# Applied Databases

**Lecture 16**
*Suffix Array, Burrows-Wheeler Transform*

Sebastian Maneth

*University of Edinburgh - March 10th, 2016*

# Outline

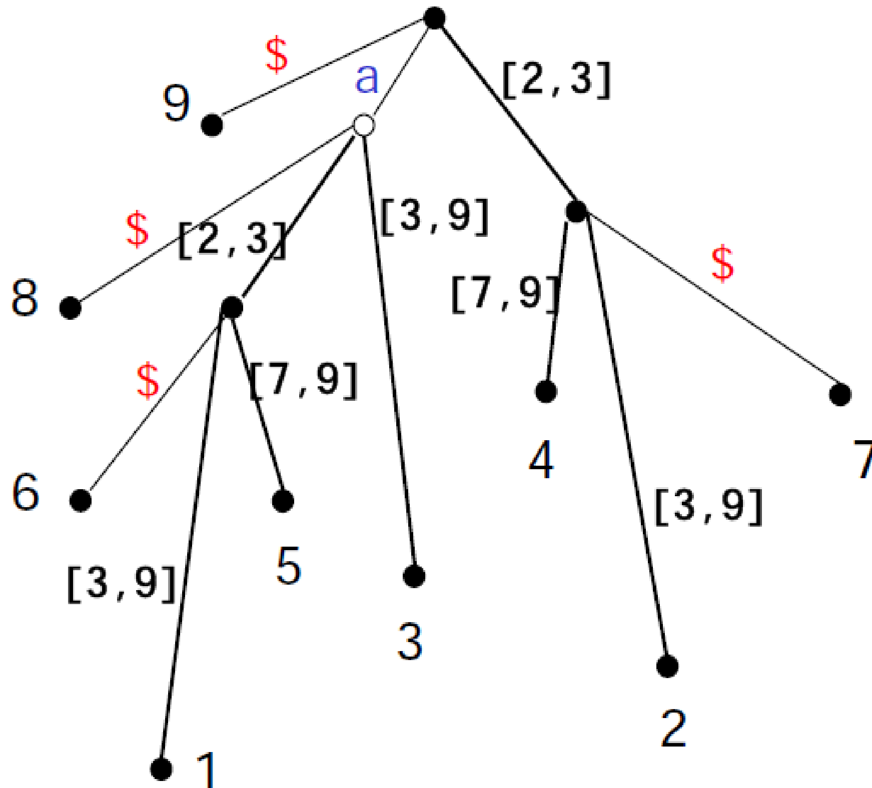
1. Suffix Array
2. Burrows-Wheeler Transform

# Online String-Matching

→ how can we do Horspool on Unicode Texts?

# Suffix Tree

```
    123456789
T = abaababa$
```




→ add end marker “$”

→ one-to-one correspondence of leaves to suffixes

→ a tree with m+1 leaves has <= 2m+1 nodes!

**Lemma**
Size of suffix tree for “T$” is linear in n=|T|, i.e., in O(n).

→ search time still O(|P|), as for suffix trie!
→ perfect data structure for our task!

# Suffix Tree Construction

Good news:
**Suffix tree *can* be constructed in linear time!**

Complex construction algorithms

→ Weiner 1973

→ McCreight 1976

→ Ukkonen 1995

Linear time only for *constant-size alphabets*!
Otherwise, O(n log n)

→ Farach 1997

Linear time for **any integer alphabet**,
drawn from a polynomial range

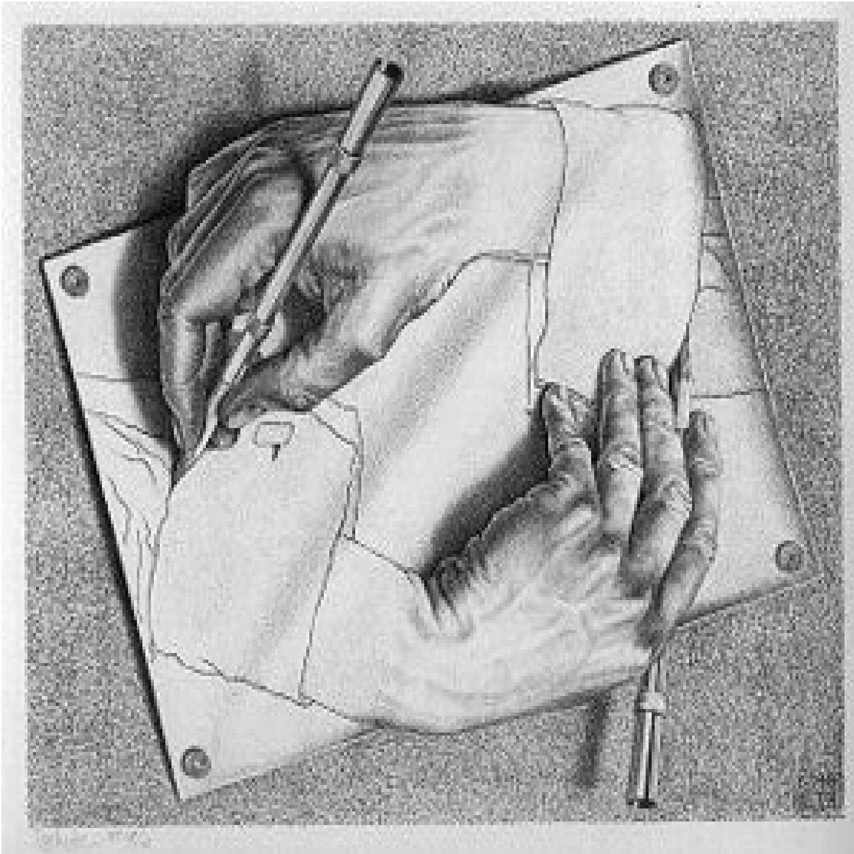
→ again a big breakthrough

# Suffix Tree

Many applications
→ E.g., infinite-window Lempel-Ziv like compression

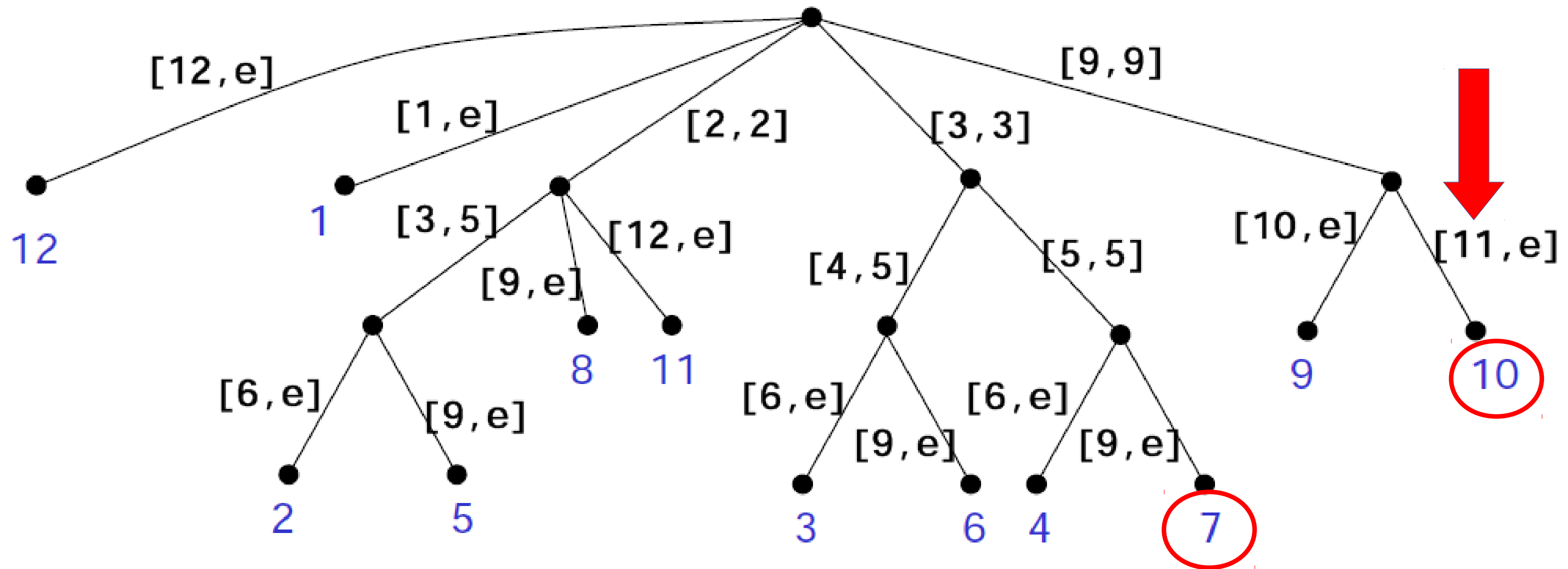
a b a abaa aba baba ab b → a b a (1,4) (1,3) (9,4) (1,2) b

(position, length)



M. C. Escher (1948)

T = mississippi$

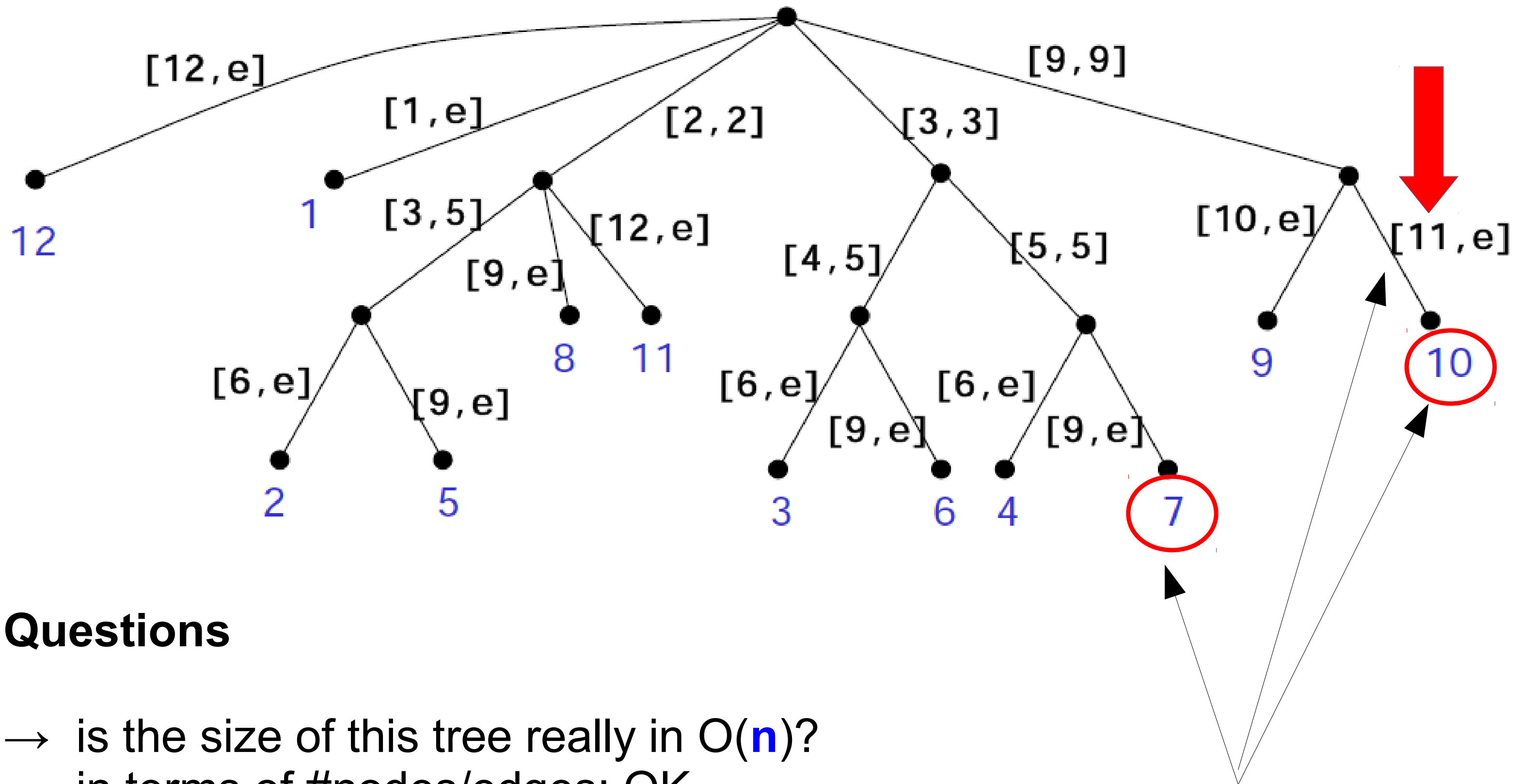



**Questions**

→ is the size of this tree really in O(**n**)?
→ in terms of #nodes/edges: OK

→ how about the sizes of labels ??

T = mississippi$



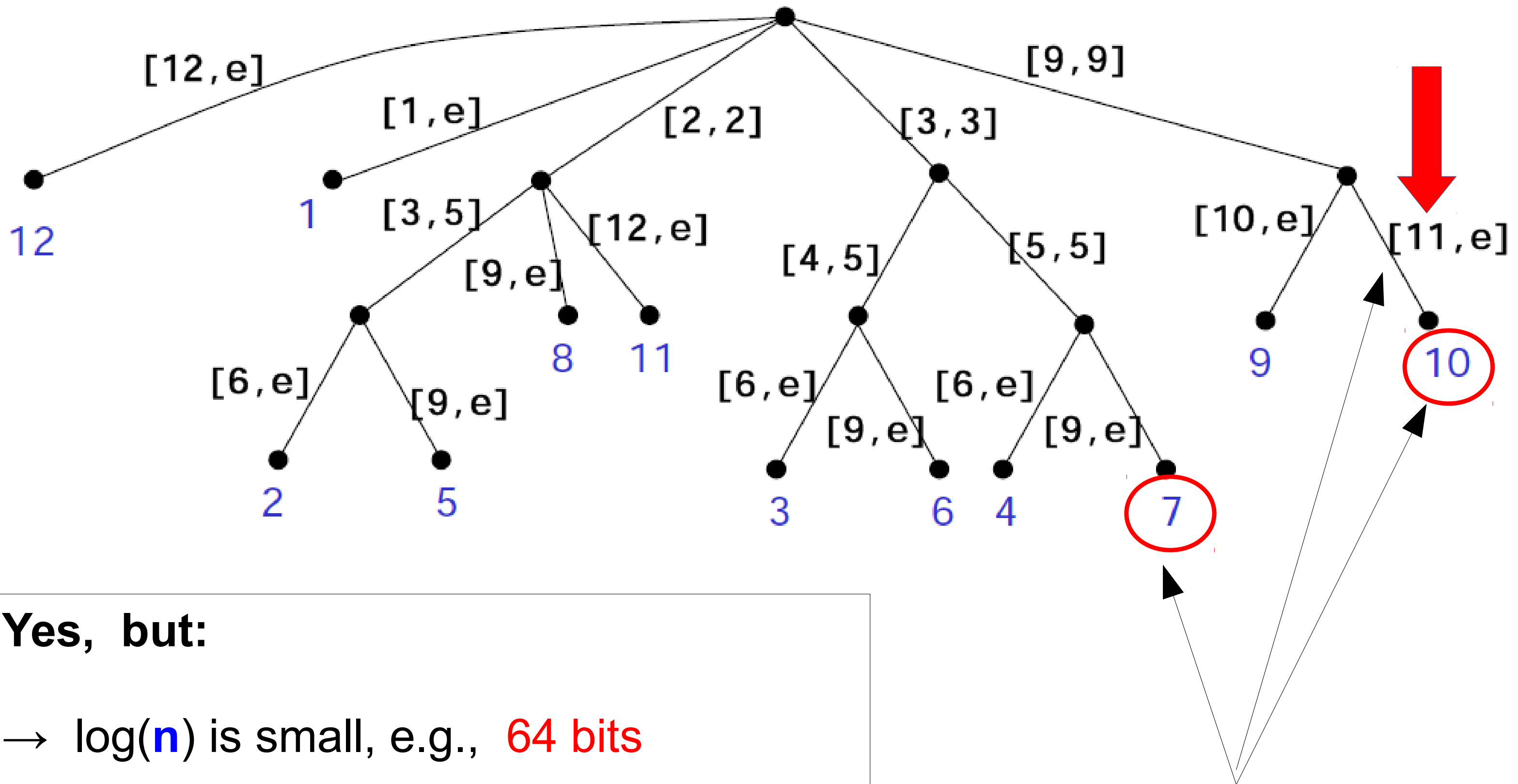

**Questions**

→ is the size of this tree really in O(**n**)?
→ in terms of #nodes/edges: OK

→ how about the sizes of labels ??

each requires **log(n)** bits!?

T = mississippi$



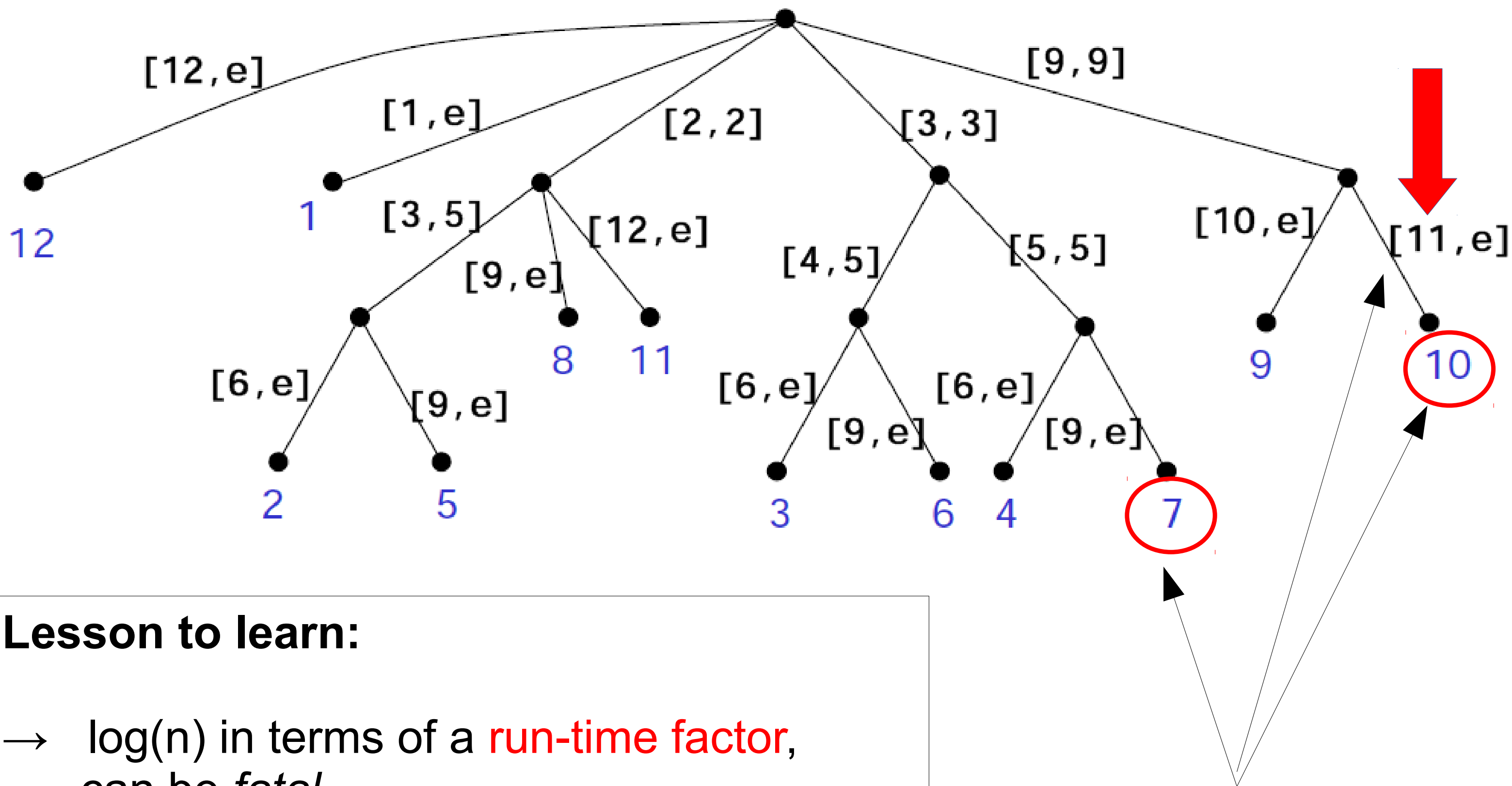

**Yes, but:**

→ log(**n**) is small, e.g., 64 bits

→ can be considered constant!

each requires **log(n)** bits!?

T = mississippi$

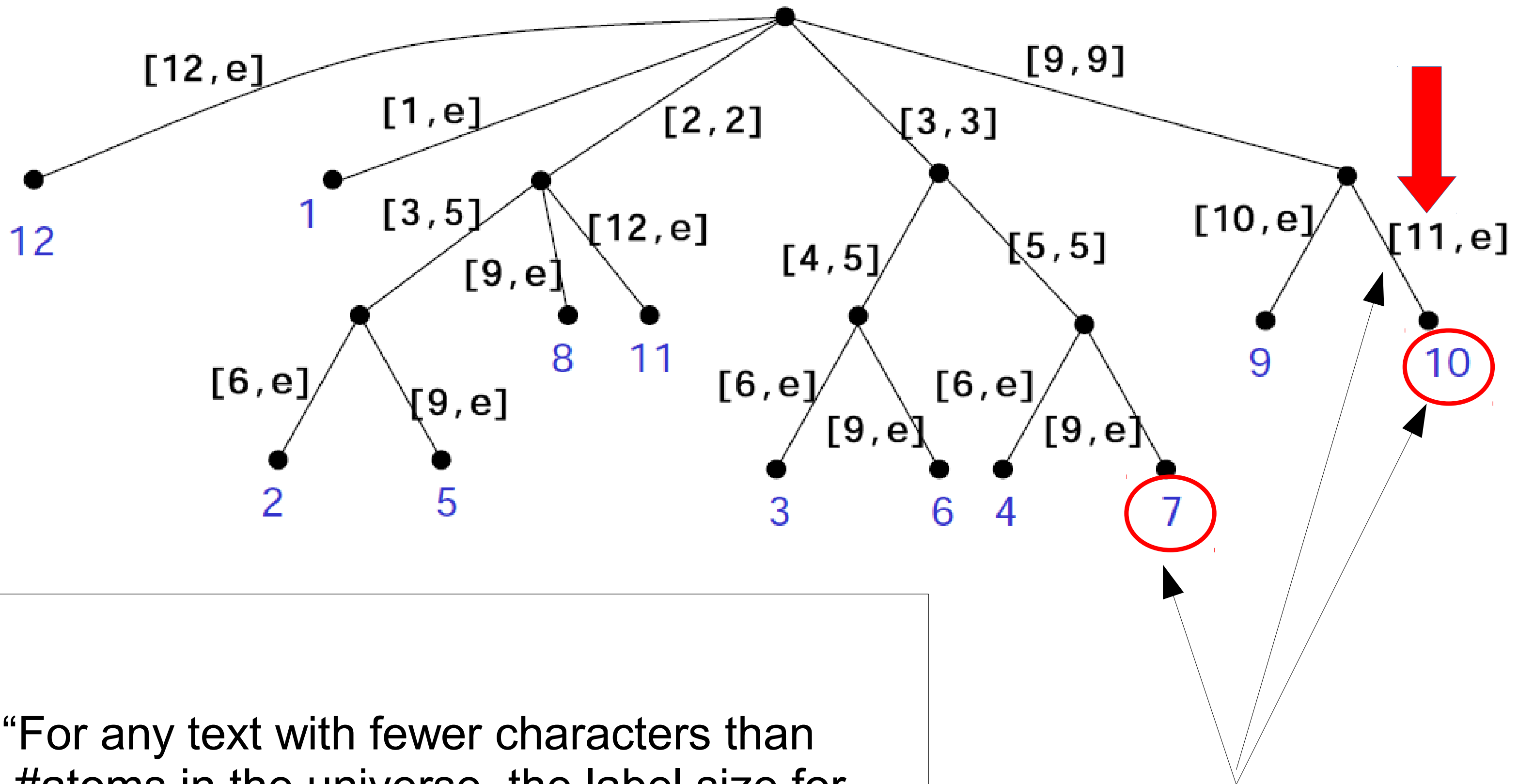



**Lesson to learn:**

→ log(n) in terms of a run-time factor, can be *fatal*

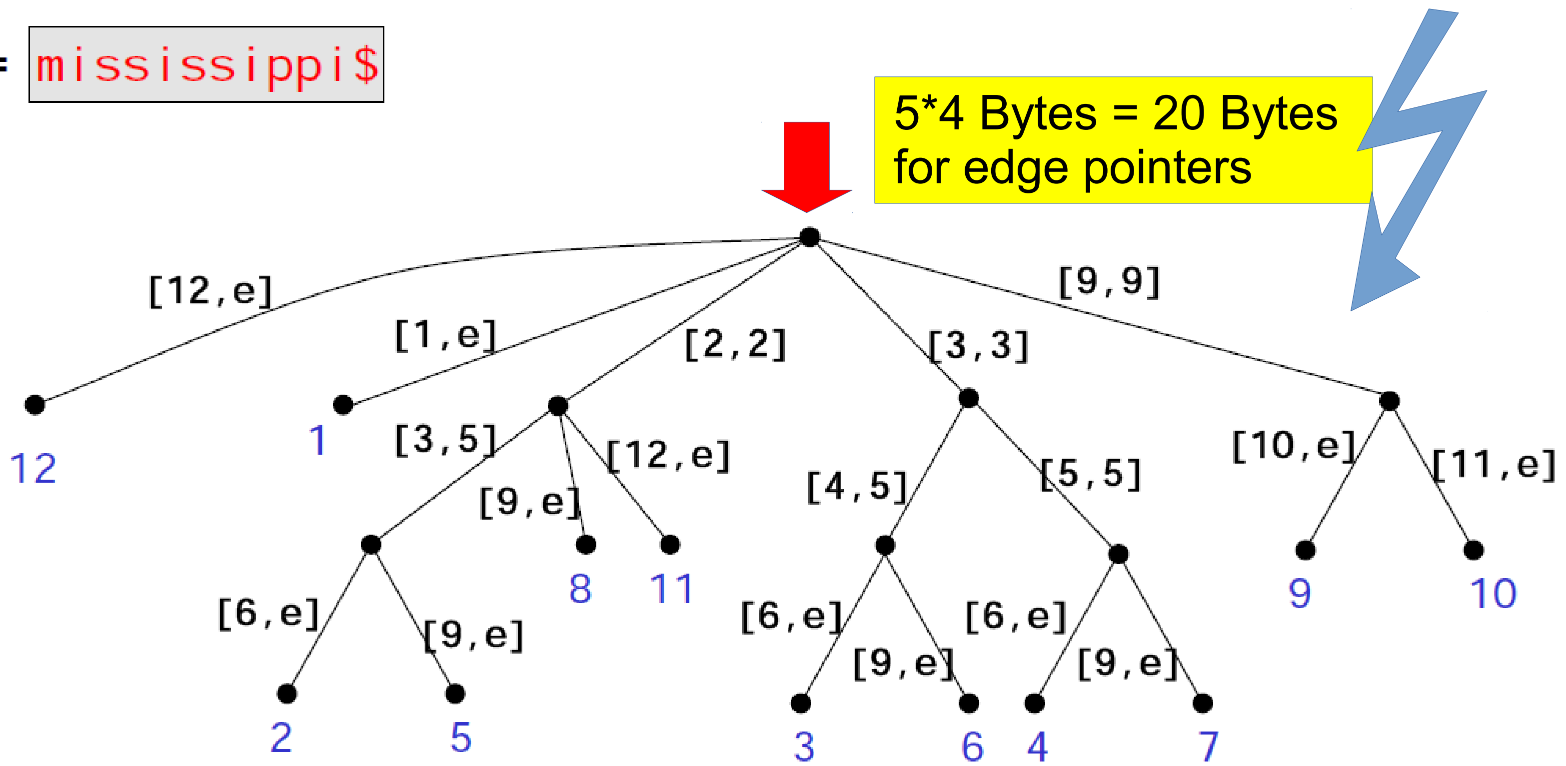
→ in terms of a space-factor, it is *fine*! how long will a text be? 2^64???

**4 / 8 Bytes** each is enough!




"For any text with fewer characters than #atoms in the universe, the label size for the suffix tree is a constant of **x bits**.. "

**x Bits** each is enough!




→ label size is not an issue

→ but, size of edge-pointers?

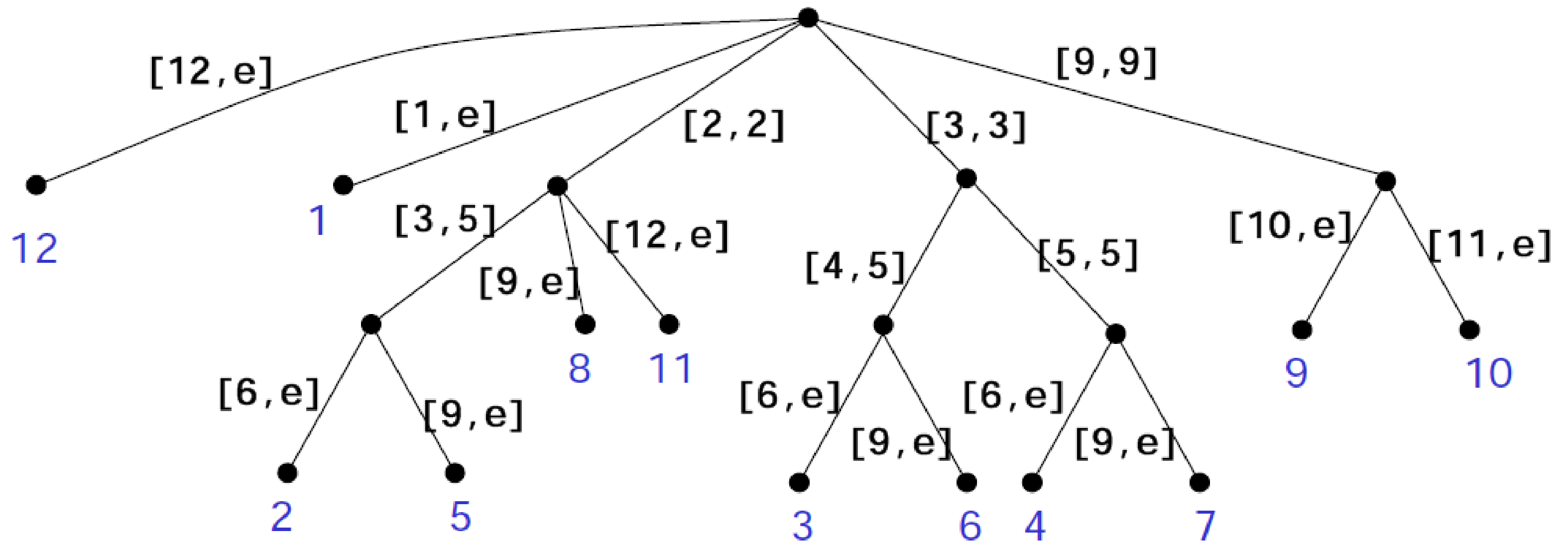
→ imagine each edge requires a 32-bit pointer!!

# Actual Space of Suffix Trees

Space for edge-pointers is problematic:

→ actual space of suffix tree, ca. **20|T|**

→ on commodity hardware, texts of more than 1GB are not doable




→ how to avoid the huge space needed for edges?

# 1. Suffix Array

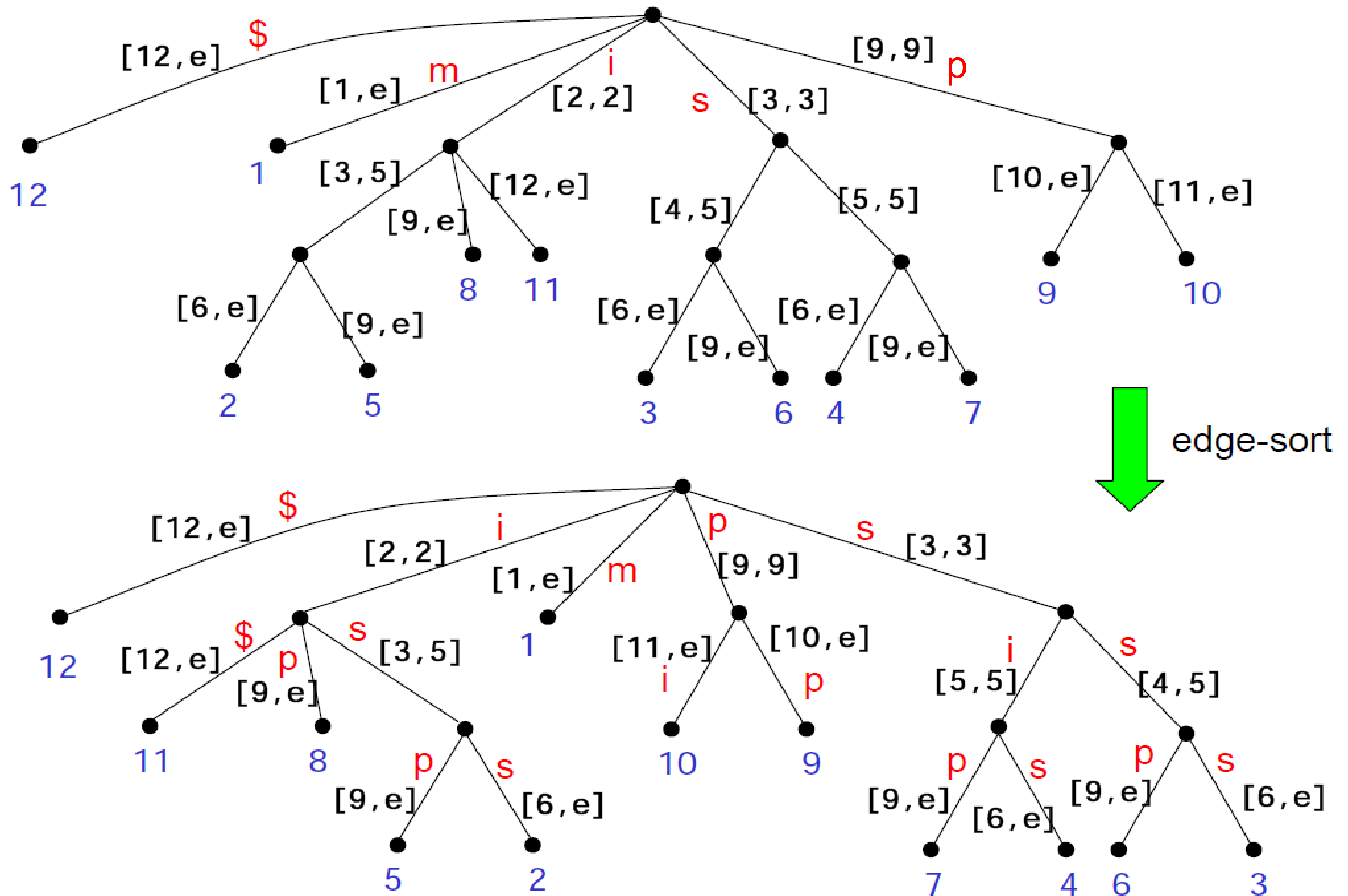
**Definition**

Given text T of length n. For i=1…n, SA[k]=i if suffix T[i…n] is at position k in the lexicographic order T's suffixes.

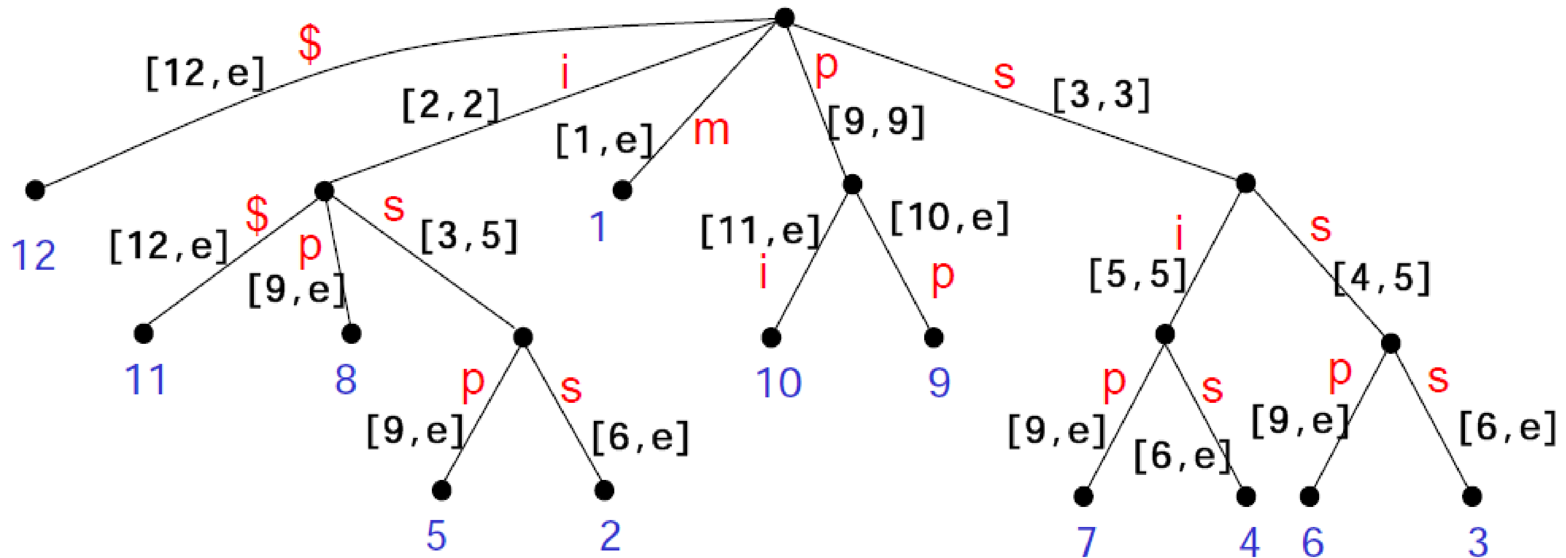
```
      123456789012
T  =  mississippi$        Order  $ < i < m < p < s

  12  $
  11  i$
   8  ippi$
   5  issippi$
   2  ississippi$
   1  mississippi$
  10  pi$
   9  ppi$
   7  sippi$
   4  sissippi$
   6  ssippi$
   3  ssissippi$
```

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

# Suffix Array Construction



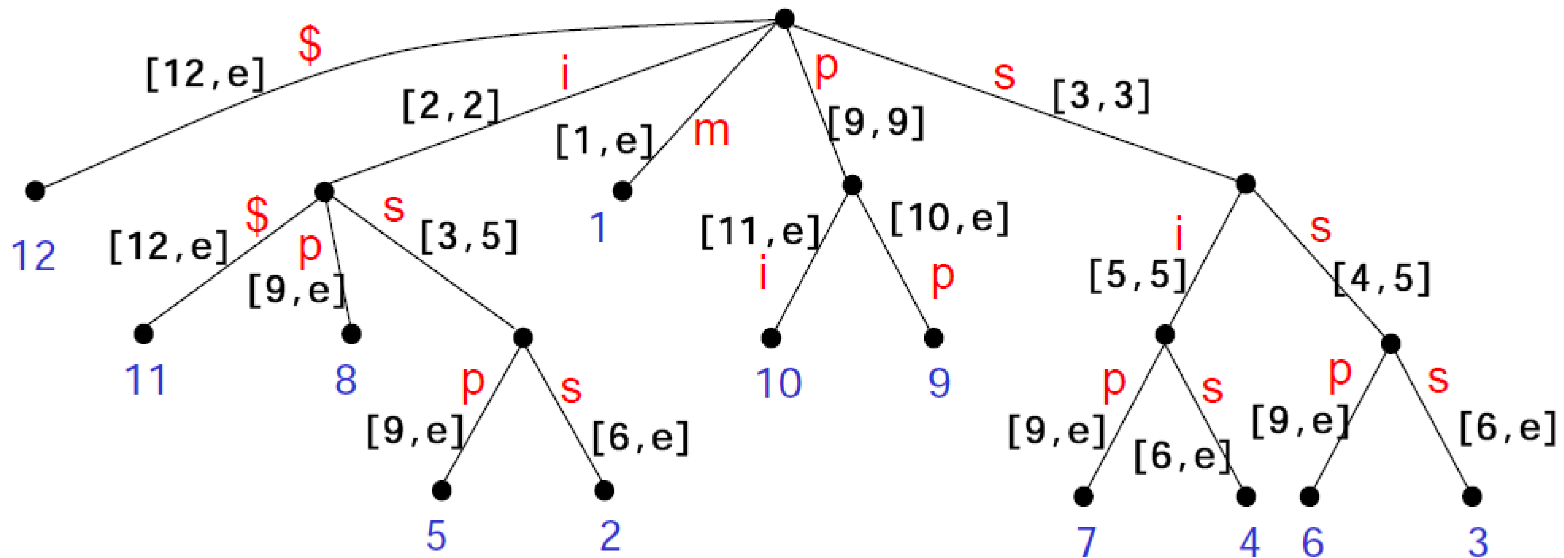
# 1. Suffix Array




→ read leaves from left-to-right!

SA(T) = [ 12,11,8,5,2,10,9,7,4,6,3 ]

# 1. Suffix Array




→ read leaves from left-to-right!

SA(T) = [ 12,11,8,5,2,10,9,7,4,6,3 ]

---

**Theorem**
The suffix array of T can be constructed in time O(|T|).

# Search

**Theorem**
Using binary search on SA(T), all occurrences of P in T can be located in O(|P| * log|T|) time.

---

```
        1234567890
 T  =  mississippi$
```

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

Search for P= issi

all occurren's consecutive in SA!

```
12  $
11  i$
 8  ippi$
 5  issippi$
 2  ississippi$
 1  mississippi$
10  pi$
 9  ppi$
 7  sippi$
 4  sissippi$
 6  ssippi$
 3  ssissippi$
```

Binary search for start-index:
L=1, R=|T|=n
Repeat
  M = $\lceil (L+R-1)/2 \rceil$
  If P $\leq_{lex}$ T[M…M+|P|] then R:=M else L:=M
Until M does not change.

# Search

```
     1234567890
T  = mississippi$
```

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

L

M

issi ≤ miss?

R

| | |
|---|---|
| 12 | $ |
| 11 | i$ |
| 8 | ippi$ |
| 5 | issippi$ |
| 2 | ississippi$ |
| 1 | mississippi$ |
| 10 | pi$ |
| 9 | ppi$ |
| 7 | sippi$ |
| 4 | sissippi$ |
| 6 | ssippi$ |
| 3 | ssissippi$ |

Binary search for start-index:

L=1, R=|T|=n

Repeat

  $M = \lceil (L+R-1)/2 \rceil$

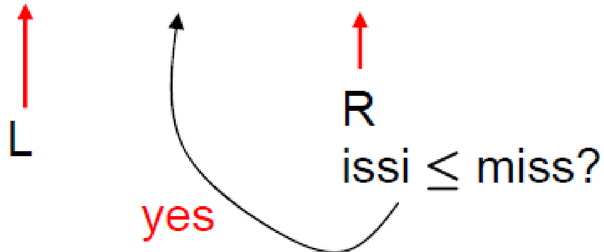
  If P $\leq_{lex}$ T[M…M+|P|] then R:=M else L:=M

Until M does not change.

# Search

```
     1234567890
T = mississippi$
```

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]




issi ≤ ippi?

| | |
|---|---|
| 12 | $ |
| 11 | i$ |
| 8 | ippi$ |
| 5 | issippi$ |
| 2 | ississippi$ |
| 1 | mississippi$ |
| 10 | pi$ |
| 9 | ppi$ |
| 7 | sippi$ |
| 4 | sissippi$ |
| 6 | ssippi$ |
| 3 | ssissippi$ |

Binary search for start-index:
L=1, R=|T|=n
Repeat
  M = $\lceil (L+R-1)/2 \rceil$
  If P $\leq_{lex}$ T[M…M+|P|] then R:=M else L:=M
Until M does not change.

# Search

```
    1234567890
T = mississippi$
```

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

L M R

issi $\le$ issi

| | |
|---|---|
| 12 | $ |
| 11 | i$ |
| 8 | ippi$ |
| 5 | issippi$ |
| 2 | ississippi$ |
| 1 | mississippi$ |
| 10 | pi$ |
| 9 | ppi$ |
| 7 | sippi$ |
| 4 | sissippi$ |
| 6 | ssippi$ |
| 3 | ssissippi$ |

Binary search for start-index:
L=1, R=|T|=n
Repeat
  M = $\lceil (L+R-1)/2 \rceil$
  If P $\le_{lex}$ T[M…M+|P|] then R:=M else L:=M
Until M does not change.

# Search

```
    1234567890
T = mississippi$
```

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

L R

issi ≤ issi

| | |
|---|---|
| 12 | $ |
| 11 | i$ |
| 8 | ippi$ |
| 5 | issippi$ |
| 2 | ississippi$ |
| 1 | mississippi$ |
| 10 | pi$ |
| 9 | ppi$ |
| 7 | sippi$ |
| 4 | sissippi$ |
| 6 | ssippi$ |
| 3 | ssissippi$ |

Binary search for start-index:
L=1, R=|T|=n
Repeat
  M = $\lceil (L+R-1)/2 \rceil$
  If P $\leq_{lex}$ T[M…M+|P|] then R:=M else L:=M
Until M does not change.

# Search

```
     1234567890
T  = mississippi$
```

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

R

issi ≤ issi

M does not change.
➔ Return R

Binary search for start-index:
L=1, R=|T|=n
Repeat
  M = ⌈(L+R-1)/2⌉
  If P $\leq_{lex}$ T[M…M+|P|] then R:=M else L:=M
Until M does not change.

| | |
|---|---|
| 12 | $ |
| 11 | i$ |
| 8 | ippi$ |
| 5 | issippi$ |
| 2 | ississippi$ |
| 1 | missisippi$ |
| 10 | pi$ |
| 9 | ppi$ |
| 7 | sippi$ |
| 4 | sissippi$ |
| 6 | ssippi$ |
| 3 | ssissippi$ |

# Search

Start-index

```
     1234567890
T  = mississippi$

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]
```

```
12 $
11 i$
 8 ippi$
 5 issippi$
 2 ississippi$
 1 mississippi$
10 pi$
 9 ppi$
 7 sippi$
 4 sissippi$
 6 ssippi$
 3 ssissippi$
```

Binary search for end-index:

L=1, R=|T|=n

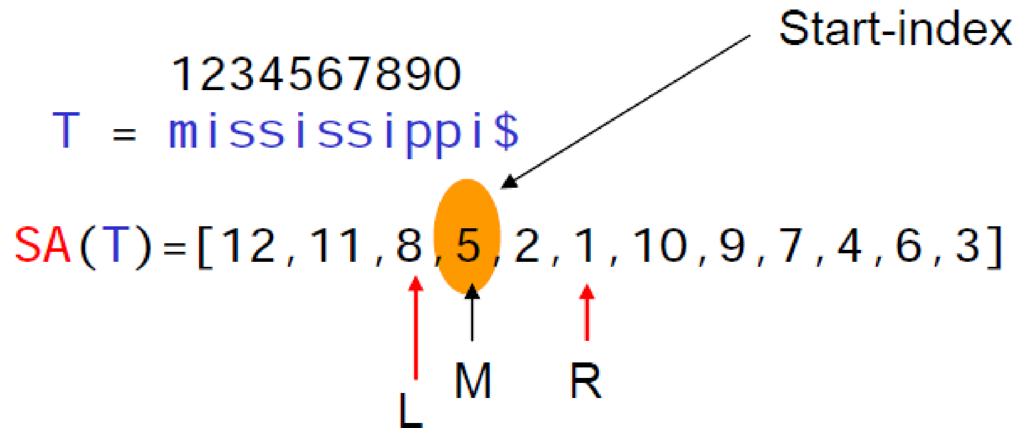
Repeat

   M = $\lceil (L+R-1)/2 \rceil$

   If P $<_{lex}$ T[M…M+|P|] then R:=M else L:=M

Until M does not change.

# Search




not( issi $<$ issi )

➔

12 $
11 i$
8 ippi$
5 issippi$
2 ississippi$
1 mississippi$
10 pi$
9 ppi$
7 sippi$
4 sissippi$
6 ssippi$
3 ssissippi$

Binary search for end-index:

L=1, R=|T|=n

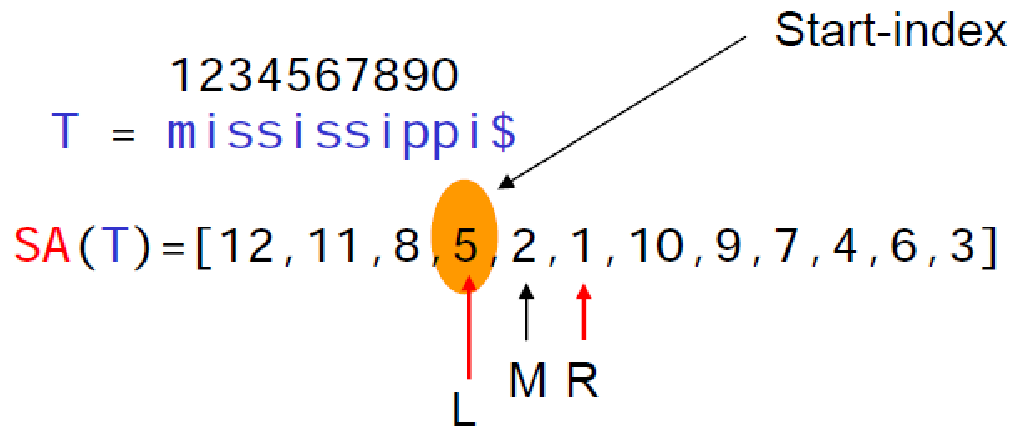
Repeat

  $M = \lceil (L+R-1)/2 \rceil$

  If $P <_{lex} T[M \ldots M+|P|]$ then R:=M else L:=M

Until M does not change.

# Search




not( issi < issi )

➔ L:=M

Binary search for end-index:

L=1, R=|T|=n

Repeat

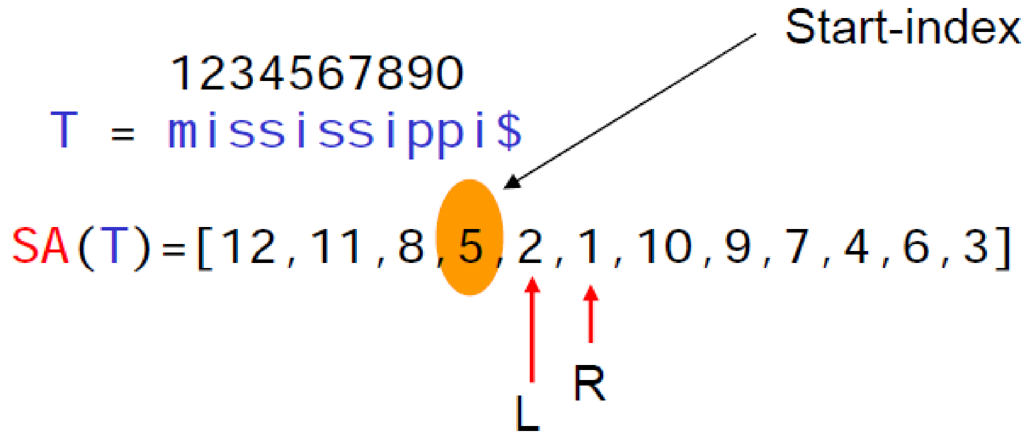
  M = $\lceil (L+R-1)/2 \rceil$

  If P $<_{lex}$ T[M…M+|P|] then R:=M else L:=M

Until M does not change.

12 $
11 i$
8 ippi$
5 issippi$
2 ississippi$
1 mississippi$
10 pi$
9 ppi$
7 sippi$
4 sissippi$
6 ssippi$
3 ssissippi$

# Search

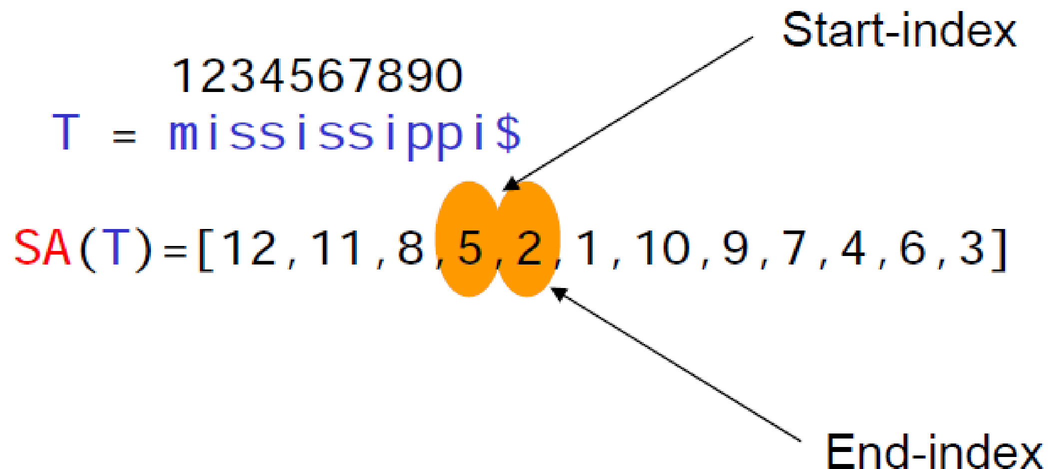



M does not change.
➔ Return L

Binary search for end-index:
L=1, R=|T|=n
Repeat
  M = ⌈(L+R-1)/2⌉
  If P $<_{lex}$ T[M…M+|P|] then R:=M else L:=M
Until M does not change.

```
12 $
11 i$
 8 ippi$
 5 issippi$
 2 ississippi$
 1 mississippi$
10 pi$
 9 ppi$
 7 sippi$
 4 sissippi$
 6 ssippi$
 3 ssissippi$
```

# Search




12 $
11 i$
8 ippi$
5 issippi$
2 ississippi$
1 mississippi$
10 pi$
9 ppi$
7 sippi$
4 sissippi$
6 ssippi$
3 ssissippi$

Binary search for end-index:
L=1, R=|T|=n
Repeat
  $M = \lceil (L+R-1)/2 \rceil$
  If P $<_{lex}$ T[M…M+|P|] then R:=M else L:=M
Until M does not change.

# Search

**Theorem**
Using binary search on SA(T), all occurrences of P in T can be located in $O(|P| * \log|T|)$ time.

---

**Note**
This is a pessimistic bound!
We *almost never* need $O(|P|)$ time for one lexicographic comparison!

On random strings, this should run in $O(|P| + \log|T|)$ time.

# Search

**Theorem**
Using binary search on SA(T), all occurrences of P in T can be located in O(|P| * log|T|) time.

---

**Note**
This is a pessimistic bound!
We *almost never* need O(|P|) time for one lexicographic comparison!

On random strings, this should run in O(|P| + log|T|) time.

→ O(|P| + log|T|) in practise, using a simple trick

→ O(|P| + log|T|) guaranteed, using LCP-array

LCP(k,j) = longest common prefix of T[SA[k]...]
and T[SA[j]...]

# 1. Suffix Arrays

→ much more space efficient than Suffix Tree
→ used in pratise (suffix tree more used in theory)

---

→ Suffix Array Construction, without Suffix Trees?

[ Linear Work Suffix Array Construction,
J. Kärkkäinen, Sanders, Burkhardt,
*Journal of the ACM*, 2006 ]

→ See also:

[ A taxonomy of suffix array construction algorithms,
S. J. Puglisi, W. F. Smyth, A. Turpin,
*ACM Computing Surveys* 39, 2007 ]

# 2. Burrows-Wheeler Transform

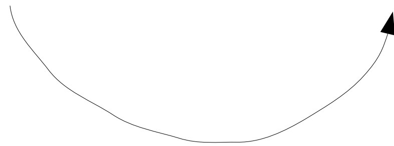
T = banana$

banana$
$banana
a$banan
na$bana
ana$ban
nana$ba
anana$b

# 2. Burrows-Wheeler Transform

T = banana$

| | |
|---|---|
| banana$ | $banana |
| $banana | a$banan |
| a$banan | ana$ban |
| na$bana | anana$b |
| ana$ban | banana$ |
| nana$ba | na$bana |
| anana$b | nana$ba |



sort them
lexicographically

$ < a < b < c < . . .

# 2. Burrows-Wheeler Transform

T = banana$

What information is captured by the **first column**?

| | |
|---|---|
| banana$ | $banana |
| $banana | a$banan |
| a$banan | ana$ban |
| na$bana | anana$b |
| ana$ban | banana$ |
| nana$ba | na$bana |
| anana$b | nana$ba |

sort them lexicographically

$ < a < b < c < . . .

# 2. Burrows-Wheeler Transform

T = banana$

```
banana$        $banana
$banana        a$banan
a$banan        ana$ban
na$bana        anana$b
ana$ban        banana$
nana$ba        na$bana
anana$b        nana$ba
```

sort them lexicographically

What information is captured by the **first column**?

→ sorted #occ of each letter:

– one time "$"
– three times "a"
– one time "b"
– two times "n"

$ < a < b < c < . . .

# 2. Burrows-Wheeler Transform

T = banana$

```
banana$          $banana
$banana          a$banan
a$banan          ana$ban
na$bana          anana$b
ana$ban          banana$
nana$ba          na$bana
anana$b          nana$ba
```

sort them lexicographically

What information is captured by the **first column**?

→ sorted #occ of each letter:

– one time “$”
– three times “a”
– one time “b”
– two times “n”

Can you retrieve the original text T, given only the first column?

$ < a < b < c < . . .

# 2. Burrows-Wheeler Transform

T = banana$

| Rotations | Sorted |
| --- | --- |
| banana$ | $banana |
| $banana | a$banan |
| a$banan | ana$ban |
| na$bana | anana$b |
| ana$ban | banana$ |
| nana$ba | na$bana |
| anana$b | nana$ba |

sort them lexicographically

$ < a < b < c < . . .

What information is captured by the **first column**?

→ sorted #occc of each letter:

– one time "$"
– three times "a"
– one time "b"
– two times "n"

Can you retrieve the original text T, given only the first column?

Of course not!!

# 2. Burrows-Wheeler Transform

T = banana$

| | |
|---|---|
| banana$ | $banana |
| $banana | a$banan |
| a$banan | ana$ban |
| na$bana | anana$b |
| ana$ban | banana$ |
| nana$ba | na$bana |
| anana$b | nana$ba |

sort them lexicographically

$ < a < b < c < . . .

What information is captured by the **first column**?

→ sorted #occ of each letter:

– one time "$"
– three times "a"
– one time "b"
– two times "n"

Can you retrieve the original text T, given only the first column?

**Note**
→ each column contains the same letters ($, 3*a, b, 2*n)

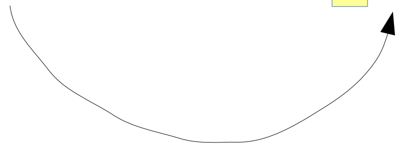
# 2. Burrows-Wheeler Transform

```
T = banana$
```

What information is captured by the **second column**?

→ “sorting with respect to considering one previous letter”

```
banana$          $banana
$banana          a$banan
a$banan          ana$ban
na$bana          anana$b
ana$ban          banana$
nana$ba          na$bana
anana$b          nana$ba
```



sort them lexicographically

$\$ < a < b < c < \ . \ . \ .$

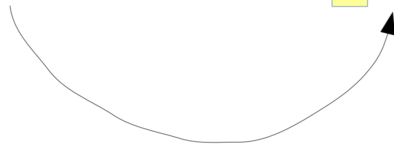
# 2. Burrows-Wheeler Transform

T = banana$

What information is captured by the **second column**?

→ “sorting with respect to considering one previous letter”

| | |
|---|---|
| banana$ | $banana |
| $banana | a$banan |
| a$banan | ana$ban |
| na$bana | anana$b |
| ana$ban | banana$ |
| nana$ba | na$bana |
| anana$b | nana$ba |



sort them lexicographically

Can you retrieve the original T from the second column only?

$ < a < b < c < . . .

# 2. Burrows-Wheeler Transform

T = banana$

| | |
|---|---|
| banana$ | $banana |
| $banana | a$banan |
| a$banan | ana$ban |
| na$bana | anana$b |
| ana$ban | banana$ |
| nana$ba | na$bana |
| anana$b | nana$ba |

sort them lexicographically

$ < a < b < c < . . .

What information is captured by the **second column**?

→ "sorting with respect to considering one previous letter"

Can you retrieve the original T from the second column only?

Can you retrieve the first column, given the second one?

# 2. Burrows-Wheeler Transform

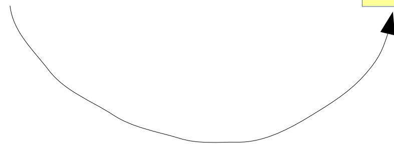
T = banana$

What information is captured by the **third column**?

→ “sorting with respect to considering *two previous letters*”

```
banana$
$banana
a$banan
na$bana
ana$ban
nana$ba
anana$b
```

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```



sort them lexicographically

$ < a < b < c < . . .

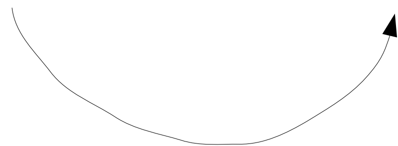
# 2. Burrows-Wheeler Transform

```
T = banana$
```

What information is captured by the **last column**?

→ "sorting with respect to considering ***all*** *previous letters*"

| Rotations | Sorted |
| --- | --- |
| banana$ | $banana |
| $banana | a$banan |
| a$banan | ana$ban |
| na$bana | anana$b |
| ana$ban | banana$ |
| nana$ba | na$bana |
| anana$b | nana$ba |



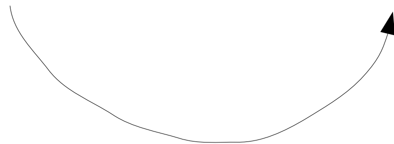
sort them lexicographically

$\$ < a < b < c < \ldots$

# 2. Burrows-Wheeler Transform

T = banana$

| | |
|---|---|
| banana$ | $banana |
| $banana | a$banan |
| a$banan | ana$ban |
| na$bana | anana$b |
| ana$ban | banana$ |
| nana$ba | na$bana |
| anana$b | nana$ba |



sort them lexicographically

$ < a < b < c < . . .

What information is captured by the **last column**?

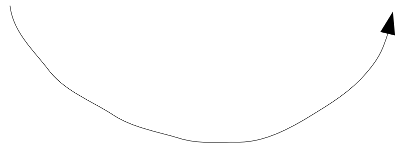
→ "sorting with respect to considering ***all*** *previous letters*"

Can you retrieve the original text T, given only the last column?

# 2. Burrows-Wheeler Transform

```
T = banana$
```

```
banana$        $banana
$banana        a$banan
a$banan        ana$ban
na$bana        anana$b
ana$ban        banana$
nana$ba        na$bana
anana$b        nana$ba
```



sort them
lexicographically

What information is captured
by the **last column**?

→ "sorting with respect to considering
***all*** *previous letters*"

Can you retrieve the original text T,
given only the last column?

**YES, you can!!**

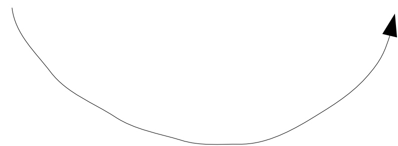
```
$ < a < b < c <   . . .
```

# 2. Burrows-Wheeler Transform

T = banana$

Burrows-Wheeler Transform L
of text T

| | |
|---|---|
| banana$ | $banana |
| $banana | a$banan |
| a$banan | ana$ban |
| na$bana | anana$b |
| ana$ban | banana$ |
| nana$ba | na$bana |
| anana$b | nana$ba |



sort them
lexicographically

$ < a < b < c < . . .

# 2. Burrows-Wheeler Transform

T = banana$

Burrows-Wheeler Transform L of text T

| | |
|---|---|
| banana$ | $banana |
| $banana | a$banan |
| a$banan | ana$ban |
| na$bana | anana$b |
| ana$ban | banana$ |
| nana$ba | na$bana |
| anana$b | nana$ba |

sort them lexicographically

Given last column L, how can we reconstruct the original text T?

$ < a < b < c < . . .

# 2. Burrows-Wheeler Transform

Naively:

a
n
n
b
$
a
a

# 2. Burrows-Wheeler Transform

Naively:

a
n
n
b
$
a
a

sort →

$
a
a
a
b
n
n

← first column!

# 2. Burrows-Wheeler Transform

Naively:

a
n
n
b
$
a
a

sort

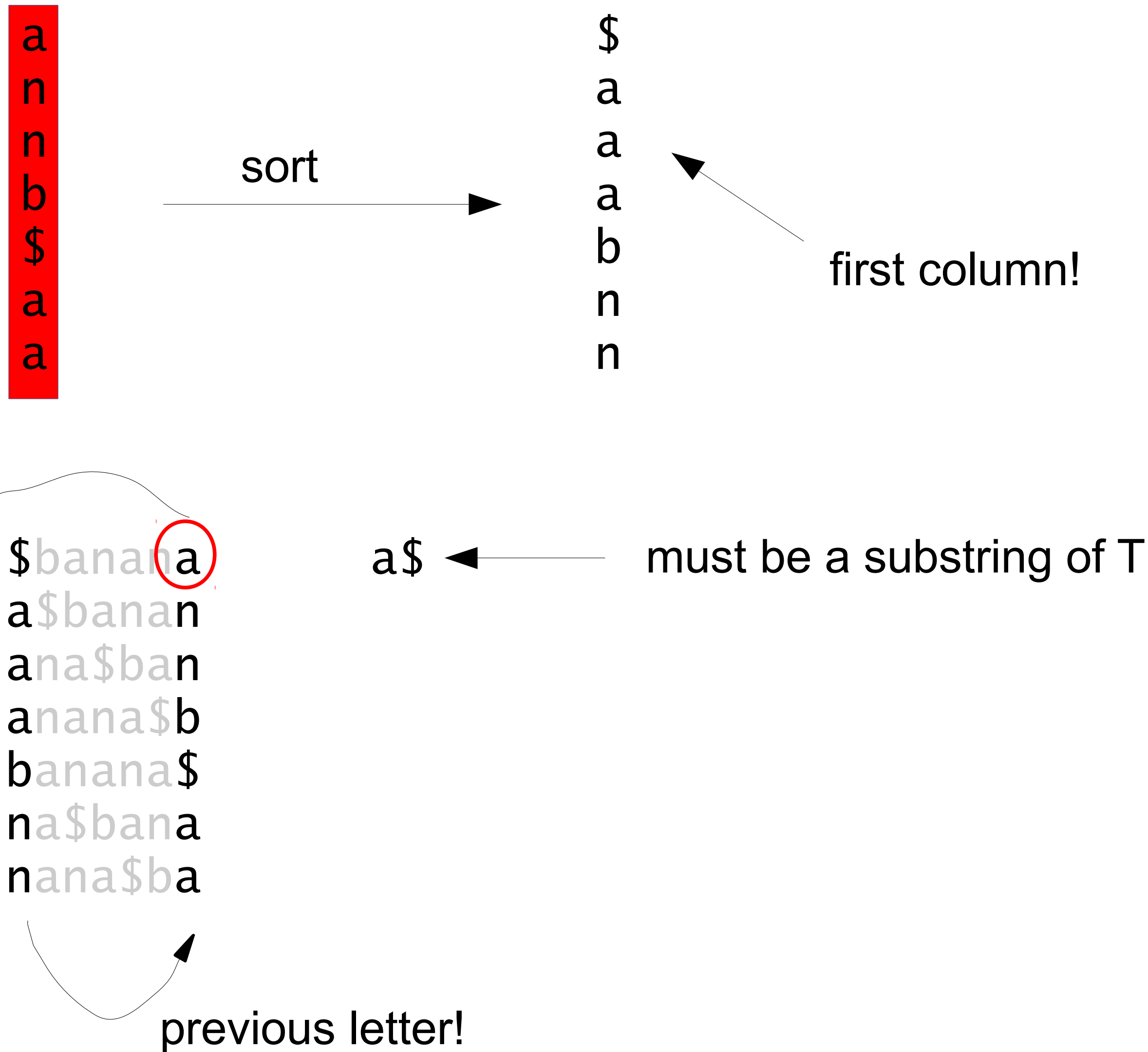
$
a
a
a
b
n
n

first column!

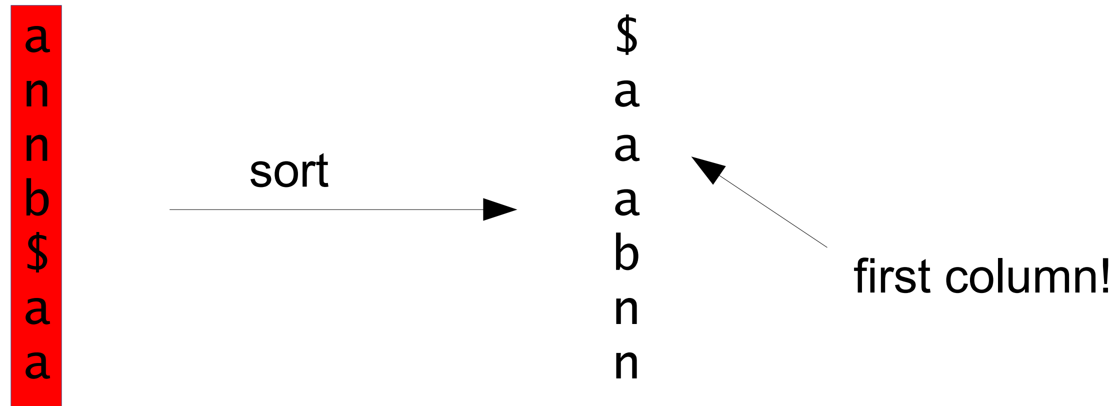
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba

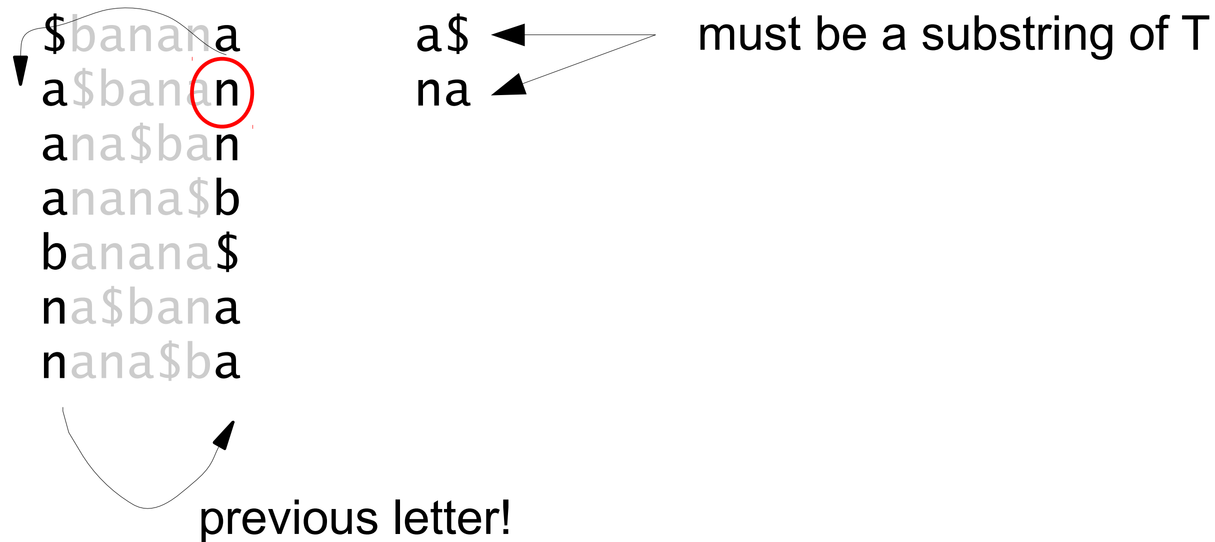
previous letter!

# 2. Burrows-Wheeler Transform



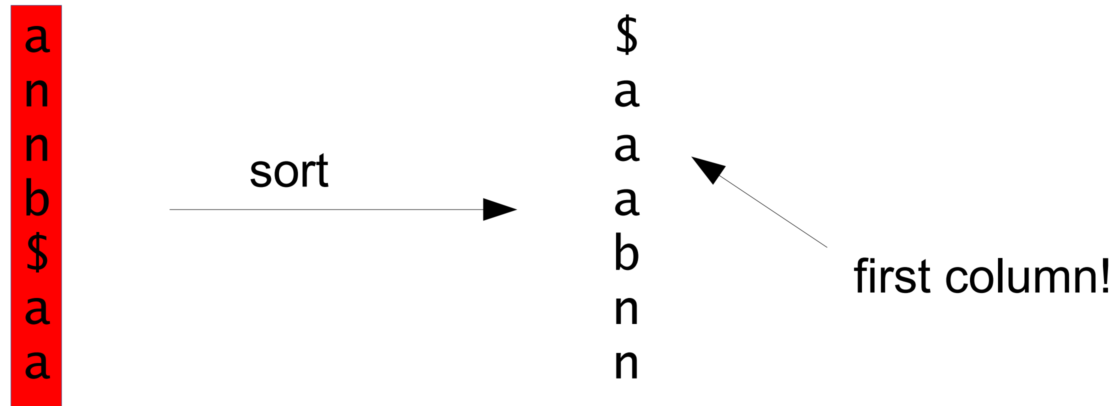
# 2. Burrows-Wheeler Transform

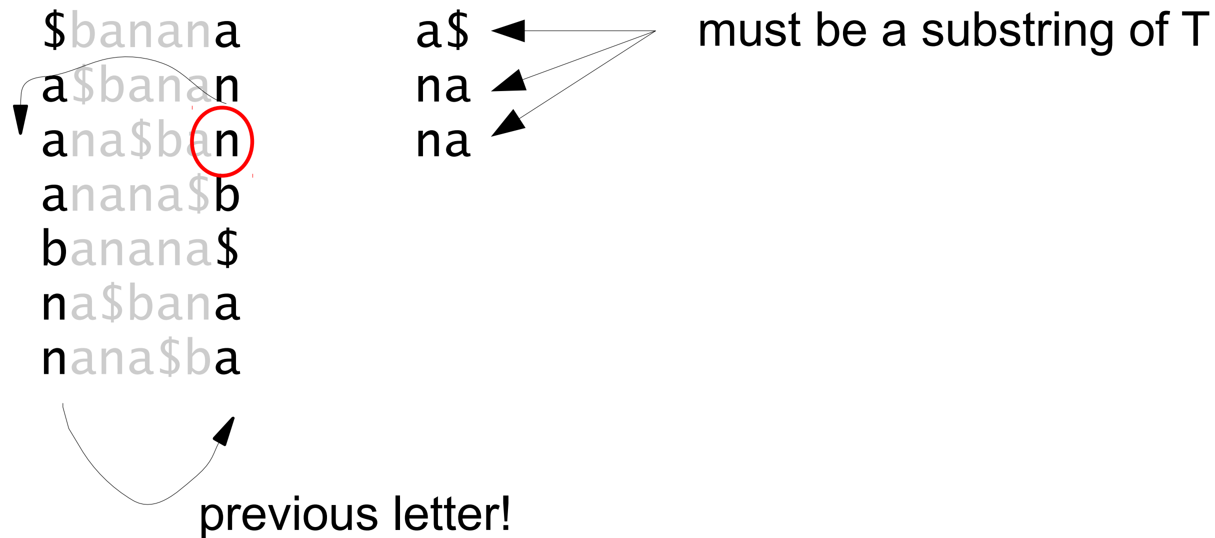


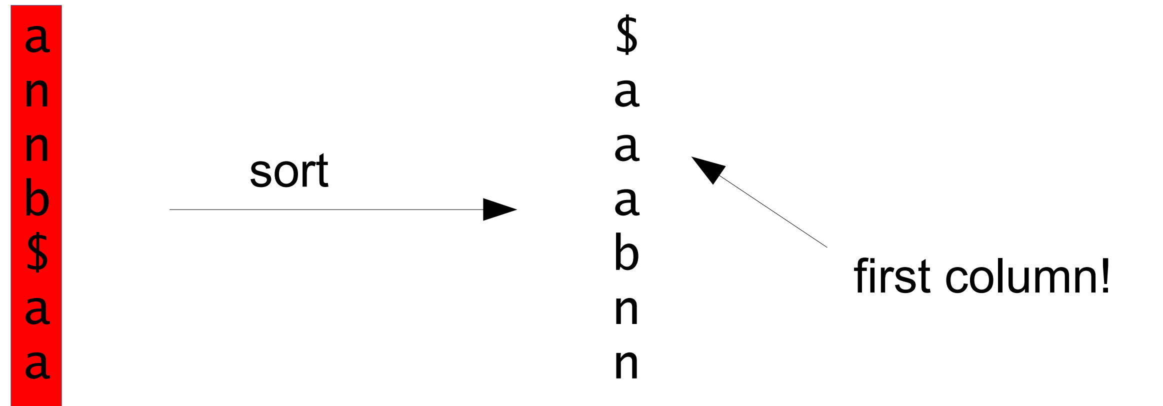
# 2. Burrows-Wheeler Transform

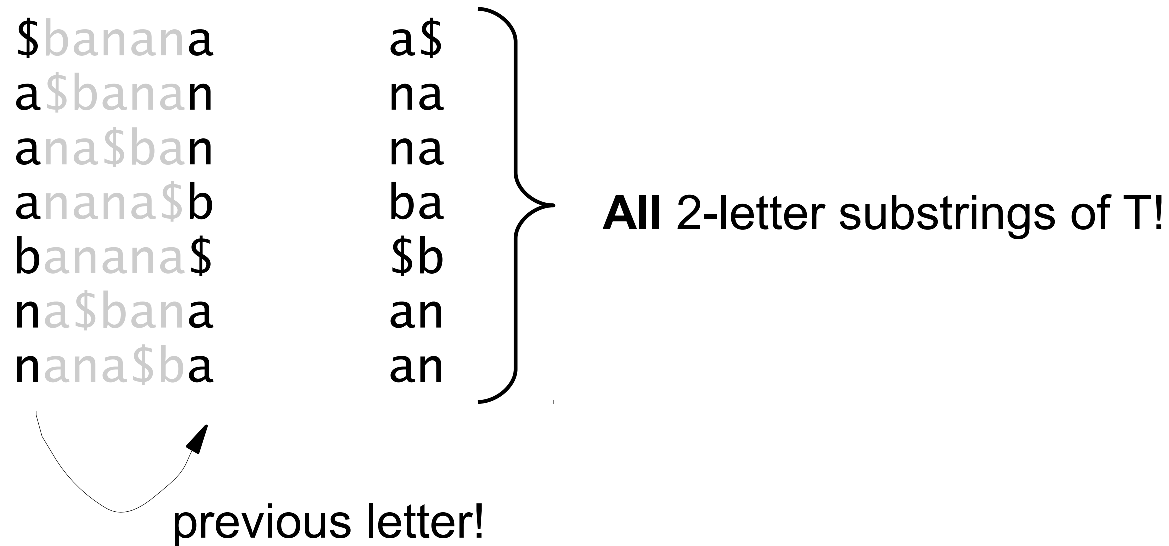
Naively:



# 2. Burrows-Wheeler Transform



# 2. Burrows-Wheeler Transform

Naively:

| | | |
|---|---|---|
| a<br>n<br>n<br>b<br>$<br>a<br>a | sort → | $<br>a<br>a<br>a<br>b<br>n<br>n |

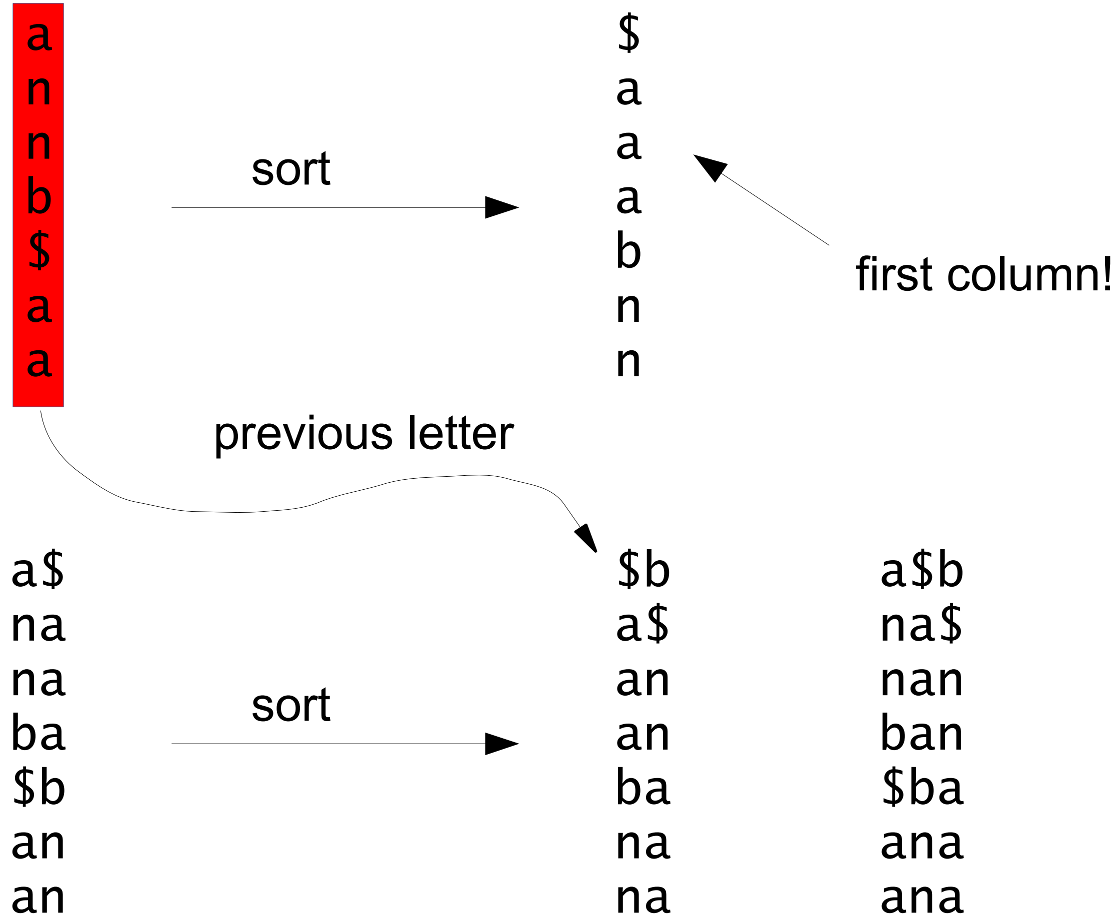
first column!

| | | |
|---|---|---|
| a$<br>na<br>na<br>ba<br>$b<br>an<br>an | sort → | $b<br>a$<br>an<br>an<br>ba<br>na<br>na |

columns 1 and 2.

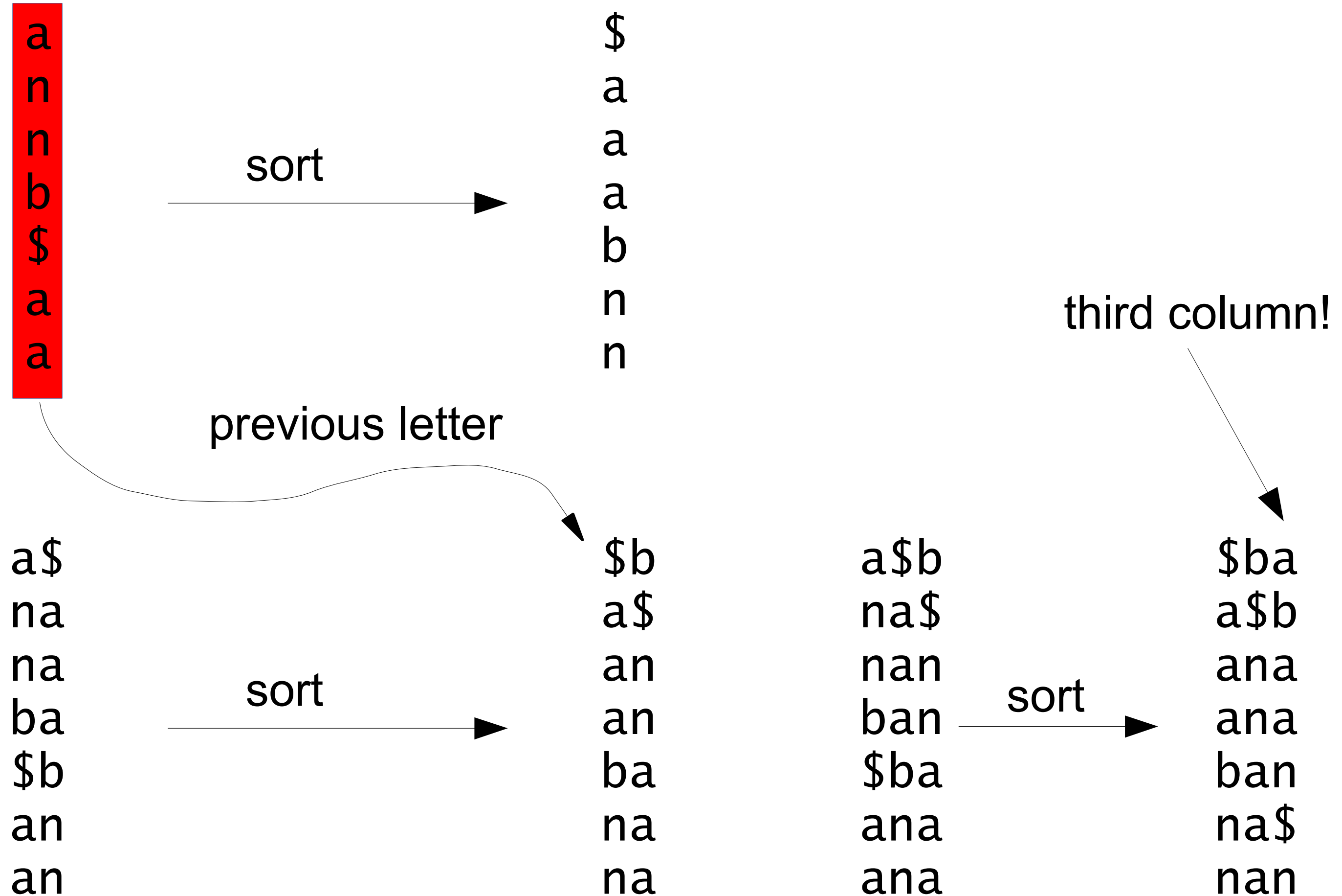
# 2. Burrows-Wheeler Transform

Naively:

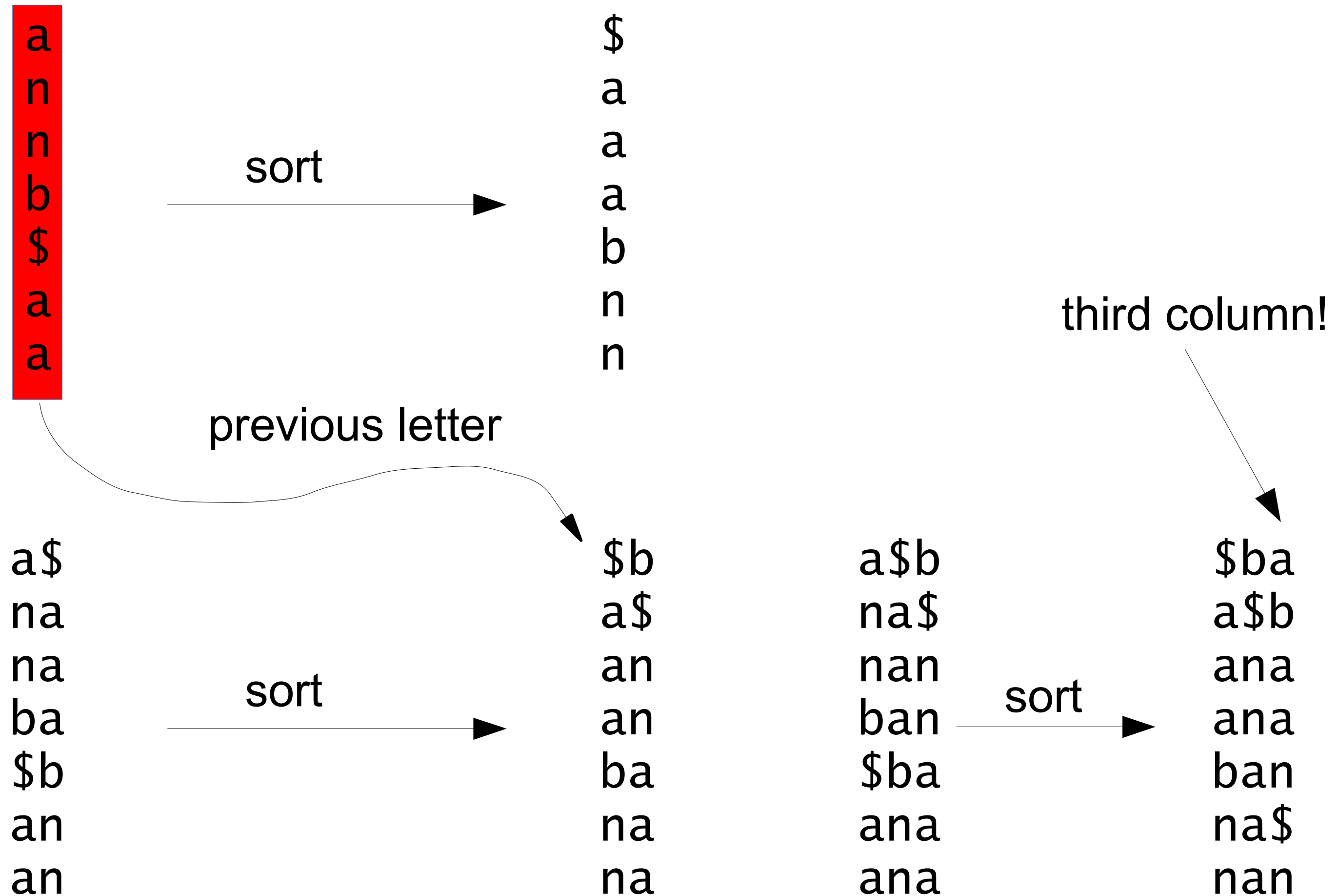


# 2. Burrows-Wheeler Transform

Naively:



# 2. Burrows-Wheeler Transform




Et cetera

# 2. Burrows-Wheeler Transform

Naive method: very expensive! (many sortings)

$rank_b$( L, k ) = #occ of b in L, up to position k (excluding k)

```
     1 2 3 4 5 6 7
L  = a n n b $ a a
```

$rank_n$( L, 4 ) = 2

# 2. Burrows-Wheeler Transform

Naive method: very expensive! (many sortings)

$rank_b(L, k)$ = #occc of b in L, up to position k (excluding k)

```
      1 2 3 4 5 6 7
L  =  a n n b $ a a
```

$rank_n(L, 4) = 2$

$rank_n(L, 3) = 1$

# 2. Burrows-Wheeler Transform

Naive method: very expensive! (many sortings)

$rank_b$( L, k ) = #occ of b in L, up to position k (excluding k)

```
    1 2 3 4 5 6 7
L = a n n b $ a a
```

$rank_n$( L, 4 ) = 2

$rank_n$( L, 3 ) = 1

$rank_a$( L, 7 ) = 2

**O(log |S|) time**
(after linear time preprocessing of L)

# 2. Burrows-Wheeler Transform

1$
2a
 a
 a
5b
6n
 n

$rank_b(L, k)$ = #occ of b in L, up to position k-1

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| L = | a | n | n | b | $ | a | a |

| | $ | a | b | m |
|---|---|---|---|---|
| C | 1 | 2 | 5 | 6 |

first line starting with the letter

Last-to-Front Mapping

$LF(k) = C[L[k]] + rank_{L[k]}(L, k)$

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

second "a", coming from the top

where is the second "a" from top, in the first column?

# 2. Burrows-Wheeler Transform

1$
2a
a
a
5b
6n
n

$rank_b(L, k)$ = #occ of b in L, up to position k-1

```
     1 2 3 4 5 6 7
L = a n n b $ a a
```

```
  $ a b m
C 1 2 5 6
```

first line starting with the letter

Last-to-Front Mapping

$LF(k) = C[L[k]] + rank_{L[k]}(L, k)$

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

second "a", coming from the top
where is the second "a" from top, in the first column?

$LF(6) = C[L[6]] + rank_a(L, 6)$ = C["a"] + 1 = 2 + 1 = 3

# Decoding

$rank_b$( L, k ) = #occ of b in L, up to position k-1

```
     1 2 3 4 5 6 7           $ a b m
L  = a n n b $ a a         C 1 2 5 6
```

first line starting with the letter

Last-to-Front Mapping

LF(k)=C[L[k]] + $rank_{L[k]}$( L, k )

previous letter!

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

→ start with $ (position 5)

→ LF(5) = 1

→ L[1] = a [current decoding: "a$"]

# Decoding

$rank_b(L, k)$ = #occ of b in L, up to position k-1

```
    1 2 3 4 5 6 7           $ a b m
L = a n n b $ a a         C 1 2 5 6
```

first line starting with the letter

Last-to-Front Mapping

$LF(k) = C[L[k]] + rank_{L[k]}(L, k)$

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

→ start with $ (position 5)
→ LF(5) = 1
→ L[1] = a [current decoding: "a$"]

→ LF(1) = 2
→ L[2] = n [ "ba$" ]

# Decoding

$rank_b(L, k)$ = #occ of b in L, up to position k-1

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| L = | a | n | n | b | $ | a | a |

|   | $ | a | b | m |
|---|---|---|---|---|
| C | 1 | 2 | 5 | 6 |

first line starting with the letter

Last-to-Front Mapping

$LF(k) = C[L[k]] + rank_{L[k]}(L, k)$

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

→ start with $ (position 5)
→ LF(5) = 1
→ L[1] = a [current decoding: "a$"]

→ LF(1) = 2
→ L[2] = n [ "ba$" ]

→ LF(2) = 6
→ L(6) = "a" [ "ana$" ]

# Decoding

$rank_b( L, k )$ = #occc of b in L, up to position k-1

```
      1 2 3 4 5 6 7            $ a b m
L  =  a n n b $ a a          C 1 2 5 6
```

first line starting with the letter

Last-to-Front Mapping

$LF(k) = C[L[k]] + rank_{L[k]}( L, k )$

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

→ start with $ (position 5)
→ LF(5) = 1
→ L[1] = a [current decoding: "a$"]

→ LF(1) = 2, L[2] = n [ "ba$" ]
→ LF(2) = 6, L(6) = "a" [ "ana$" ]
→ LF(6) = 3, L(3) = "n" [ "nana$" ]

# Decoding

$rank_b(L, k)$ = #occc of b in L, up to position k-1

```
      1 2 3 4 5 6 7          $ a b m
L  =  a n n b $ a a        C 1 2 5 6
```

first line starting with the letter

Last-to-Front Mapping

$LF(k) = C[L[k]] + rank_{L[k]}(L, k)$

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

→ start with $ (position 5)
→ LF(5) = 1
→ L[1] = a [current decoding: "a$"]

→ LF(1) = 2, L[2] = n [ "ba$" ]
→ LF(2) = 6, L(6) = "a" [ "ana$" ]
→ LF(6) = 3, L(3) = "n" [ "nana$" ]
→ LF(3) = 7, L(7) = "a" [ "anana$" ]

# Decoding

$rank_b(L, k)$ = #occ of b in L, up to position k-1

```
     1 2 3 4 5 6 7           $ a b m
L  = a n n b $ a a         C 1 2 5 6
```

first line starting with the letter

Last-to-Front Mapping

$LF(k) = C[L[k]] + rank_{L[k]}(L, k)$

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

→ start with $ (position 5)
→ LF(5) = 1
→ L[1] = a [current decoding: "a$"]

→ LF(1) = 2, L[2] = n [ "ba$" ]
→ LF(2) = 6, L(6) = "a" [ "ana$" ]
→ LF(6) = 3, L(3) = "n" [ "nana$" ]
→ LF(3) = 7, L(7) = "a" [ "anana$" ]
→ LF(7) = 4, L(4) = ""b" [ "banana$" ]

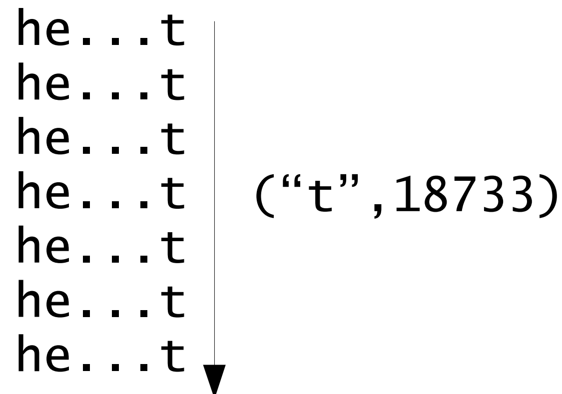
# 2. Burrows-Wheeler Transform

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

What is special about the BTW?

→ has many repeating characters (WHY?)
→ can be run-length compressed!

Imagine the word “the” appears many times in a text.




→ main motivation
→ used in “bzip2” compressor

# 2. Burrows-Wheeler Transform

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

What is special about the BTW?

→ efficient backward search!

→ counting #occ's of pattern P in O(|P| log |S|) time!

# Backward Search on BWT

T = banana$

Burrows-Wheeler Transform L of text T

| | |
|---|---|
| banana$ | $banana |
| $banana | a$banan |
| a$banan | ana$ban |
| na$bana | anana$b |
| ana$ban | banana$ |
| nana$ba | na$bana |
| anana$b | nana$ba |

| C | $ | a | b | n |
|---|---|---|---|---|
| | 1 | 2 | 5 | 6 |

123
P = ana
[sp,ep] = [2,4]

**Backward search for** Pattern P[1]..P[m]

→ Initial range: [sp,ep] with sp=C[P[m]] and ep=C[P[m]+1]-1
Then [s,e] with

$s = C[P[i]] + rank_{L[i]}(L, sp-1)$

$e = C[P[i]] + rank_{L[i]}(L, ep) - 1$

# Backward Search on BWT

T = banana$

Burrows-Wheeler Transform L of text T

| | | L |
|---|---|---|
| banana$ | $banan | a |
| $banana | a$bana | n |
| a$banan | ana$ba | n |
| na$bana | anana$ | b |
| ana$ban | banana | $ |
| nana$ba | na$ban | a |
| anana$b | nana$b | a |

| C | $ | a | b | n |
|---|---|---|---|---|
| | 1 | 2 | 5 | 6 |

123

P = ana

[sp,ep] = [2,4]

s=C["n"] + $rank_n$(L,1)

= 6 + 0 = 6

e= 6 + $rank_n$(L,4) - 1

= 6 + 2 – 1 = 7

**Backward search** for Pattern P[1]..P[m]

$$s = C[P[i]] + rank_{L[i]}(L, sp-1)$$

$$e = C[P[i]] + rank_{L[i]}(L, ep) - 1$$

# Backward Search on BWT

T = banana$

Burrows-Wheeler Transform L of text T

| | |
|---|---|
| banana$ | $banana |
| $banana | a$banan |
| a$banan | ana$ban |
| na$bana | anana$b |
| ana$ban | banana$ |
| nana$ba | na$bana |
| anana$b | nana$ba |

| C | $ | a | b | n |
|---|---|---|---|---|
| | 1 | 2 | 5 | 6 |

123

P = ana

[sp,ep] = [2,4]

sp=6

ep=7

s=C["a"] + $rank_a$(L,5)
= 2 + 1 = 3

e = 1 + rank_a(L,7) =
2 + 3 – 1 = 4

**Backward search** for Pattern P[1]..P[m]

$s = C[P[i]] + rank_{L[i]}(L, sp-1)$

$e = C[P[i]] + rank_{L[i]}(L, ep) - 1$

**Done!**
[3,4]=final range
→ 2 Occs of "ana"

# BWT Construction

→ use the suffix array SA(T)!

→ BWT[ k ] = T[ SA[ k ] – 1 ]   (assuming T[0]=$)

# BWT Construction

→ use the suffix array SA(T)!

→ BWT[ k ] = T[ SA[ k ] – 1 ] (assuming T[0]=$)

e.g.

```
   1234567
SA[banana$] = [7,6,4,2,1,5,3]

T[6] T[5] T[3] T[1] T[0] T[4] T[2]
  a    n    n    b    $    a    a
```

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

# BWT Construction

→ use the suffix array SA(T)!

→ BWT[ k ] = T[ SA[ k ] – 1 ]  (assuming T[0]=$)

→ explain why this equation is correct!

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

# END
# Lecture 16