The BWT as a compressed index

Giovanni Manzini

Off-line pattern searching

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

Since the collection is know 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.

Enriching the suffix array with additional information 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 ... |T|

⇒ |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

This is 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

Note: the Suffix Array supports arbitrary substring searches. If you only want to search for words you can use less space.

It turns out that there is no gap: you can have simultaneously small storage and fast retrieval.

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.

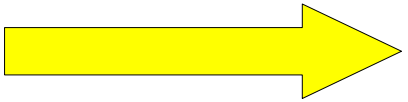
The last column of the resulting matrix is the BWT.

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

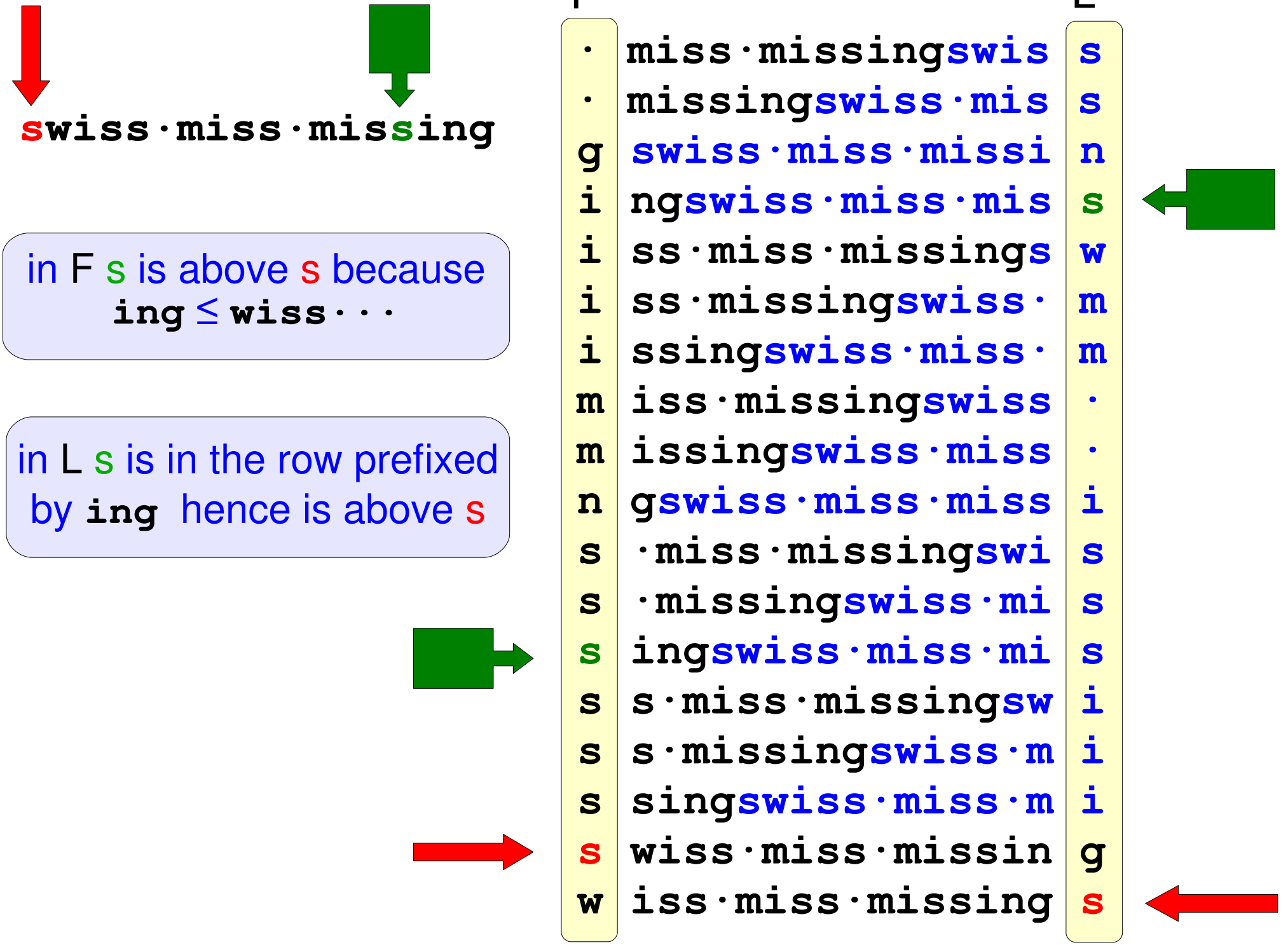
Burrows-Wheeler Transform

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


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

Summing up

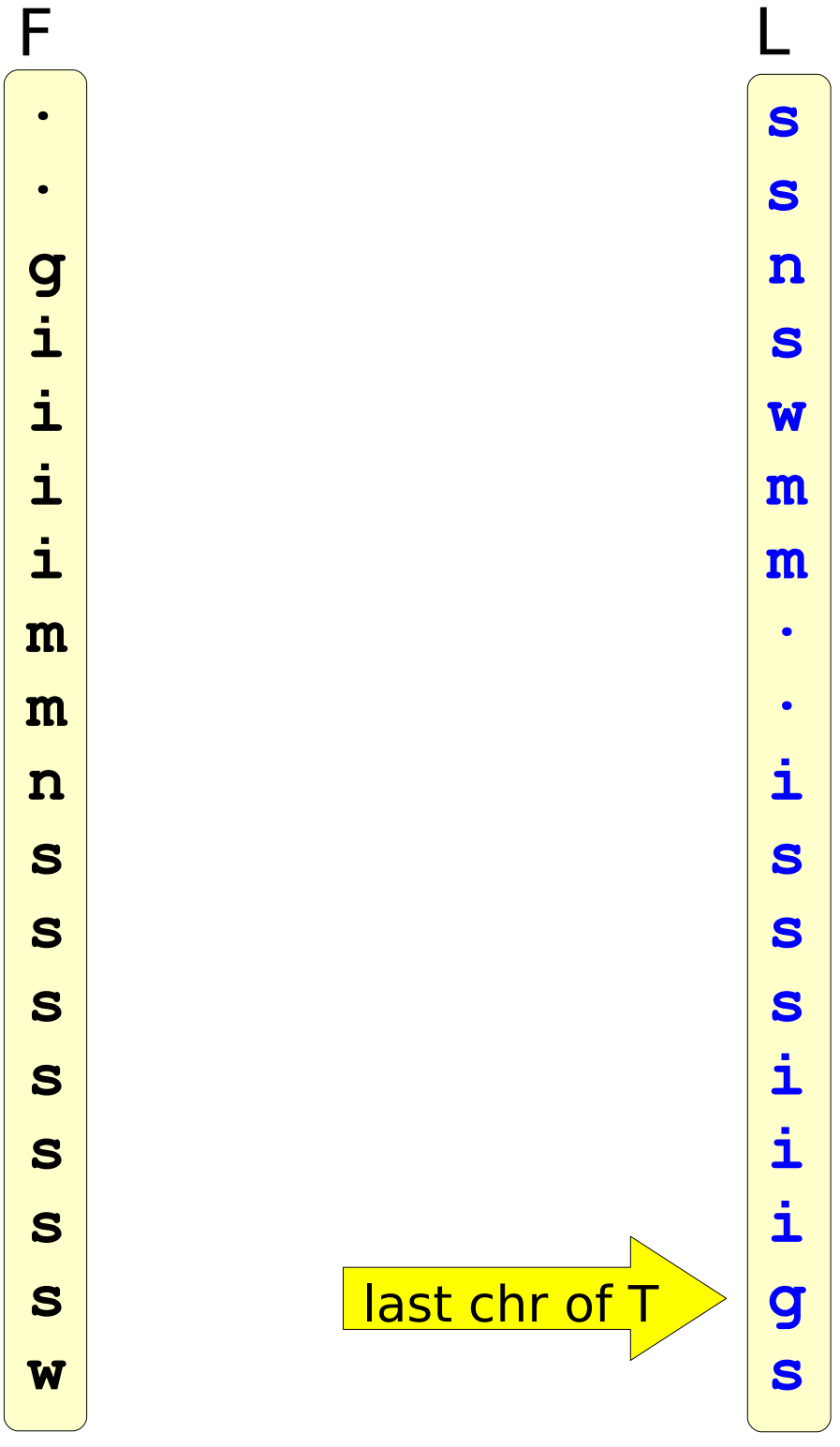
The relative order of the occurrences of any given character in F and L is the same

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

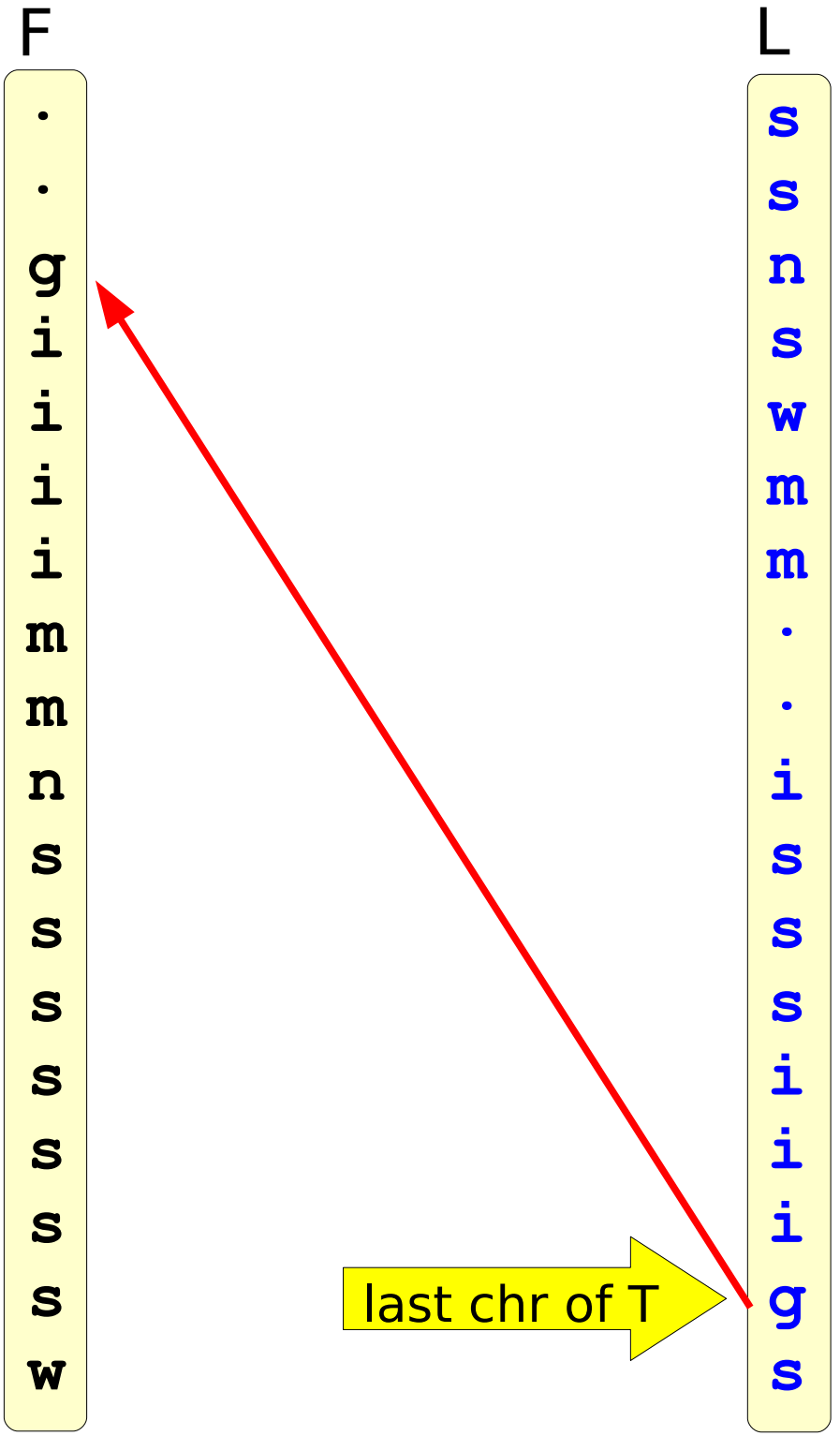
Using the LF map we retrieve T right-to-left

T = g



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

T = g

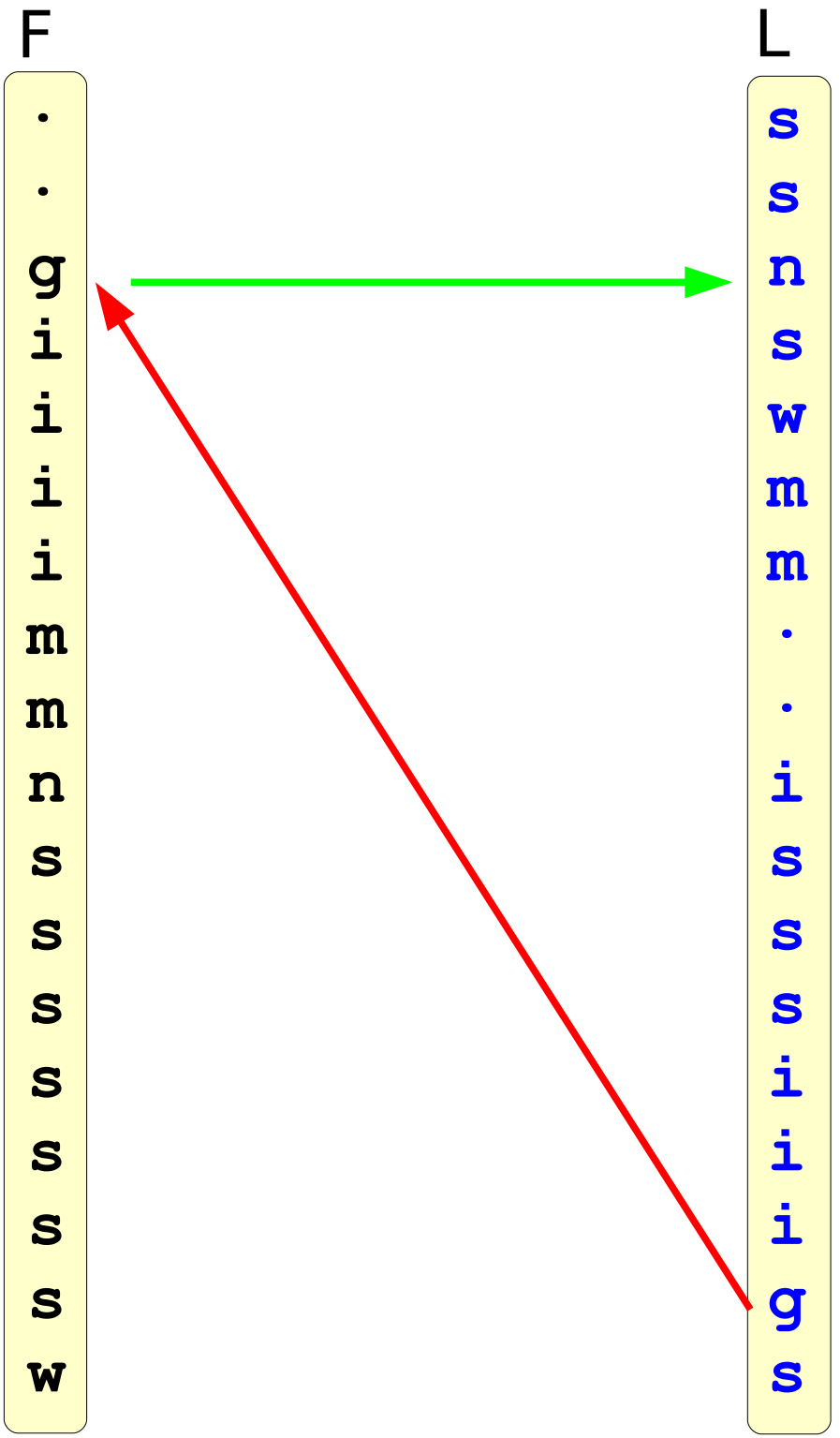


Using the LF map we retrieve T right-to-left

T = ng

...and so on

We now show that the LF-map can be used also for pattern searching

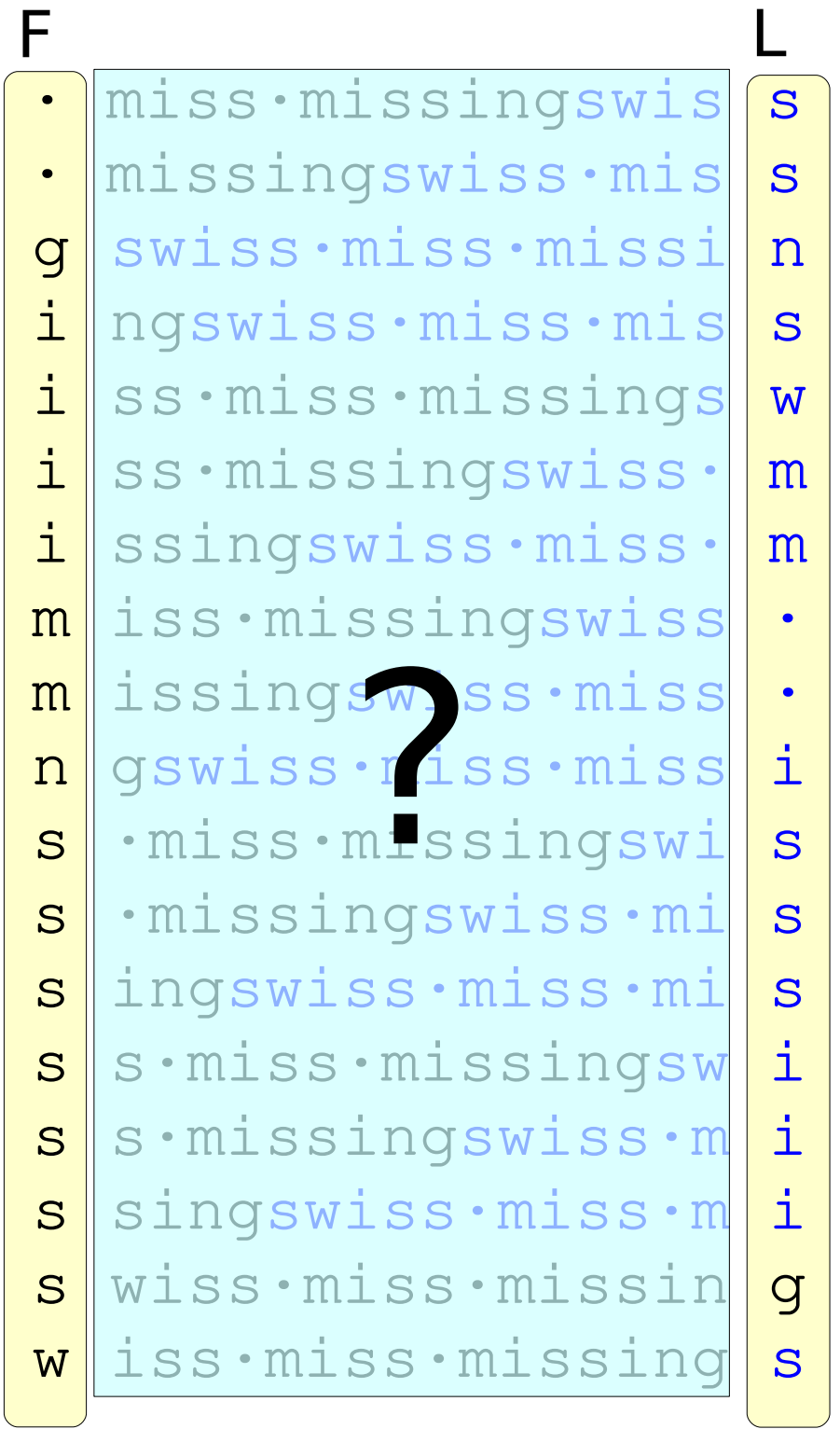


Search using the BWT

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



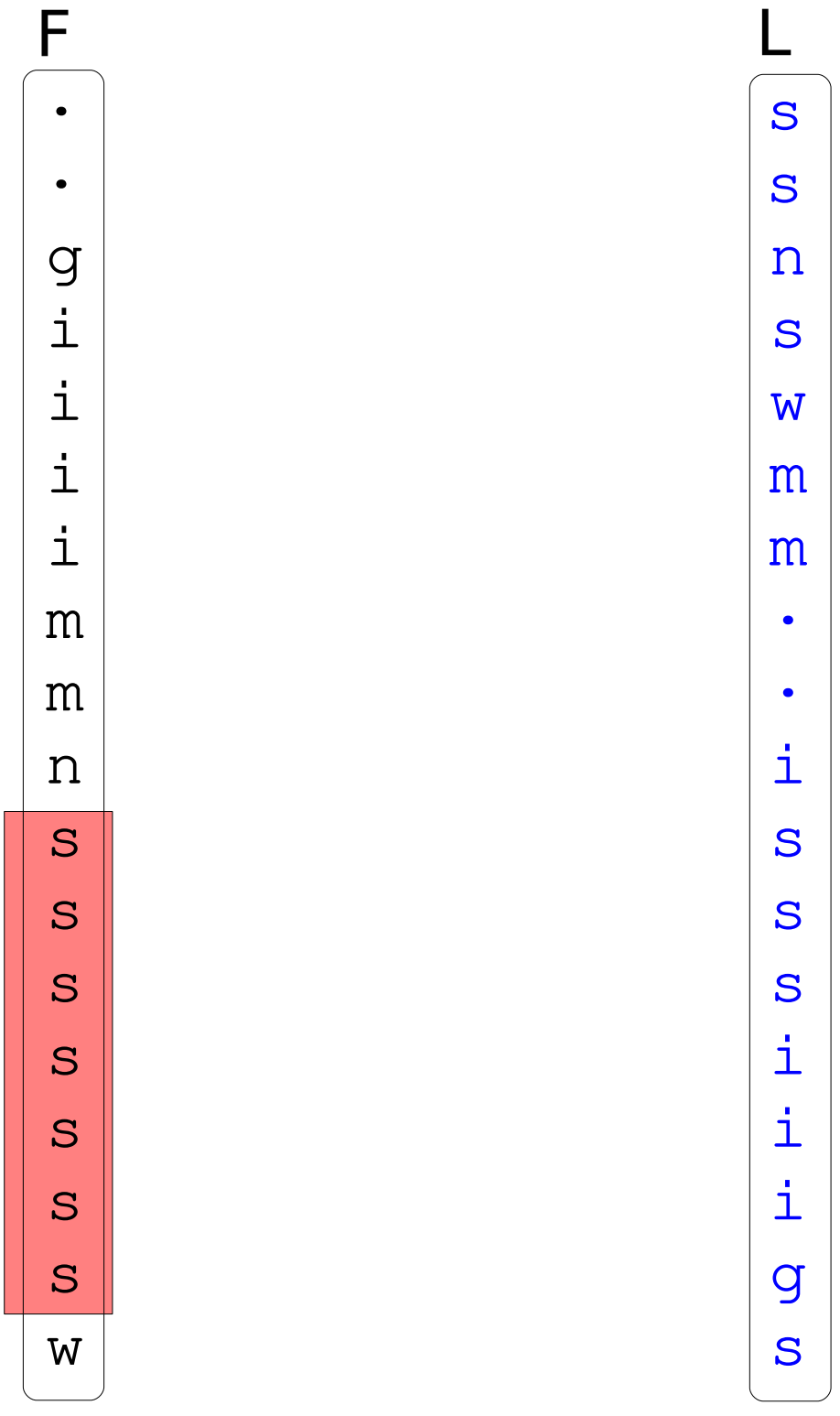
Backward search

The rows prefixed by s are easy to find using F

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

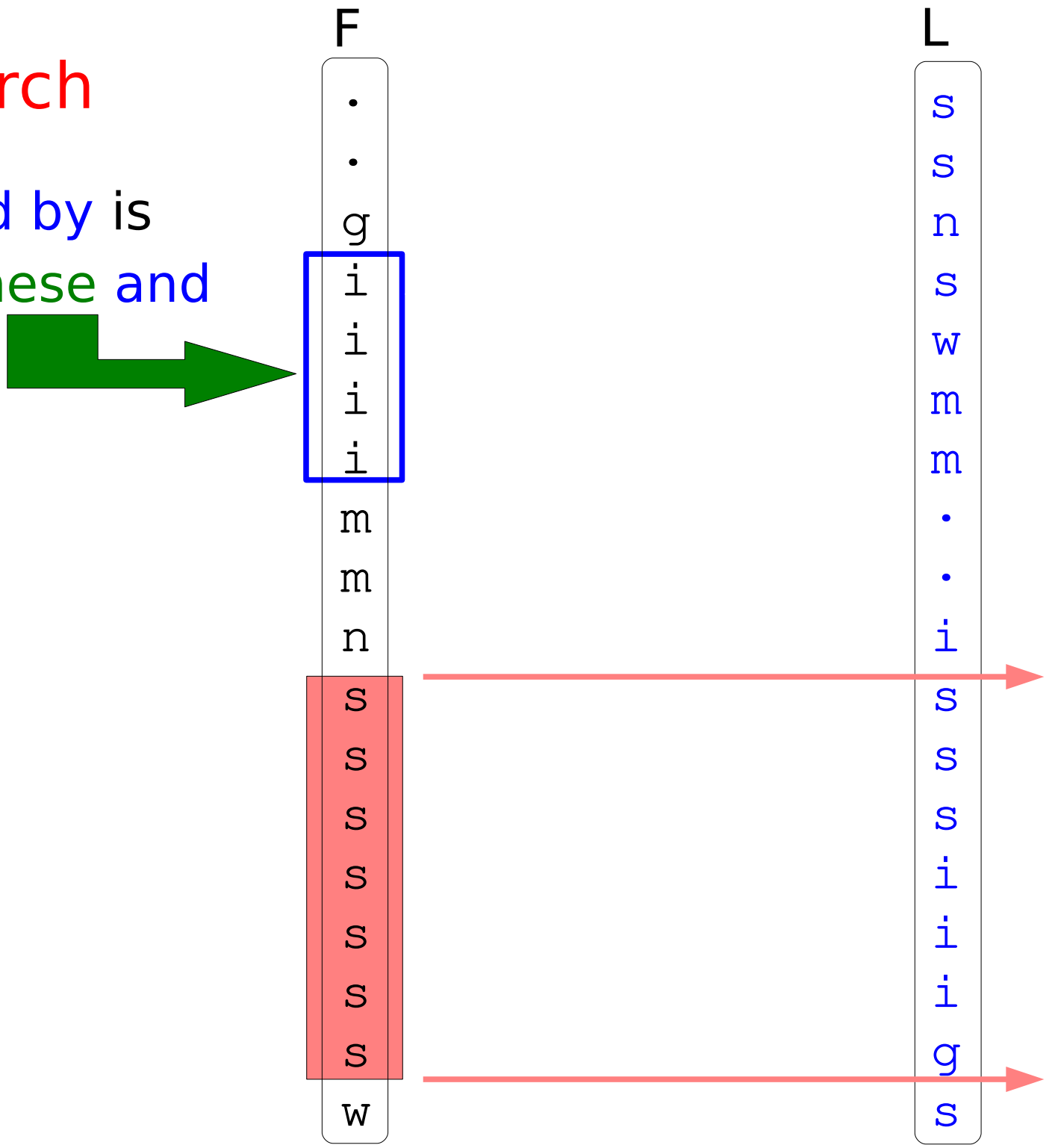
· →1 g →3 i →4
m →8 n →10
s →11 w →18

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



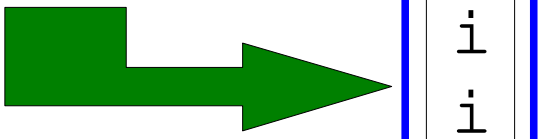
Backward search

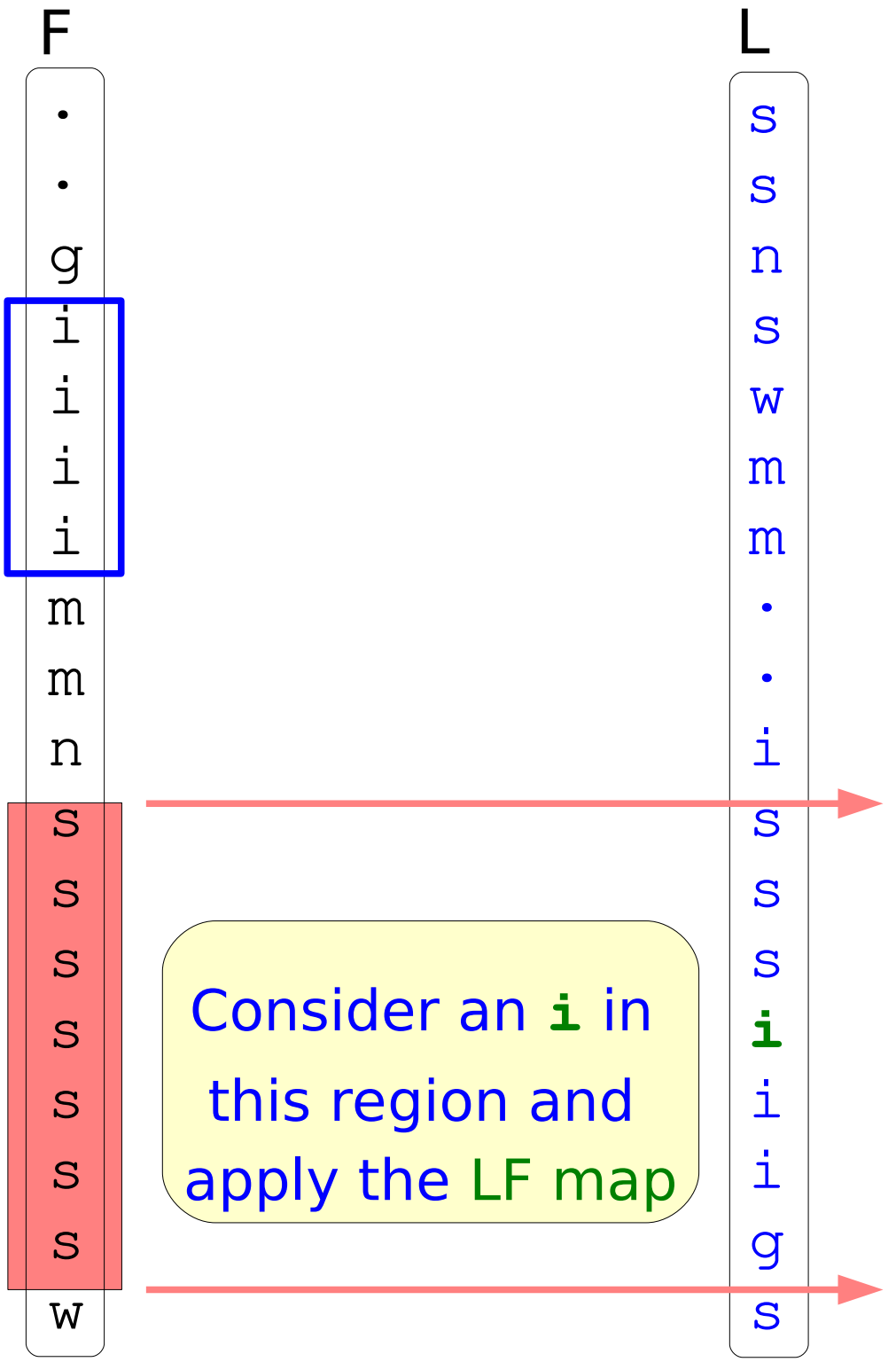
The rows prefixed by is are a subset of these and are consecutive.



Backward search

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

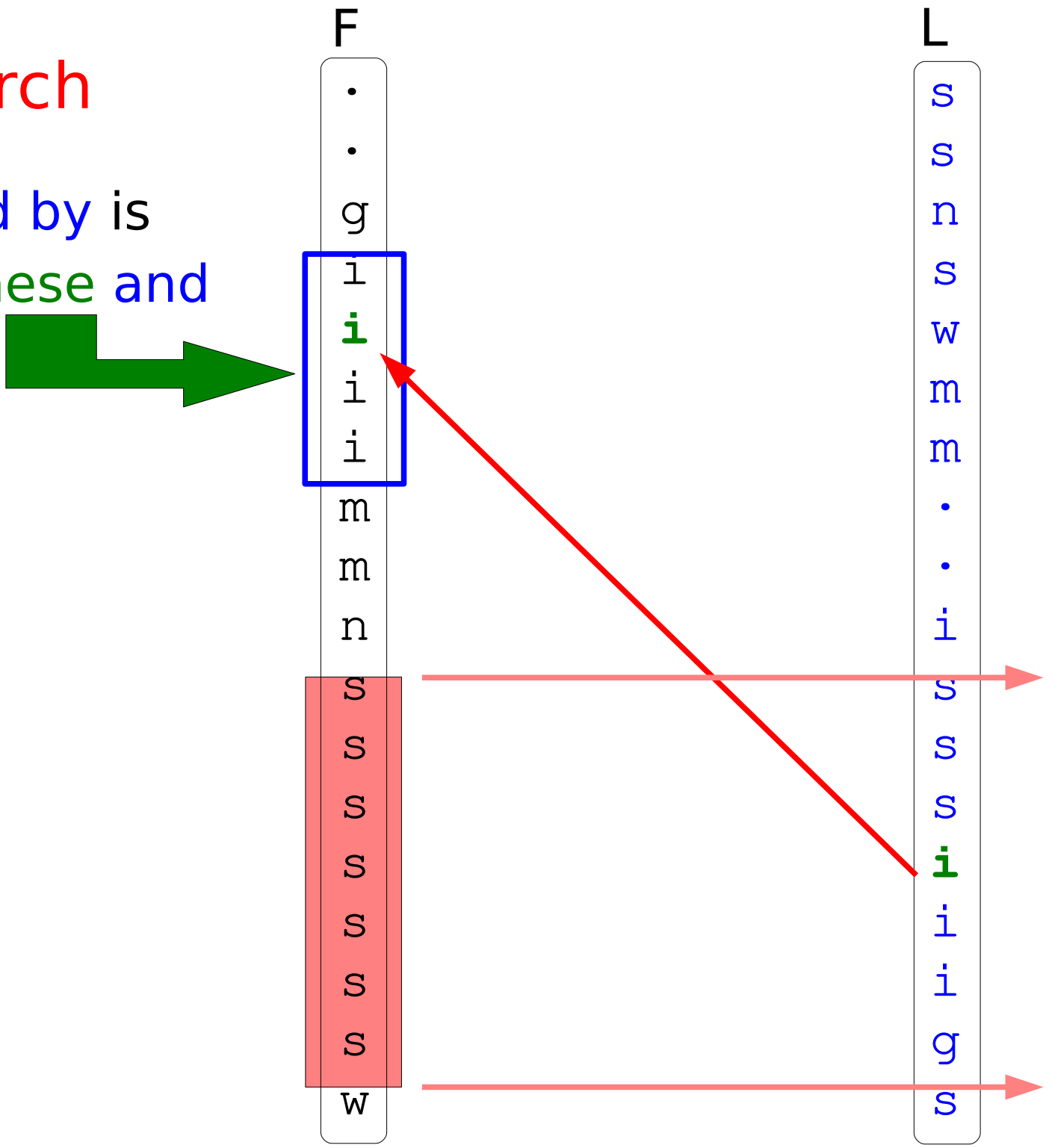




Consider an **i** in this region and apply the LF map

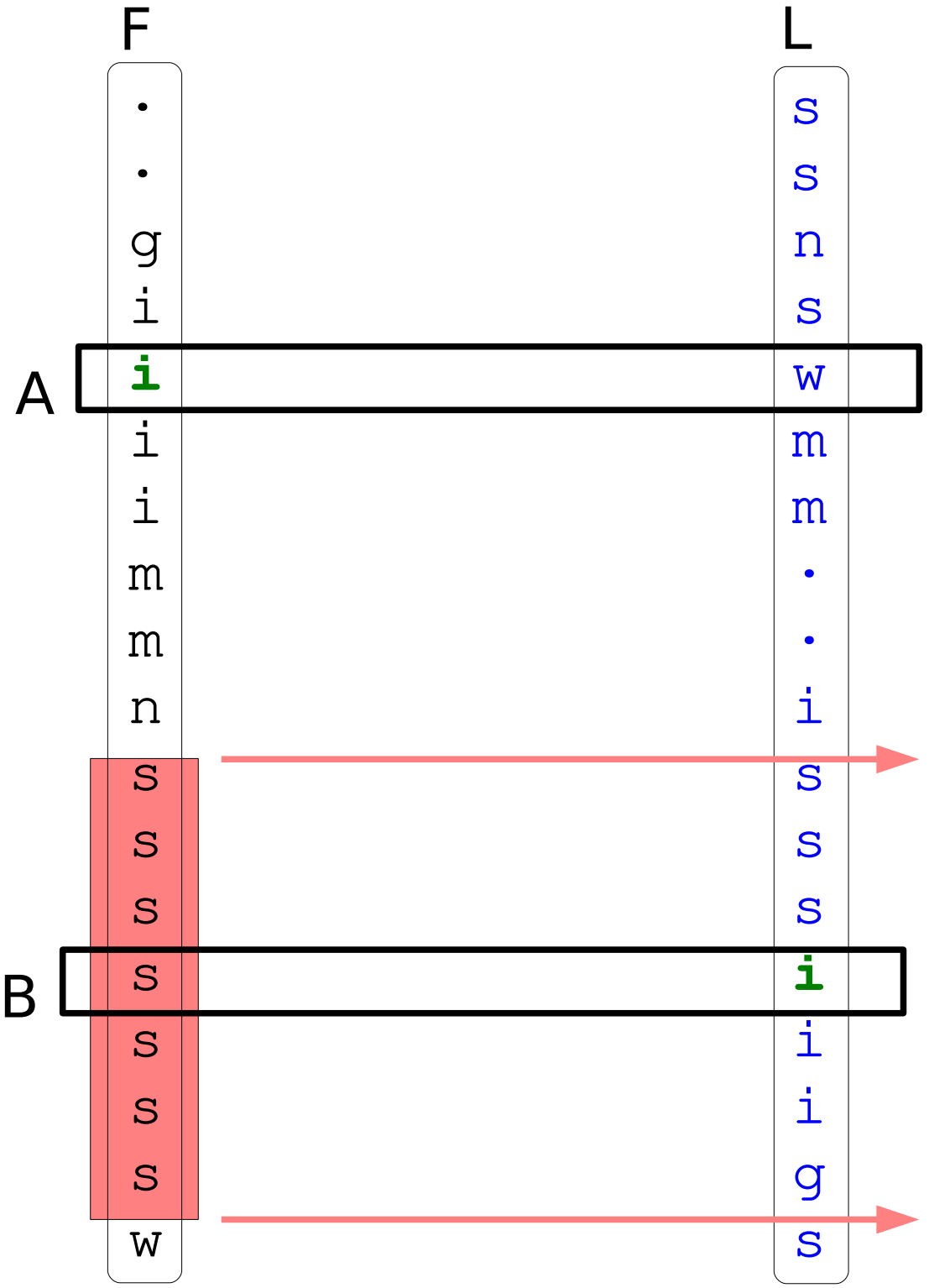
Backward search

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



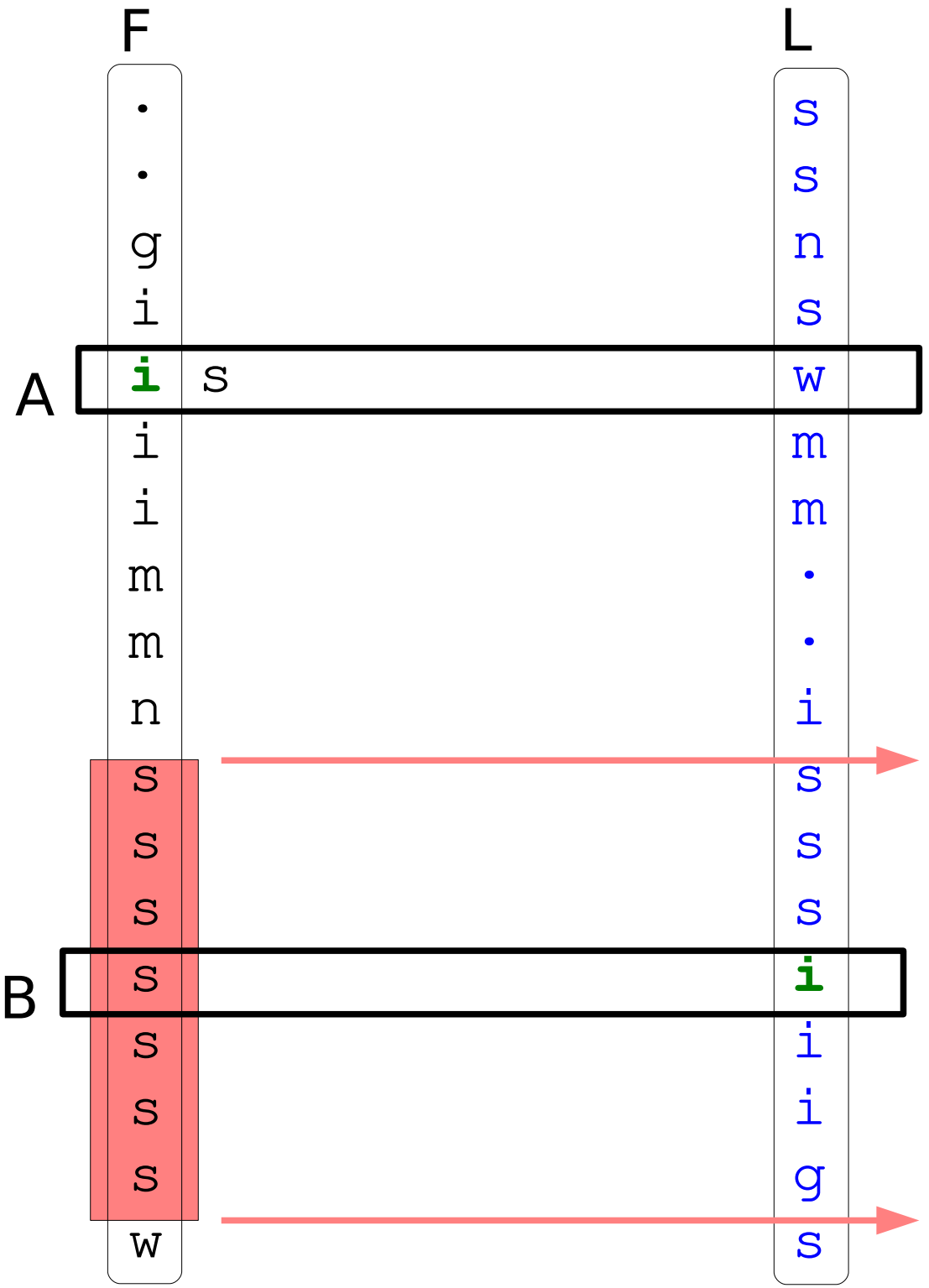
Backward search

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

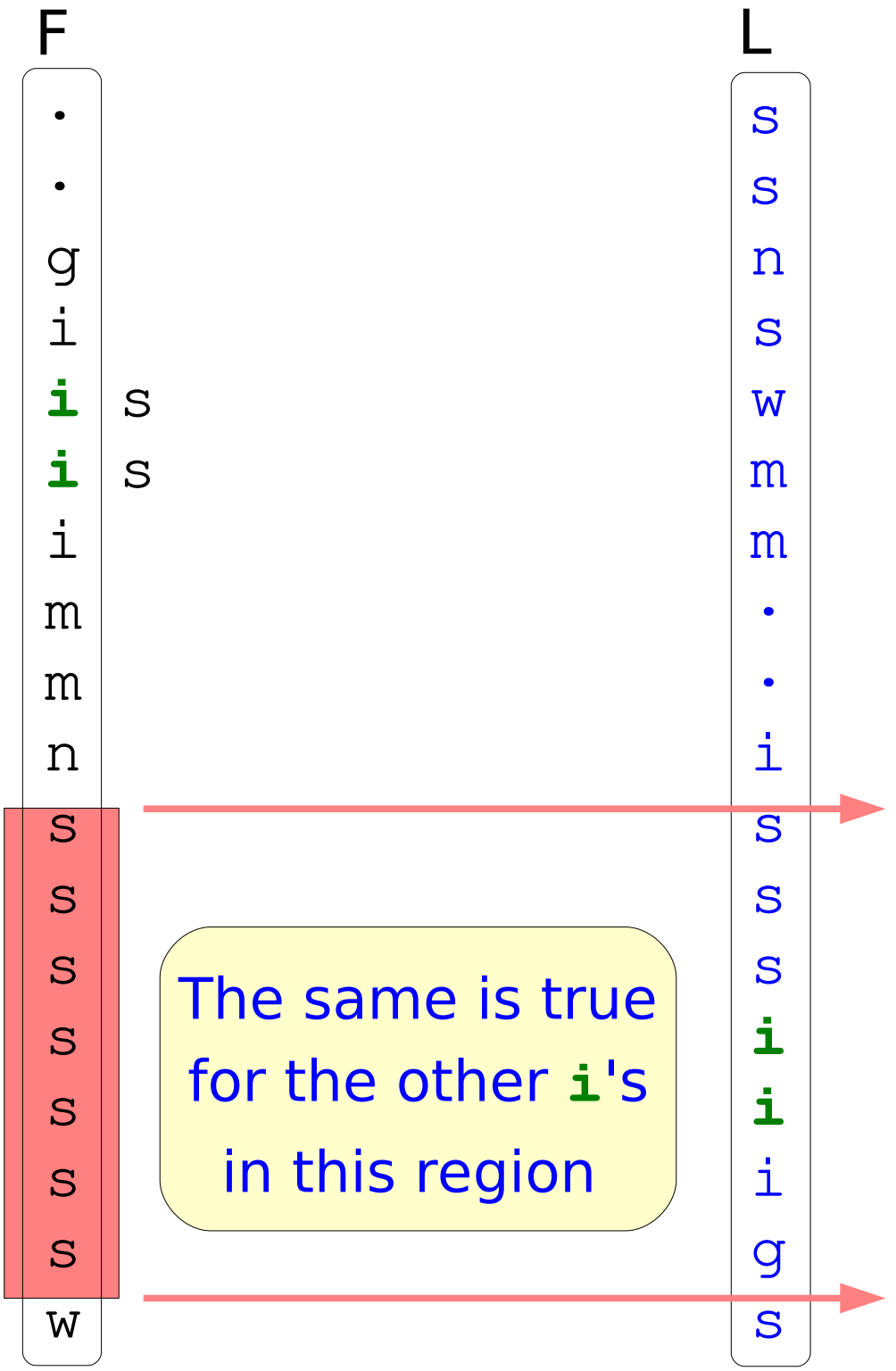


Backward search

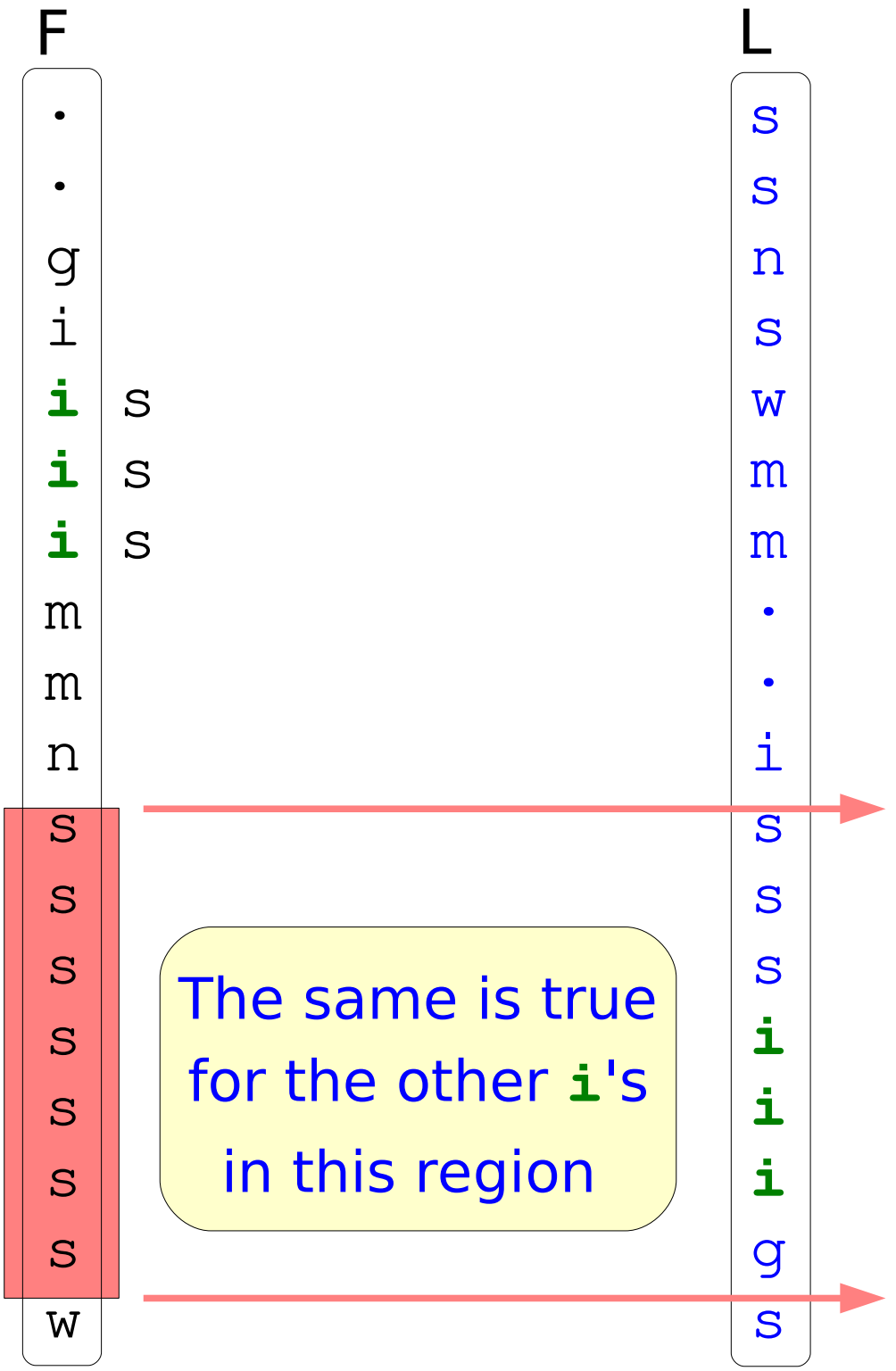
Since the two i's are the same, row A is row B left-shifted by 1.



Backward search

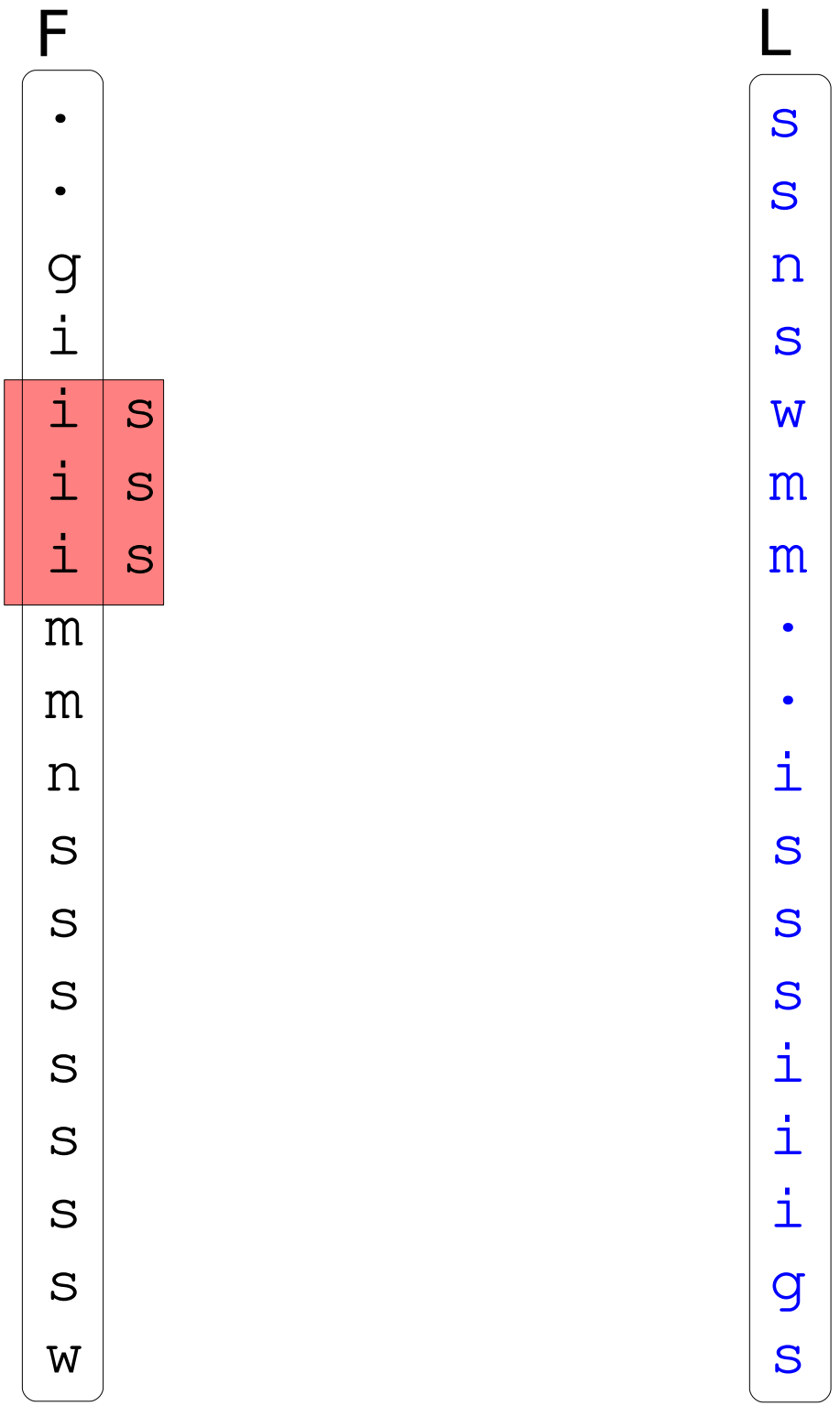


Backward search



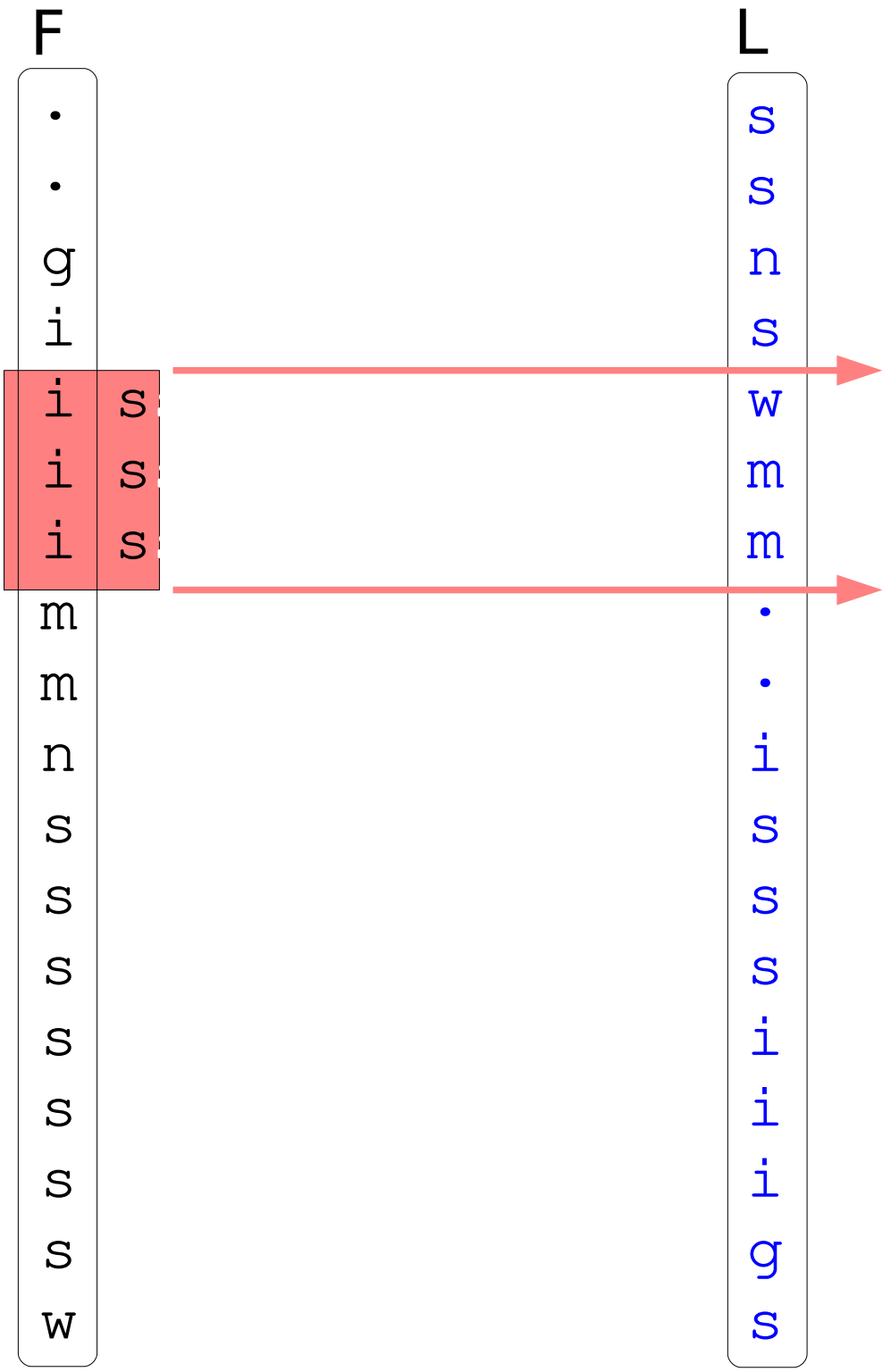
Backward search

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



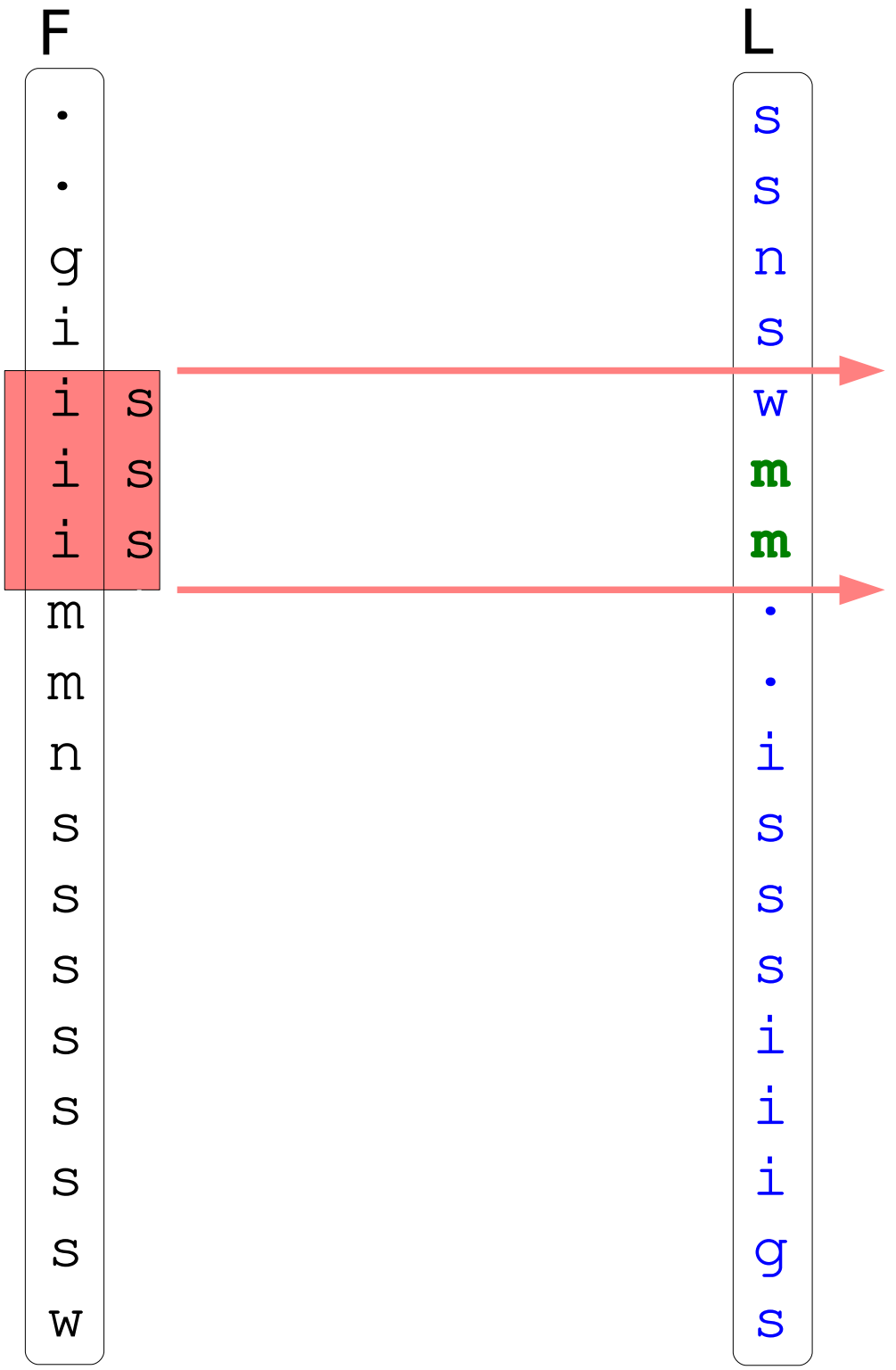
Backward search

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



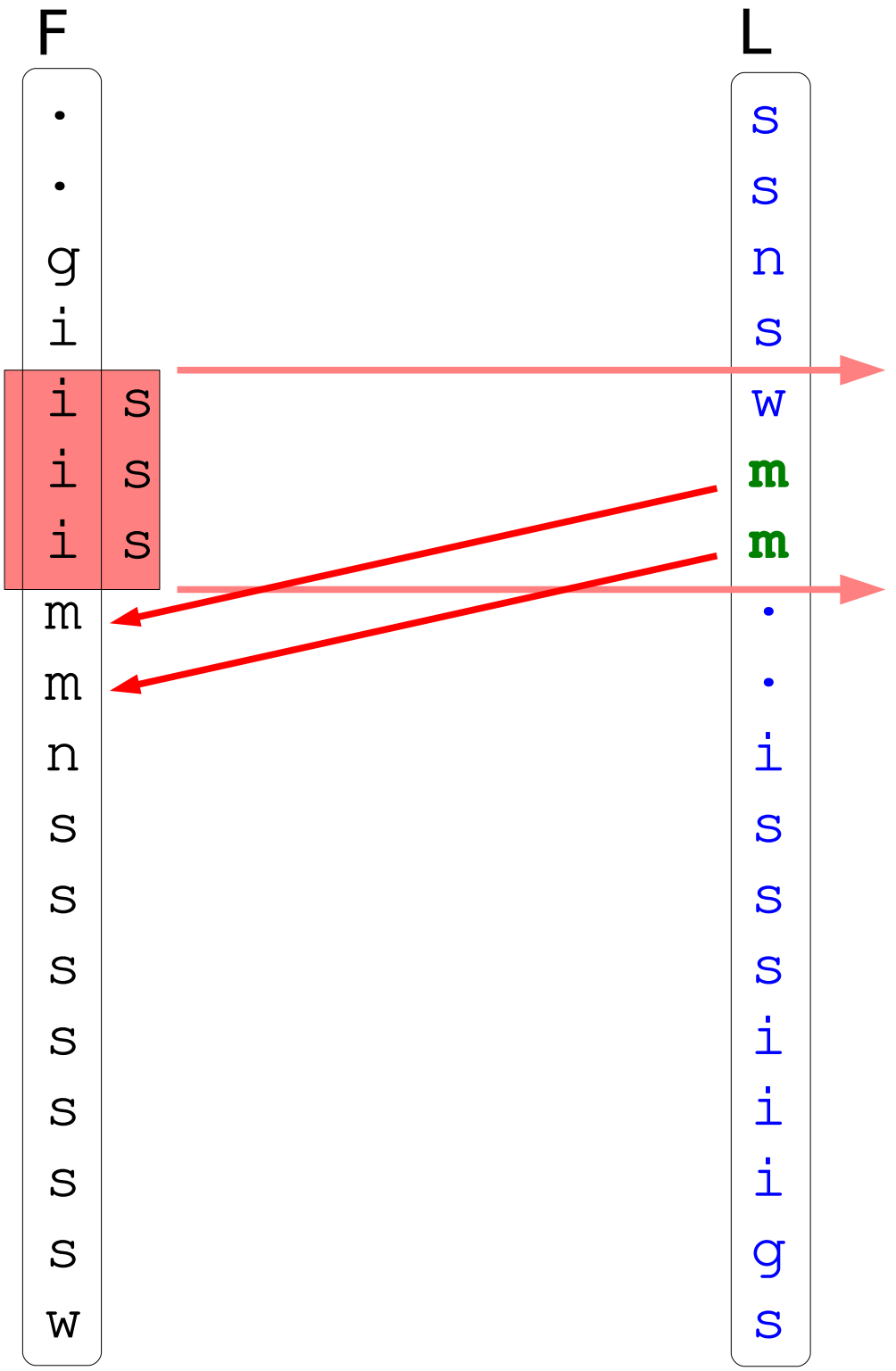
Backward search

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



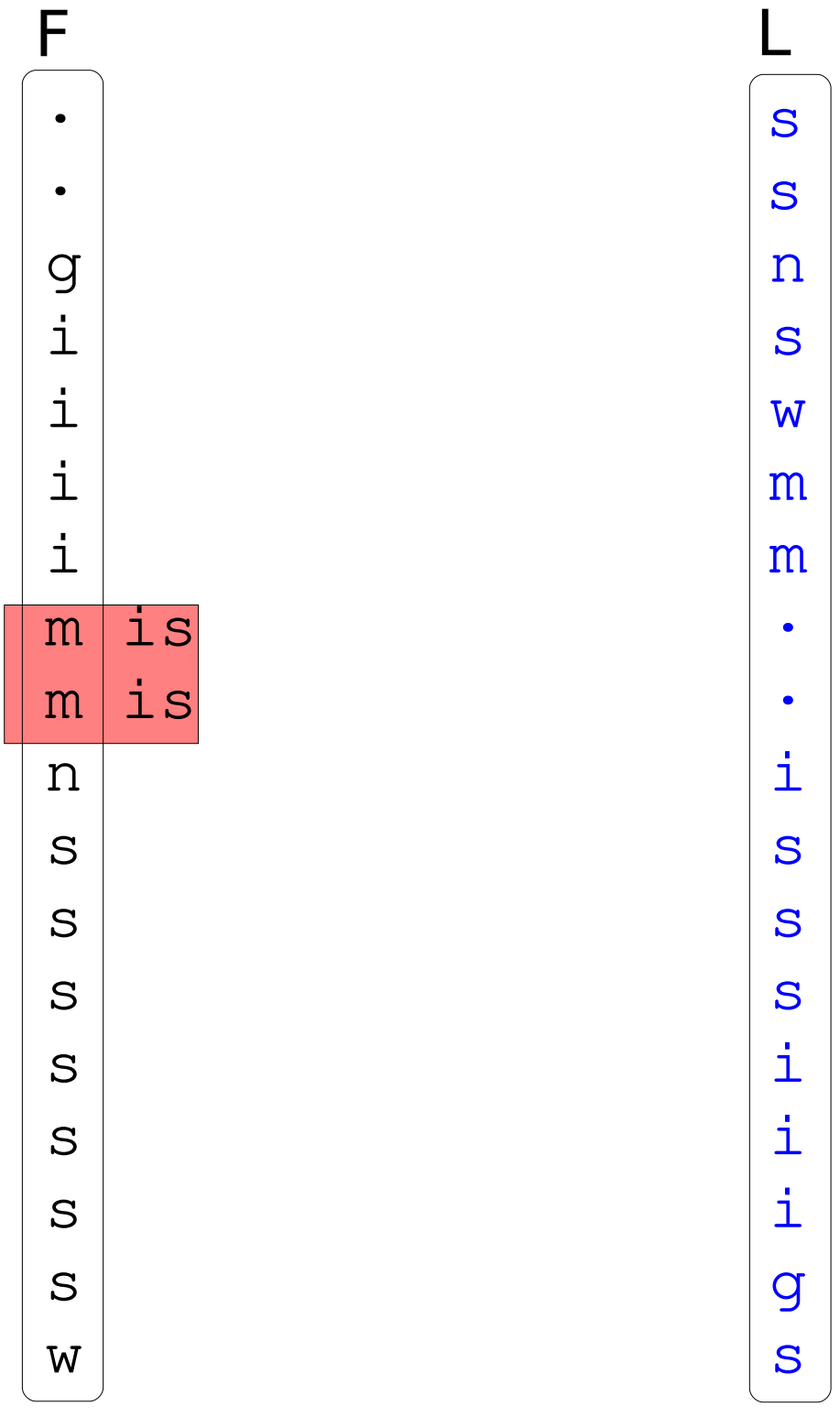
Backward search

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

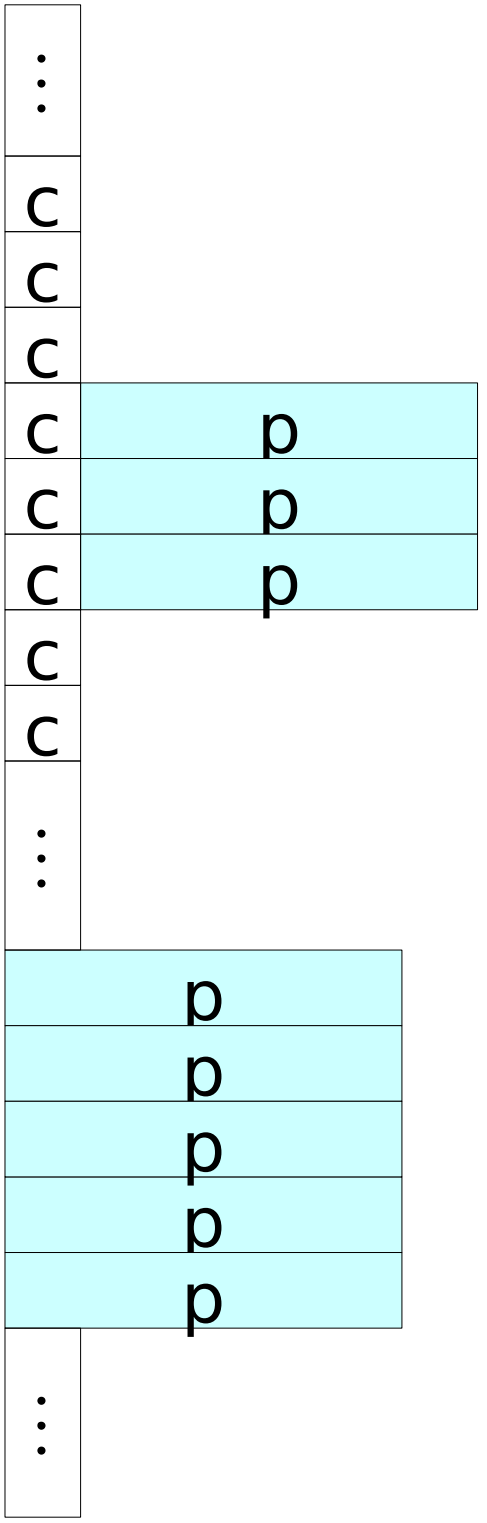
We have found the rows prefixed by **mis**!

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

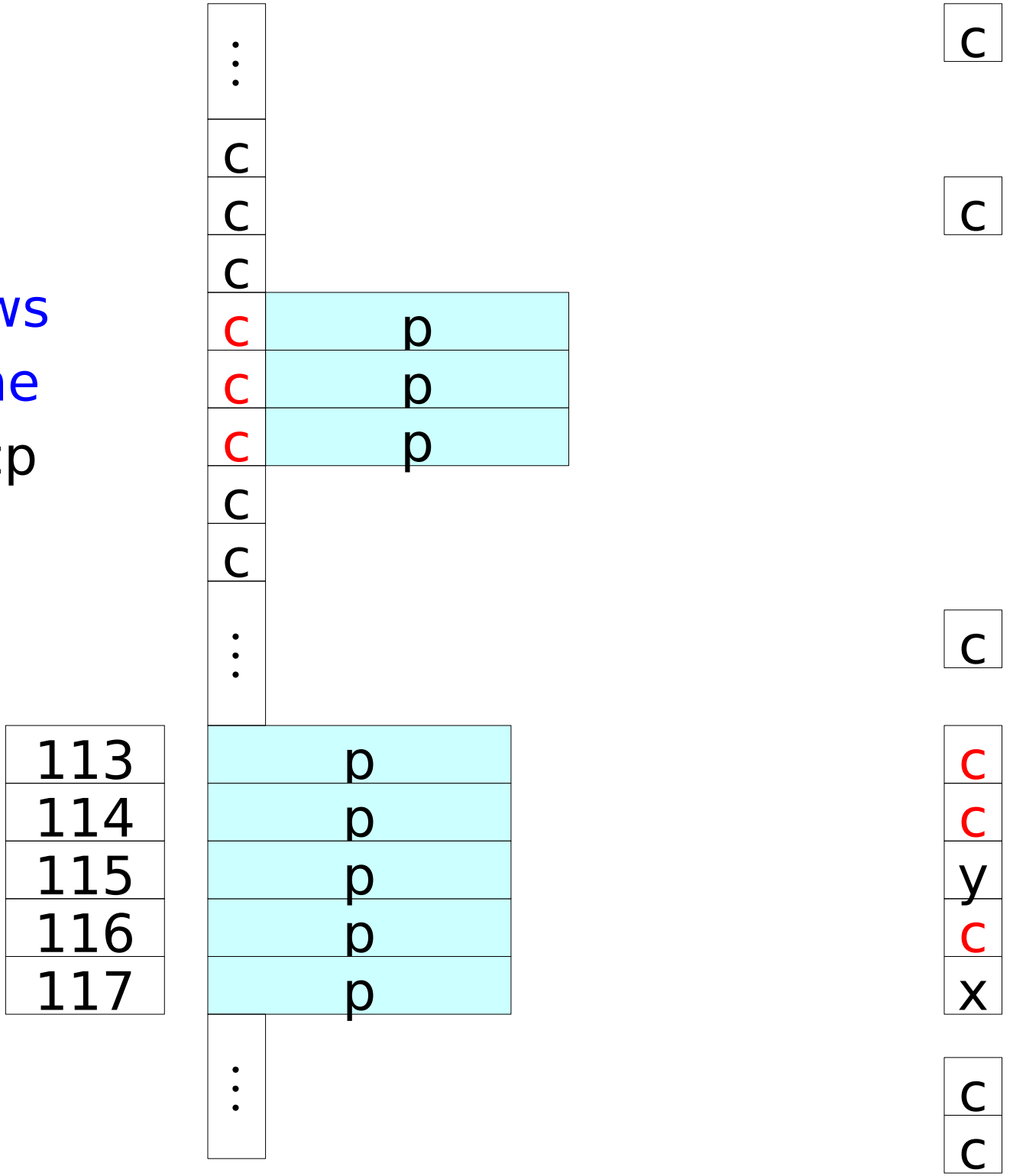
A closer look

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



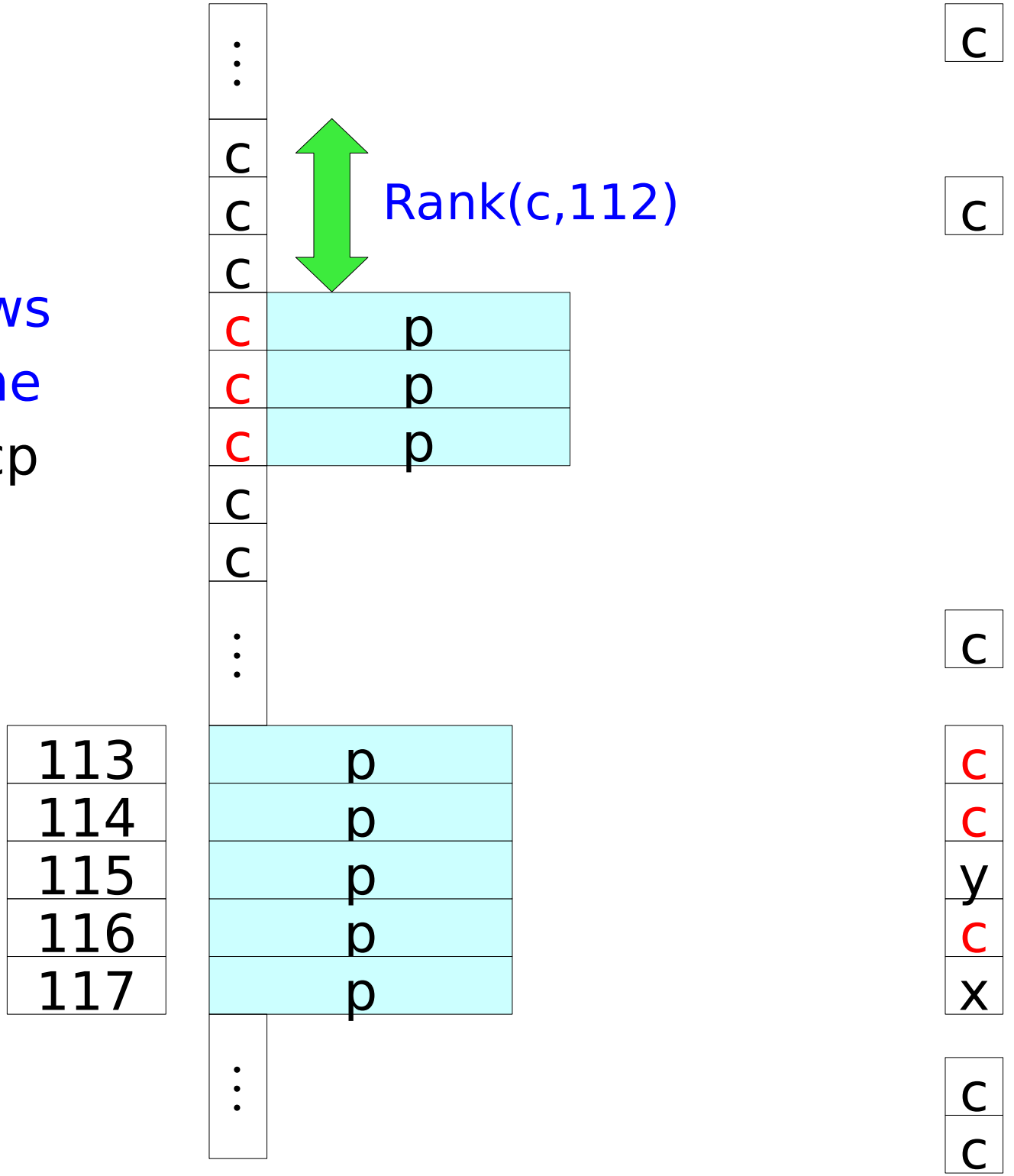
A closer look

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



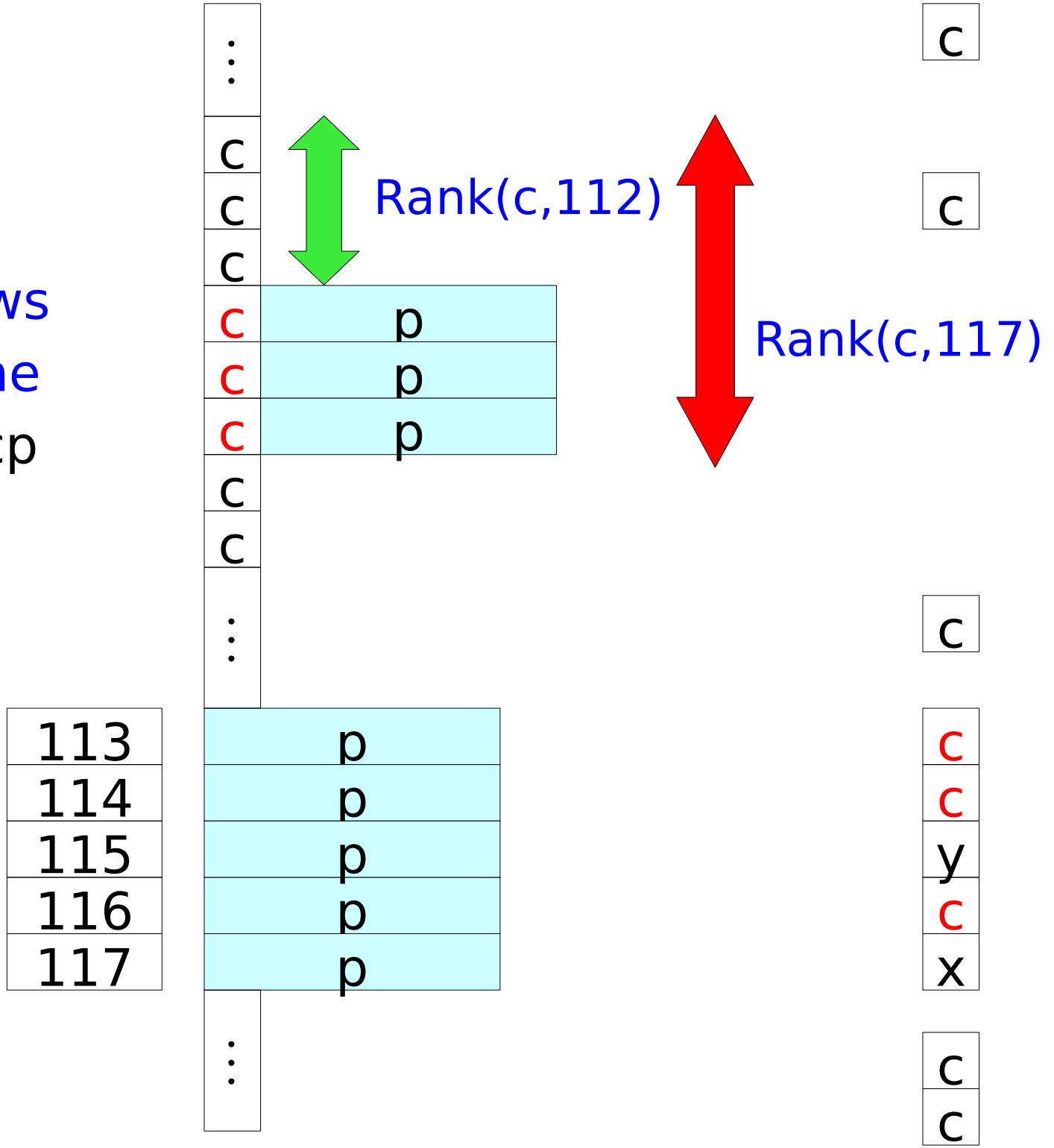
A closer look

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.

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.