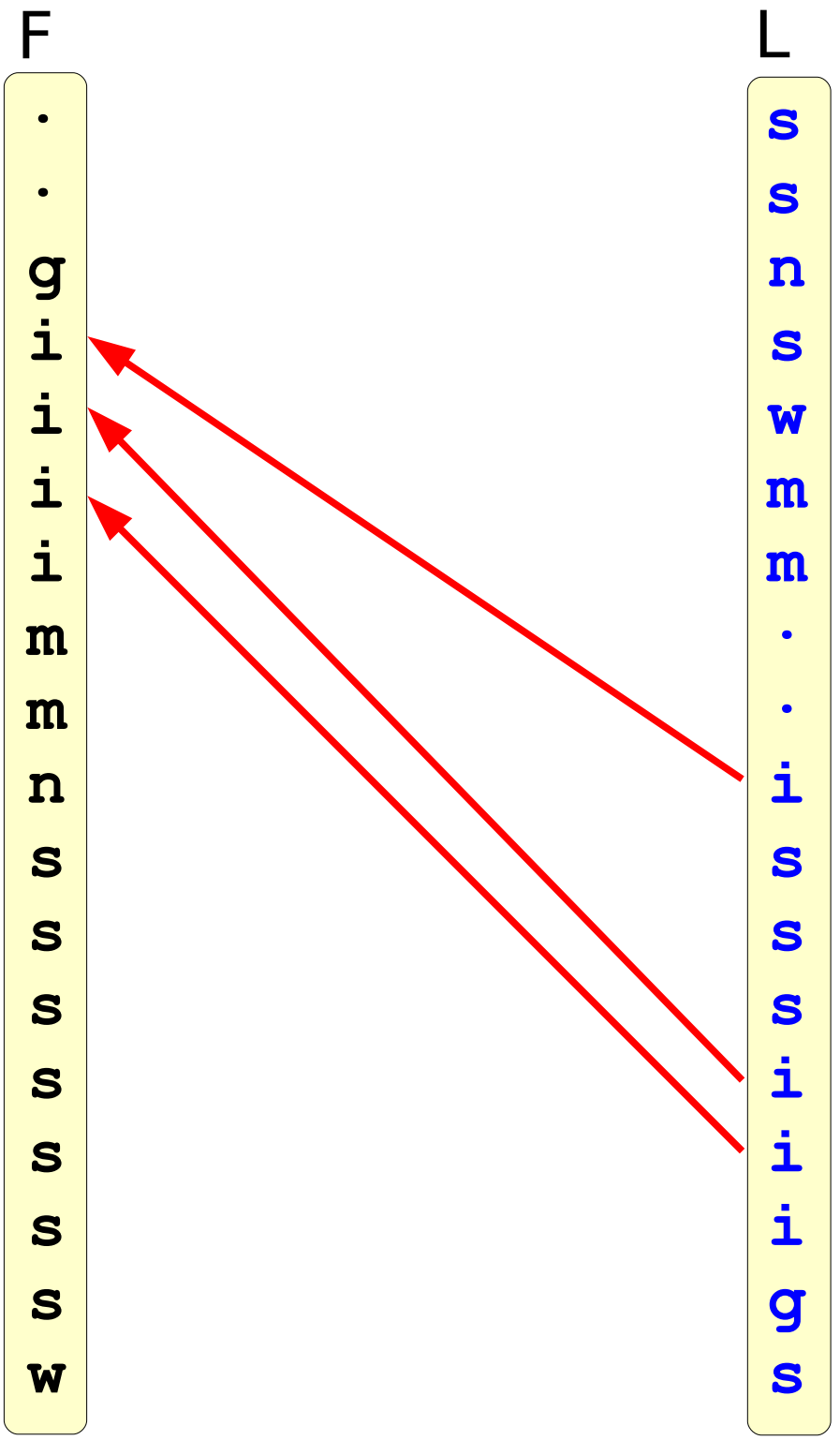


We can map each character in L to its image in F



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

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



Summing up

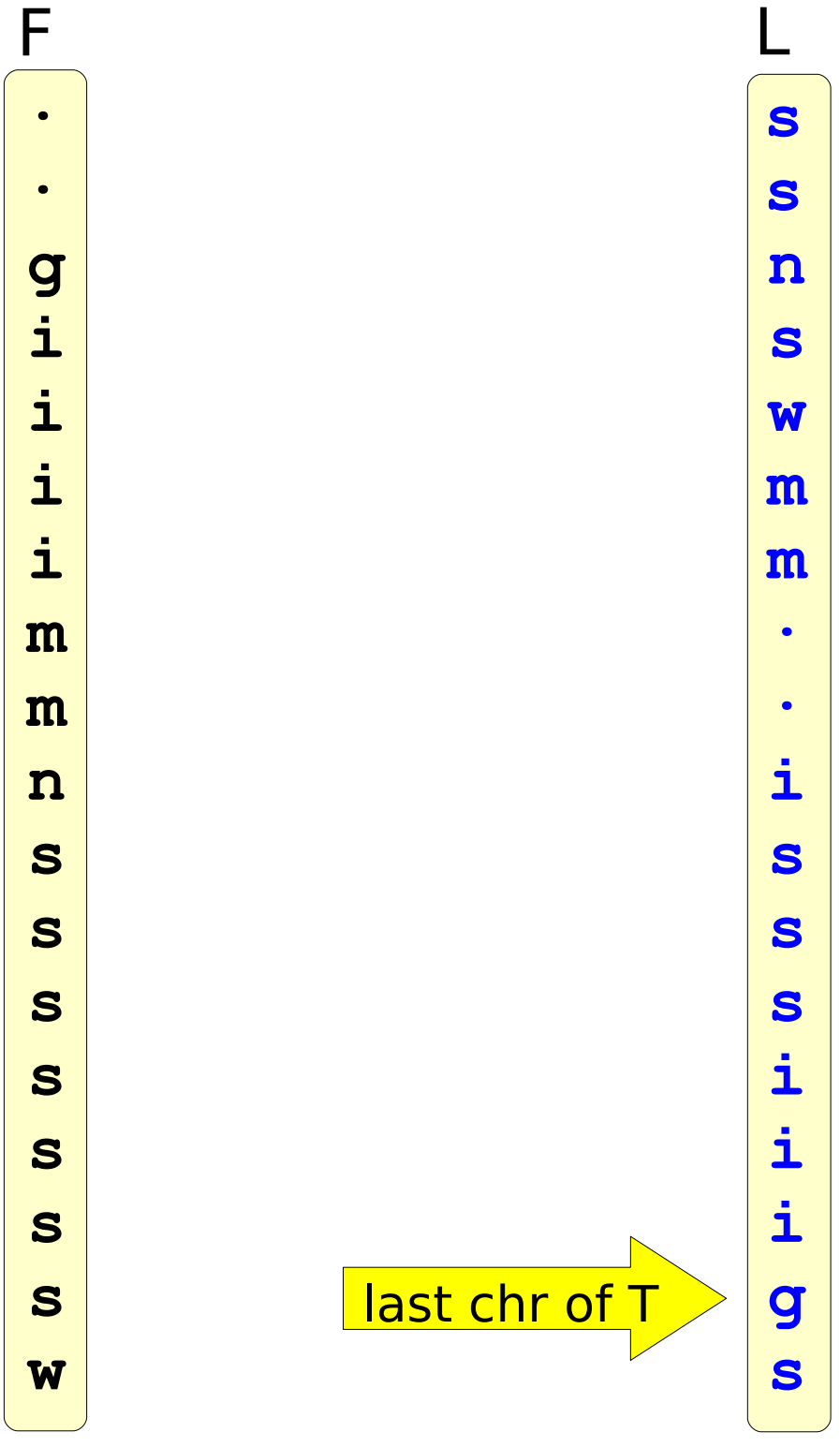
The relative order of the occurrences of any given character in F and L is the same

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

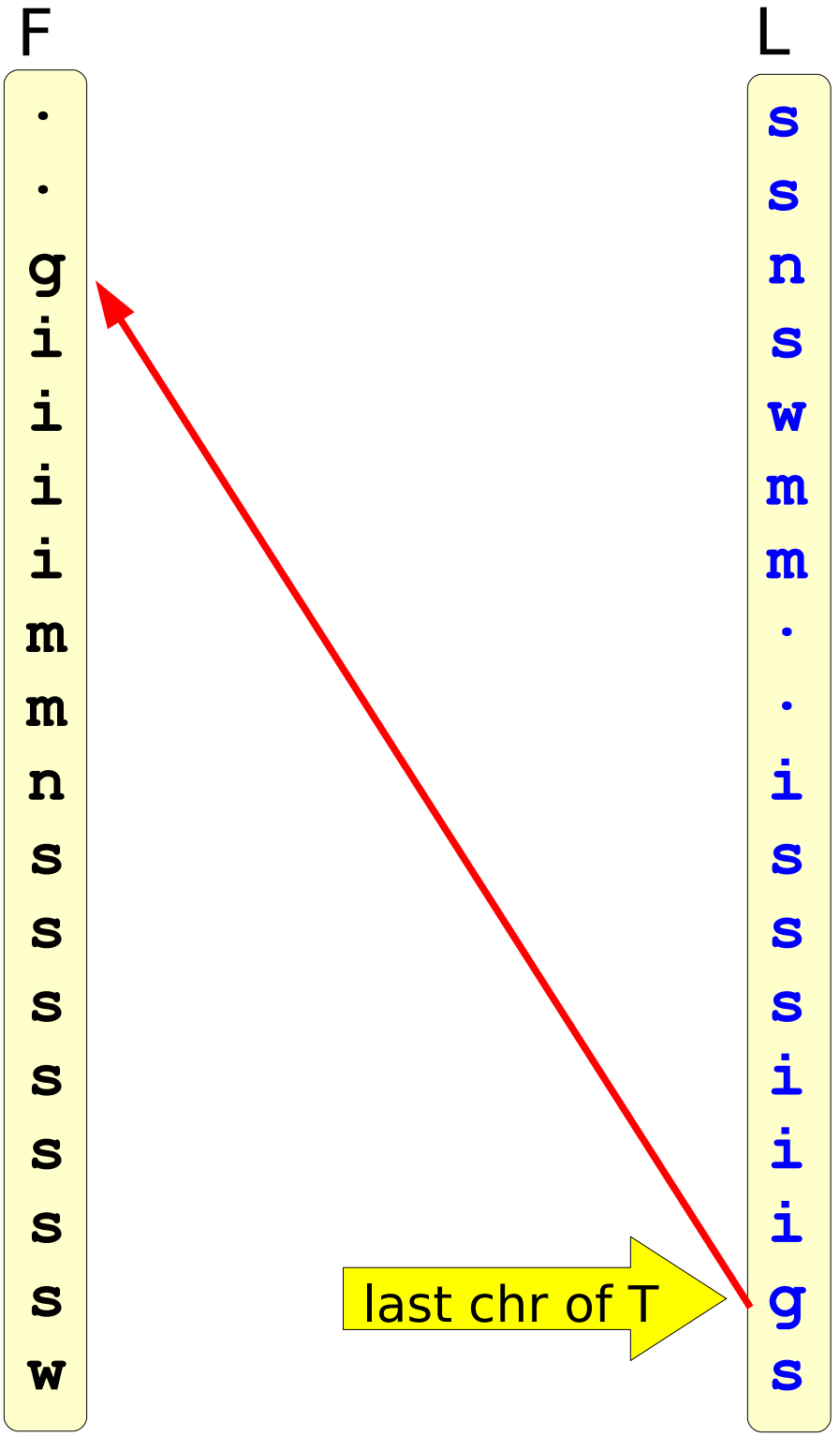
Using the LF map we retrieve T right-to-left

T = g



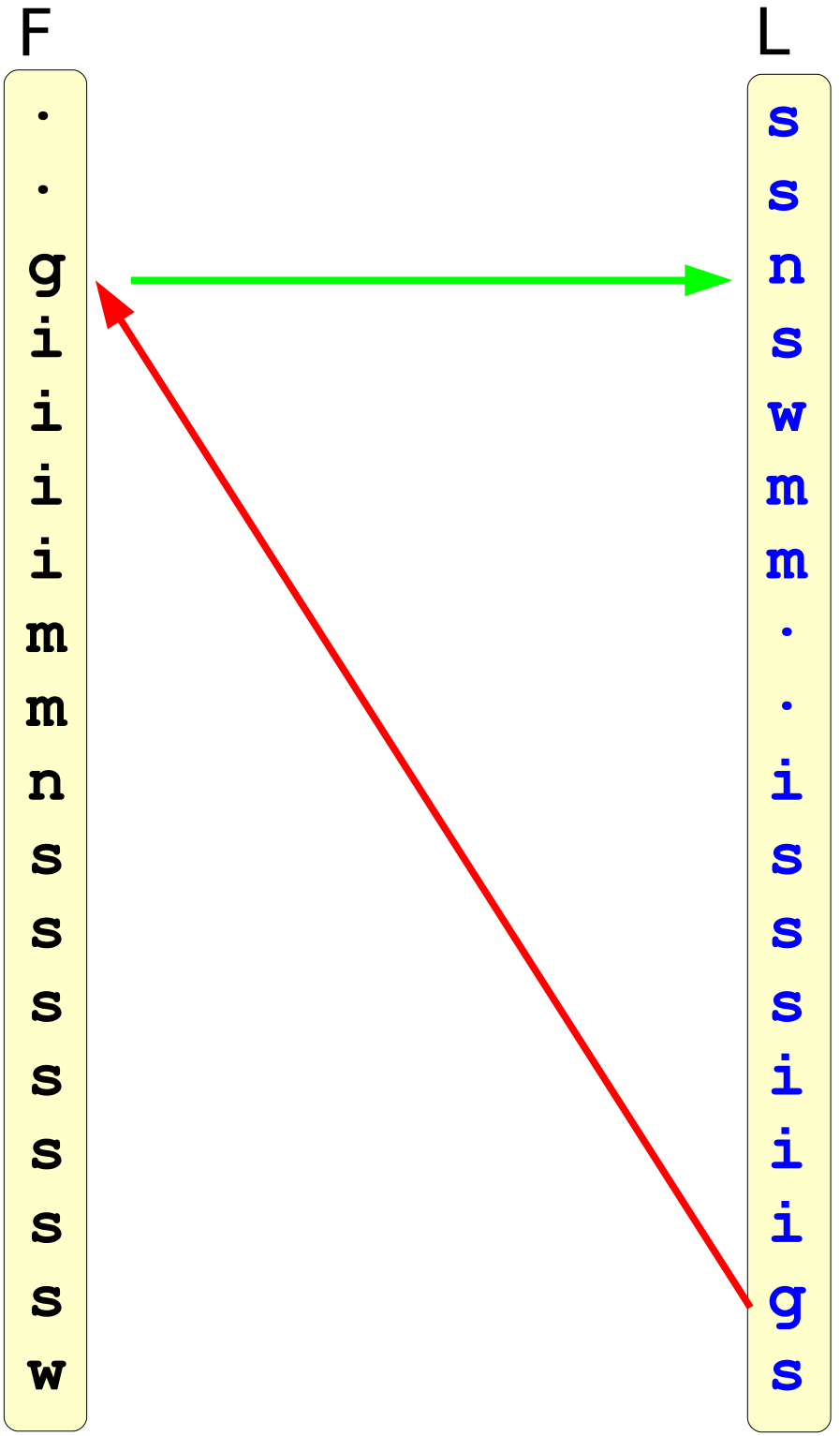
Using the LF map we retrieve T right-to-left

T = g



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

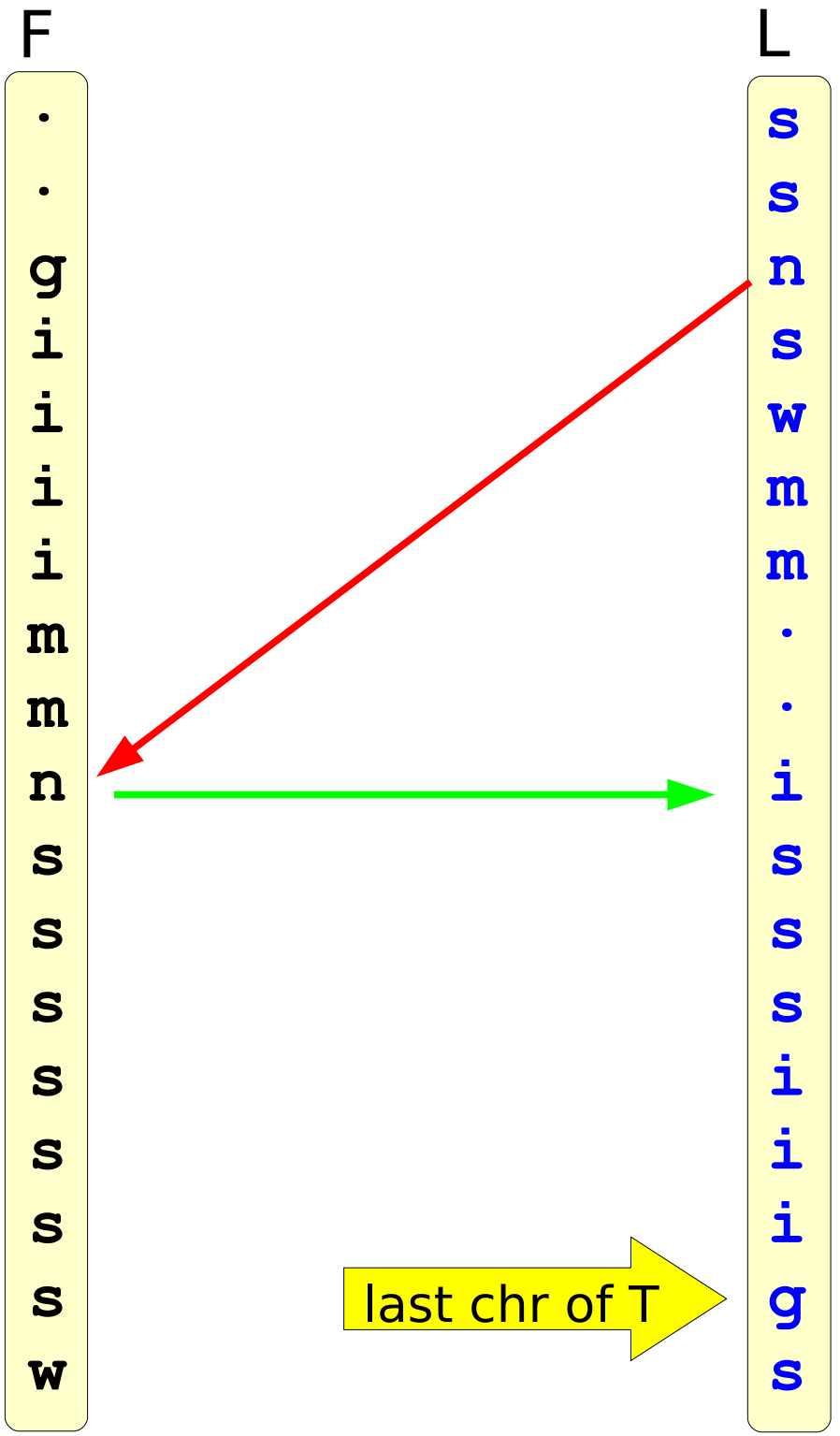
T = ng



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

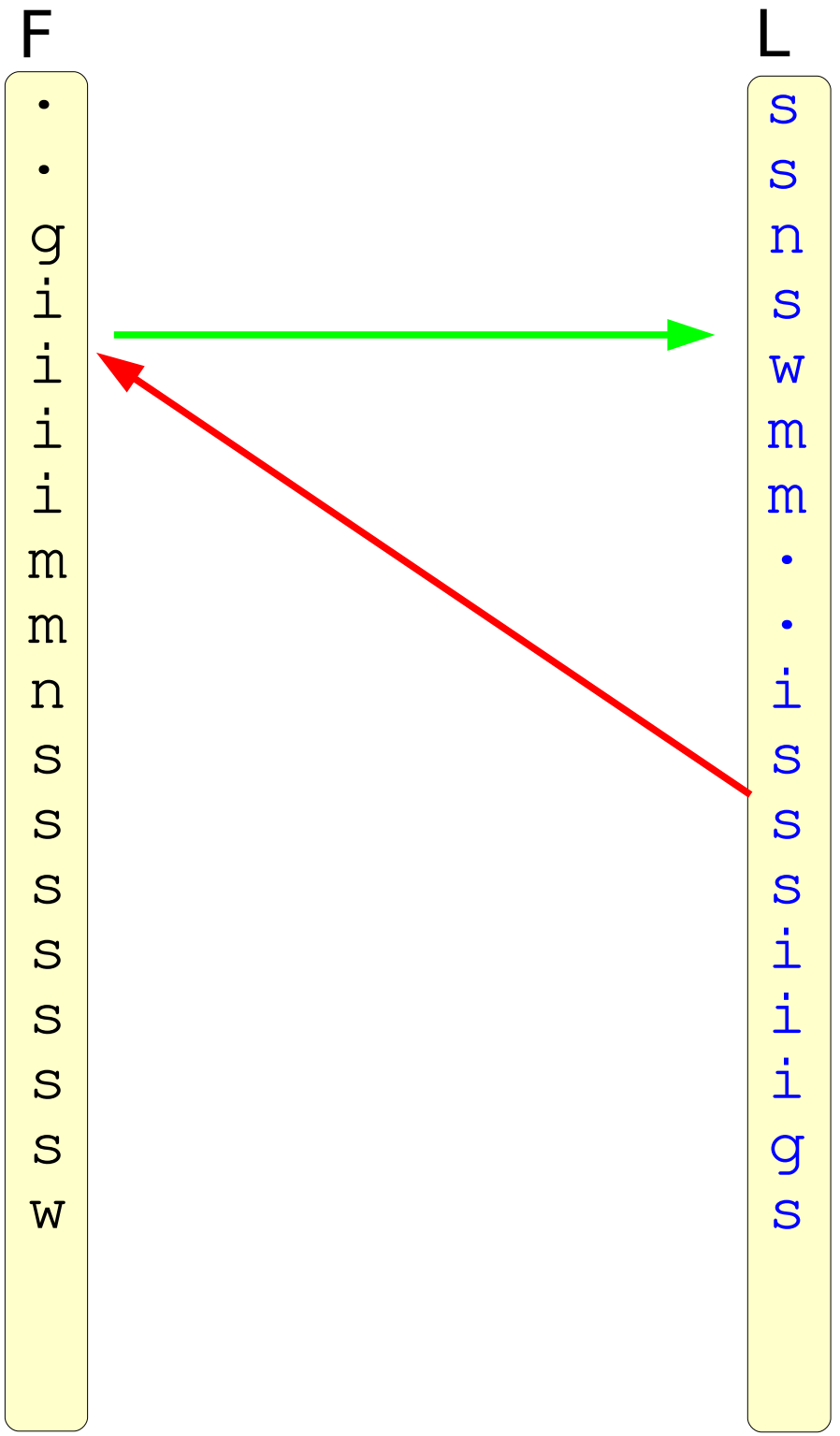
T = ing

...and so on



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

T = **sing**

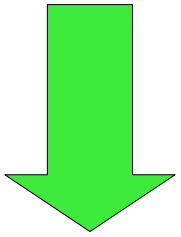


What about compression?

Since BWT is a permutation of T, why should it be “easier” to compress?

The reason is that BWT is usually “locally homogeneous”

The rows prefixed by any given substring are consecutive in the SA



The characters preceding any given substring are consecutive in the BWT

	L
·miss·missingswis	s
·missingswiss·mis	s
gswiss·miss·missi	n
ingswiss·miss·mis	s
iss·miss·missings	w
iss·missingswiss·	m
issingswiss·miss·	m
miss·missingswiss	·
missingswiss·miss	·
ngswiss·miss·miss	i
s·miss·missingswi	s
s·missingswiss·mi	s
singswiss·miss·mi	s
ss·miss·missingsw	i
ss·missingswiss·m	i
ssingswiss·miss·m	i
swiss·miss·missin	g
wiss·miss·missing	s

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

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 $L[i]$ to be the
i	n turn, set $R[i]$ 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.

Even a simple run length encoding gives good compression

The bzip2 compressor based on the BWT is usually better than gzip but not of xz or lzma, or zstd

BWT running time

- If we compute the Suffix Array using a linear time algorithm (eg. the skew algorithm) then computing the BWT takes linear time
- Inverting the BWT takes linear time (see python code), but because of the jumping in the L column, there are many cache misses so the constant factor is relatively high

For a while the BWT appeared just as an original tool for data compression.

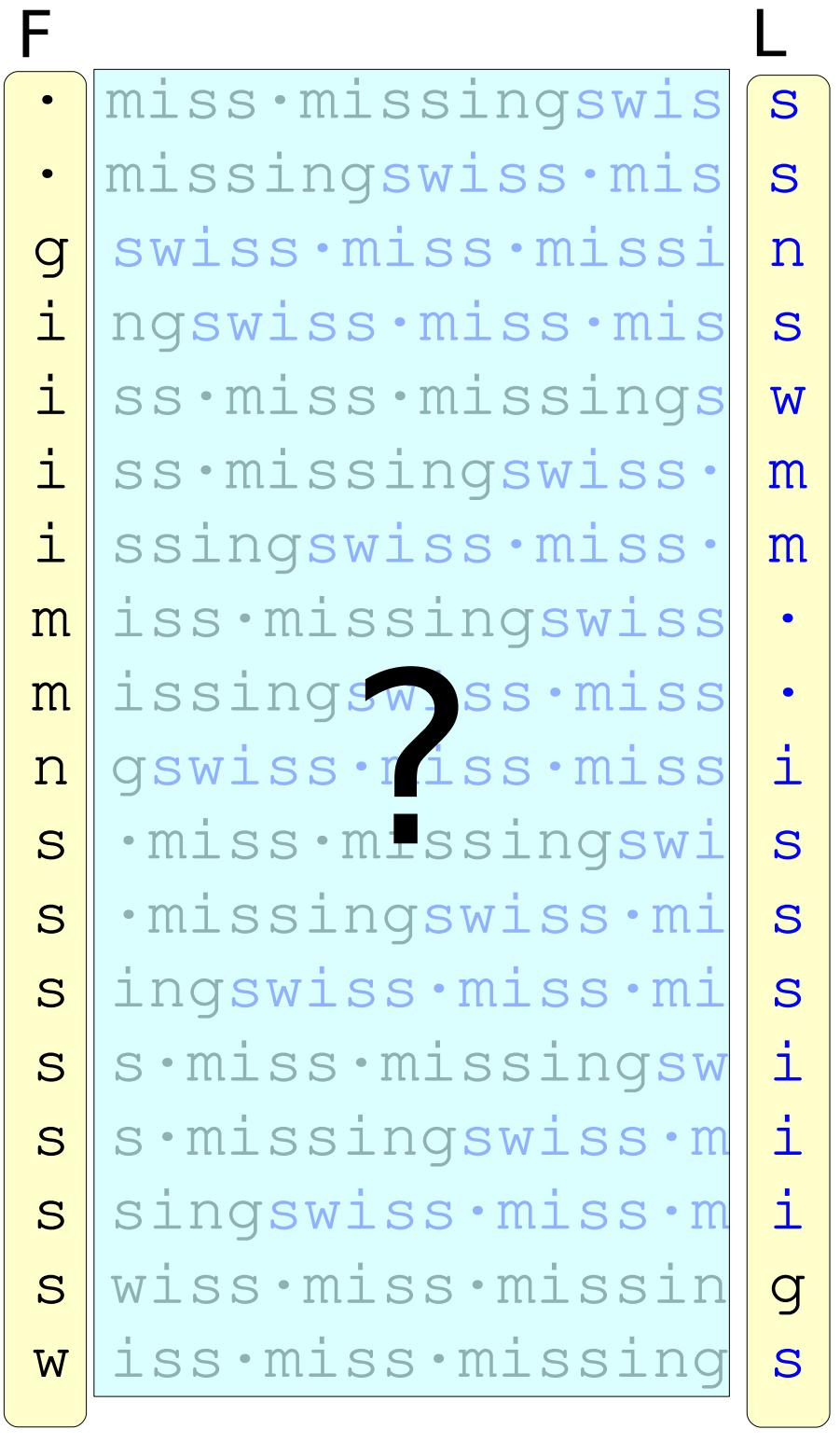
Starting with the year 2000, researchers started to realize that the similarity between the BWT and the Suffix Array can be exploited to use the BWT as a sort of “compressed Suffix Array”

Search using the BWT

Suppose we want to count the occurrences of mis

Working with only F and L we successively find the range of rows prefixed by:

s, is, mis



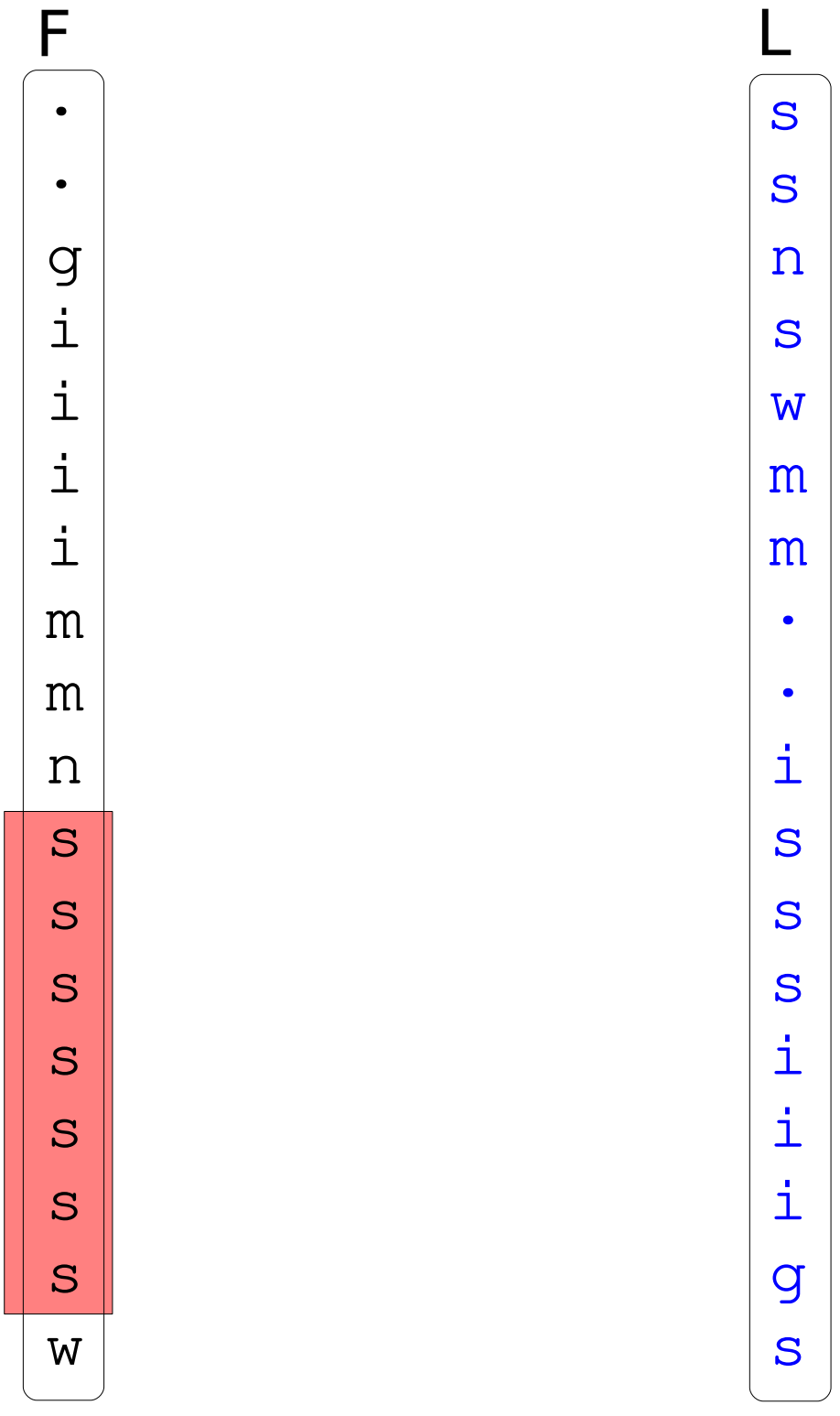
Backward search

The rows prefixed by s are easy to find using F

We represent F storing the position of the first occ of each symbol:

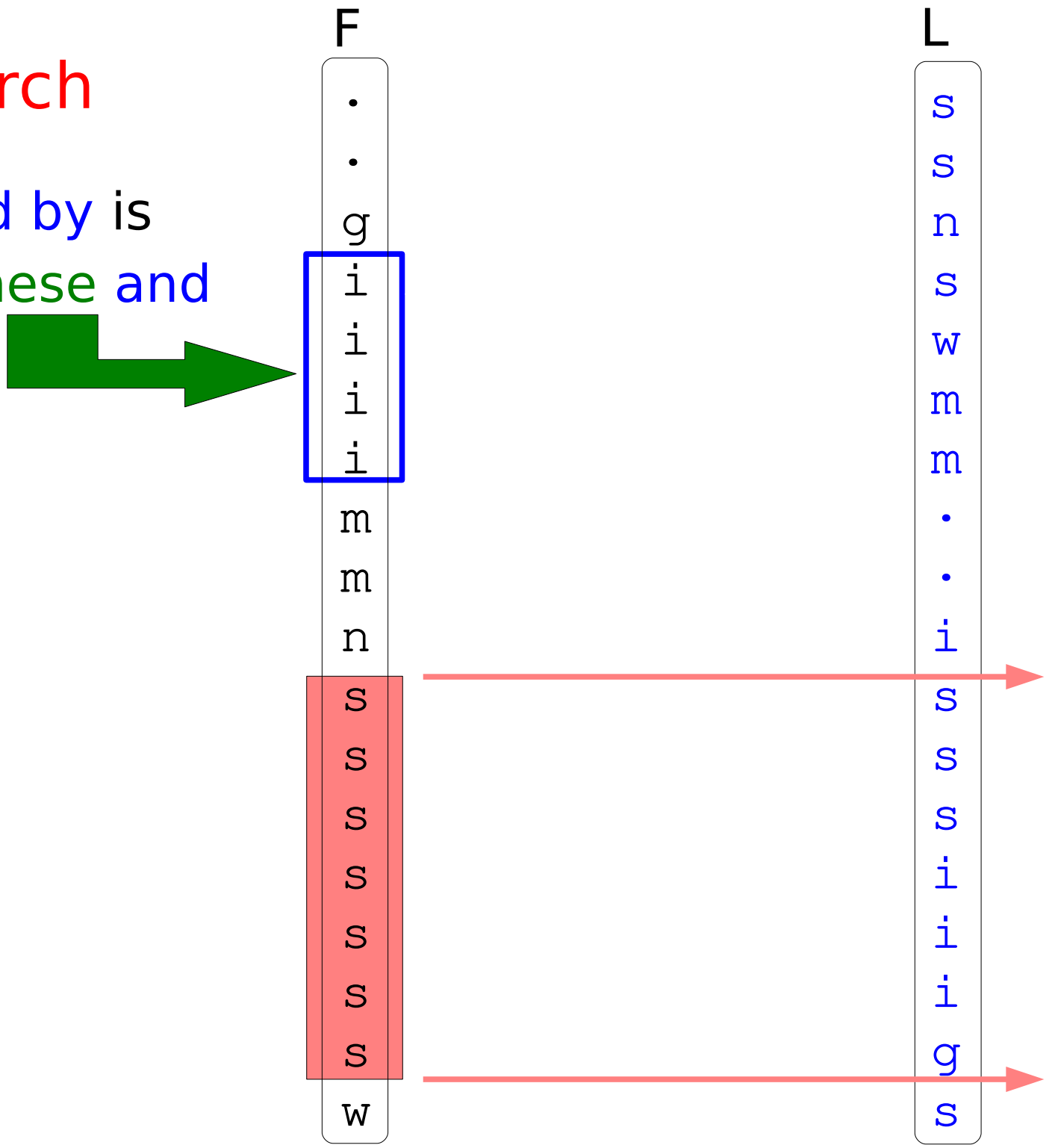
· →1 g →3 i →4
m →8 n →10
s →11 w →18

the total cost is:
O(|A| log t) bits



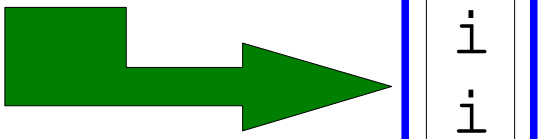
Backward search

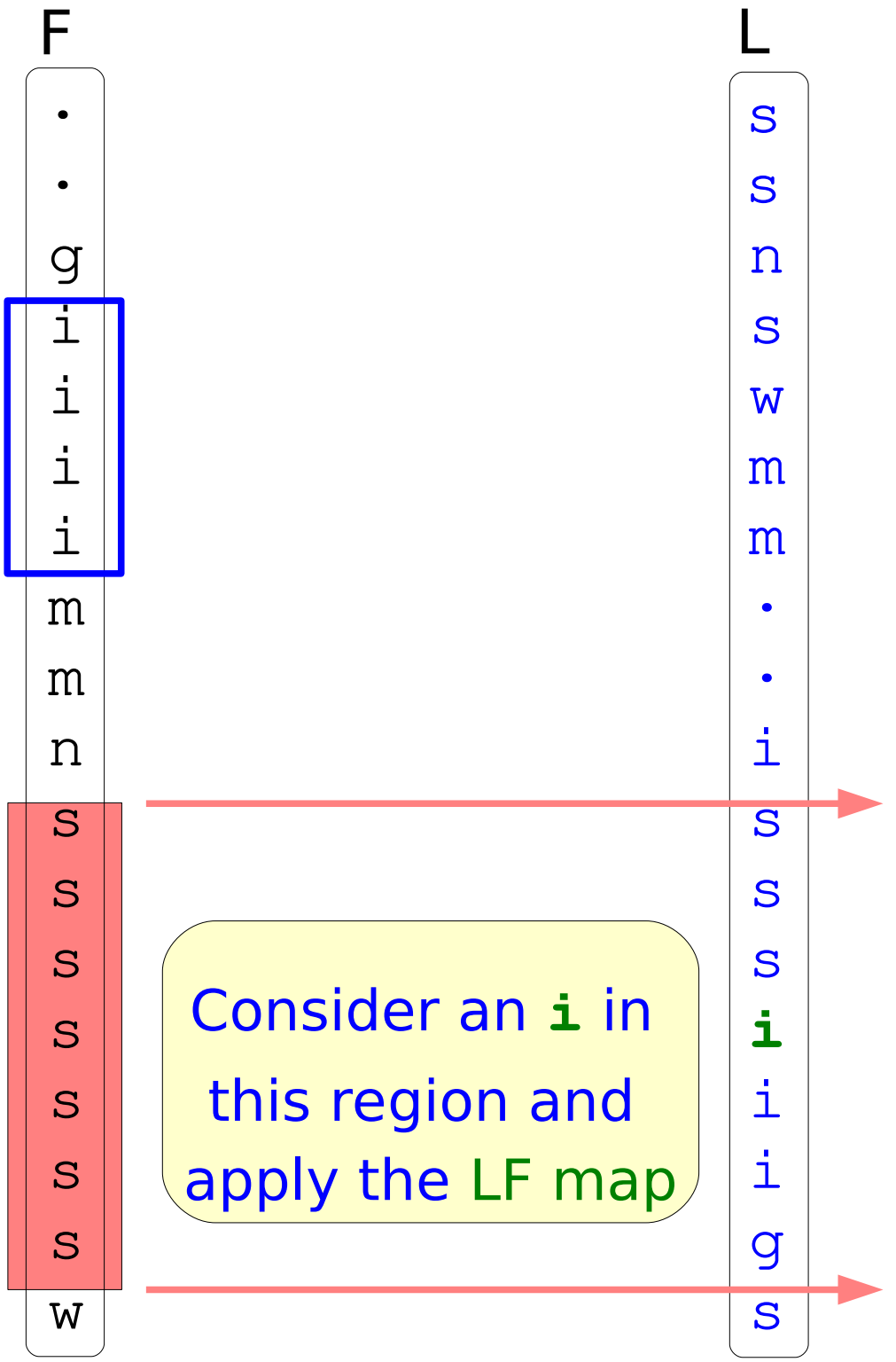
The rows prefixed by is are a subset of these and are consecutive.



Backward search

The rows prefixed by is are a subset of these and are consecutive.

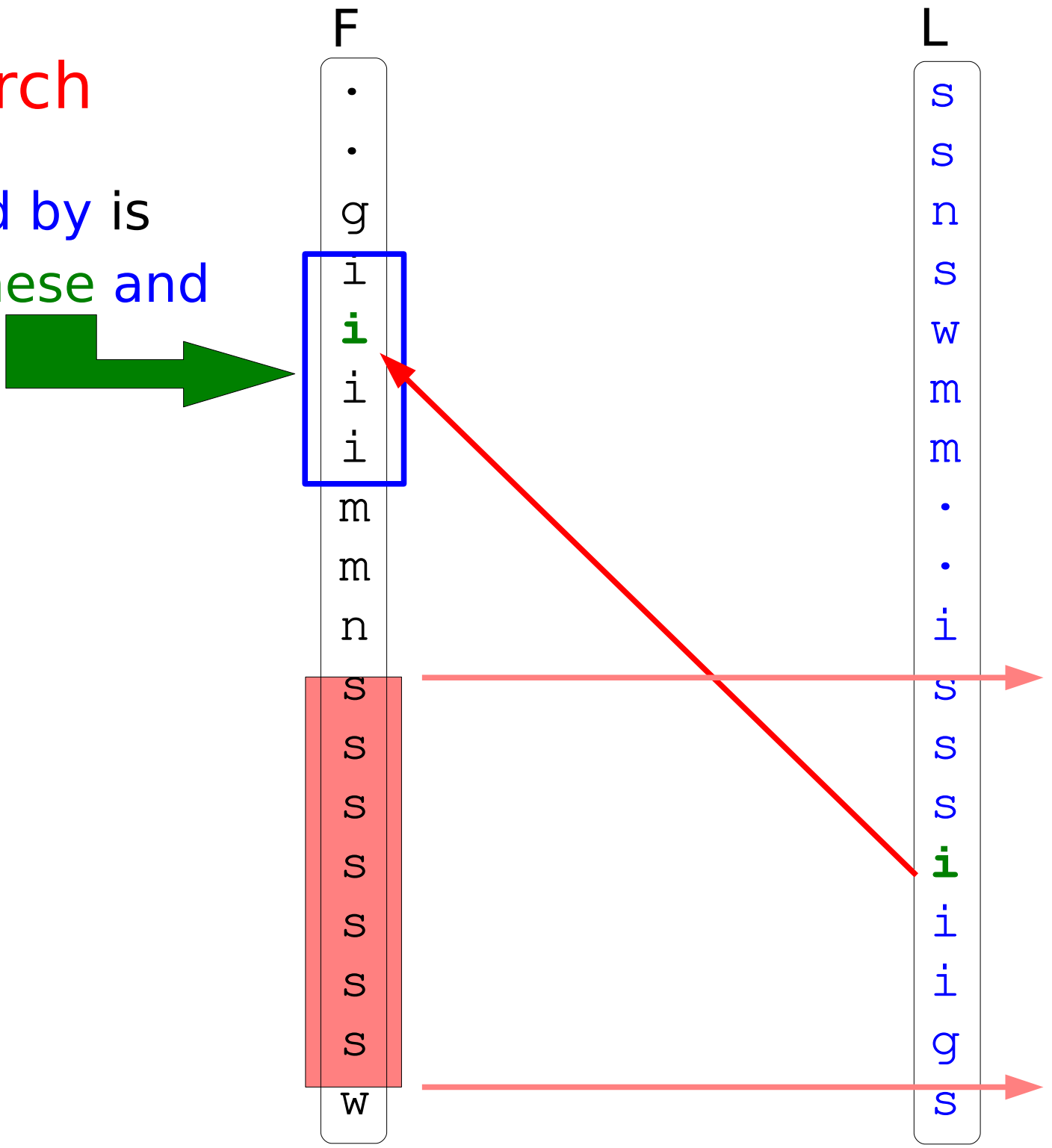




Consider an **i** in this region and apply the LF map

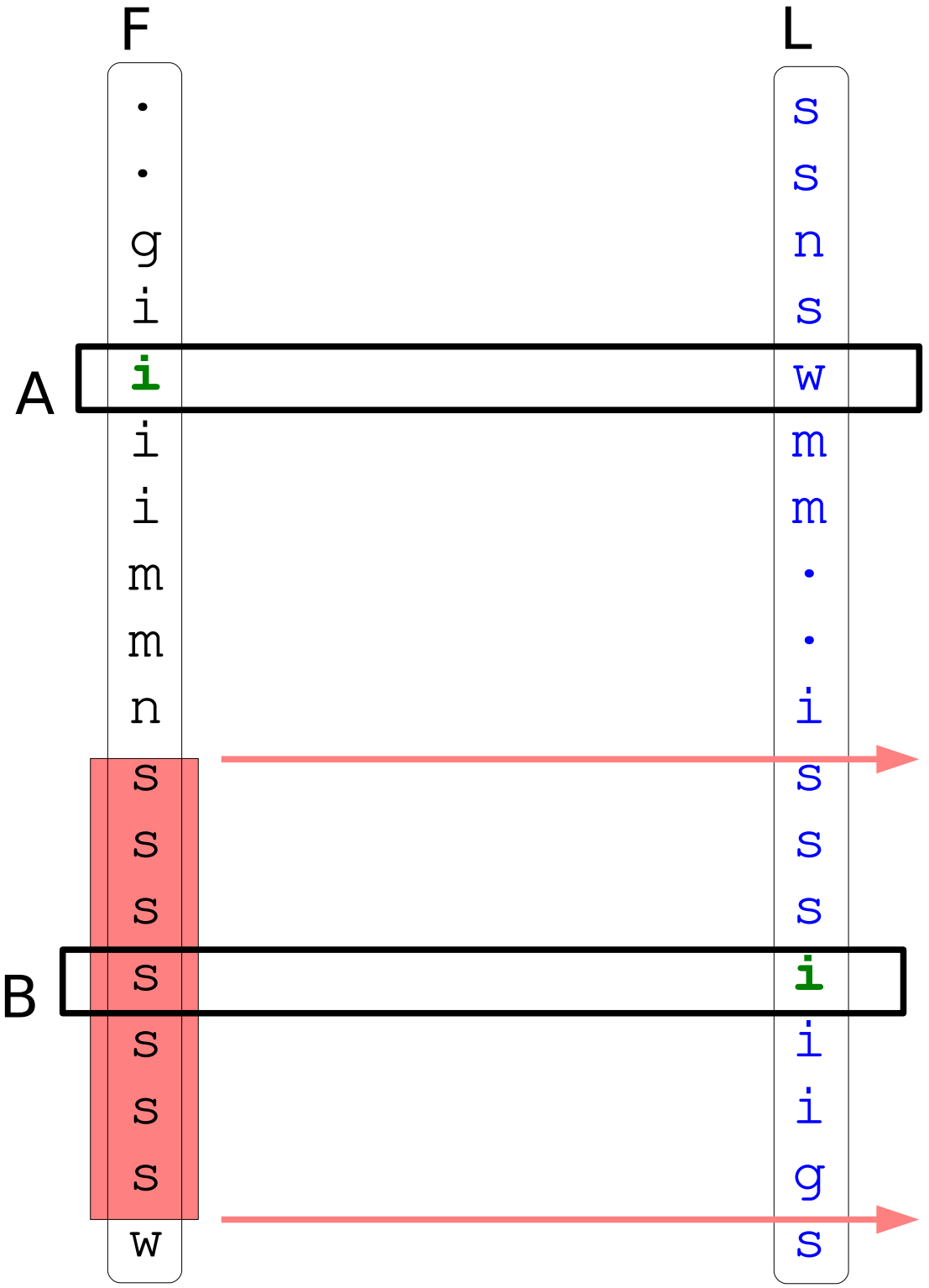
Backward search

The rows prefixed by is are a subset of these and are consecutive.



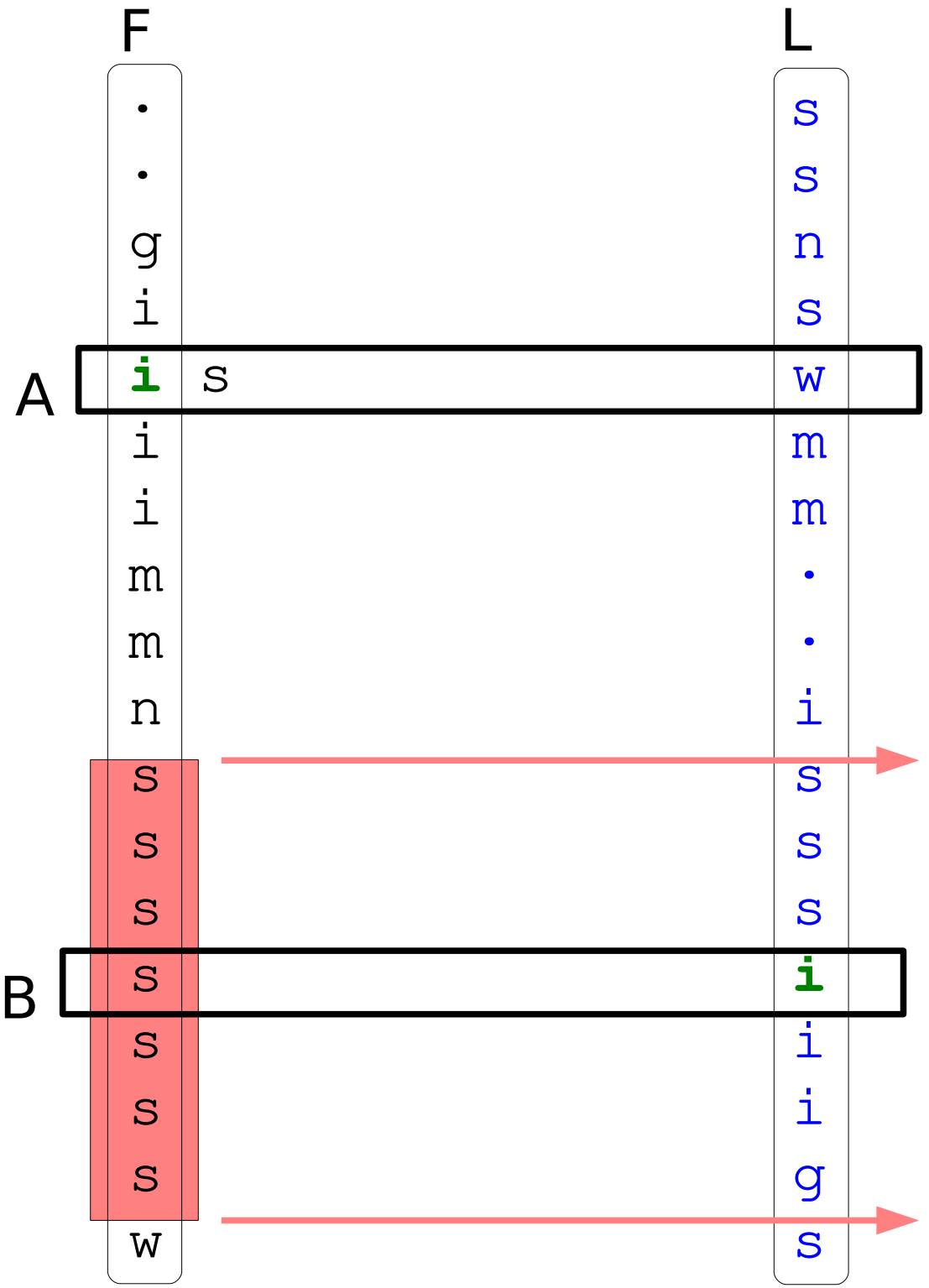
Backward search

Since the two i's are the same, row A is row B left-shifted by 1.

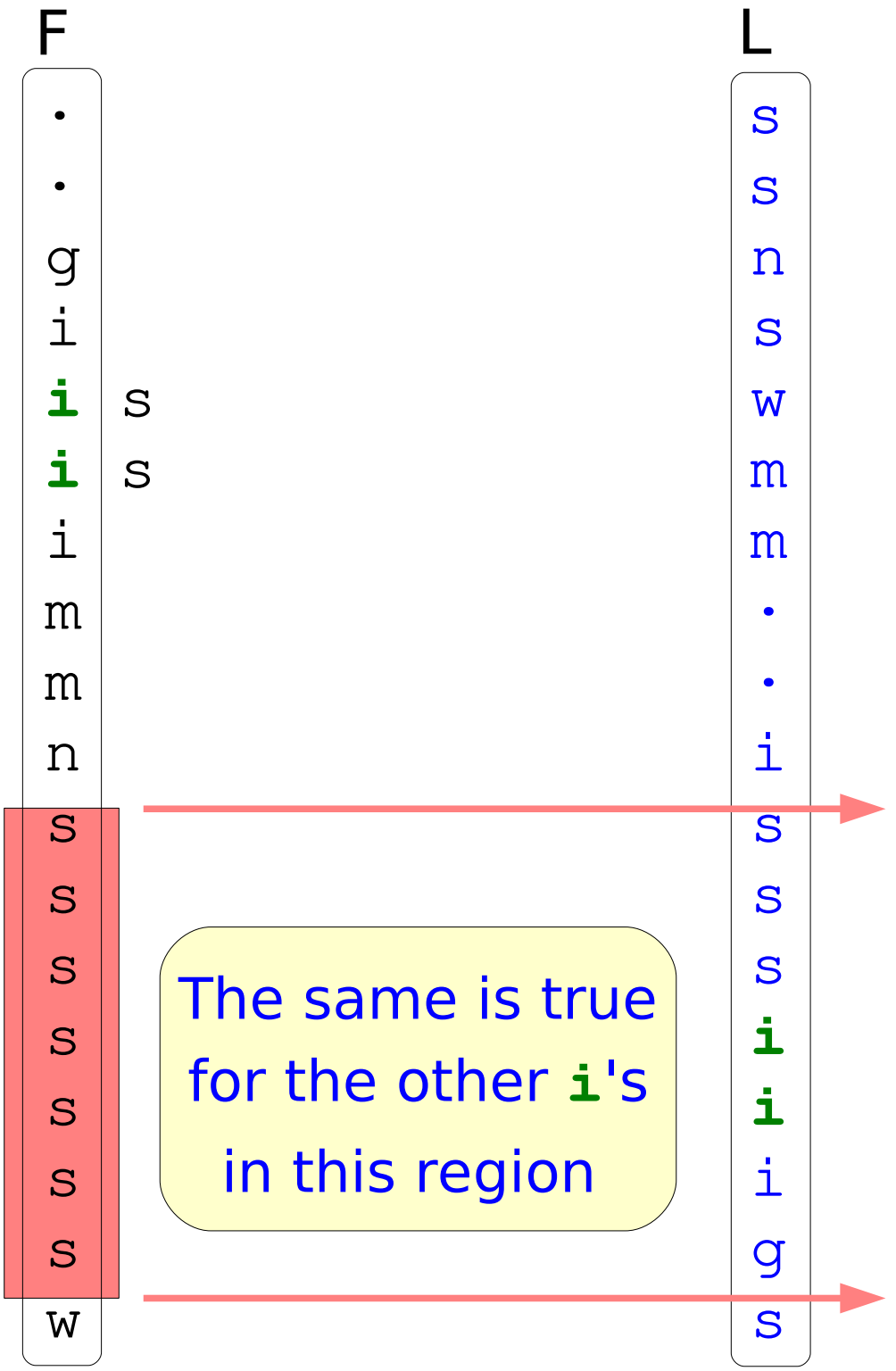


Backward search

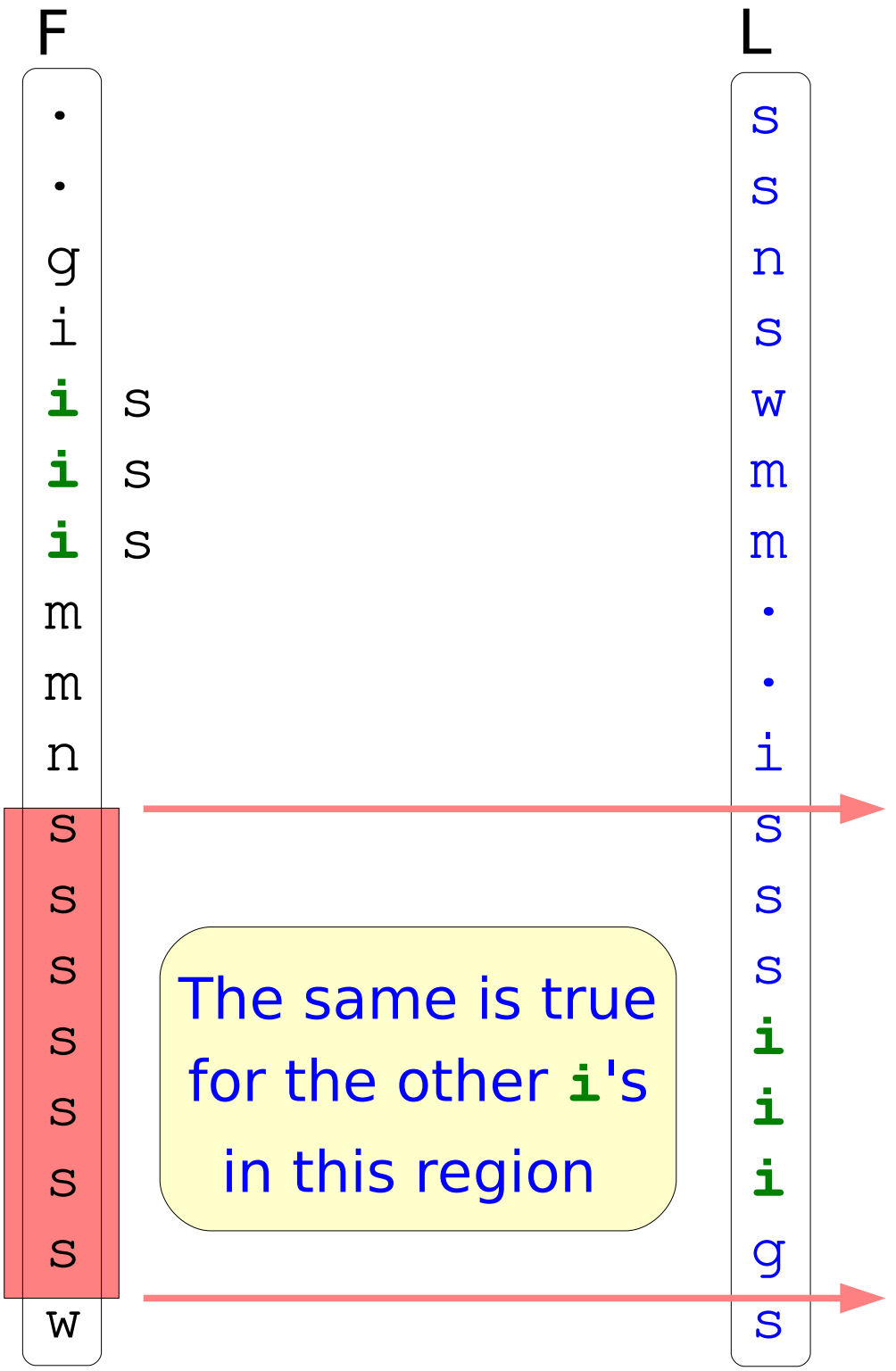
Since the two i's are the same, row A is row B left-shifted by 1.



Backward search

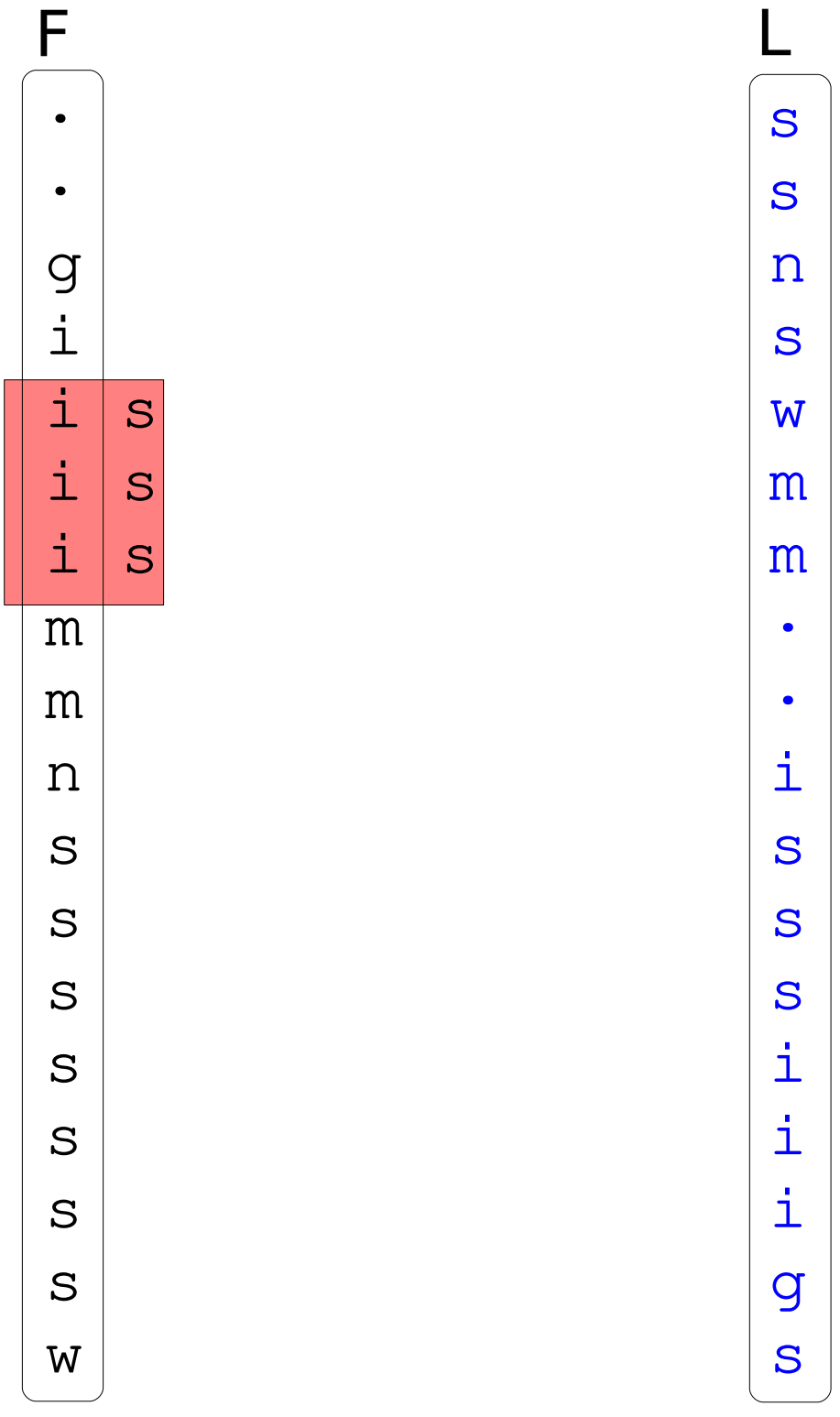


Backward search



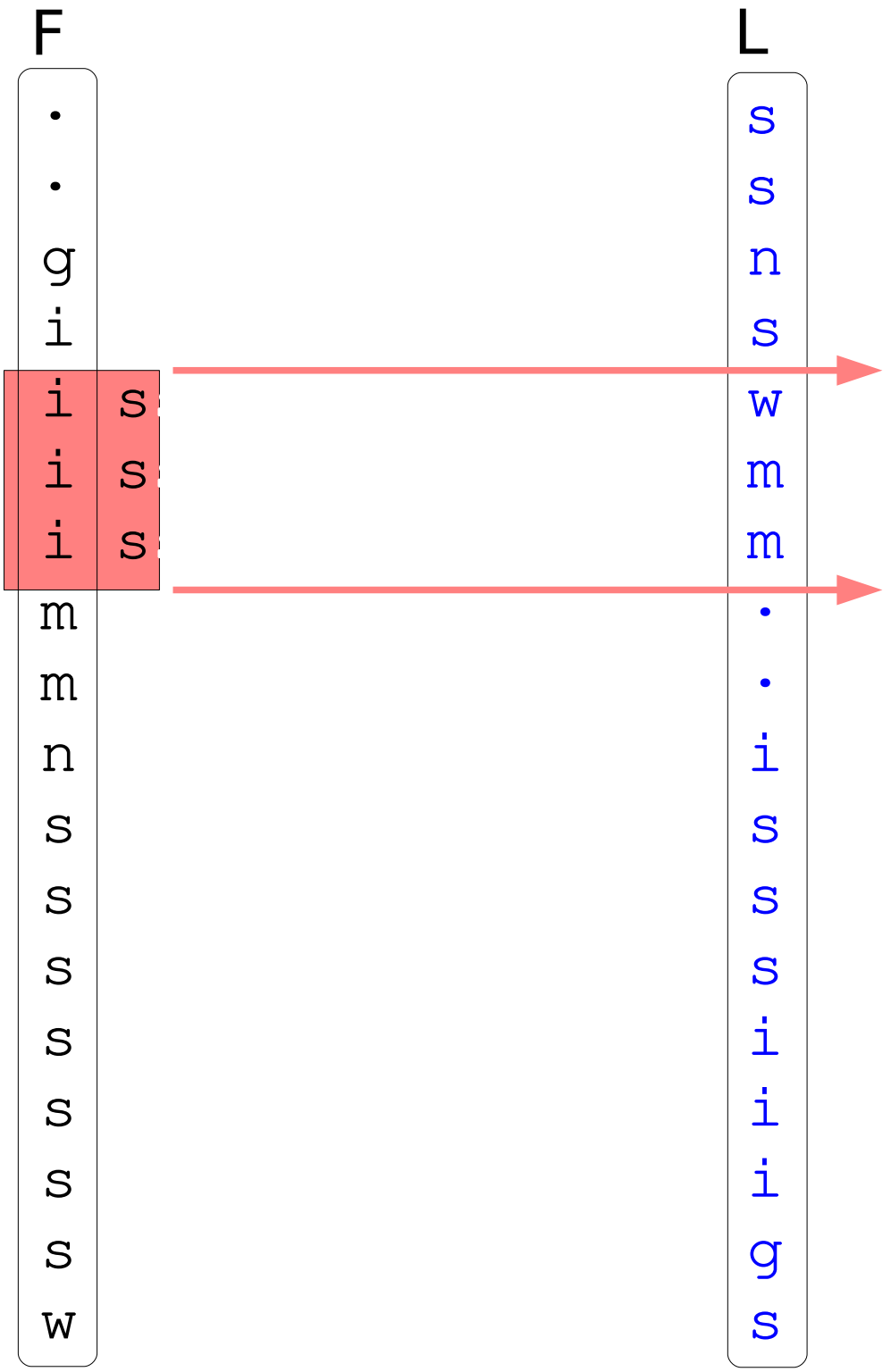
Backward search

We have found the rows prefixed by **is**!



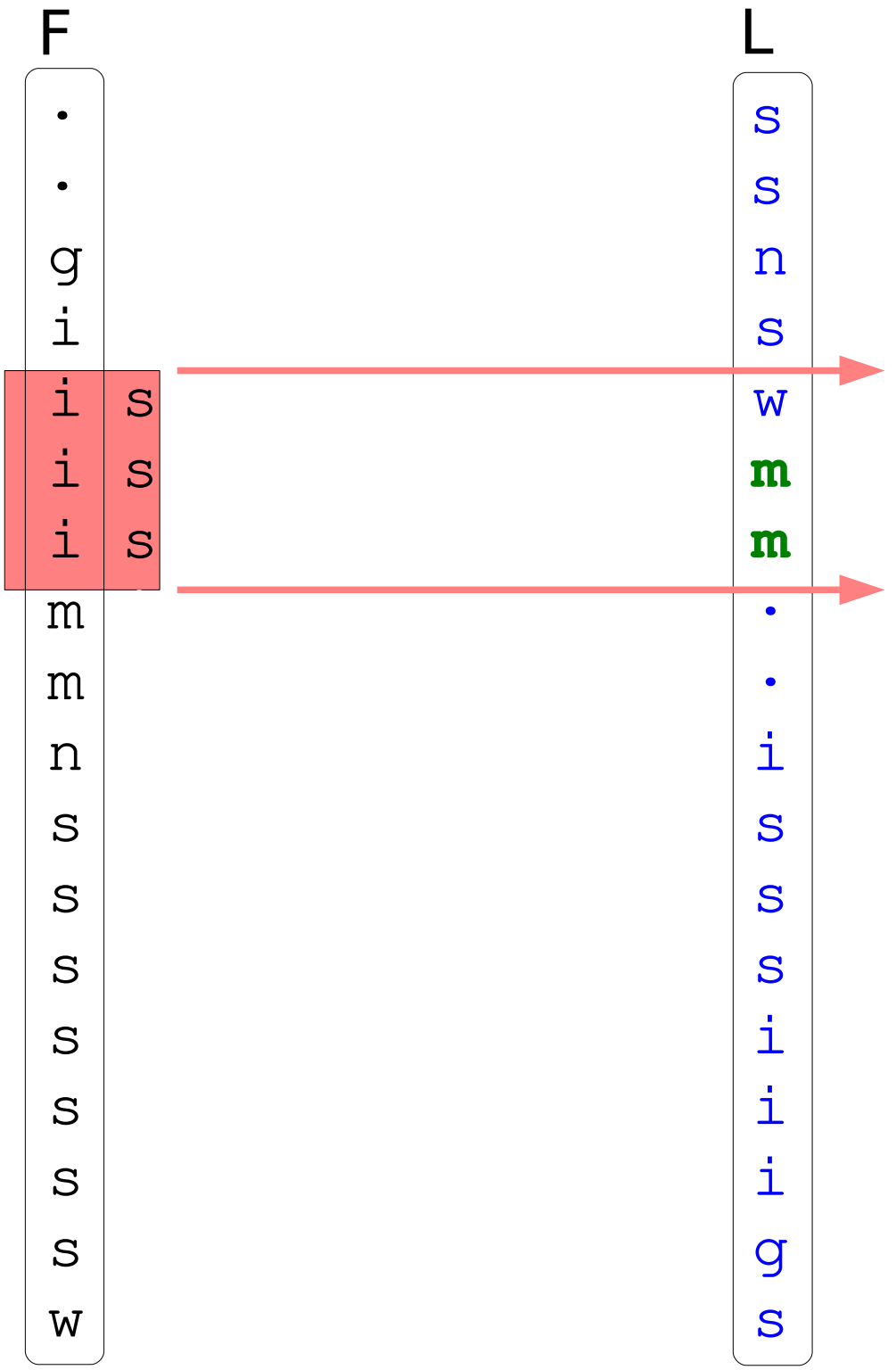
Backward search

To find the rows prefixed by **mis** we proceed in the same way



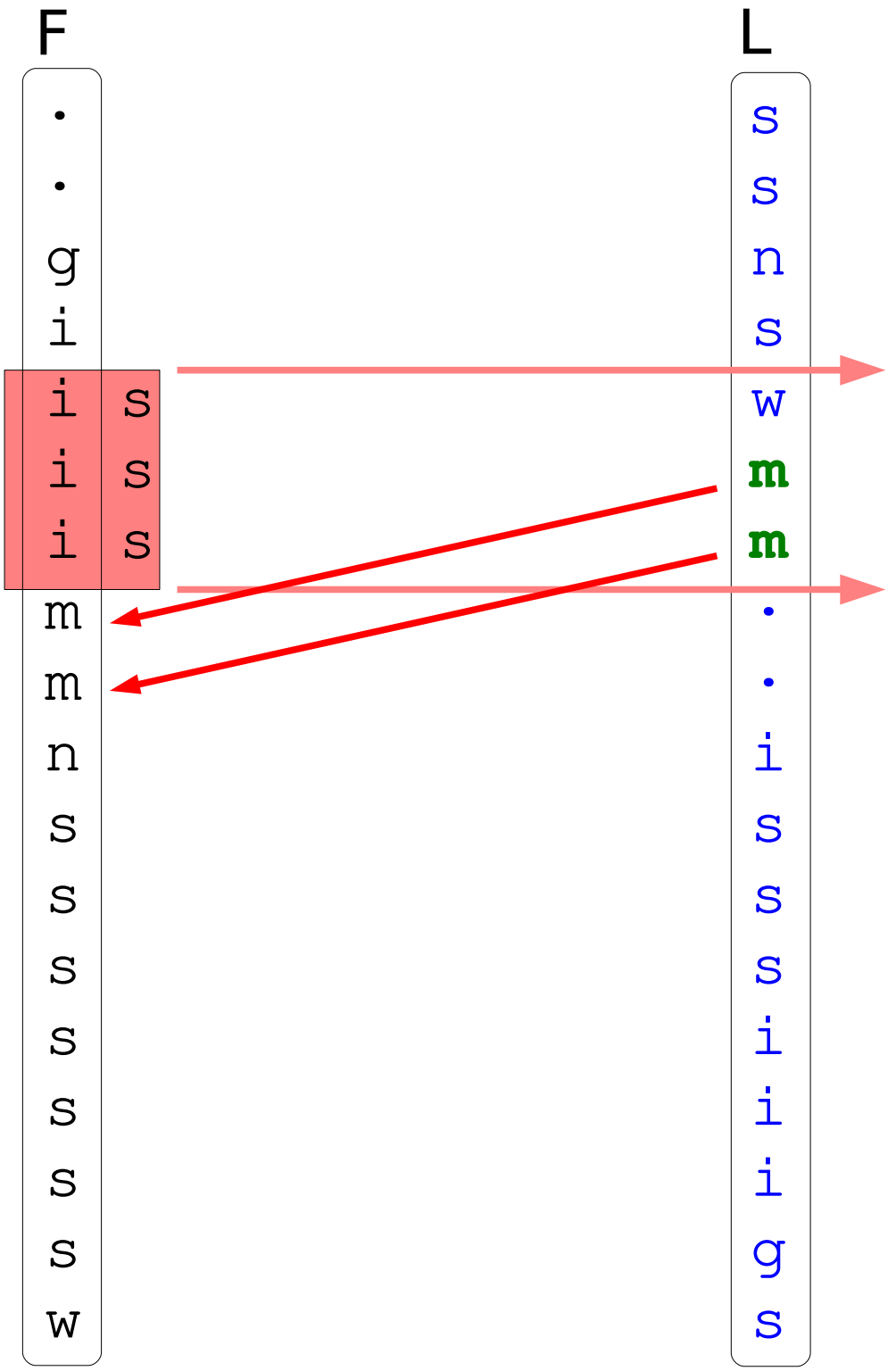
Backward search

To find the rows prefixed by **mis** we proceed in the same way



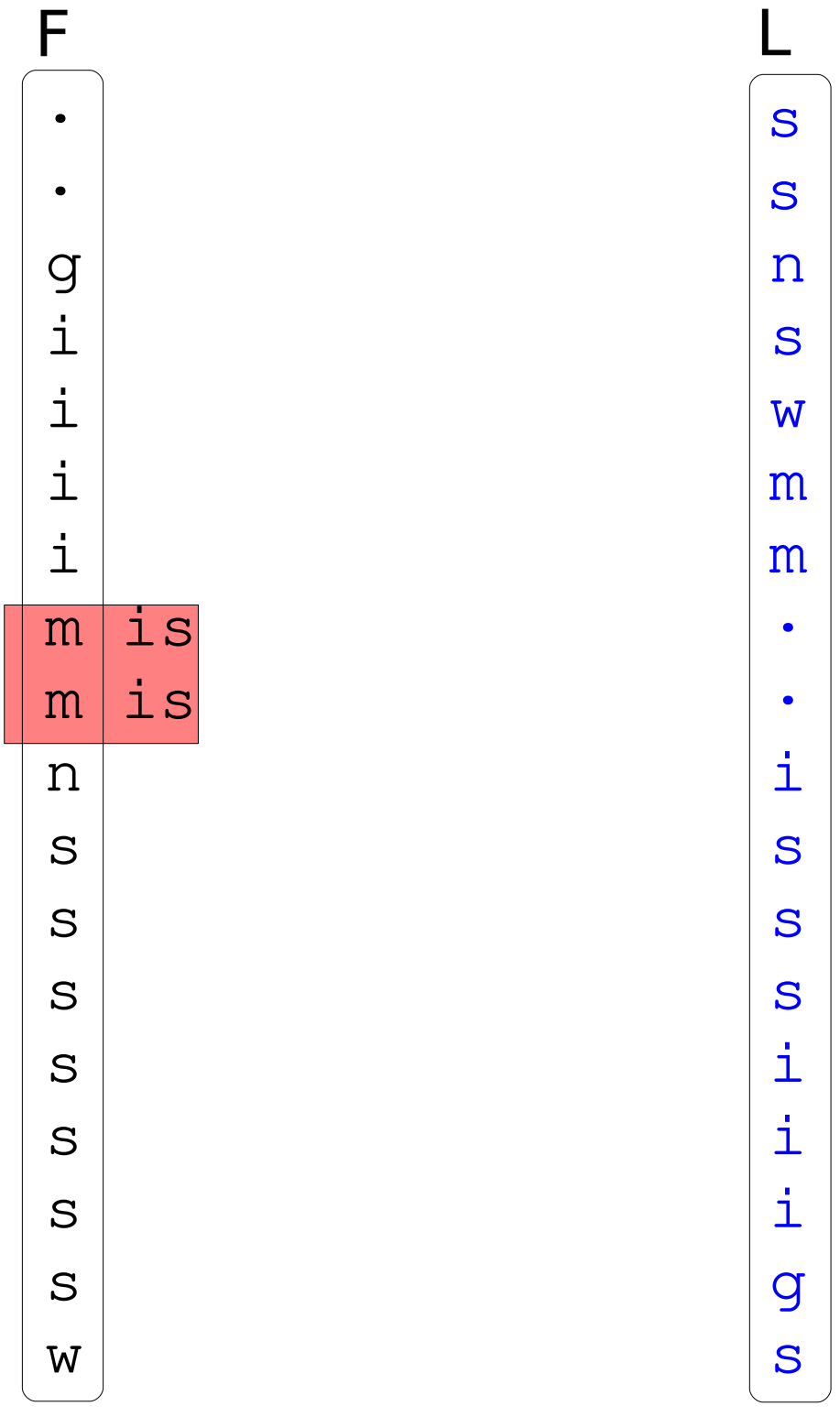
Backward search

To find the rows prefixed by **mis** we proceed in the same way



Backward search

We have found the rows prefixed by **mis**!



Backward search

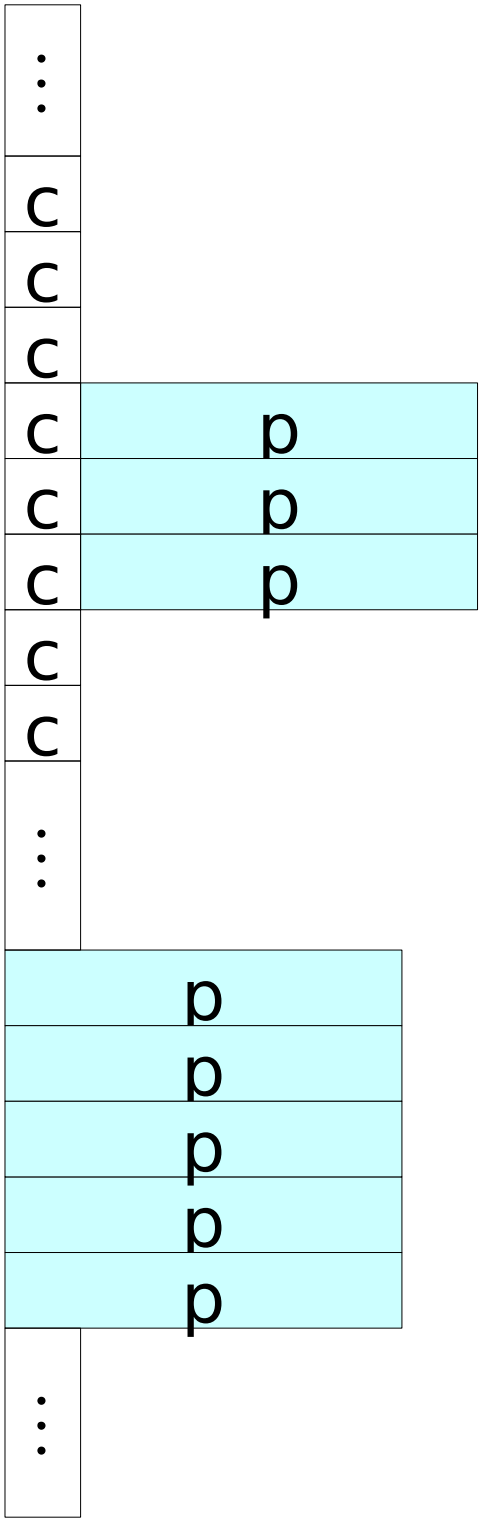
We have found the rows prefixed by **mis**!

F		L
·	miss·missingswis	s
·	missingswiss·mis	s
g	swiss·miss·missi	n
i	ngswiss·miss·mis	s
i	ss·miss·missings	w
i	ss·missingswiss·	m
i	ssingswiss·miss·	m
m	iss·missingswiss	·
m	issingswiss·miss	·
n	gswiss·miss·miss	i
s	·miss·missingswi	s
s	·missingswiss·mi	s
s	ingswiss·miss·mi	s
s	s·miss·missingsw	i
s	s·missingswiss·m	i
s	singswiss·miss·m	i
s	wiss·miss·missin	g
w	iss·miss·missing	s

Even if we only have F and L we can work as if we had the SA

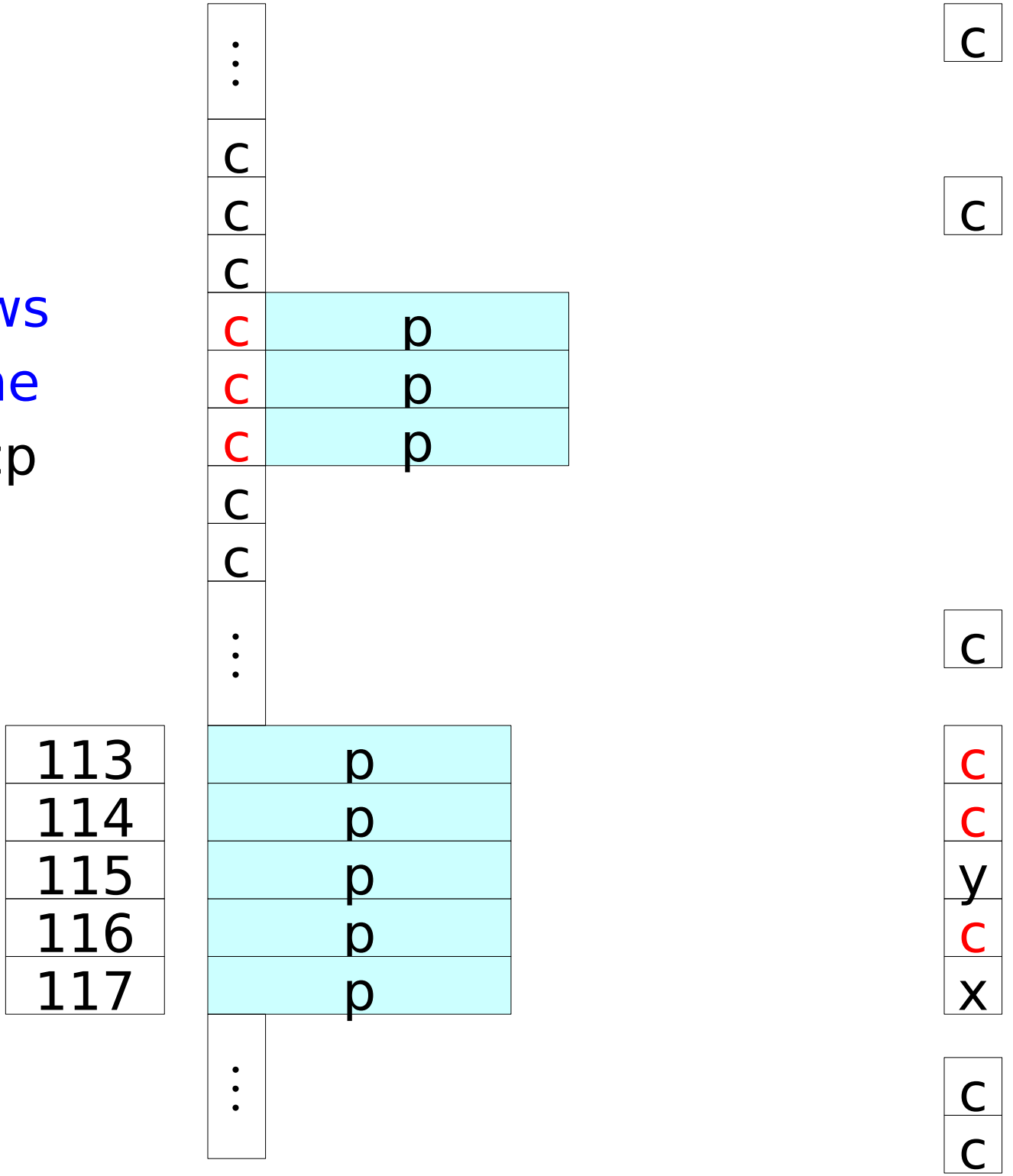
A closer look

The basic step is going from the rows prefixed by p to the rows prefixed by cp



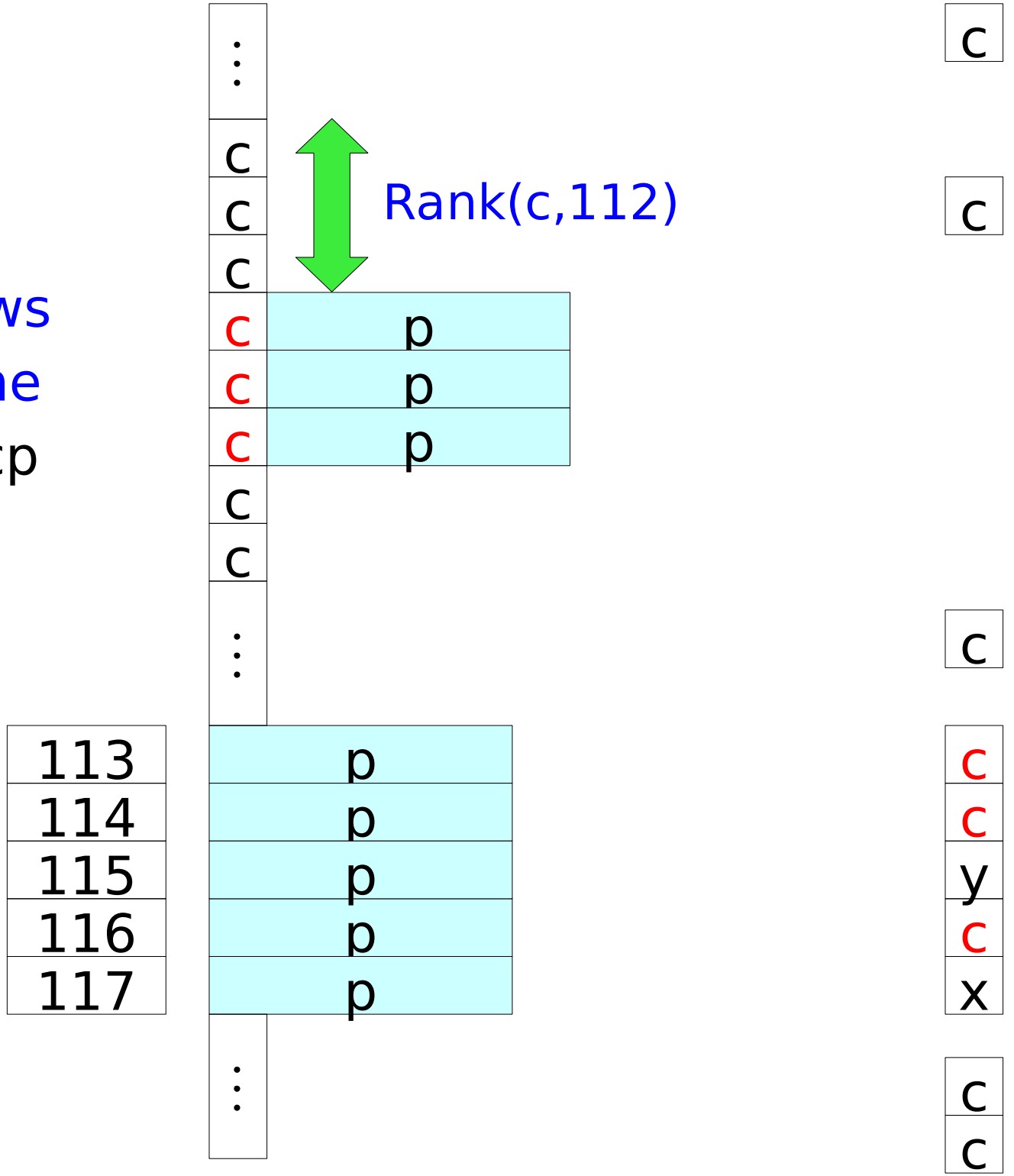
A closer look

The basic step is going from the rows prefixed by p to the rows prefixed by cp



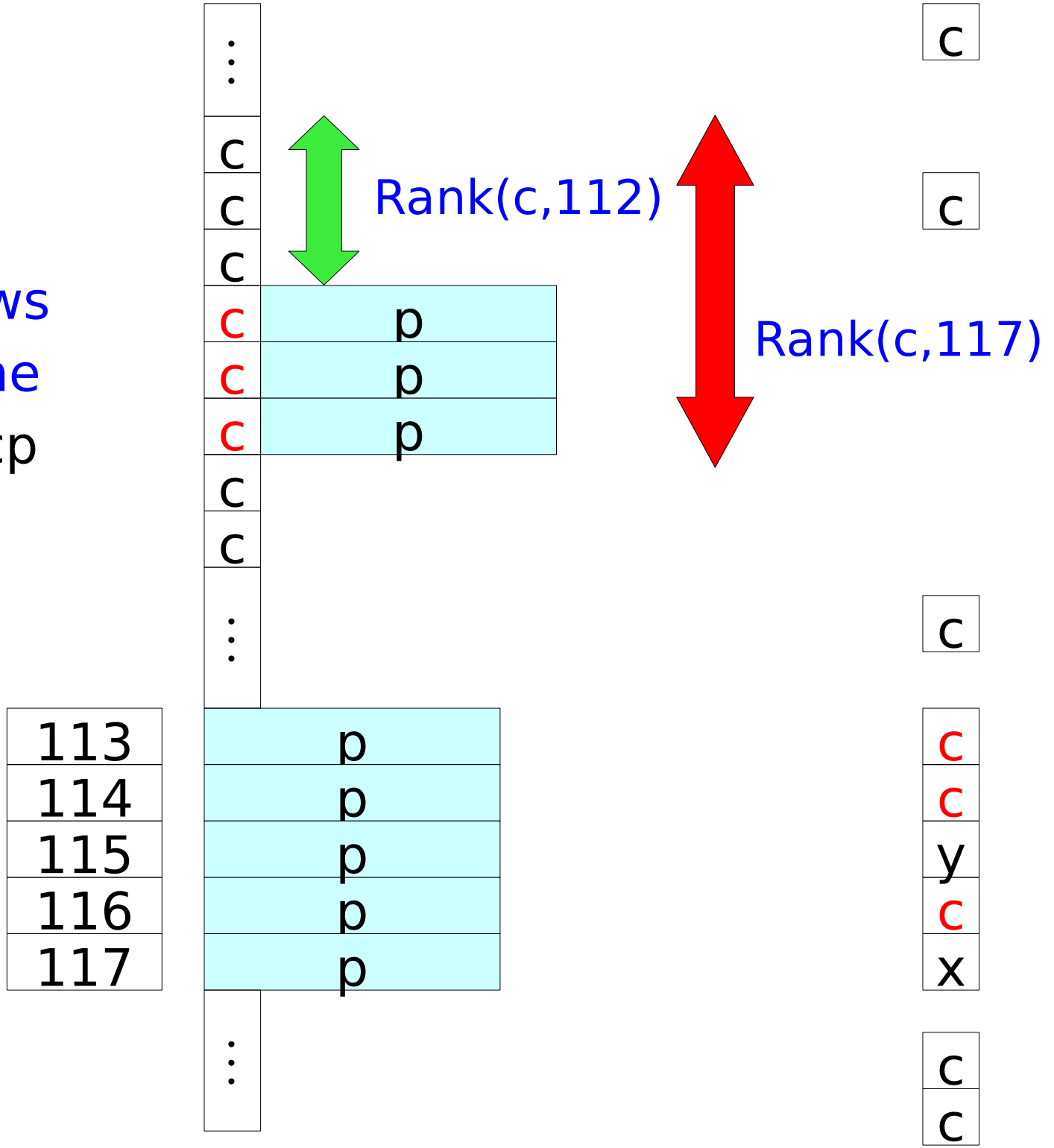
A closer look

The basic step is going from the rows prefixed by p to the rows prefixed by cp



A closer look

The basic step is going from the rows prefixed by p to the rows prefixed by cp



Summing up

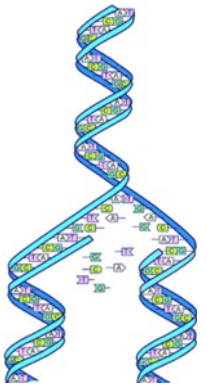
Each basic step requires two rank queries on L

We can do a rank query in O(log|A|) time on a compressed sequence using a Wavelet Tree

Finding the range of rows prefixed by a pattern P takes O(|P| log|A|) time (no dependency on |T|!).

We have a compressed representation of T supporting fast queries.

A happy coincidence



First draft of Human Genome Project completed

- People was looking for tools to search in the human genome
- Traditional indices did not fit in RAM
- Compressed indices did!
- Bowtie [Langmead et al. 09], BWA [Li et al, 09], SOAP2 [Li et al, 09] are extremely popular aligner based on a BWT index.

What does this mean?

- To efficiently search in a 1GB file it was necessary to build a full text index of size 5GB
- Now a BWT-compressed file of size ~300MB was able to store the original text and a full text index
- An unexpected 16x space reduction!