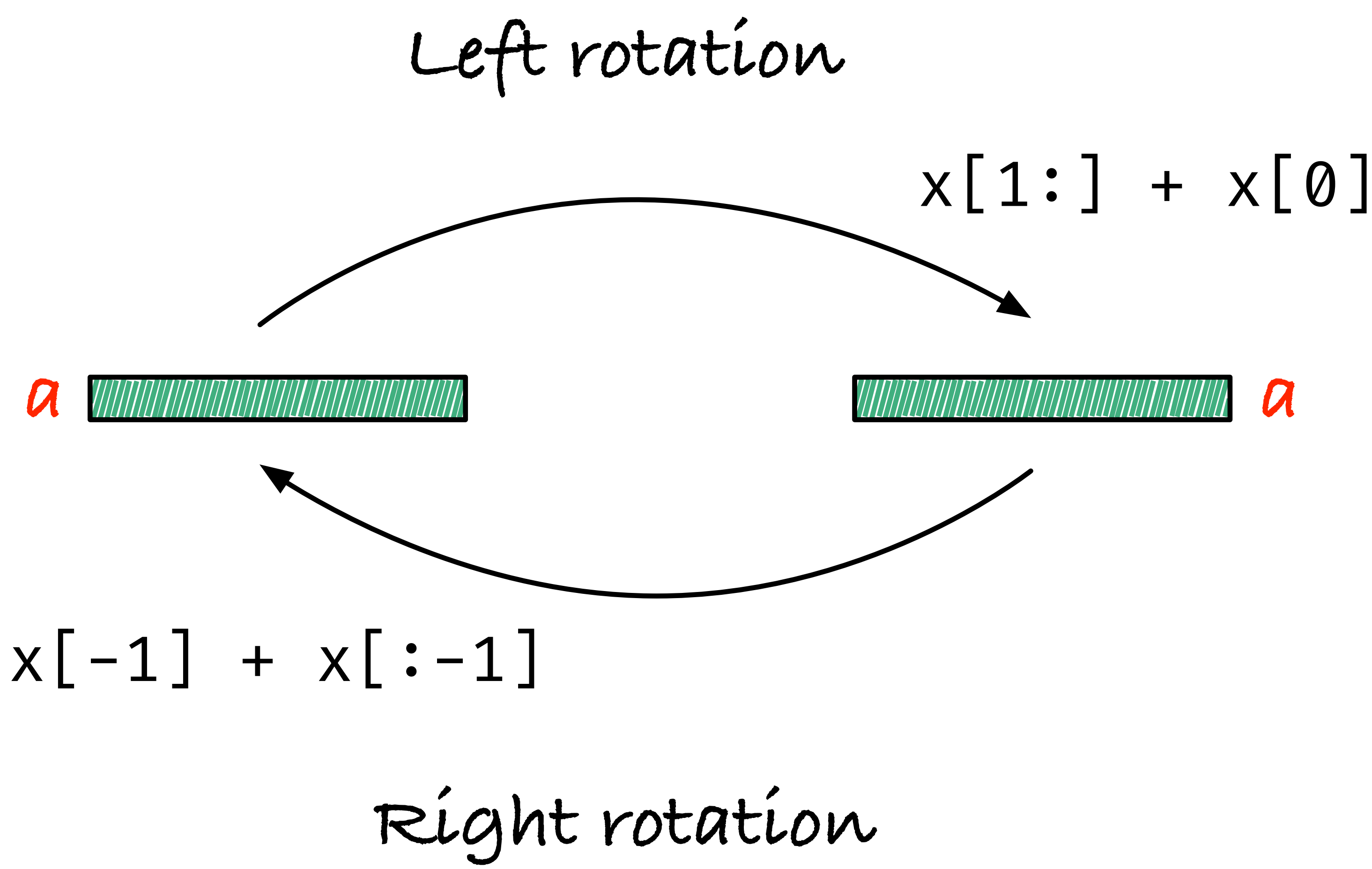
Burrows-Wheeler transform and FM-index search

(FM-index, after Paolo Ferragina and Giovanni Manzini)

Burrows-Wheeler transform

String rotations



String rotations

-4: ppi$mississi

-3: pi$mississip

-2: i$mississipp

-1: $mississippi

...sippi$mississippi$mississippi$mississippi$mississippi$mississippi$miss...

0: mississippi$

+1: ississippi$m

+2: ssissippi$mi

+3: sissippi$mis

+4: issippi$miss

(infinite or just x[i] == x[i%n])

Burrows-Wheeler transform

x = mississippi$

Take all rotations

```
 0: mississippi$
 1: ississippi$m
 2: ssissippi$mi
 3: sissippi$mis
 4: issippi$miss
 5: ssippi$missi
 6: sippi$missis
 7: ippi$mississ
 8: ppi$mississi
 9: pi$mississip
10: i$mississipp
11: $mississippi
```

Sort the rotations

```
11: $mississippi
10: i$mississipp
 7: ippi$mississ
 4: issippi$miss
 1: ississippi$m
 0: mississippi$
 9: pi$mississip
 8: ppi$mississi
 6: sippi$missis
 3: sissippi$mis
 5: ssippi$missi
 2: ssissippi$mi
```

Take the last column

bwt(x) = ipssm$pissii

Burrows-Wheeler transform

x = mississippi$

Take all rotations

```
 0: mississippi$
 1: ississippi$m
 2: ssissippi$mi
 3: sissippi$mis
 4: issippi$miss
 5: ssippi$missi
 6: sippi$missis
 7: ippi$mississ
 8: ppi$mississi
 9: pi$mississip
10: i$mississipp
11: $mississippi
```

Sort the rotations

Burrows-Wheeler matrix

```
11  $mississippi
10  i$mississipp
 7  ippi$mississ
 4  issippi$miss
 1  ississippi$m
 0  mississippi$
 9  pi$mississip
 8  ppi$mississi
 6  sippi$missis
 3  sissippi$mis
 5  ssippi$missi
 2  ssissippi$mi
```

Take the l

bwt(x) is the last column in the Burrows-Wheeler matrix for x

bwt(x) = ipssm$pissii

Burrows-Wheeler transform

- Burrows-Wheeler matrix is $O(n^2)$ space
  - Takes $O(n^2)$ time to fill in all rotations
  - Takes $O(n^2)$ time to radix sort it

Burrows-Wheeler transform

- Burrows-Wheeler matrix is $O(n^2)$ space
  - Takes $O(n^2)$ time to fill in all rotations
  - Takes $O(n^2)$ time to radix sort it
  - (We don't want any of this…)

BWT and suffix arrays

Sorted rotations

```
11: $mississippi
10: i$mississipp
 7: ippi$mississ
 4: issippi$miss
 1: ississippi$m
 0: mississippi$
 9: pi$mississip
 8: ppi$mississi
 6: sippi$missis
 3: sissippi$mis
 5: ssippi$missi
 2: ssissippi$mi
```

sa(x)

```
11: $mississippi
10: i$mississipp
 7: ippi$mississ
 4: issippi$miss
 1: ississippi$m
 0: mississippi$
 9: pi$mississip
 8: ppi$mississi
 6: sippi$missis
 3: sissippi$mis
 5: ssippi$missi
 2: ssissippi$mi
```

BWT and suffix arrays

$x[i-1]$ i $rot_i(x)[-1]$ $x[j-1]$ j $rot_j(x)[-1]$

...sippi$mississippi$mississippi$mississippi$mississippi$miss...

BWT and suffix arrays

sa(x) bwt(x)

```
11: $mississippi
10: i$mississipp
 7: ippi$mississ
 4: issippi$miss
 1: ississippi$m
 0: mississippi$
 9: pi$mississip
 8: ppi$mississi
 6: sippi$missis
 3: sissippi$mis
 5: ssippi$missi
 2: ssissippi$mi
```

sa(x)-1 bwt(x)

```
10: i$mississippi
 9: pi$mississipp
 6: sippi$mississ
 3: sissippi$miss
 0: mississippi$m
-1: $mississippi$     Special case(?)
 8: ppi$mississip
 7: ippi$mississi
 5: ssippi$missis
 2: ssissippi$mis
 4: issippi$missi
 1: ississippi$mi
```

$bwt(x)[i] = \$$ if $sa[i] == 0$

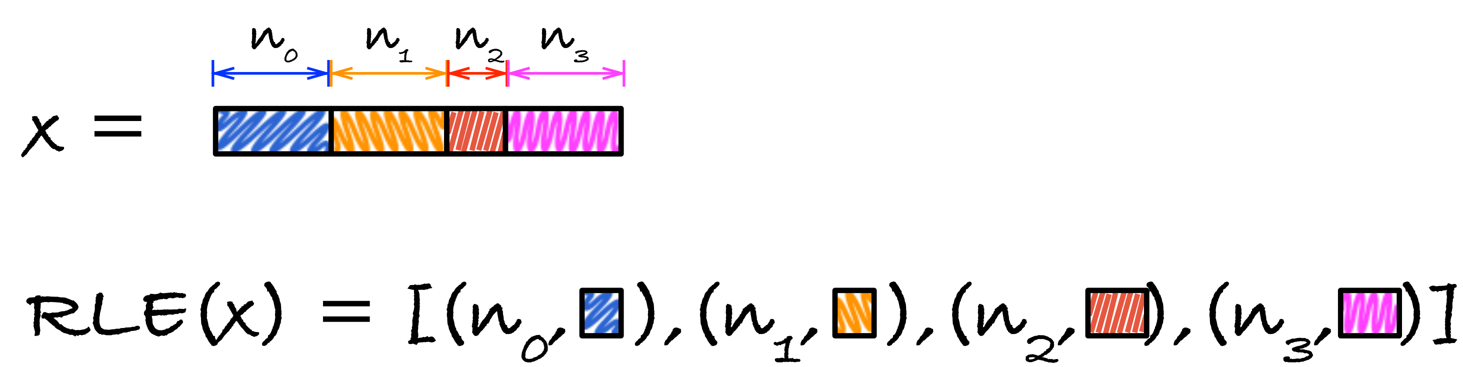
$bwt(x)[i] = x[sa[i]-1]$ otherwise

Why BWT?

- BWT is used in compression
    - e.g. in bzip2 (combined with many other things). It's the "b" in the name
- Compression likes low entropy (low randomness)
    - You get low randomness if some characters are much more frequent than others in a (sub-)string
    - We don't have time to talk compression and information theory, but I will give you a bit of an intuition (and nothing more)…

Run length encoding

- Given a string x, split it into blocks of consecutive identical characters, then encode each block as its length and one character.
  - E.g. mississippi = (1,m),(1,i),(2,s),(1,i),(2,s),(1,i),(2,p),(1,i)



- This is exactly what we do with CIGAR strings

Run length encoding

- Run length encoding isn’t always a good idea. Strings can get longer when you “compress them”.
- There is usually more magic in compression, but RLE is enough to illustrate why BWT is interesting
  - (and take my word that BWT is useful beyond RLE)

BWT tends to group characters

```
$the_day_the_damned_dog_died
_damned_dog_died$the_day_the
_day_the_damned_dog_died$the
_died$the_day_the_damned_dog
_dog_died$the_day_the_damned
_the_damned_dog_died$the_day
amned_dog_died$the_day_the_d
ay_the_damned_dog_died$the_d
d$the_day_the_damned_dog_die
d_dog_died$the_day_the_damne
damned_dog_died$the_day_the_
day_the_damned_dog_died$the_
died$the_day_the_damned_dog_
dog_died$the_day_the_damned_
e_damned_dog_died$the_day_th
e_day_the_damned_dog_died$th
ed$the_day_the_damned_dog_di
ed_dog_died$the_day_the_damn
g_died$the_day_the_damned_do
he_damned_dog_died$the_day_t
he_day_the_damned_dog_died$t
ied$the_day_the_damned_dog_d
mned_dog_died$the_day_the_da
ned_dog_died$the_day_the_dam
og_died$the_day_the_damned_d
the_damned_dog_died$the_day_
the_day_the_damned_dog_died$
y_the_damned_dog_died$the_da
```

x = "the_day_the_damned_dog_died$"

bwt(x) = "deegdyddee____hhinottdamd_$a"

RLE(x) = 1t1h1e1_1d1a1y1_1t1h1e1_1d1a1m1n1e1d1_1d1o1g1_1d1i1e1d1$

RLE(bwt(x)) = 1d2e1g1d1y2d2e4_2h1i1n1o2t1d1a1m1d1_1$1a

because BWT sorts by right-context

SA: x[i] sorted wrt x[i:] a

BWT: x[i] sorted wrt x[i+1:]+x[i] a

x = "the_day_the_damned_dog_died$"

day_the_damned_dog_died$
damned_dog_died$
dog_died$
died$

```
...
d_dog_died$the_day_the_damne
damned_dog_died$the_day_the_
day_the_damned_dog_died$the_
died$the_day_the_damned_dog_
dog_died$the_day_the_damned_
e_damned_dog_died$the_day_th
...
```

```
...
_the_damned_dog_died$the_day
amned_dog_died$the_day_the_d
ay_the_damned_dog_died$the_d
d$the_day_the_damned_dog_die
d_dog_died$the_day_the_damne
...
he_day_the_damned_dog_died$t
ied$the_day_the_damned_dog_d
mned_dog_died$the_day_the_da
ned_dog_died$the_day_the_dam
og_died$the_day_the_damned_d
the_damned_dog_died$the_day
```

Why not use the first column?

```
$the_day_the_damned_dog_died
_damned_dog_died$the_day_the
_day_the_damned_dog_died$the
_died$the_day_the_damned_dog
_dog_died$the_day_the_damned
_the_damned_dog_died$the_day
amned_dog_died$the_day_the_d
ay_the_damned_dog_died$the_d
d$the_day_the_damned_dog_die
d_dog_died$the_day_the_damne
damned_dog_died$the_day_the_
day_the_damned_dog_died$the_
died$the_day_the_damned_dog_
dog_died$the_day_the_damned_
e_damned_dog_died$the_day_th
e_day_the_damned_dog_died$th
ed$the_day_the_damned_dog_di
ed_dog_died$the_day_the_damn
g_died$the_day_the_damned_do
he_damned_dog_died$the_day_t
he_day_the_damned_dog_died$t
ied$the_day_the_damned_dog_d
mned_dog_died$the_day_the_da
ned_dog_died$the_day_the_dam
og_died$the_day_the_damned_d
the_damned_dog_died$the_day_
the_day_the_damned_dog_died$
y_the_damned_dog_died$the_da
```

x = "the_day_the_damned_dog_died$"

bwt(x) = "deegdyddee___hhinottdamd_$a"

RLE(x) = 1t1h1e1_1d1a1y1_1t1h1e1_1d1a1m1n1e1d1_1d1o1g1_1d1i1e1d1$

RLE(bwt(x)) = 1d2e1g1d1y2d2e4_2h1i1n1o2t1d1a1m1d1_1$1a

RLE(1stCol(x)) = 1$5_2a6d4e1g2h1i1m1n1o2t1y

Two things we want from compression:

- We want compr(x) to be small
- We *also* want to get x back: x = inv(compr(x))

- The first column in the BWT matrix is optimal for RLE
- We cannot reconstruct x from it
- We can from the last column

Characters in first and last come in the same order...

(when identifying the characters by their position in the original string x$)

$\$_0\,\texttt{mississipp}\,i_0$

$i_0\,\texttt{\$mississip}\,p_0$

$i_1\,\texttt{ppi\$missis}\,s_0$

$i_2\,\texttt{ssippi\$mis}\,s_1$

$i_3\,\texttt{ssissippi\$}\,m_0$

$m_0\,\texttt{ississippi}\,\$_0$

$p_0\,\texttt{i\$mississi}\,p_1$

$p_1\,\texttt{pi\$mississ}\,i_1$

$s_0\,\texttt{ippi\$missi}\,s_2$

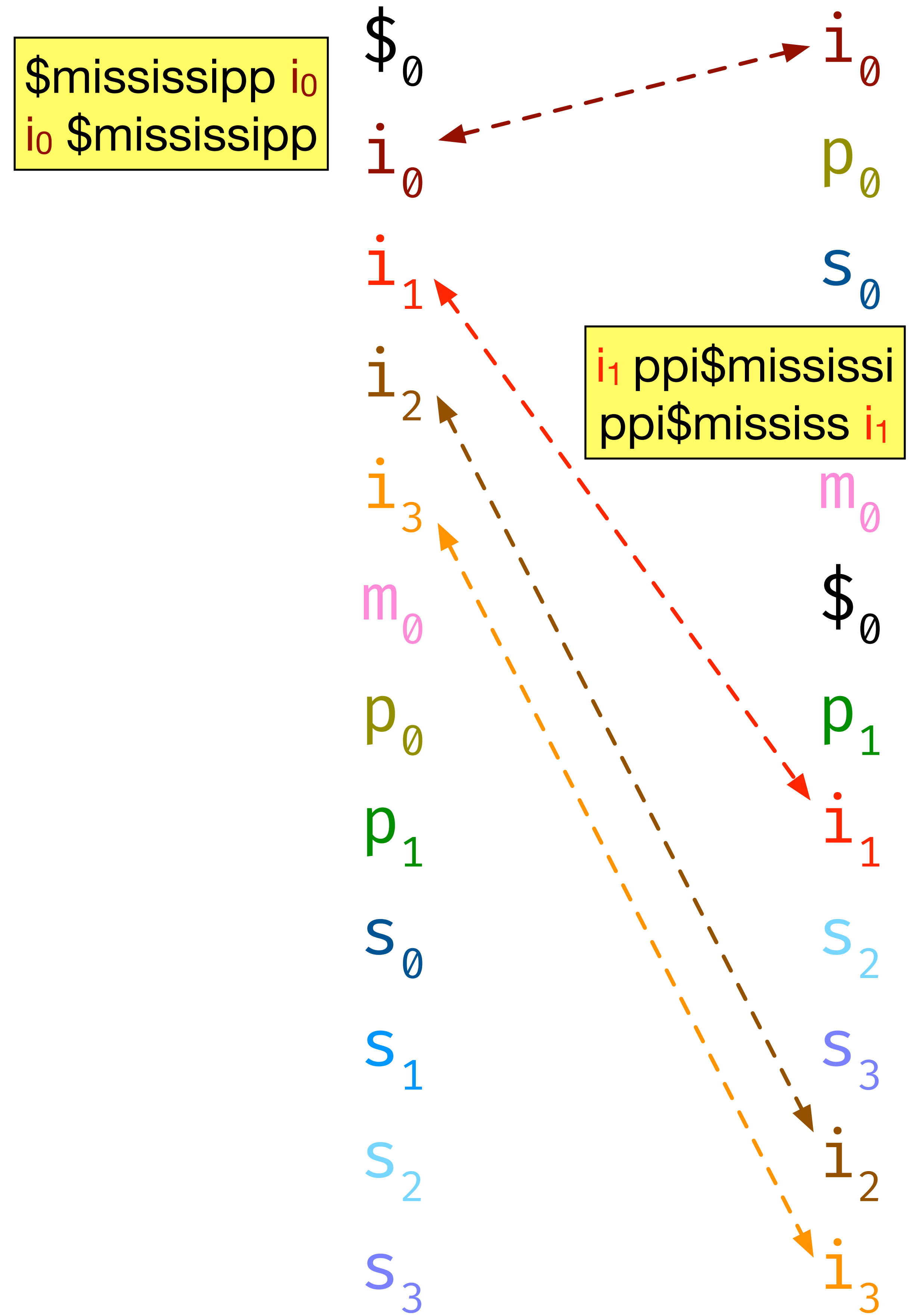
$s_1\,\texttt{issippi\$mi}\,s_3$

$s_2\,\texttt{sippi\$miss}\,i_2$

$s_3\,\texttt{sissippi\$m}\,i_3$

Characters in first and last come in the same order…

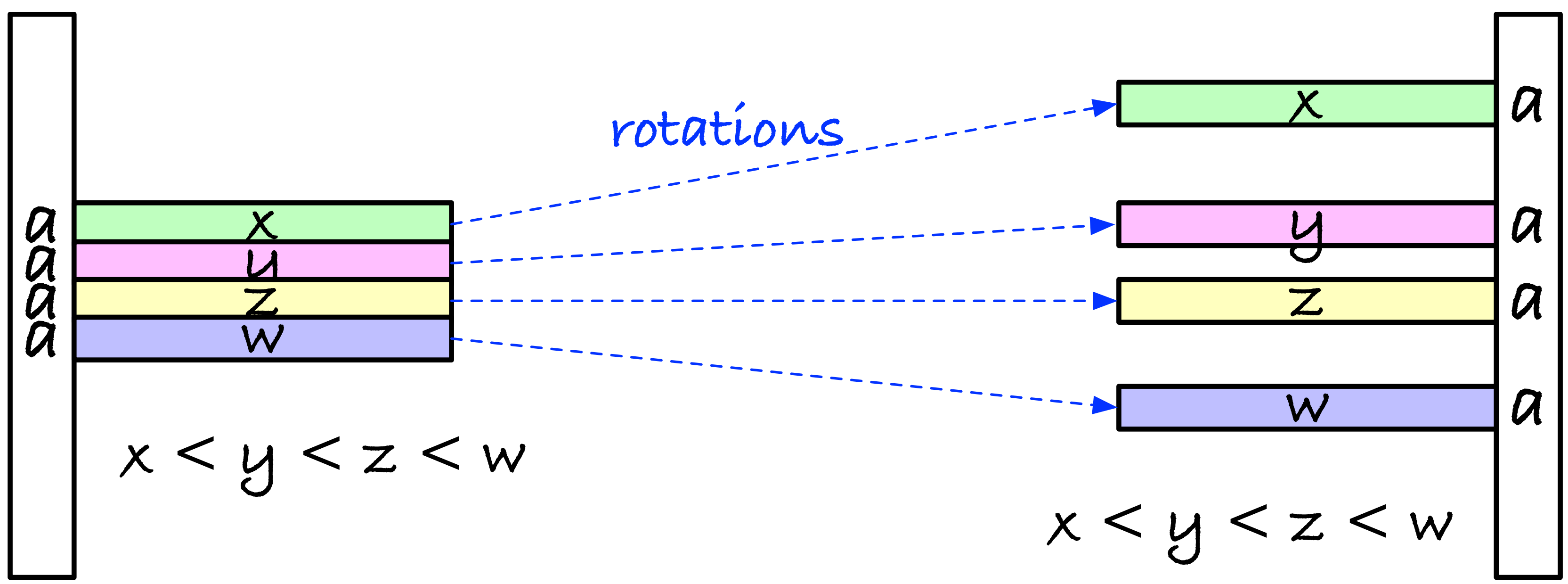
(when identifying the characters by their position in the original string x$)



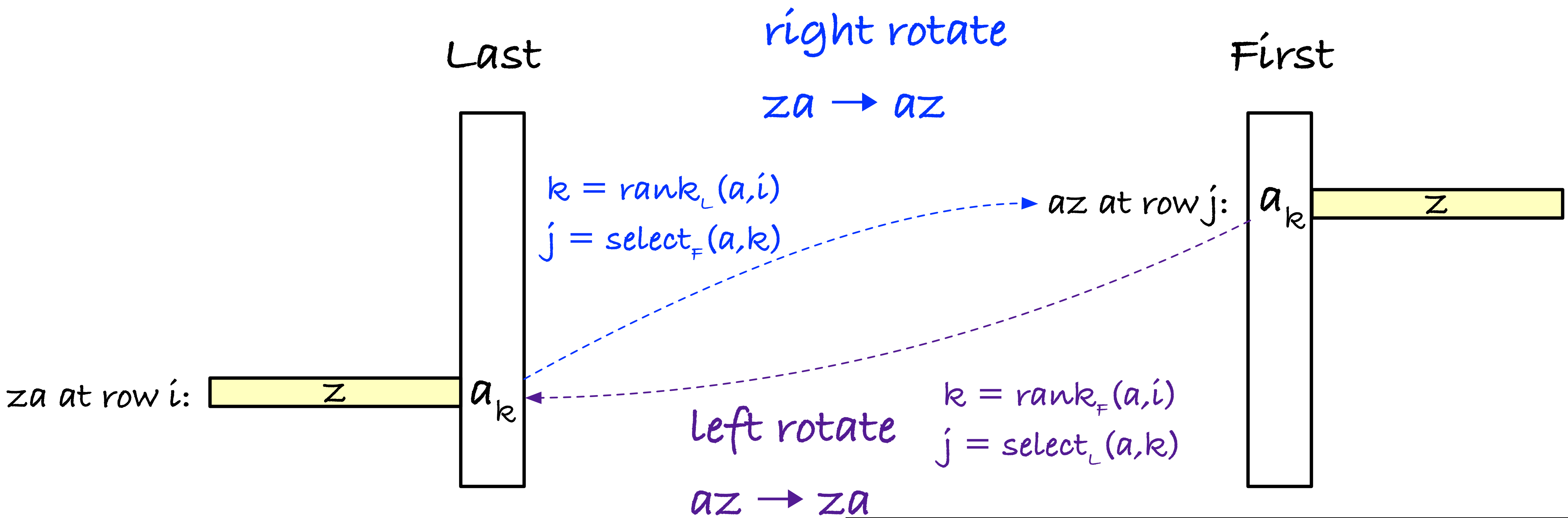
Characters come in the same order

a-bin in first column of BWT matrix

Placement in last column given by order of suffixes of the first column...



Rotating in the BWT matrix



$$\mathrm{rank}_A(a,i) = |\{j < i : A[j] == a\}|$$

$$\mathrm{select}_A(a,k) = \text{pos of the } k\text{'th } a \text{ in } A$$

These are common algorithmic operations. Here on strings (the columns are strings). We need to implement them somehow.

We can rotate using only first and last (no middle columns needed)

We can always rebuild first from last (we only need to count characters)

Reversing transformation

Input L = bwt(mississippi\$) = ipssm\$pissii

Preprocess as needed for rank and select...

$i = select_L(\$,0)$

$\$_0$mississipp$i_0$
i_0\$mississipp_0
$i_1$ppi\$mississ_0
$i_2$ssippi\$miss_1
$i_3$ssissippi\$$m_0$
$m_0$ississippi$\$_0$
p_0i\$mississip_1
$p_1$pi\$mississi_1
$s_0$ippi\$missis_2
$s_1$issippi\$mis_3
$s_2$sippi\$missi_2
$s_3$sissippi\$mi_3

i

Reversing transformation

emit L[i] ($)

	$\$_0$	i_0
	i_0	p_0
	i_1	s_0
	i_2	s_1
	i_3	m_0
i	m_0	$\boxed{\$_0}$
	p_0	p_1
	p_1	i_1
	s_0	s_2
	s_1	s_3
	s_2	i_2
	s_3	i_3

$i = \text{select}_F(L[i], \text{rank}_L(L[i], i))$

[now L[i] is the character just before the previous L[i] in string x$]

i	$\$_0$	i_0
	i_0	p_0
	i_1	s_0
	i_2	s_1
	i_3	m_0
	m_0	$\$_0$
	p_0	p_1
	p_1	i_1
	s_0	s_2
	s_1	s_3
	s_2	i_2
	s_3	i_3

Emitted: $

Reversing transformation

i

emit L[i] (i)

$\$_0$	$\boxed{i_0}$
i_0	p_0
i_1	s_0
i_2	s_1
i_3	m_0
m_0	$\$_0$
p_0	p_1
p_1	i_1
s_0	s_2
s_1	s_3
s_2	i_2
s_3	i_3

$i = \mathrm{select}_F(L[i], \mathrm{rank}_L(L[i], i))$

[now L[i] is the character just before the previous L[i] in string x$]

i

$\$_0$	i_0
i_0	p_0
i_1	s_0
i_2	s_1
i_3	m_0
m_0	$\$_0$
p_0	p_1
p_1	i_1
s_0	s_2
s_1	s_3
s_2	i_2
s_3	i_3

Emitted: i$

Reversing transformation

i

emit L[i]

$\$_0$	i_0
i_0	$\boxed{p_0}$
i_1	s_0
i_2	s_1
i_3	m_0
m_0	$\$_0$
p_0	p_1
p_1	i_1
s_0	s_2
s_1	s_3
s_2	i_2
s_3	i_3

$i = \mathrm{select}_F(L[i], \mathrm{rank}_L(L[i], i))$

[now L[i] is the character just before the previous L[i] in string x$]

i

$\$_0$	i_0
i_0	p_0
i_1	s_0
i_2	s_1
i_3	m_0
m_0	$\$_0$
p_0	p_1
p_1	i_1
s_0	s_2
s_1	s_3
s_2	i_2
s_3	i_3

Emitted: pi$

Reversing transformation

	$\$_0$	i_0
	i_0	p_0
	i_1	s_0
	i_2	s_1
emit L[i]	i_3	m_0
	m_0	$\$_0$
i	p_0	$\boxed{p_1}$
	p_1	i_1
	s_0	s_2
	s_1	s_3
	s_2	i_2
	s_3	i_3

$i = \mathrm{select}_F(L[i], \mathrm{rank}_L(L[i], i))$

[now L[i] is the character just before the previous L[i] in string x$]

	$\$_0$	i_0
	i_0	p_0
	i_1	s_0
	i_2	s_1
	i_3	m_0
	m_0	$\$_0$
	p_0	p_1
i	p_1	i_1
	s_0	s_2
	s_1	s_3
	s_2	i_2
	s_3	i_3

Emitted: ppi$

Reversal as (implicit) rotations

We don't x or its rotations,
but we can see the last character
for all of them...

L[i]

row $select_L(\$,0)$	mississippi$
rotate right	$mississippi
rotate right	i$mississipp
rotate right	pi$mississip
rotate right	ppi$mississi
rotate right	ippi$mississ
rotate right	sippi$missis
rotate right	ssippi$missi
rotate right	issippi$miss
rotate right	sissippi$mis
rotate right	ssissippi$mi
rotate right	ississippi$m

You start with the rotation that ends in \$. Then you keep rotating to get the letter to the left of the current in x, until you have seen n characters. Then you have seen x reversed (so reverse the result or store it backwards).

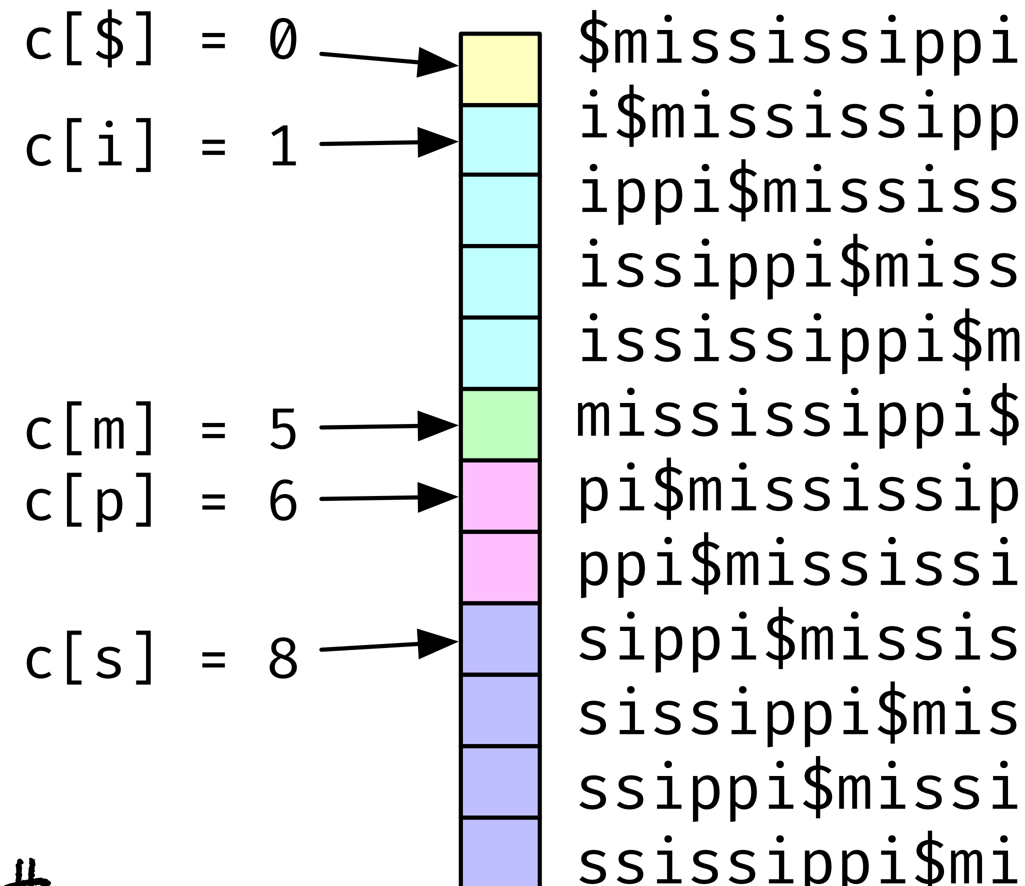
Running time

- You do $rank_L(L[i], i)$ and $select_F(L[i],k)$ n times.
- We shall see that we can do both operations in O(1).
- The running time for reversal: bwt(x) → x is O(n).

Select in first column

c[a] = index of bin a

Just bucketing as we have seen a thousand times before.



x = mississippi$

counts = {$: 1, i: 4, m: 1, p: 2, s: 4}

buckets = {$: 0, i: 1, m: 5, p: 6, s: 8}

Notice: we don't need the first column for this. We can count in the last, that we already have.

Select in first column

$select_F(a,k)$ = index of k'th a
= a number k in the bucket

$select_F(a,k) = C[a] + k$

We can build the C table in O(n)
and then we can select in O(1)

c[i] = 1

c[i] + 0
c[i] + 1
c[i] + 2
c[i] + 3

$\$_0$mississippi
i_0\$mississipp
$i_1$ppi\$mississ
$i_2$ssippi\$miss
$i_3$ssissippi\$m
$m_0$ississippi\$
p_0i\$mississip
$p_1$pi\$mississi
$s_0$ippi\$missis
$s_1$issippi\$mis
$s_2$sippi\$missi
$s_3$sissippi\$mi

Rank in last column

O table just brute force rank...

$O[0,:] = 0$
$O[i,a] = O[i-1,a] + (L[i-1]==a)$

$\text{rank}_L(a,k) = O[k,a]$

We can build the O table in $O(n)$
(because σ is $O(1)$)
and then we can select in $O(1)$

	O[$,	i,	m,	p,	s]
$\$_0$mississipp$i_0$	0,	0,	0,	0,	0
i_0\$mississipp_0	0,	1,	0,	0,	0
$i_1$ppi\$mississ_0	0,	1,	0,	1,	0
$i_2$ssippi\$miss_1	0,	1,	0,	1,	1
$i_3$ssissippi\$$m_0$	0,	1,	0,	1,	2
$m_0$ississippi$\$_0$	0,	1,	1,	1,	2
p_0i\$mississip_1	1,	1,	1,	1,	2
$p_1$pi\$missisi_1	1,	1,	1,	2,	2
$s_0$ippi\$missis_2	1,	2,	1,	2,	2
$s_1$issippi\$mis_3	1,	2,	1,	2,	3
$s_2$sippi\$missi_2	1,	2,	1,	2,	4
$s_3$sissippi\$mi_3	1,	3,	1,	2,	4
(add an extra row, $O[n+1,:]$, it pays off later) →	1,	4,	1,	2,	4

The O table is O(σn) integers (or O(σ n log n) bits). We call that O(n) with constant alphabet and the RAM model.

We can fill the O table in O(σn) by running line by line or column by column.

Rank in last column

O[i,a] is the number of a's in bwt(x) before index i

(which is what $rank_{bwt(x)}(a,i)$ is)

O[7,$] = 1

O[$, i, m, p, s]

	$	i	m	p	s
0:	0,	0,	0,	0,	0
1:	0,	1,	0,	0,	0
2:	0,	1,	0,	1,	0
3:	0,	1,	0,	1,	1
4:	0,	1,	0,	1,	2
5:	0,	1,	1,	1,	2
6:	1,	1,	1,	1,	2
7:	$\boxed{1,}$	1,	1,	2,	2
8:	1,	2,	1,	2,	2
9:	1,	2,	1,	2,	3
10:	1,	2,	1,	2,	4
11:	1,	3,	1,	2,	4
12:	1,	4,	1,	2,	4

bwt(x)

i
p
s
s
m
$\boxed{\$}$
p
i
s
s
i
i

7

Rank in last column

O[i,a] is the number of a's in bwt(x) before index i

(which is what $rank_{bwt(x)}(a,i)$ is)

O[7,i] = 1

O[$, i, m, p, s]

	$	i	m	p	s
0:	0	0	0	0	0
1:	0	1	0	0	0
2:	0	1	0	1	0
3:	0	1	0	1	1
4:	0	1	0	1	2
5:	0	1	1	1	2
6:	1	1	1	1	2
7:	1	$\boxed{1}$	1	2	2
8:	1	2	1	2	2
9:	1	2	1	2	3
10:	1	2	1	2	4
11:	1	3	1	2	4
12:	1	4	1	2	4

bwt(x)

$\boxed{i}$
p
s
s
m
$
p
$\boxed{i}$ 7
s
s
i
i

(we don't count the i at index 7)

Rank in last column

O[i,a] is the number of a's
in bwt(x) before index i

(which is what $\text{rank}_{\text{bwt}(x)}(a,i)$ is)

O[7,m] = 1

	$	i	m	p	s
0:	0	0	0	0	0
1:	0	1	0	0	0
2:	0	1	0	1	0
3:	0	1	0	1	1
4:	0	1	0	1	2
5:	0	1	1	1	2
6:	1	1	1	1	2
7:	1	1	$\boxed{1}$	2	2
8:	1	2	1	2	2
9:	1	2	1	2	3
10:	1	2	1	2	4
11:	1	3	1	2	4
12:	1	4	1	2	4

bwt(x)

i
p
s
s
$\boxed{m}$
$
p
i
s
s
i
i

7

Rank in last column

O[i,a] is the number of a's in bwt(x) before index i

(which is what $rank_{bwt(x)}(a,i)$ is)

O[7,p] = 2

	O[$,	i,	m,	p,	s]
0:	0,	0,	0,	0,	0
1:	0,	1,	0,	0,	0
2:	0,	1,	0,	1,	0
3:	0,	1,	0,	1,	1
4:	0,	1,	0,	1,	2
5:	0,	1,	1,	1,	2
6:	1,	1,	1,	1,	2
7:	1,	1,	1,	$\boxed{2}$,	2
8:	1,	2,	1,	2,	2
9:	1,	2,	1,	2,	3
10:	1,	2,	1,	2,	4
11:	1,	3,	1,	2,	4
12:	1,	4,	1,	2,	4

bwt(x)

i
$\boxed{p}$
s
s
m
$
$\boxed{p}$
i
s
s
i
i

7

Rank in last column

O[i,a] is the number of a's in bwt(x) before index i

(which is what $\text{rank}_{\text{bwt}(x)}(a,i)$ is)

O[7,s] = 2

	O[$	i	m	p	s]
0:	0,	0,	0,	0,	0
1:	0,	1,	0,	0,	0
2:	0,	1,	0,	1,	0
3:	0,	1,	0,	1,	1
4:	0,	1,	0,	1,	2
5:	0,	1,	1,	1,	2
6:	1,	1,	1,	1,	2
7:	1,	1,	1,	2,	$\boxed{2}$
8:	1,	2,	1,	2,	2
9:	1,	2,	1,	2,	3
10:	1,	2,	1,	2,	4
11:	1,	3,	1,	2,	4
12:	1,	4,	1,	2,	4

bwt(x)

i
p
$\boxed{s}$
$\boxed{s}$
m
$
p
i
s
s
i
i

7

Rank in last column

O[i,a] is the number of a's in bwt(x) before index i

(which is what $\text{rank}_{bwt(x)}(a,i)$ is)

O[16,:] == character counts in x

	O[$	i	m	p	s]
0:	0	0	0	0	0
1:	0	1	0	0	0
2:	0	1	0	1	0
3:	0	1	0	1	1
4:	0	1	0	1	2
5:	0	1	1	1	2
6:	1	1	1	1	2
7:	1	1	1	2	2
8:	1	2	1	2	2
9:	1	2	1	2	3
10:	1	2	1	2	4
11:	1	3	1	2	4
12:	1	4	1	2	4

bwt(x)

i
p
s
s
m
$
p
i
s
s
i
i

12

Rank in last column

You can easy represent row zero implicitly

```
O[i,a] = 0 if i == 0 else O'[i-1,a]
```

In the LM index search you never look up in column $, so you don't need it.

(It is still a big table, though...)

O[	$	i	m	p	s]
0:	0	0	0	0	0
1:	0	1	0	0	0
2:	0	1	0	1	0
3:	0	1	0	1	1
4:	0	1	0	1	2
5:	0	1	1	1	2
6:	1	1	1	1	2
7:	1	1	1	2	2
8:	1	2	1	2	2
9:	1	2	1	2	3
10:	1	2	1	2	4
11:	1	3	1	2	4
12:	1	4	1	2	4

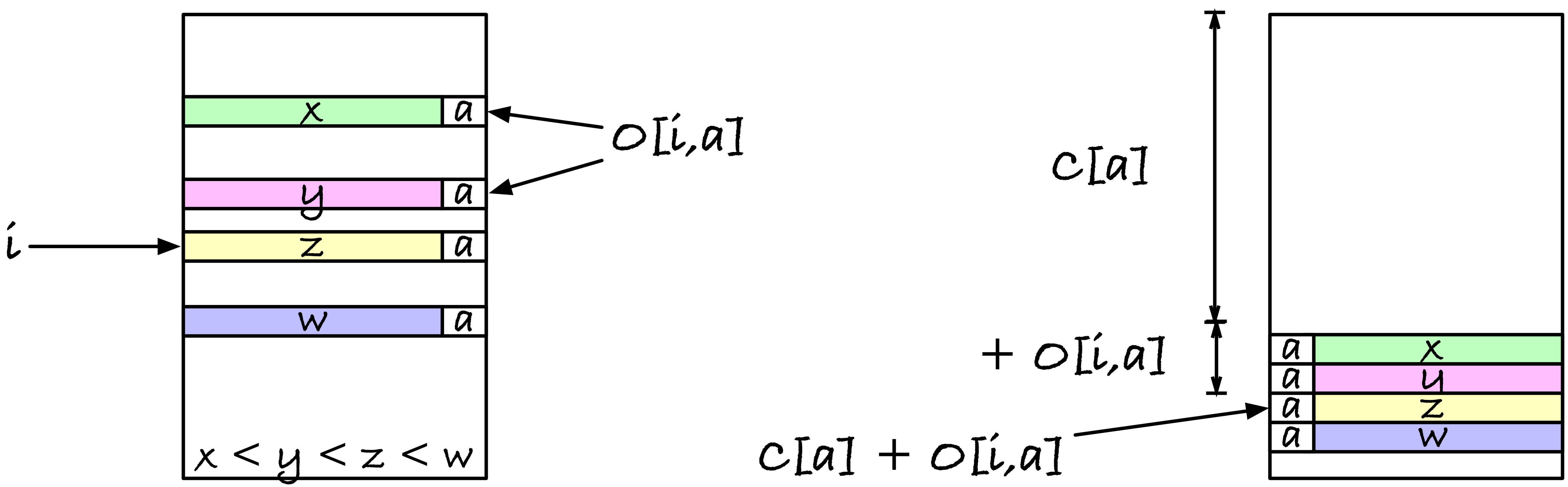
We can talk about ways to reducing the size in one of the mystery lectures at the end of the semester...

FM index

Ferragina and Manzini index

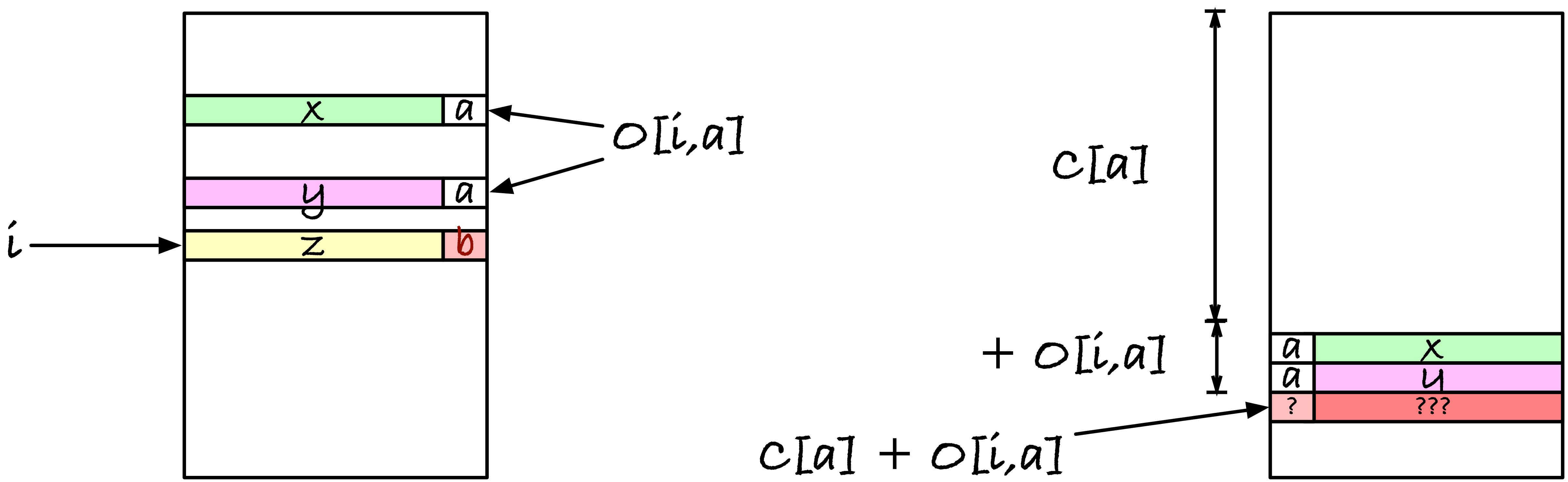
General jump

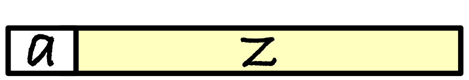
Rotating the last character to the front...



General jump

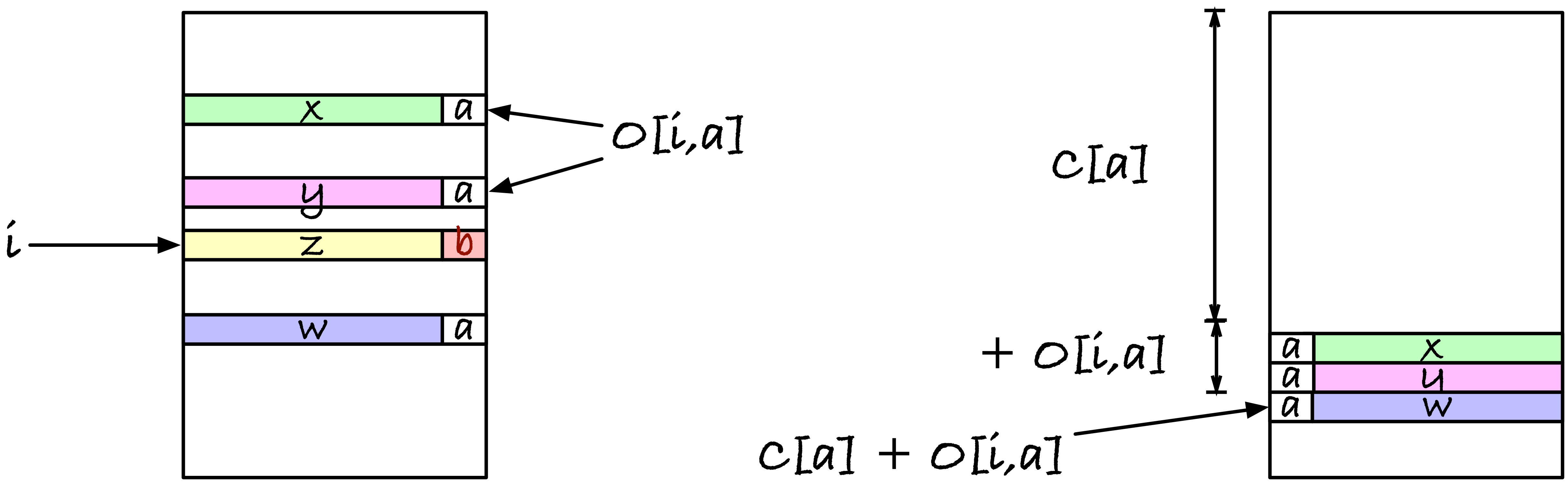
Rotating the last character to the front...



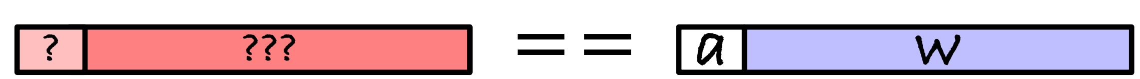
is the smallest rotation ≥


General jump

Rotating the last character to the front...



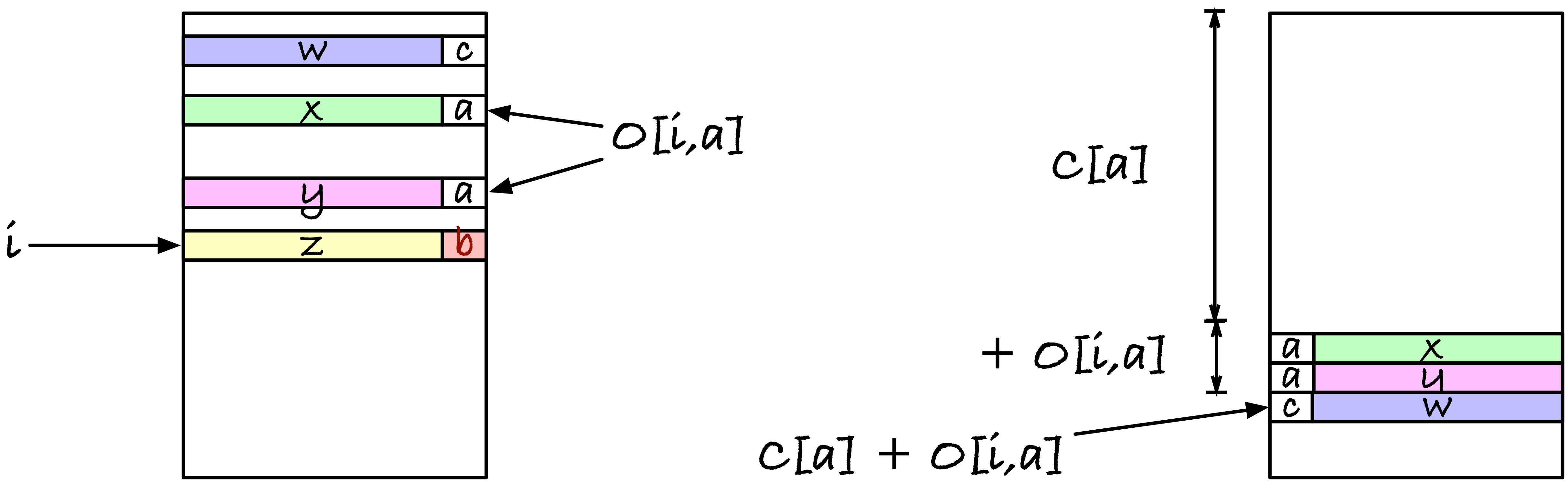
Assume

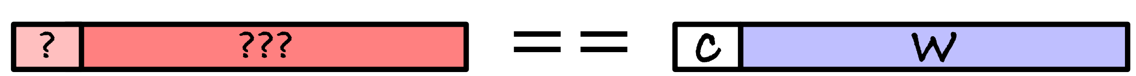


Then w ≥ z (or it would go above C[a]+O[i,a]), and it is the smallest aw since the a-bin is sorted.

General jump

Rotating the last character to the front...

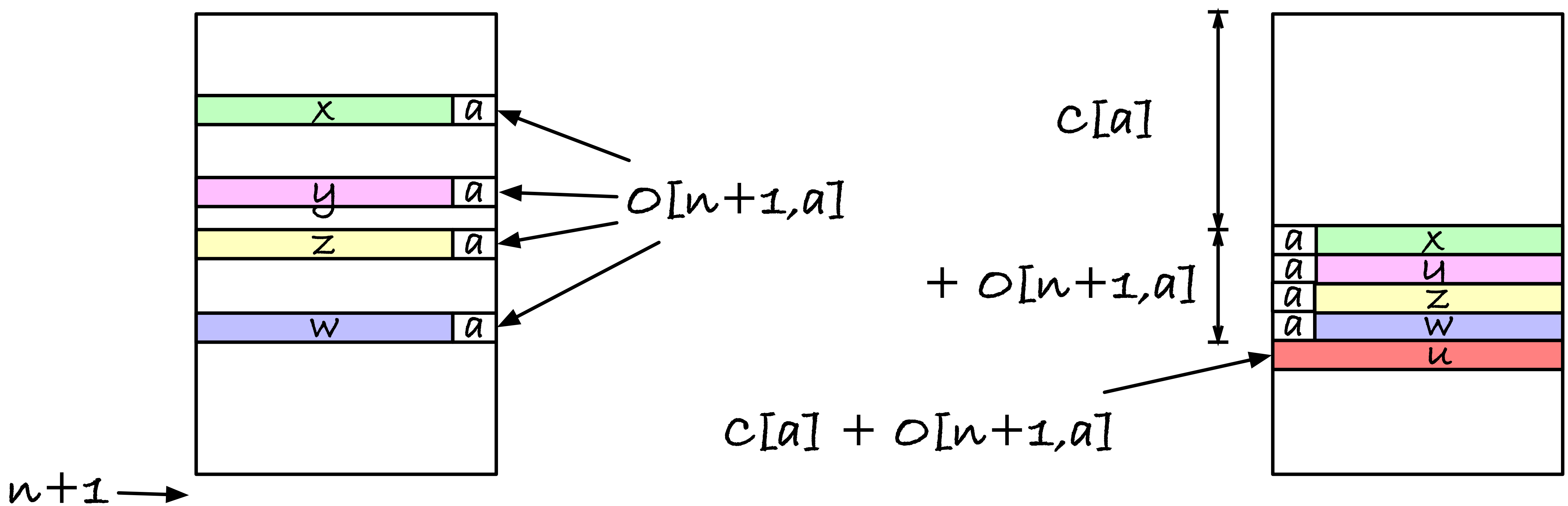


Assume 

Then cw is the smallest rotation after those that start with a, and since all rotations that start with a, ax, are smaller than az (or we wouldn't point beyond the a-bin), it is the smallest greater than az.

General jump

Special case for row n - 1



u is the smallest string after the a-bin

FM index search

Start with the smallest and one-past-the-largest rotations. "Rotate" according to reverse p.

```
L, R = 0, n+1
for a in reversed(p):
    if L == R: break
    L = C[a] + O[L,a]
    R = C[a] + O[R,a]
```

x = "mississippi"
p = "ssi"

L

$mississippi

rotate "i"

$mississippi
i$mississipp
ippi$mississ

rotate "s"

ppi$mississi
[si$mississip]
sippi$missis
sissippi$mis

rotate "s"

sissippi$mis
ssippi$missi
ssissippi$mi

R

ssissippi$mi

rotate "i"

ississippi$m
mississippi$
pi$mississip

rotate "s"

sissippi$mis
[smississippi]
ssippi$missi
ssissippi$mi

rotate "s"

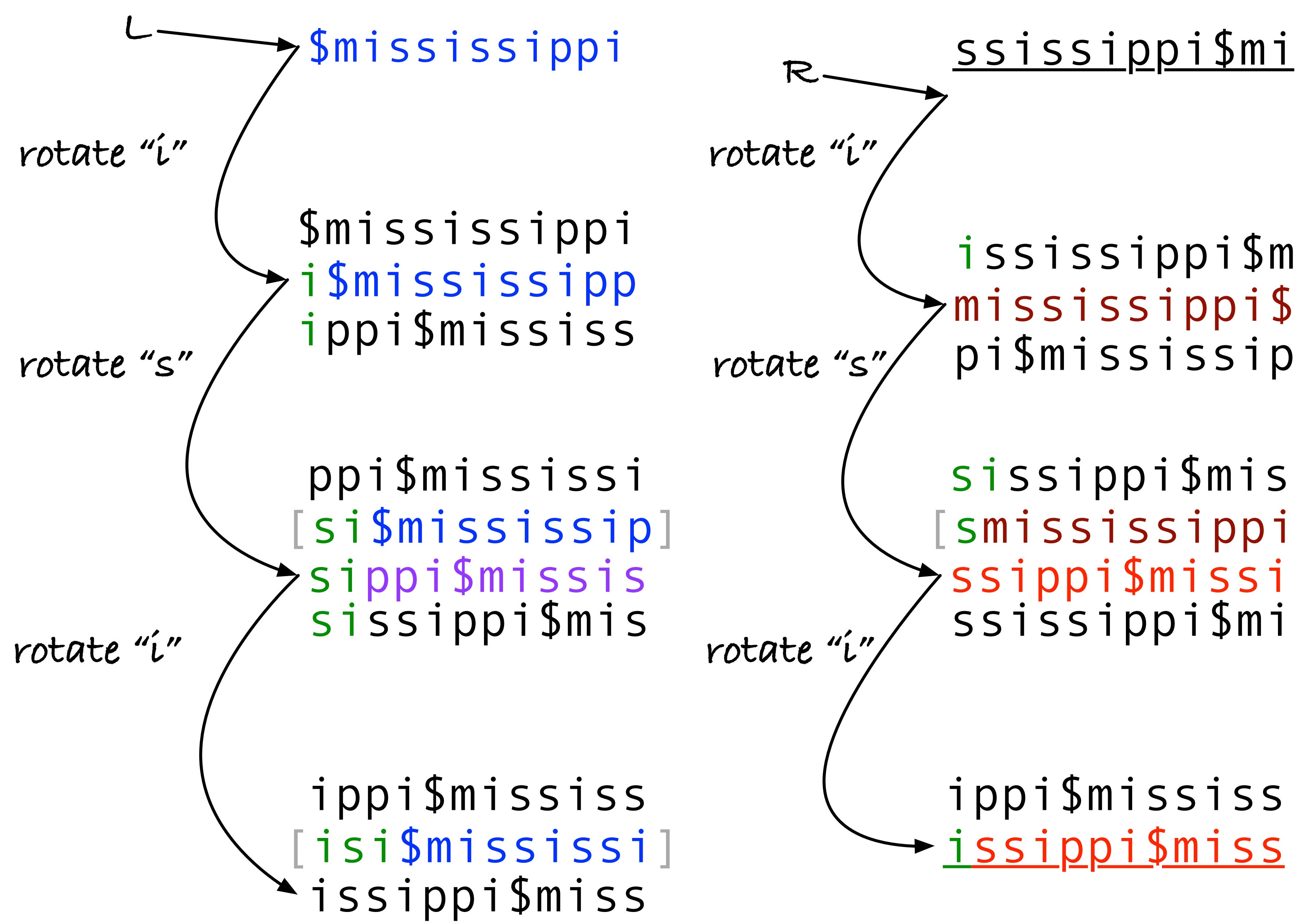
ssissippi$mi
[sssippi$miss]

FM index search

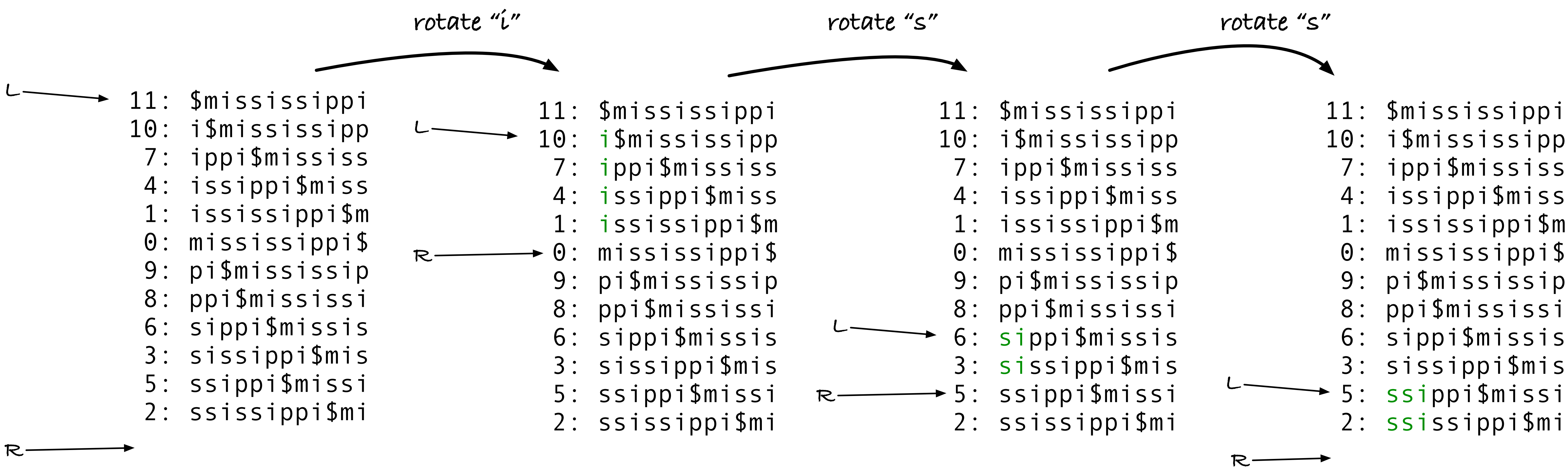
Start with the smallest and one-past-the-largest rotations. "Rotate" according to reverse p.

```
L, R = 0, n+1
for a in reversed(p):
    if L == R: break
    L = C[a] + O[L,a]
    R = C[a] + O[R,a]
```

x = "mississippi"
p = "isi"



FM index search



Matches at sa[i] for i in range(L,R)

Running time

- Preprocessing: O(n)
    - Suffix array [to get bwt(x)]
    - C and O tables
- Search: O(m)


That's all Folks!