Applied Databases

Lecture 16
Suffix Array, Burrows-Wheeler Transform

Sebastian Maneth

University of Edinburgh - March 16th, 2017

Outline

1. Suffix Array
2. Burrows-Wheeler Transform

Outline

1. Suffix Array

2. Burrows-Wheeler Transform

Lecture 17: XPath
Lecture 18: XSLT

Lecture 19: Recap I
Lecture 20: Recap I

Lecture 21: guest lecture “NULLs considered harmful” (April-3)
(to be confirmed)

Lecture 22: no lecture! (April-6)

1. Suffix Array

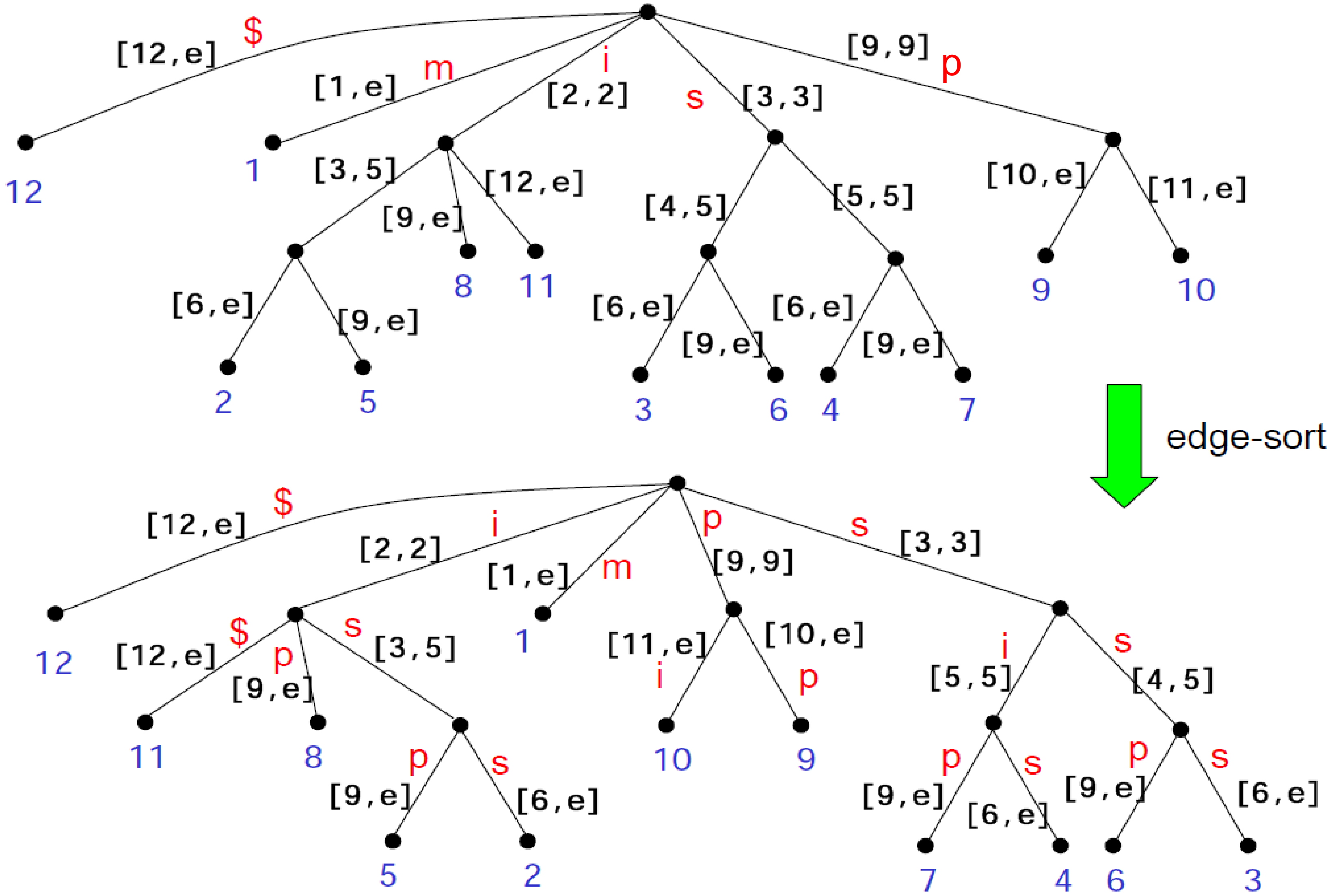
Definition

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.

```
     1234567890
T  = mississippi$        Order  $ < i < m < p < s

   12 $
   11 i$
    8 ippi$                SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]
    5 issippi$
    2 ississippi$
    1 mississippi$
   10 pi$
    9 ppi$
    7 sippi$
    4 sissippi$
    6 ssippi$
    3 ssissippi$
```

Suffix Array Construction



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

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.

12 $
11 i$
M1 → 8 ippi$
5 issippi$
2 ississippi$
M2 → 1 mississippi$
10 pi$
9 ppi$
7 sippi$
4 sissippi$
6 ssippi$
3 ssissippi$

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]...]

History [wikipedea]
The LCP array was introduced in 1993, by Udi Manber and Gene Myers alongside the suffix array in order to improve the running time of their string search algorithm.
Gene Myers later became the vice president of Informatics Research at Celera Genomics, and Udi Manber the vice president of engineering at Google.

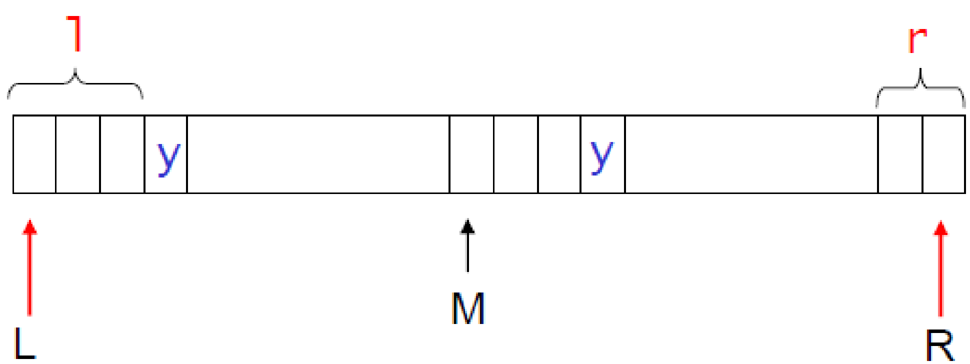
Case 1: $l = r$ then proceed as original algorithm (start at $m+1$)

Case 2: $l \neq r$ (wlog, assume that $l > r$).

(a) LCP(L,M) > l, then let L:=M. ➔ no comparisons!

(b) LCP(L,M) < l, then let R:=M and r:=LCP(L,M). ➔ no comparisons!

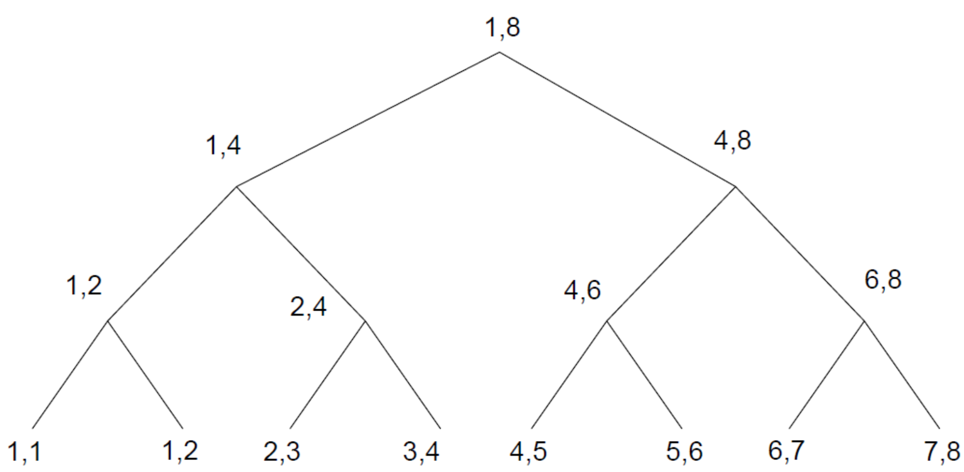
(c) LCP(L,M) = l, then start comparing from $l+1$.



→ during binary search, `l` and `r` never decrease.
→ when a character is examined, starts at `max(l,r)`
→ If k characters are examined, then `max(l,r)` increases by k-1

→ max(l,r) character may have been checked before, but next Character in P has not! ➔ one $\leq$ 1 redundant check per iteration
➔ $\leq \log_2|T|$ redundant checks in total

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



Suffix Arrays

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

→ Suffix Array Construction, without Suffix Trees?

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

→ See also:

[A taxonomy of suffix array construction algorithms,
Puglisi, Smyth, Turpin,
ACM Computing Surveys 39, 2007]

linked from course web page

2. Burrows-Wheeler Transform

T = banana$

generate all
cyclic shifts of T

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$

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?

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”

= sorting of all two-letter substrings!

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

sort them lexicographically

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

```
$ < 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 **second column**?

→ "sorting with respect to considering one previous letter"

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

$ < 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?

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

```
$ < 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 **third column**?

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

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

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

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

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

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

2. Burrows-Wheeler Transform

T = banana$

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

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

sort them lexicographically

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

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

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

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

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

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

Why is this useful?

E.g., if T contains many occ's of “the”, then BWT[T] contains many consecutive “t” letters.

→ BWT[T] is easily compressible by simple run-length code

→ this is the idea behind bzip2

$ < 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

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

a$ ← must be a substring of T

previous letter!

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

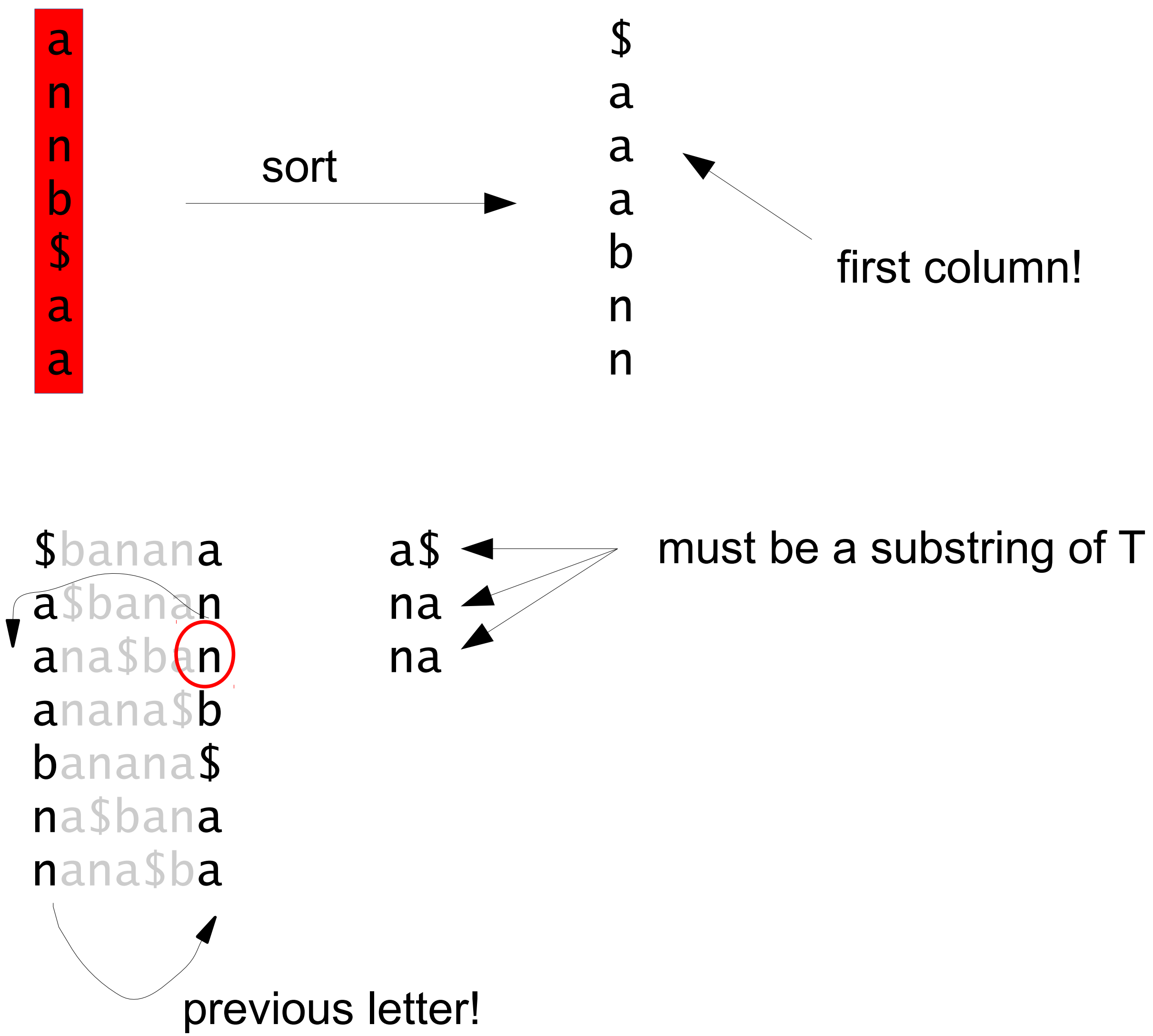
previous letter!

a$
na

← must be a substring of T

2. Burrows-Wheeler Transform

Naively:



2. Burrows-Wheeler Transform

Naively:

a		$
n		a
n	sort →	a
b		a
$		b
a		n
a		n

← first column!

$banana	a$	
a$banan	na	
ana$ban	na	
anana$b	ba	**All** 2-letter substrings of T!
banana$	$b	
na$bana	an	
nana$ba	an	

previous letter!

2. Burrows-Wheeler Transform

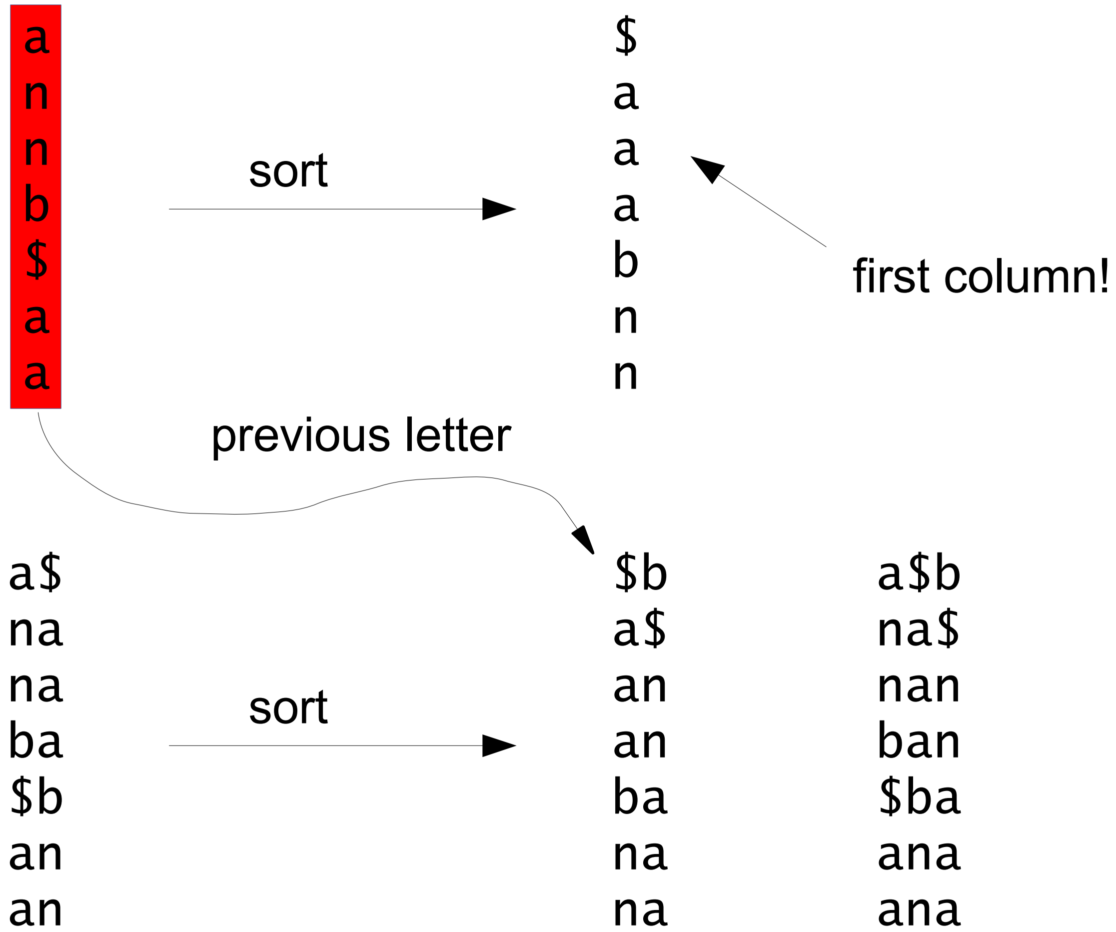
Naively:

a	sort →	$	
n		a	
n		a	← first column!
b		a	
$		b	
a		n	
a		n	

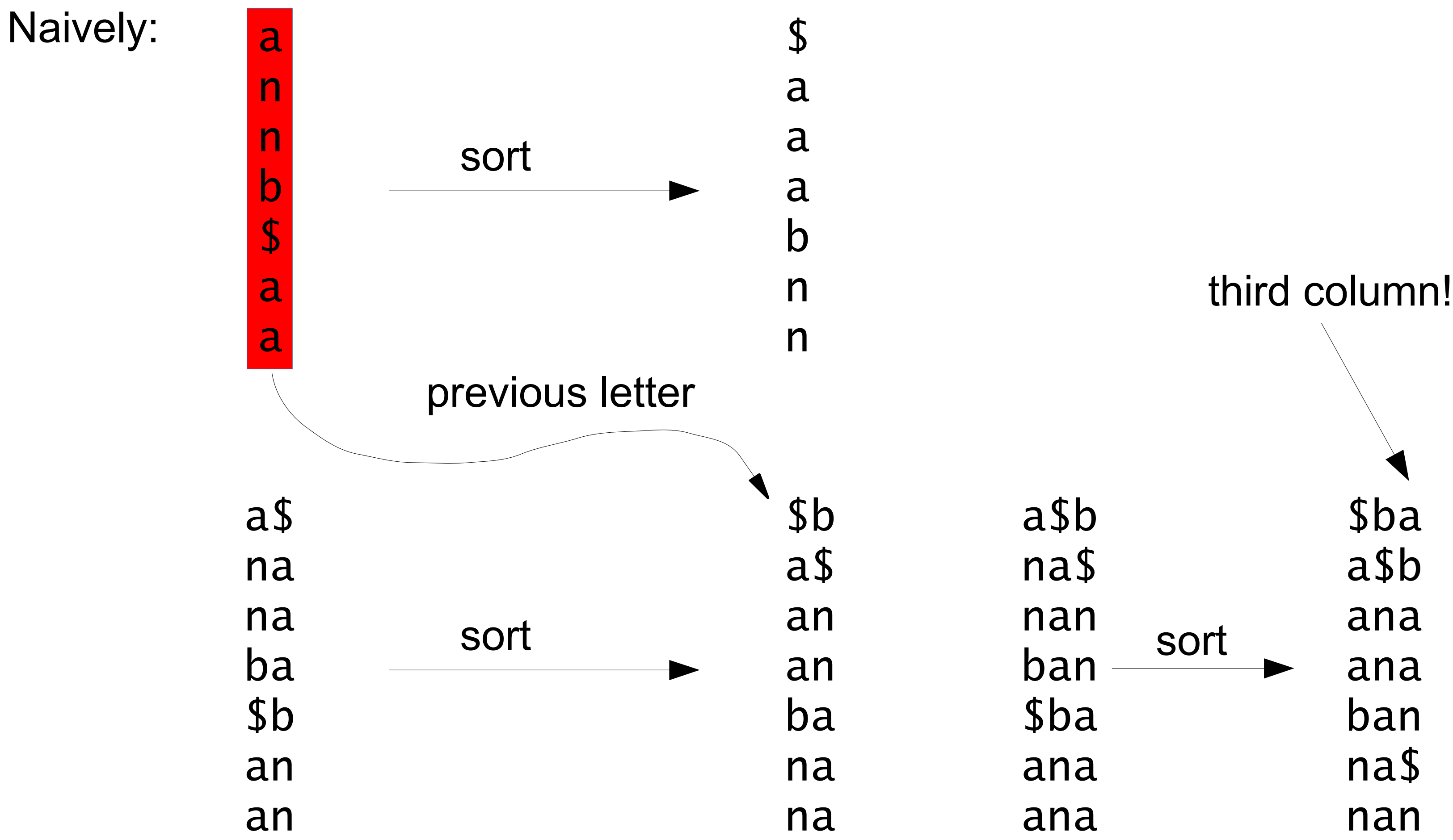
a$	sort →	$b	
na		a$	
na		an	← columns 1 and 2.
ba		an	
$b		ba	
an		na	
an		na	

2. Burrows-Wheeler Transform

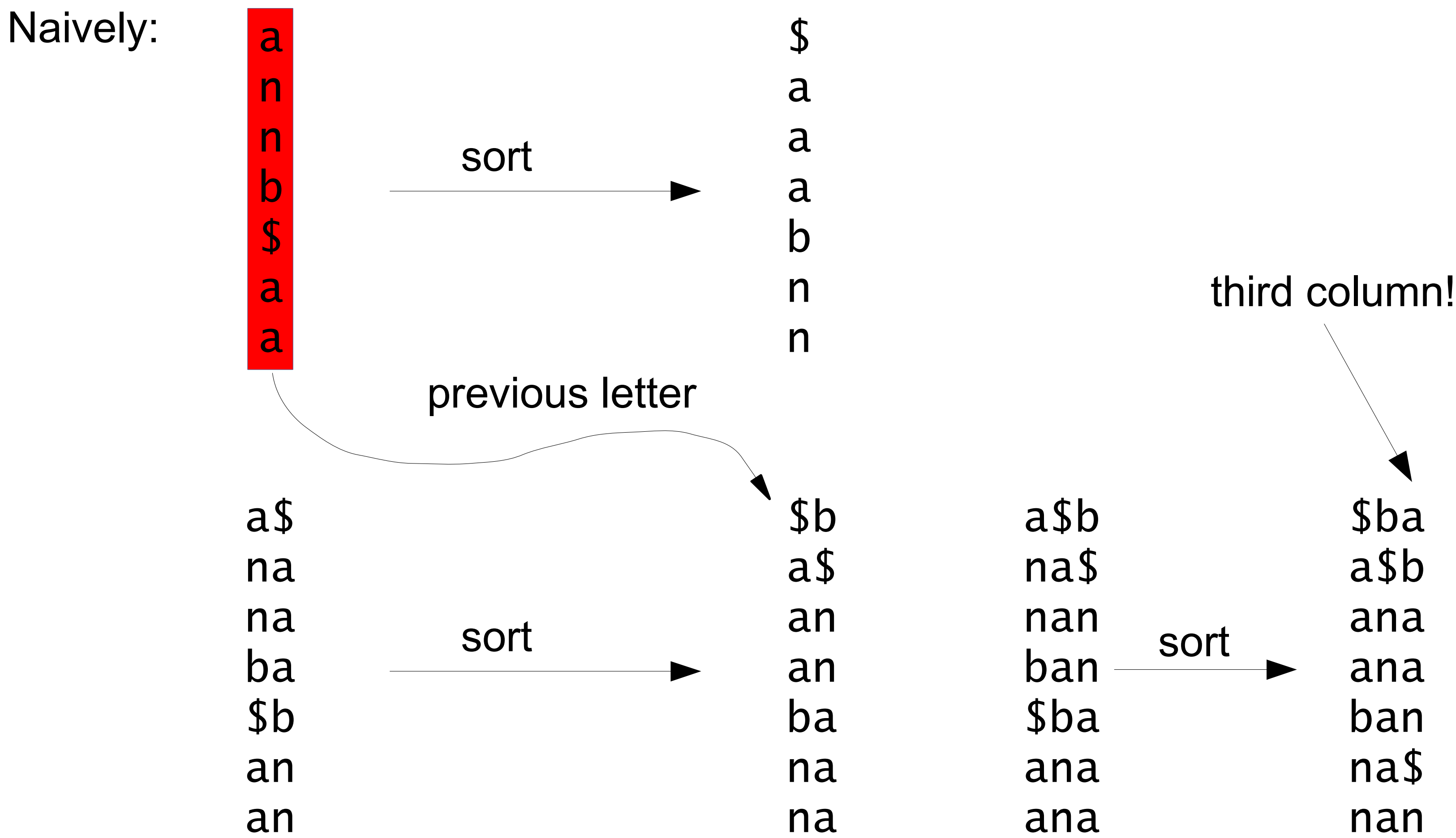
Naively:



2. Burrows-Wheeler Transform



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)

$rank_b(L, k)$ = #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

$rank_n(L, 4) = 2$

$rank_n(L, 3) = 1$

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
```

$rank_n(L, 4) = 2$

$rank_n(L, 3) = 1$

$rank_a(L, 7) = 2$

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

2. Burrows-Wheeler Transform

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

first line starting with the letter in the first column

Last-to-Front Mapping

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

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

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

2. Burrows-Wheeler Transform

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

first line starting with the letter in the first column

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?

$LF(6) = C[L[6]] + rank_a(L, 6) = C[\text{“a”}] + 1 = 2 + 1 = 3$

Decoding

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

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

	$	a	b	n
C	1	2	5	6

first line starting with the letter in the first column

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

$rank_b(L, k)$ = #occc 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	n
C	1	2	5	6

first line starting with the letter in the first column

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	n
C	1	2	5	6

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) = #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	n
C	1	2	5	6

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

$rank_b(L, k)$ = #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	n
C	1	2	5	6

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
L =	a	n	n	b	$	a	a

	$	a	b	n
C	1	2	5	6

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.

```
he...t
he...t
he...t
he...t   (“t”,18733)
he...t
he...t
he...t
```

→ 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

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]

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]

$$s = C[P[i]] + \text{rank}_{L[i]}(L, sp-1)$$

$$e = C[P[i]] + \text{rank}_{L[i]}(L, ep) - 1$$

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

→ How can we construct BWT[T]??

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