Backward Search FM-Index

(Full-text index in Minute space)

Paper by
Ferragina & Manzini

Presentation by
Yuval Rikover

MOTIVATION

- Combine Text compression with indexing (discard original text).
- Count and locate P by looking at only a small portion of the compressed text.
- Do it efficiently:
  - Time: $O(p)$
  - Space: $O(n\ H_k(T)) + o(n)$

HOW DOES IT WORK?

- Exploit the relationship between the *Burrows-Wheeler Transform* and the Suffix Array data structure.
- Compressed suffix array that encapsulates both the ***compressed text*** and the ***full-text indexing information.***
- Supports two basic operations:
  - **Count** – return number of occurrences of P in T.
  - **Locate** – find all positions of P in T.

BUT FIRST – COMPRESSION

- Process T[1,..,n] using Burrows-Wheeler Transform
  - Receive string L[1,..,n]   (permutation of T)

- Run Move-To-Front encoding on L
  - Receive L^{MTF}[1,…,n]

- Encode runs of zeroes in L^{MTF} using run-length encoding
  - Receive L^{rle}

- Compress L^{rle} using variable-length prefix code
  - Receive Z   (over alphabet {0,1})

BURROWS-WHEELER TRANSFORM

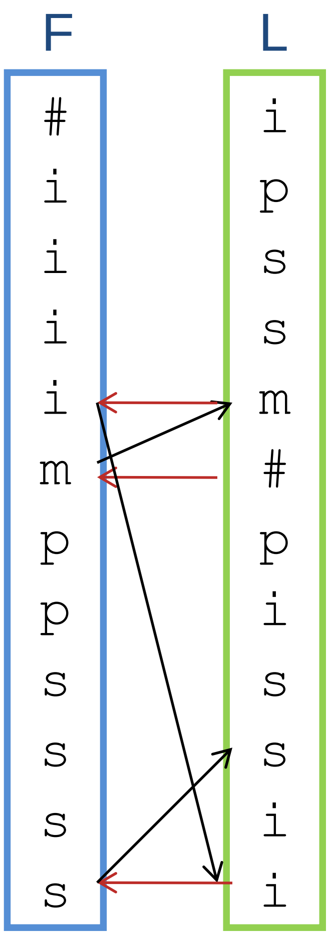
- Every column is a permutation of T.
- Given row i, char L[i] precedes F[i] in original T.
- Consecutive char's in L are adjacent to similar strings in T.
- Therefore – L usually contains long runs of identical char's.

F		L
#	mississipp	i
i	#mississip	p
i	ppi#missis	s
i	ssippi#mis	s
i	ssissippi#	m
m	ississippi	#
p	i#mississi	p
p	pi#mississ	i
s	ippi#missi	s
s	issippi#mi	s
s	sippi#miss	i
s	sissippi#m	i

BURROWS-WHEELER TRANSFORM

Reminder: Recovering T from L

1. Find F by sorting L
2. First char of T? m
3. Find m in L
4. L[i] precedes F[i] in T. Therefore we get
   mi
5. How do we choose the correct i in L?
   - The i's are in the same order in L and F
   - As are the rest of the char's
6. i is followed by s: mis
7. And so on….



MOVE-TO-FRONT

- Replace each char in L with the number of distinct char's seen since its last occurrence.
- Keep MTF[1,…,|Σ|] array, sorted lexicographically.
- Runs of identical char's are transformed into runs of zeroes in L^{MTF}

L	i	p	s	s	m	#	p	i	s	s	i	i
	1	3	4	0	4	4	3	4	4	0	1	0

- Bad example
- For larger, English texts, we will receive more runs of zeroes, and dominancy of smaller numbers.
- The reason being that BWT creates clusters of similar char's.

RUN-LENGTH ENCODING

- Replace any sequence of zeroes 0^m with:
  - (m+1) in binary
  - LSB first
  - Discard MSB
- Add 2 new symbols – ***0,1***
- L^{rle} is defined over **{ *0*,*1*,1,2,…,|Σ|}**

L: ii ppp sp ss smm #e ip p i# si ss s i i ii

L^{MTF}: 21 43 0 40 50 0 45 45 43 4 45 42 50 01 1 00

L^{rle}	2	4	***1***	5	***0***	5	5	4	4	5	2	5	***0***	1	***0***

To give a meatier example (not really), we'll change our text to:

T = pipeMississippi#

4. 0000 → 4+1 = 5 → **101** → **101** → **10**
5. 00000→ 5+1 = 6 → **110** → **011** → **01**
6. 000000 → 6+1 = 7 → **111** → **111** → **11**
7. 0000000 → 7+1 = 8 → **1000** → **0001**→**000**

RUN-LENGTH ENCODING

- Replace any sequence of zeroes 0^m with:
  - (m+1) in binary
  - LSB first
  - Discard MSB
- Add 2 new symbols – ***0,1***
- L^{rle} is defined over **{ *0*,*1*,**1,2**,…,|Σ|}**

<u>How to retrieve m</u>

- given a binary number $b_0 b_1,\ldots,b_k$
- Replace each bit b_j with a sequence of $(b_j+1)\cdot 2^j$ zeroes
- **10** → $(1+1)\cdot 2^0+(0+1)\cdot 2^1=4$ Zeroes

<u>Example</u>

1. 0 → 1+1 = 2 → **10** → **01** → **0**
2. 00 → 2+1 = 3 → **11** → **11** → **1**
3. 000 → 3+1 = 4 → **100** → **001** → **00**
4. 0000 → 4+1 = 5 → **101** → **101** → **10**
5. 00000→ 5+1 = 6 → **110** → **011** → **01**
6. 000000 → 6+1 = 7 → **111** → **111** → **11**
7. 0000000 → 7+1 = 8 → **1000** → **0001**→**000**

VARIABLE-LENGTH PREFIX CODING

011 00101 11 00110 10 00110 00110 00101 00101 00110 011 00110 10 010 10

L^{rle}	2	4	***1***	5	***0***	5	5	4	4	5	2	5	***0***	1	***0***

over alphabet {0,1}:

- ***1*** → 11 ***0*** → 10
- For i = 1,2,…, |Σ| - 1
  - $\lfloor \log(i+1) \rfloor$ zeroes
  - Followed by binary representation of i+1 which takes 1+ $\lfloor \log(i+1) \rfloor$
  - For a total of 1+2$\lfloor \log(i+1) \rfloor$ bits

Example

1. i=1 → $\lfloor \log(2) \rfloor$ 0's, bin(2) → 010
2. i=2 → $\lfloor \log(3) \rfloor$ 0's, bin(3) → 011
3. i=3 → $\lfloor \log(4) \rfloor$ 0's, bin(4) → 00100
4. i=4 → $\lfloor \log(5) \rfloor$ 0's, bin(5) → 00101
5. i=5 → $\lfloor \log(6) \rfloor$ 0's, bin(6) → 00110
6. i=6 → $\lfloor \log(7) \rfloor$ 0's, bin(7) → 00111
7. i=7 → $\lfloor \log(8) \rfloor$ 0's, bin(8) → 0001000

Compression - Space Bound

- In 1999, Manzini showed the following upper bound for BWT compression ratio:
  - $8nH_k(T)+(0.08+1)\cdot n+\log(n)+g(k,|\Sigma|)$

$$|Z| \le 5nH_k(T)+O(\log n)$$

$$nH_k(T) \le nH_k^*(T) \le nH_k(T)+O(\log n)$$

- In 2001, Manzini showed that for every k, the above compression method is bounded by:

$$5nH_k^*(T)+g(k,|\Sigma|)$$

NEXT: COUNT P IN T

- **Backward-search algorithm**
- Uses only L (output of BWT)
- Relies on 2 structures:
  - C[1,…,|Σ|] : C[c] contains the total number of text chars in T which are alphabetically smaller then c (including repetitions of chars)
  - Occ(c,q): number of occurrences of char c in prefix L[1,q]

<u>Example</u>

- C[ ] for T = mississippi#

1	5	6	8
i	m	p	s

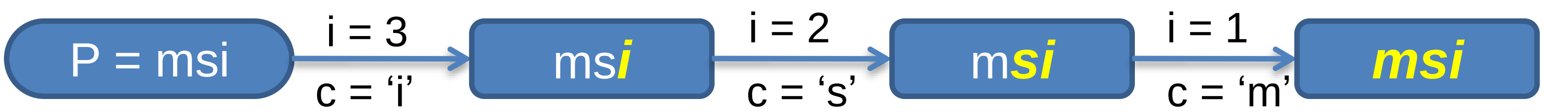
- occ(s, 5) = 2
- occ(s,12) = 4

Occ ≡ Rank

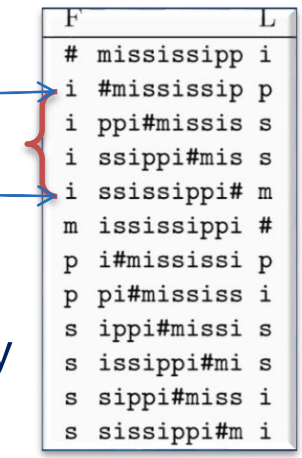
F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

BACKWARD-SEARCH

- Works in p iterations, from p down to 1



- Remember that the BWT matrix rows = sorted suffixes of T
  - All suffixes prefixed by pattern P, occupy a continuous set of rows
  - This set of rows has starting position ***First***
  - and ending position ***Last***
  - So, (Last – First +1) gives total pattern occurrences



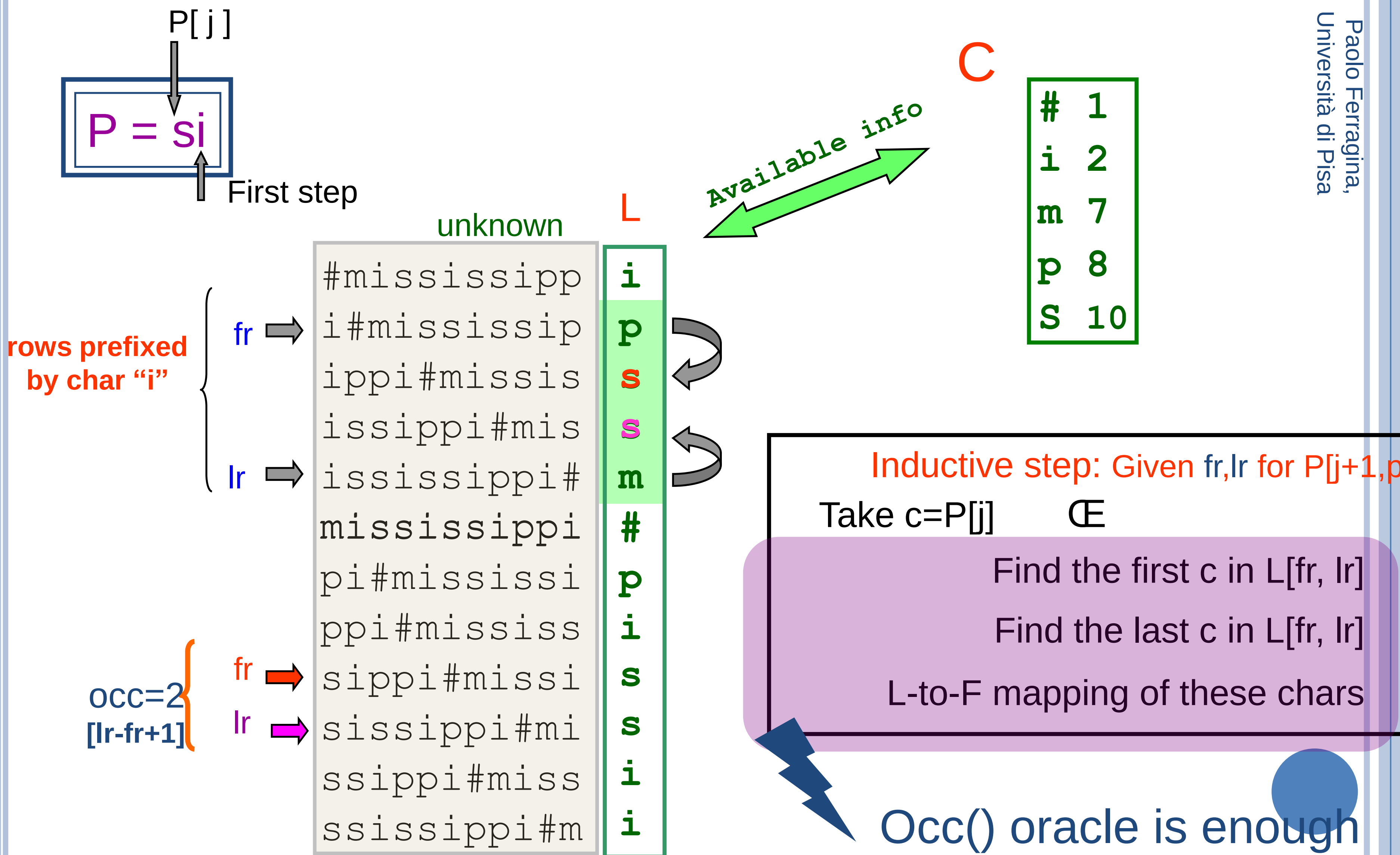
- At the end of the i-th phase, ***First*** points to the first row prefixed by P[i,p], and ***Last*** points to the last row prefiex by P[i,p].

Algorithm backward_search($P[1,p]$)

(1) $i \leftarrow p$, $c \leftarrow P[p]$, First $\leftarrow C[c]+1$, Last $\leftarrow C[c+1]$;
(2) **while** ((First $\leq$ Last) **and** $(i \geq 2)$) **do**
(3) $\quad c \leftarrow P[i-1]$;
(4) $\quad$ First $\leftarrow C[c] + \mathrm{Occ}(c, \mathrm{First}-1)+1$;
(5) $\quad$ Last $\leftarrow C[c] + \mathrm{Occ}(c, \mathrm{Last})$;
(6) $\quad i \leftarrow i-1$;
(7) **if** (Last < First) **then return** "no rows prefixed by $P[1,p]$" **else return** $\langle$First, Last$\rangle$.

Substring search in T (Count the pattern occurrences)

Paolo Ferragina, Università di Pisa



BACKWARD-SEARCH EXAMPLE

- **P = pssi**
  - i = 3
  - c = ‘s’
  - First = C[‘s’] + Occ(‘s’,1) +1 = 8+0+1 = 9
  - Last = C[‘s’] + Occ(‘s’,5) = 8+2 = 10
  - (Last – First + 1) = 2

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

First → row 2; Last → row 5

C[ ] =

i	m	p	s
1	5	6	8

Algorithm backward_search($P[1,p]$)

(1) $i \leftarrow p$, $c \leftarrow P[p]$, First $\leftarrow C[c]+1$, Last $\leftarrow C[c+1]$;
(2) **while** ((First $\leq$ Last) **and** ($i \geq 2$)) **do**
(3) $\quad c \leftarrow P[i-1]$;
(4) $\quad$ First $\leftarrow C[c]$ + Occ(c, First -1) $+1$;
(5) $\quad$ Last $\leftarrow C[c]$ + Occ(c, Last);
(6) $\quad i \leftarrow i-1$;
(7) **if** (Last < First) **then return** “no rows prefixed by $P[1,p]$” **else return** ⟨First, Last⟩.

BACKWARD-SEARCH EXAMPLE

- P = pssi
  - i = 2
  - c = 's'
  - First = C['s'] + Occ('s',8) +1 = 8+2+1 = 11
  - Last = C['s'] + Occ('s',10) = 8+4 = 12
  - (Last – First + 1) = 2

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

First → row 9; Last → row 10

C[ ] =

i	m	p	s
1	5	6	8

Algorithm backward_search($P[1,p]$)

(1) $i \leftarrow p$, $c \leftarrow P[p]$, First $\leftarrow C[c]+1$, Last $\leftarrow C[c+1]$;
(2) **while** ((First $\leq$ Last) **and** $(i \geq 2)$) **do**
(3) $c \leftarrow P[i-1]$;
(4) First $\leftarrow C[c] + \text{Occ}(c, \text{First}-1) + 1$;
(5) Last $\leftarrow C[c] + \text{Occ}(c, \text{Last})$;
(6) $i \leftarrow i-1$;
(7) **if** (Last < First) **then return** "no rows prefixed by $P[1,p]$" **else return** ⟨First, Last⟩.

BACKWARD-SEARCH EXAMPLE

- **P = pssi**
  - i = 1
  - c = ‘p’
  - First = C[‘p’] + Occ(‘p’,10) +1 = 6+2+1 = 9
  - Last = C[‘p’] + Occ(‘p’,12) = 6+2 = 8
  - (Last – First + 1) = 0

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

First → row 11, Last → row 12

C[ ] =

i	m	p	s
1	5	6	8

Algorithm backward_search($P[1,p]$)

(1) $i \leftarrow p$, $c \leftarrow P[p]$, First $\leftarrow C[c]+1$, Last $\leftarrow C[c+1]$;

(2) **while** ((First $\leq$ Last) **and** ($i \geq 2$)) **do**

(3) $\quad c \leftarrow P[i-1]$;

(4) $\quad$ First $\leftarrow C[c]$ + Occ(c, First -1) $+1$;

(5) $\quad$ Last $\leftarrow C[c]$ + Occ(c, Last);

(6) $\quad i \leftarrow i-1$;

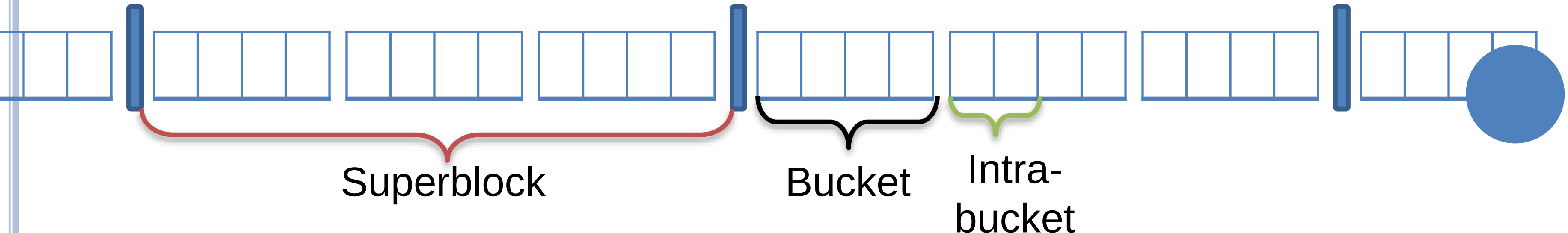
(7) **if** (Last $<$ First) **then return** “no rows prefixed by $P[1,p]$” **else return** ⟨First, Last⟩.

BACKWARD-SEARCH ANALYSIS

- Backward-search makes P iterations, and is dominated by Occ( ) calculations.
- Implement Occ( ) to run in O(1) time, using $|Z| + O\left(n \cdot \frac{\log\log n}{\log n}\right)$ bits.

  Compressed text

  - So Count will run in O(p) time, using $5nH_k(T) + O\left(n \cdot \frac{\log\log n}{\log n}\right)$ bits.
- We saw a partitioning of binary strings into Buckets and Superblocks for answering Rank( ) queries.
  - We'll use a similar solution
  - With the help of some new structures
- **General Idea**: Sum character occurrences in 3 stages



IMPLEMENTING OCC( )

- **Buckets**
  - Partition L into substrings of $l = \Theta \log n$ chars each, denoted BL_i
  - This partition induces a partition on L^{MTF}, denoted BL_i^{MTF} $\quad i = 1,..,n/l$

T = pipeMississippi# |T| = |L| = 16 $|BL_i^{MTF}| = 4$

L	i	p	p	p	s	s	m	e	i	p	#	i	s	s	i	i
L^{MTF}	2	4	0	0	5	0	5	5	4	4	5	2	5	0	1	0

  - Applying *run-length encoding* and *prefix-free encoding* on each bucket will result in n/log(n) variable-length buckets, denoted BZ_i

011 00101 11 | 00110 10 00110 00110 | 00101 00101 00110 011 | 00110 10 010 10

BZ_1 (011 00101 11) … BZ_4 (00110 10 010 10)

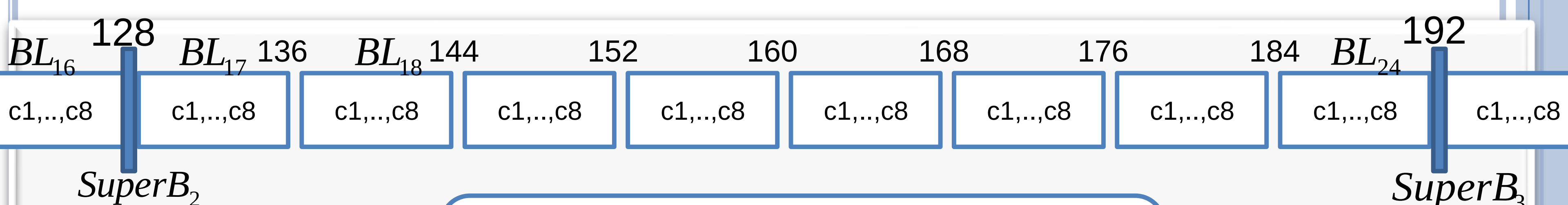
IMPLEMENTING OCC( )

- **Superblocks**
  - We also partition L into *superblocks* of size $l^2 = \Theta \log^2(n)$ each.

$$|T| = |L| = 256 \qquad \left|BL_i^{MTF}\right|_{Chars} = 8 \qquad i = 1,2,\dots,32$$

$$\left|SuperB_j\right|_{Chars} = 64 \qquad j = 1,2,3,4$$

  - Create a table for each superblock, holding for each character *c* ϵ Σ, the number of *c*'s occurrences in L, up to the start of the specific superblock.
    - Meaning, for $SuperB_j$, store occurrences of c in range $L[1,\dots, j \cdot l^2]$



$SuperB_2$

a	32
b	25
:	:
z	7

NO_2

$$O\left(\frac{n}{l^2} \cdot \log n \cdot |\Sigma|\right) = O\left(\frac{n}{\log n}\right)$$

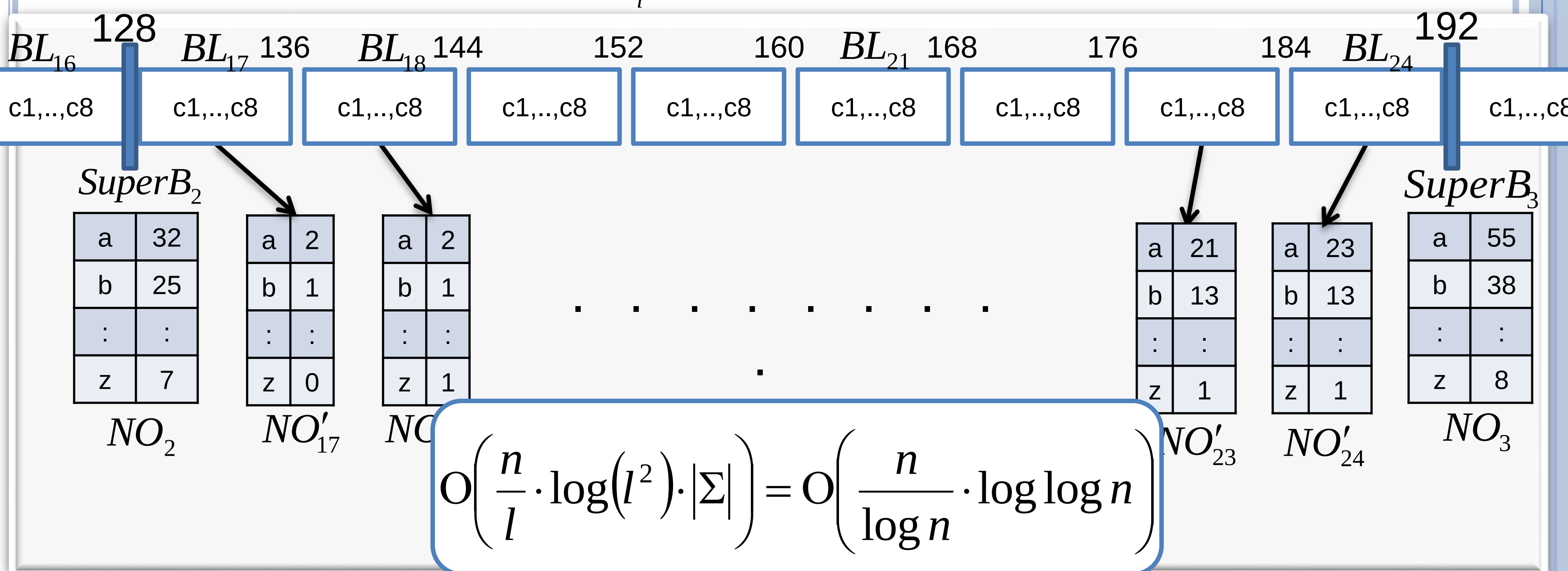
$SuperB_3$

a	55
b	38
:	:
z	8

NO_3

IMPLEMENTING OCC( )

- **<u>Back to Buckets</u>**
  - Create a similar table for buckets, but count only from current superblock's start. Denote tables as NO'_i

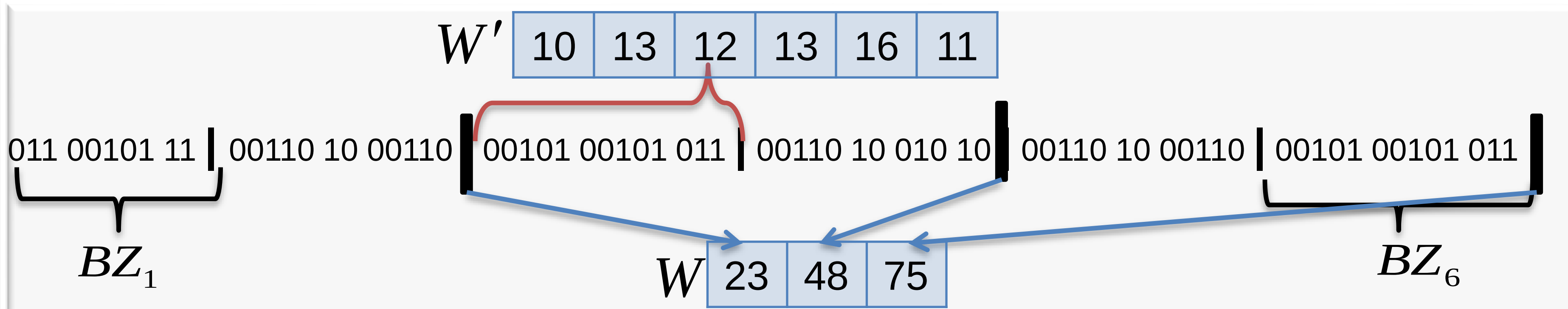


$$O\left(\frac{n}{l}\cdot\log\left(l^2\right)\cdot|\Sigma|\right)=O\left(\frac{n}{\log n}\cdot\log\log n\right)$$

- Only thing left is searching inside a bucket.
  - For example, Occ(c,164) will require counting in BL_{21}
  - But we only have the compressed string Z.
  - Need more structures

IMPLEMENTING OCC( )

- **Finding BZ_i 's starting position in Z, part 1**
  - Array $W\left[1,\ldots,n/l^2\right]$ keeps for every $SuperB_j$, the sum of the sizes of the compressed buckets $BZ_1,\ldots\ldots,BZ_{j\cdot\log n}$ (in bits).
  - Array $W'\left[1,\ldots,n/l\right]$ keeps for every bucket BZ_i , the sum of bucket sizes up to it (including), from the superblock's beginning.



$$|W| = O\left(\frac{n}{l^2}\cdot\log n\cdot\left(1+2\lfloor\log|\Sigma|\rfloor\right)\right) = O\left(\frac{n}{\log n}\right)$$

$$|W'| = O\left(\frac{n}{l}\cdot\log\left(l^2\right)\right) = O\left(\frac{n}{\log n}\cdot\log\log n\right)$$

IMPLEMENTING OCC( )

- **Finding BZ_i 's starting position in Z, part 2**
  - Given Occ('c',q)

Occ(k,166)

Find i of BL_i: $i = \lceil q / \log n \rceil$

166/8 = 20.75
→ i = 21

Find character in BL_i to count up to:
$$h = q - (i-1) \cdot \log n$$

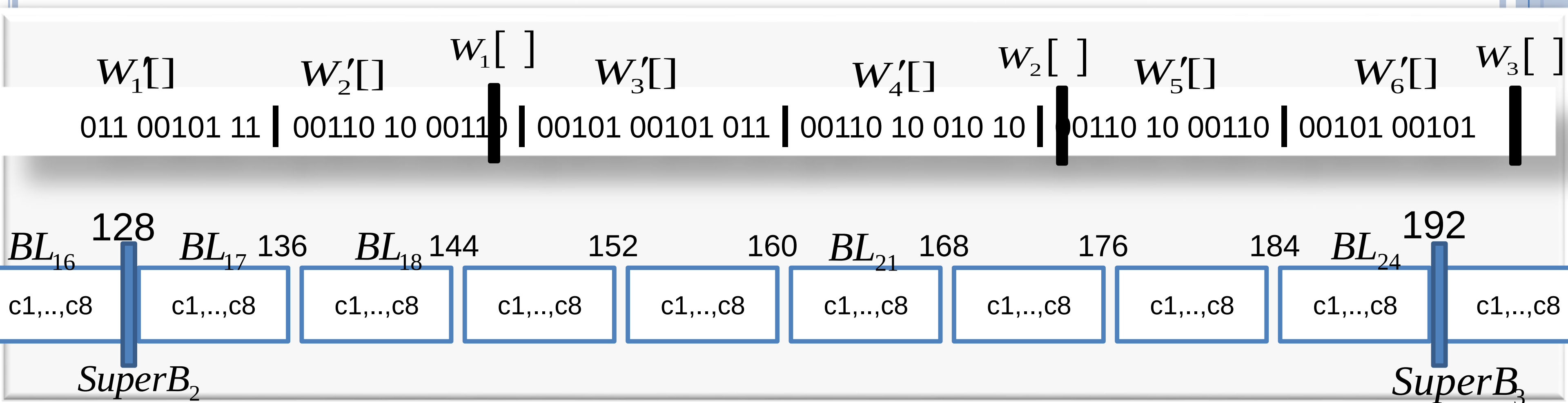
h = 166 – 20*8
→ h = 6

Find superblock $SuperB_t$ of BL_i
$$t = \left\lceil \frac{i}{\log n} \right\rceil - 1$$

(21/8) = 2.625
→ t = 2

Locate position of BZ_i in Z:
$$W[t] + W'[i-1] + 1$$

Compressed bucket is at
W[2]+W`[20]+1



IMPLEMENTING OCC( )

- We have the compressed bucket BZ_i | 011 00101 11 |
  - And we have h (how many chars to check in BZ_i). *h*
  - But BZ_i contains compressed Move-To-Front information…
  - Need more structures!
- For every i, before encoding bucket BL_i with Move-To-Front, keep the state of the MTF table

T = pipeMississippi# |T| = |L| = 16

L	i	p	p	p	s	s	m	e	i	p	#	i	s	s	i	i
L^{MTF}	2	4	0	0	5	0	5	5	4	4	5	2	5	0	1	0

#	e	i	m	p	s

MTF_1

p	i	#	e	m	s

MTF_2

e	m	s	p	i	#

MTF_3

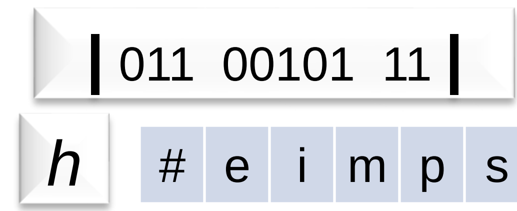
i	#	p	e	m	s

MTF_4

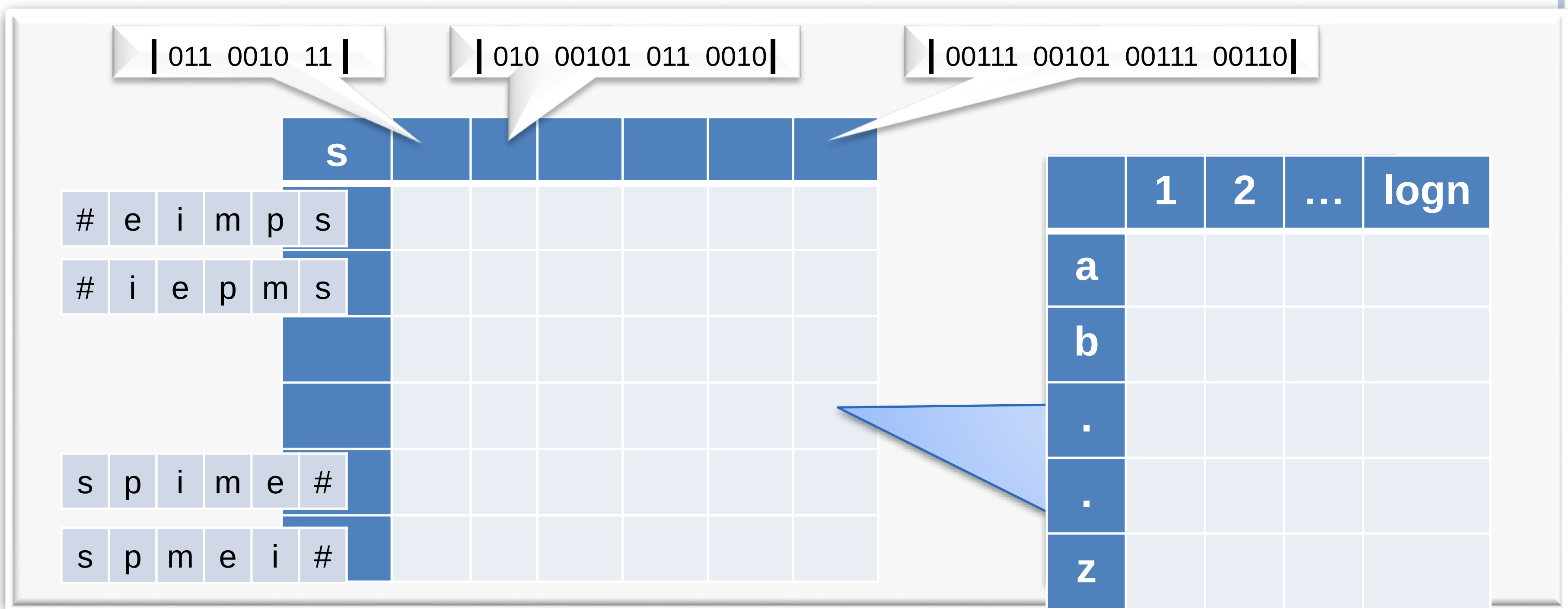
$$O\left(\frac{n}{\log n}\cdot|\Sigma|\log|\Sigma|\right)=O\left(\frac{n}{\log n}\right)$$

IMPLEMENTING OCC( )

- How do we use MTF_i to count inside BZ_i ?
- Final structure!



- Build a table $S[c, h, BZ_i, MTF_i]$, which stores the number of occurrences of ‘c’ among the first *h* characters of BL_i
  - Possible because BZ_i and MTF_i together, completely determine BL_i



IMPLEMENTING OCC( )

- Max size of a compressed bucket BZ_i : $l \cdot (1 + 2\lfloor \log|\Sigma| \rfloor)$
  - Number of possible compressed buckets : $2^{l \cdot (1 + 2\lfloor \log|\Sigma| \rfloor)} = 2^t$
- Number of possible MFT table states: $2^{|\Sigma| \log|\Sigma|}$
- Size of inner table: $|\Sigma| l$
  - Size of each entry: $\log l$
- Total size of S[ ]: $O(2^t l \log l) = O(n \cdot \log n \cdot \log\log n)$

•But – having a *linear sized index is bad* for practical uses when n is large.

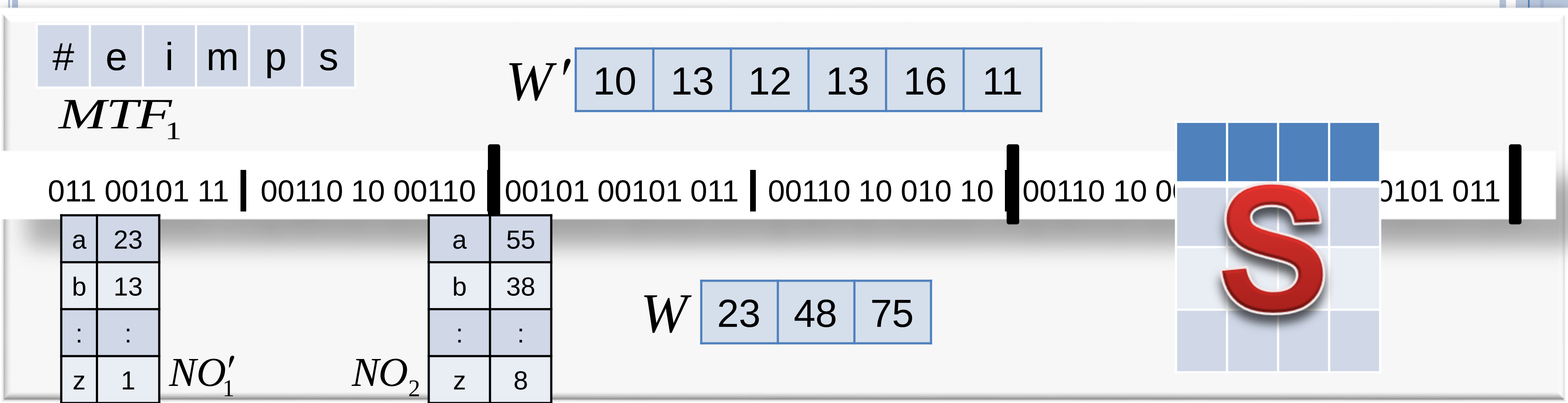
•We want to keep the index size sub-linear, specifically: $O(n^\alpha), \alpha < 1$

•Choose bucket size l, *such as:* $l \cdot (1 + 2\lfloor \log|\Sigma| \rfloor) = \alpha \log n \qquad \alpha < 1$

•We get $2^{\alpha \log n} = n^\alpha = o(n)$ → $|S| = O(n^\alpha \cdot \log n \cdot \log\log n)$

ANALYZING OCC( )

- Summing everything:
- NO & W both take $O\left(\frac{n}{\log n}\right)$
- NO' & W' both take $O\left(\frac{n}{\log n} \cdot \log\log n\right)$
- MTF takes $O\left(\frac{n}{\log n}\right)$
- S takes $O\left(n^{\alpha} \cdot \log n \cdot \log\log n\right)$
- Total Size: $|Z| + O\left(\frac{n}{\log n} \cdot \log\log n\right)$ Total Time: $O(1)$



NEXT: LOCATE P IN T

- We now want to retrieve the positions in T of the (Last – First+1) pattern occurrences.
  - **Meaning**: for every *i = first, first+1,…,Last* Find position in T of the suffix which prefixes the i-th row.
  - Denote the above as *pos(i)*

pos(9) = 7

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

P = si

- We can't find pos(i) directly, but we ***can*** do:
- Given row i (9) `s ippi#missi s` , we can find row j (11) `s sippi#miss i` such that pos(j) = pos(i) – 1
- This algorithm is called *backward_step(i)*
  - *Running time O(1)* *Uses previous structures*

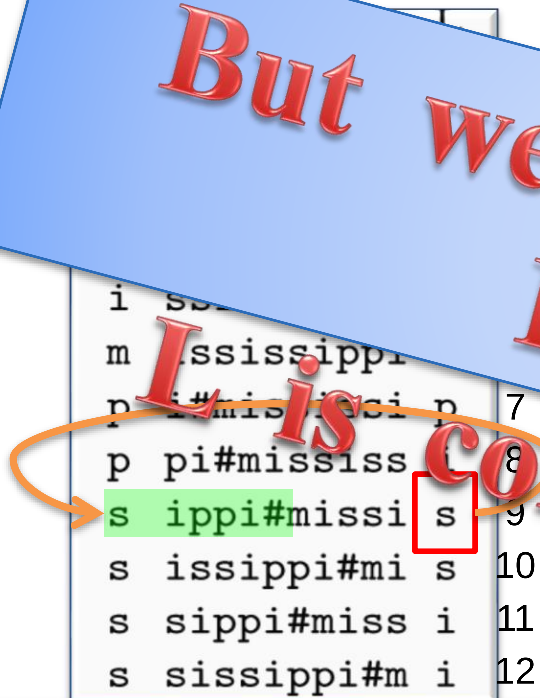
BACKWARD_STEP

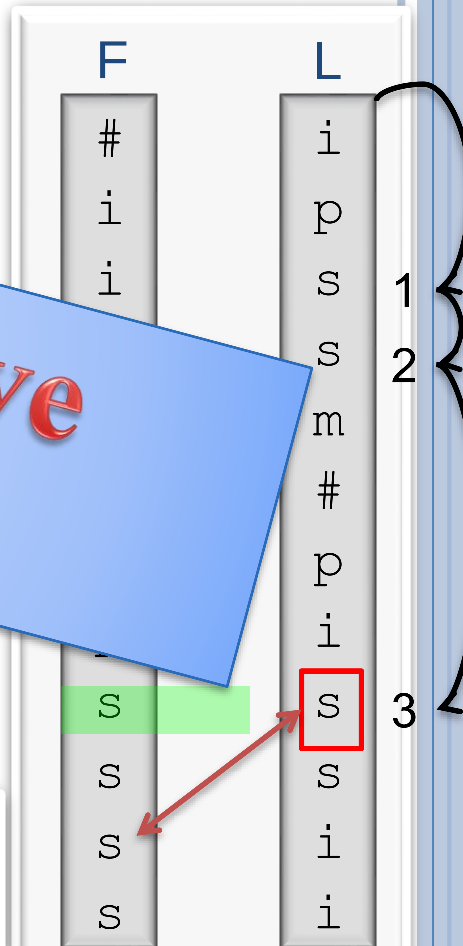
- L[i] precedes F[i] in T.
- All char's appear at the same order in L and F.

- So ideally we would just compute Occ(L[i],i)+C[L[i]]

But we don't have L[i],

L is compressed!





m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

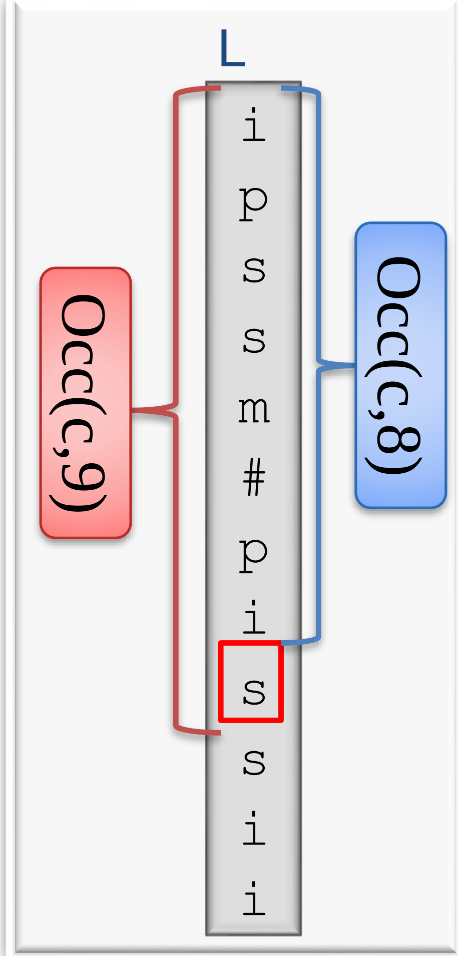
P = si

i = 9

BACKWARD_STEP

- Solution:
  - Compare *Occ(c,i)* to *Occ(c,i-1)* for every $c \in \Sigma \cup \{\#\}$
  - Obviously, will only differ at *c = L[i].*
  - Now we can compute Occ(L[i],i)+C[L[i]].

- Calling Occ() is O(1)
- $|\Sigma| = \Theta(1)$
- Therefore ***backward_step*** takes O(1) time.



P = si

i = 9

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

LOCATE P IN T: PREPROCESSING

- Now we are ready for the final algorithm.
- First, mark every $\lfloor \log^{1+\varepsilon} n \rfloor$ character from T and its corresponding row (suffix) in L.
- For each marked row r_j , store its position Pos(r_j) in data structure **S**.
- For example, querying **S** for Pos(r_3) will return 8.

	F		L	
	#	mississipp	i	1
	i	#mississip	p	2
	i	ppi#missis	s	3
	i	ssippi#mis	s	4
	i	ssissippi#	m	5
	m	ississippi	#	6
	p	i#mississi	p	7
	p	pi#mississ	i	8
first →	s	ippi#missi	s	9
last →	s	issippi#mi	s	10
	s	sippi#miss	i	11
	s	sissippi#m	i	12

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

P = si

i = 9

LOCATE P IN T

- Given row index i, find Pos(i) as follows:
  - If r_i is a marked row, return Pos(i) from **S**. DONE!
  - Otherwise - use *backward_step*(i) to find *i'* such that : Pos(*i'*) = Pos(i) – 1
    - Repeat ***t*** times until we find a marked row.
    - Then – retrieve Pos(i') from **S** and compute Pos(i) by computing: Pos(i') + *t*

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

first → (row 9)

last → (row 10)

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

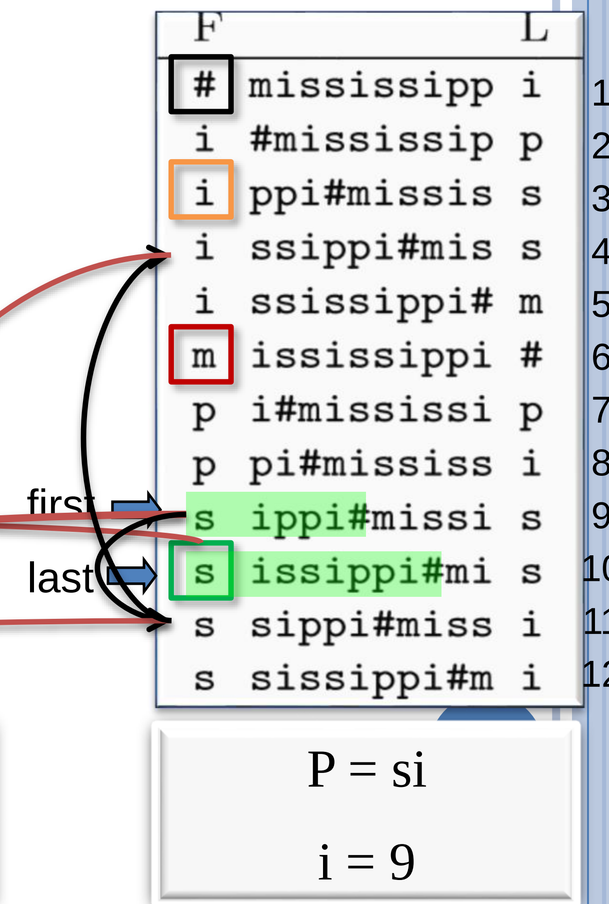
P = si

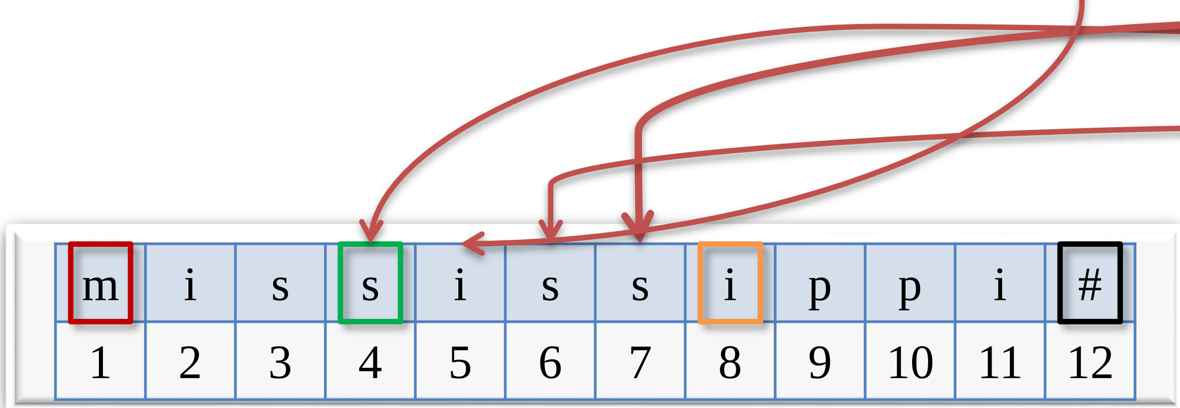
i = 9

LOCATE P IN T

Example – finding "si"

- For i = last = 10
  - r_{10} is marked – get Pos(10) from **S** :
  - *Pos(10) = 4*
- For i = first = 9
  - r_9 isn't marked. → backward_step(9)
  - backward_step(9) = r_{11} (t = 1)
  - r_{11} isn't marked either. → backward_step(11)
  - Backward_step(11) = r_4 (t = 2)



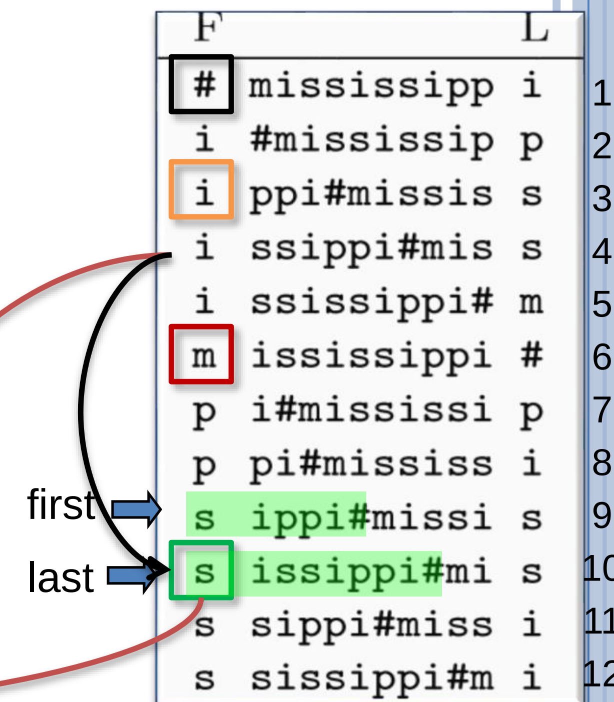


LOCATE P IN T

<u>Example – finding “si”</u>

- For *i* = last = 10
  - r_{10} is marked – get Pos(10) from **S** :
  - <u>*Pos(10) = 4*</u>

- For i = first = 9
  - r_9 isn’t marked. → backward_step(9)
  - backward_step(9) = r_{11} (t = 1)
  - r_{11} isn’t marked either. → backward_step(11)
  - Backward_step(11) = r_4 (t = 2)
  - r_4 isn’t marked either. → backward_step(4)
  - Backward_step(4) = r_{10} (t = 3)
  - r_{10} is marked – get Pos(10) from **S**. Pos(10) = 4
  - <u>Pos(9)</u> = Pos(10) + t = 4 + 3 = <u>7</u>



m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

P = si

i = 9

POS(*l*) ANALYSIS

- A marked row will be found in at most $\left|\log^{1+\varepsilon} n\right|$ iterations.
- Each iteration uses backward_step, which is O(1).
- So finding a single position takes $O\left(\log^{1+\varepsilon} n\right)$
- Finding all *occ* occurrences of P in T takes:

  $$O\left(occ \cdot \log^{1+\varepsilon} n\right)$$

  but only if querying **S** for membership is O(1)!!

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

first → row 9

last → row 10

P = si

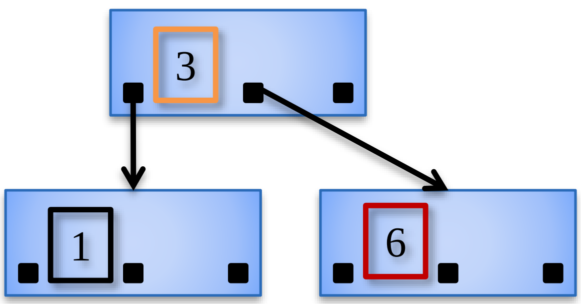
i = 9

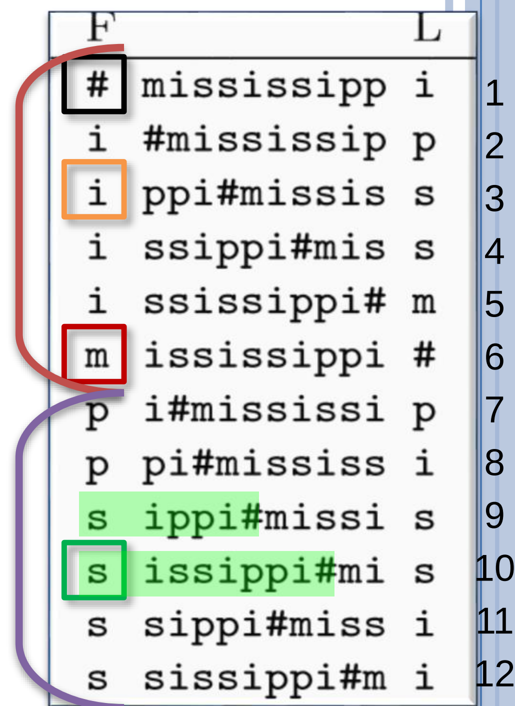
$\left|\log^{1+\varepsilon} n\right|$

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

S ANALYSIS - TIME

- Partition L's rows into buckets of $\Theta\left(\log^2 n\right)$ rows each.
- For each bucket
  - Store all marked rows in a Packed-B-Tree (unique for each row),
  - Using their distance from the beginning of the bucket as the key. (also storing the mapping)





- A tree will contain at most $O\left(\log^2 n\right)$ keys, of size $O\left(\log\left(\log^2 n\right)\right) = O\left(\log\log n\right)$ bits each.

  → O(1) access time

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

P = si

i = 9

S ANALYSIS - SPACE

- The number of marked rows is $O\left(\frac{n}{\log^{1+\varepsilon} n}\right)$
- Each key encoded in a tree takes $O(\log \log n)$ bits, and we need an additional O(logn) bits to keep the Pos(i) value.
- So **S** takes $O\left(\frac{n}{\log^{1+\varepsilon} n}(\log \log n + \log n)\right)$
- The structure we used to count P, uses $|Z| + O\left(\frac{n}{\log n} \cdot \log \log n\right)$ bits, so choose ε between 0 and 1 (because going lower than $O\left(\frac{n}{\log n} \cdot \log \log n\right)$ doesn't reduce the asymptotic space usage.)

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

P = si

i = 9

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12