# XML and Databases

**Lecture 13**
*Fast Substring Search*

Sebastian Maneth
NICTA and UNSW

*CSE@UNSW - Semester 1, 2010*

# Fast Substring Search

Recall the **`contains`**-predicate of XPath:

```
//book/abstract[contains(.,"fix")]
```

For instance the abstract node:

```
<book>..
  <abstract>This article dicusses the advantages of
  suffix arrays, for the purpose of substring search ..
  </abstract>..
</book>
```

will be returned, because it contains the substring "fix"
because it appears in the word "suffix" mentioned in the abstract text.

# Fast Substring Search

**Question**

Given a ***very large text***, how do you search for

→ All occurrences of a given keyword?
→ All occurrences of a given substring?
→ Count them (can be done faster?)

# Fast Substring Search

**Question**

Given a ***very large text***, how do you search for

→ All occurrences of a given keyword?
→ All occurrences of a given substring?
→ Count them (can be done faster?)

What we know so far:

→ can use KMP-algorithm.
for a text of length n, it only takes $O(n)$ time to locate all occurrences of the substring.

→ in a database, that is *way* to slow!!
How do you think Google indexes text for fast search??

# Fast Substring Search

**Question**

Given a ***very large text***, how do you search for

→ All occurrences of a given keyword?
→ All occurrences of a given substring?
→ Count them (can be done faster?)

We want search time to be independent of the size n of the text, but should only depend on the length of the keyword.

We are allowed to preprocess the string in linear time ("indexing").

# Fast Substring Search

**Question**

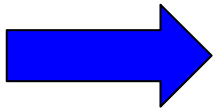
Given a ***very large text***, how do you search for

→ All occurrences of a given keyword?
→ All occurrences of a given substring?
→ Count them (can be done faster?)

**Idea 1** --we search for exact WORDS, not substrings—

Make a "dictionary" of every WORD that occurs in the text:

```
1: this[0, 89, 2098]
2: article[8, 29300]
3: …
```



Sort it!

```
1: a[90, 183, 290, … ]
2: actual[450, 9812, … ]
3: article[8, 29300]
3: …
```

# Fast Substring Search

Given a keyword $a_1a_2\ldots a_m$ of length $m$,

How much time required to locate all occurrences of the keyword?

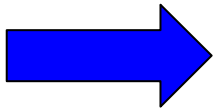
Easy: keep start rows of strings that "start with a_1" (for any letter), and within those rows, again those that "continue with letter a_2" (for all letters) Etc. (this is a tree of height=length of longest word, and branching=# different letters)

---

**Idea 1** --we search for exact WORDS, not substrings—

Make a "dictionary" of every WORD that occurs in the text:

```
1: this[0, 89, 2098]
2: article[8, 29300]
3: …
```



Sort it!

```
1: a[90, 183, 290, … ]
2: actual[450, 9812, … ]
3: article[8, 29300]
3: …
```

# Fast Substring Search

Given a keyword $a_1a_2\ldots a_m$ of length $m$,

How much time required to locate all occurrences of the keyword?

➔ only time O(m)! ☺

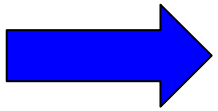
Problems (1) indexing time?!
(2) how to do substring search??

---

**Idea 1** --we search for exact WORDS, not substrings—

Make a “dictionary” of every WORD that occurs in the text:

```
1: this[0, 89, 2098]
2: article[8, 29300]
3: …
```



Sort it!

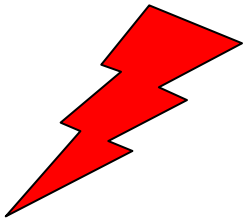
```
1: a[90, 183, 290, … ]
2: actual[450, 9812, … ]
3: article[8, 29300]
3: …
```

# Fast Substring Search

Given the text of length n, how many **substrings** are there?

→ (begin position, end position)

Quadratically many! That is, O(n^2).
Thus, it is **impossible** in linear time to list all these substrings and put them into a (sorted) dictionary!

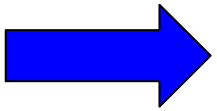


---

**Idea 1** --we search for exact WORDS, not substrings—

Make a “dictionary” of every WORD that occurs in the text:

```
1: this[0, 89, 2098]
2: article[8, 29300]
3: …
```



Sort it!

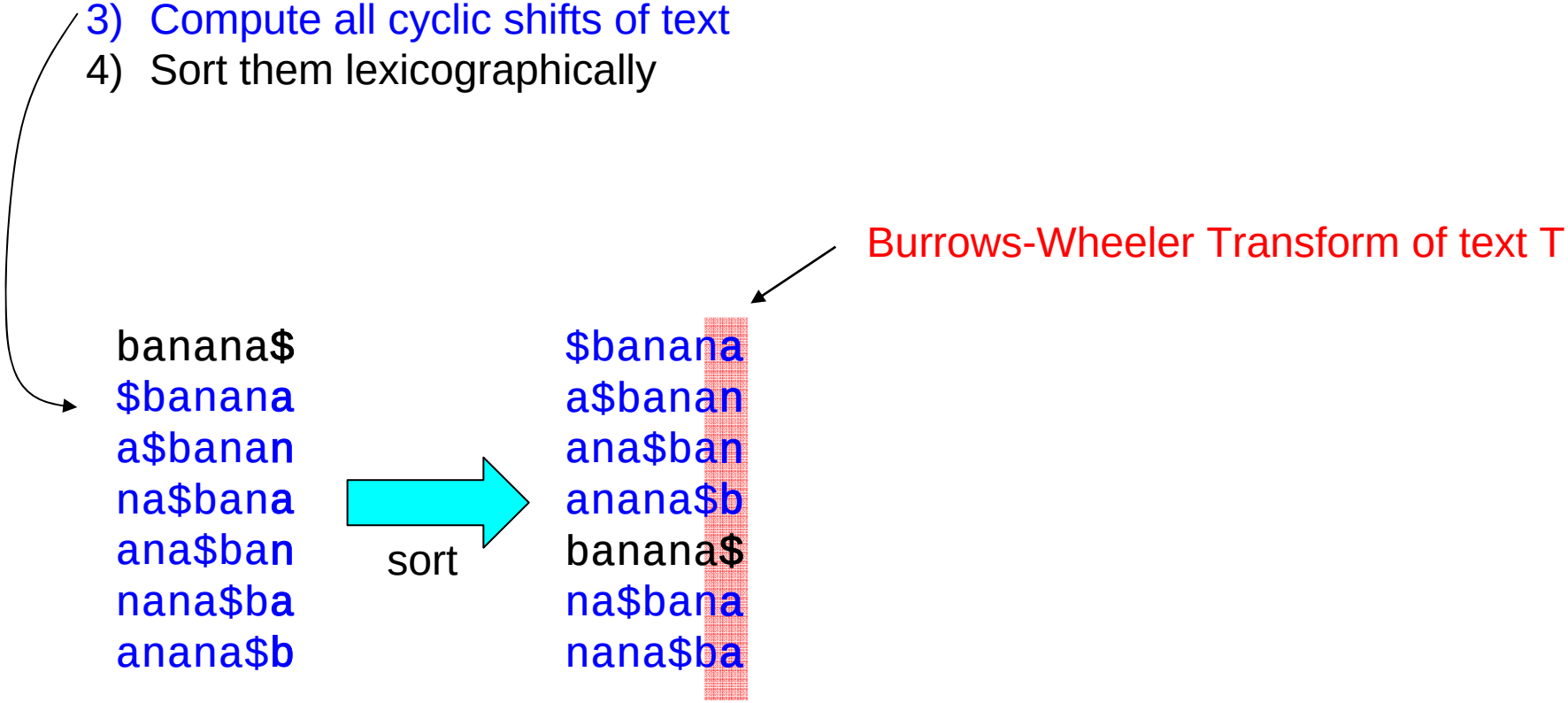
```
1: a[90, 183, 290, … ]
2: actual[450, 9812, … ]
3: article[8, 29300]
3: …
```

# The Burrows-Wheeler Transform

Idea comes from compression.
`bzip2` is based on the Burrows-Wheeler Transform!

---

1) Add an end-marker "$" to the end of the text
2) End-marker $ is smallest in ordering:
   '$' < 'a' < 'b' < 'c' < ….. < 'z' < 'A' < ….
3) Compute all cyclic shifts of text
4) Sort them lexicographically

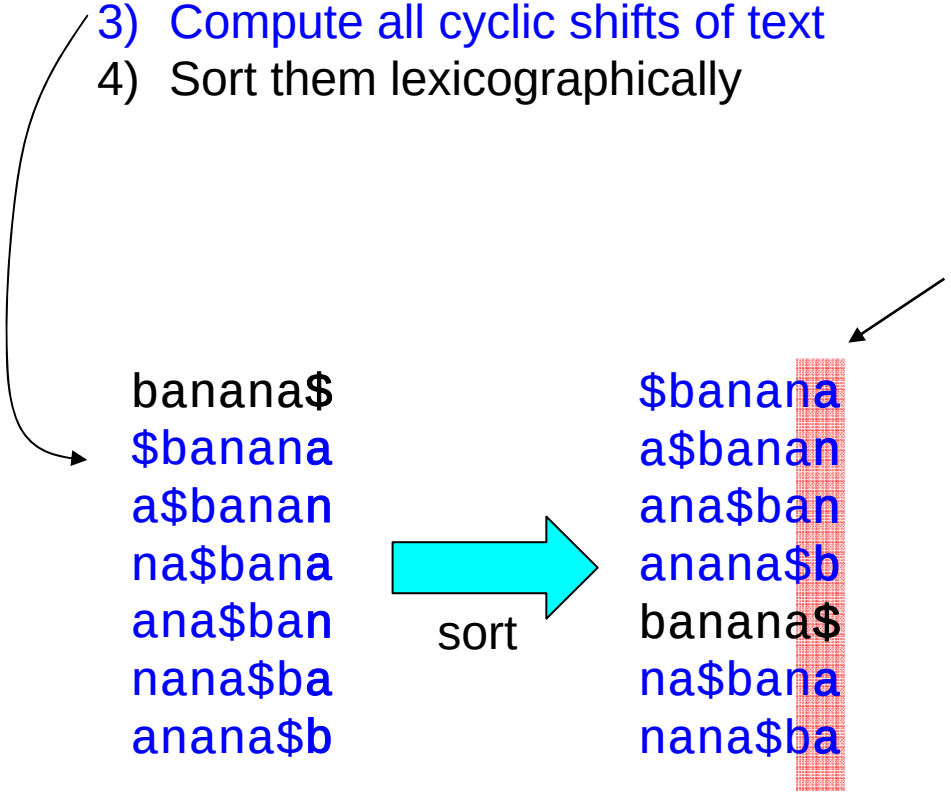


# The Burrows-Wheeler Transform

Idea comes from compression.
`bzip2` is based on the Burrows-Wheeler Transform!

---

1) Add an end-marker "$" to the end of the text
2) End-marker $ is smallest in ordering:
   '$' < 'a' < 'b' < 'c' < ….. < 'z' < 'A' < ….
3) Compute all cyclic shifts of text
4) Sort them lexicographically



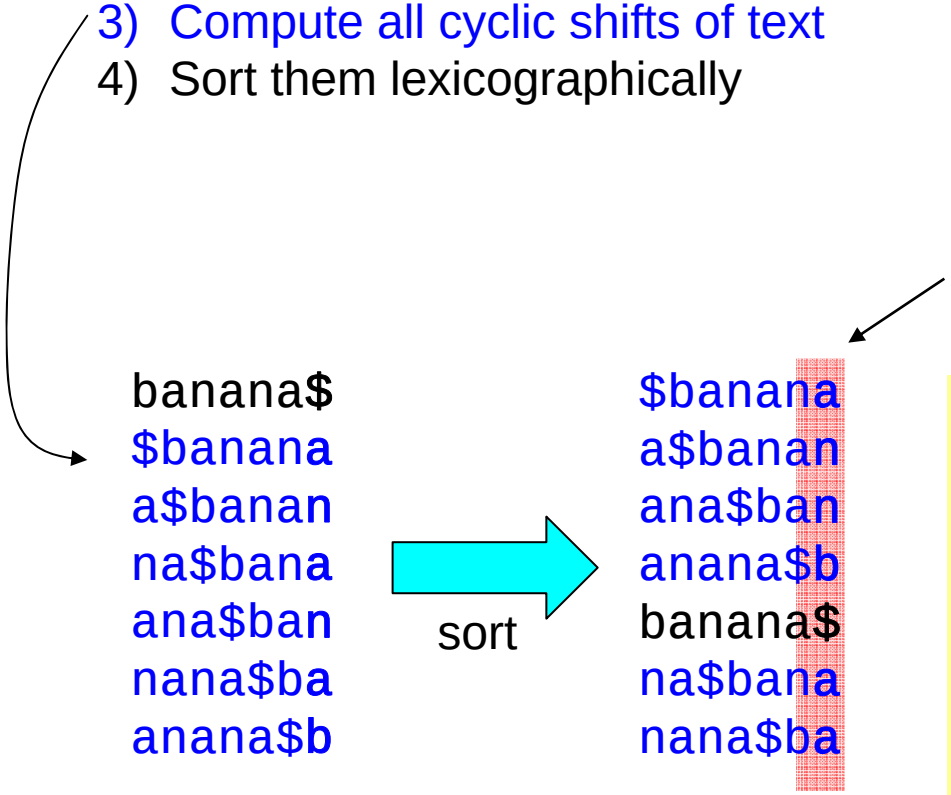

**Question**
Why do you think is the BWT good for compression?

# The Burrows-Wheeler Transform

Idea comes from compression.
`bzip2` is based on the Burrows-Wheeler Transform!

---

1) Add an end-marker "$" to the end of the text
2) End-marker $ is smallest in ordering:
   '$' < 'a' < 'b' < 'c' < ….. < 'z' < 'A' < ….
3) Compute all cyclic shifts of text
4) Sort them lexicographically




Burrows-Wheeler Transform of text T

**First row:** only tells us
how many substrings
→ start with "a" (3)
→ how many start with "b" (1)
etc.
Same for any text with these letters!
We canNOT reconstruct T from row 1!

# The Burrows-Wheeler Transform

Idea comes from compression.
`bzip2` is based on the Burrows-Wheeler Transform!

---

1) Add an end-marker "$" to the end of the text
2) End-marker $ is smallest in ordering:
   '$' < 'a' < 'b' < 'c' < ….. < 'z' < 'A' < ….
3) Compute all cyclic shifts of text
4) Sort them lexicographically

Burrows-Wheeler Transform of text T

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

canNOT reconstruct T from second row!

**Second row:** tells us
how many substrings
→start with "n", if letter before is "a" (2)
→start with "a" if letter before is "n" (2)

**First row:** only tells us
how many substrings
→ start with "a" (3)
→ how many start with "b" (1)
etc.
Same for any text with these letters!
We canNOT reconstruct T from row 1!

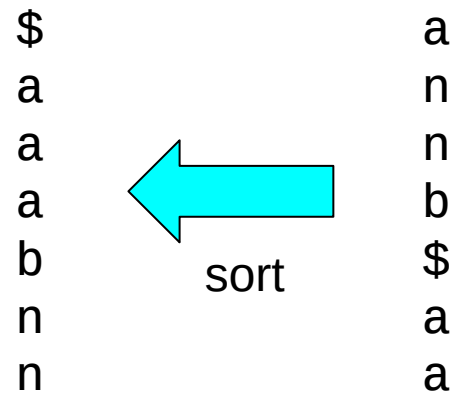
# The Burrows-Wheeler Transform

But, we can reconstruct T from the last row!! ☺

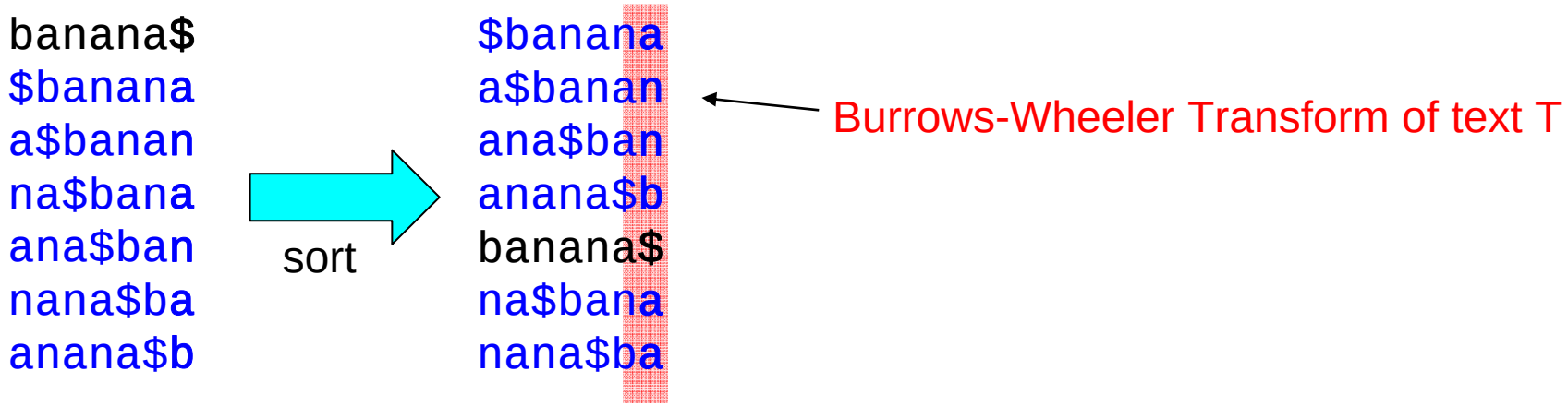
**How?**

Naïve way:

1. given “annb$aa”, sort the letters. This gives row 1!




What’s next?
Hint: this tells us all two-letter substrings!



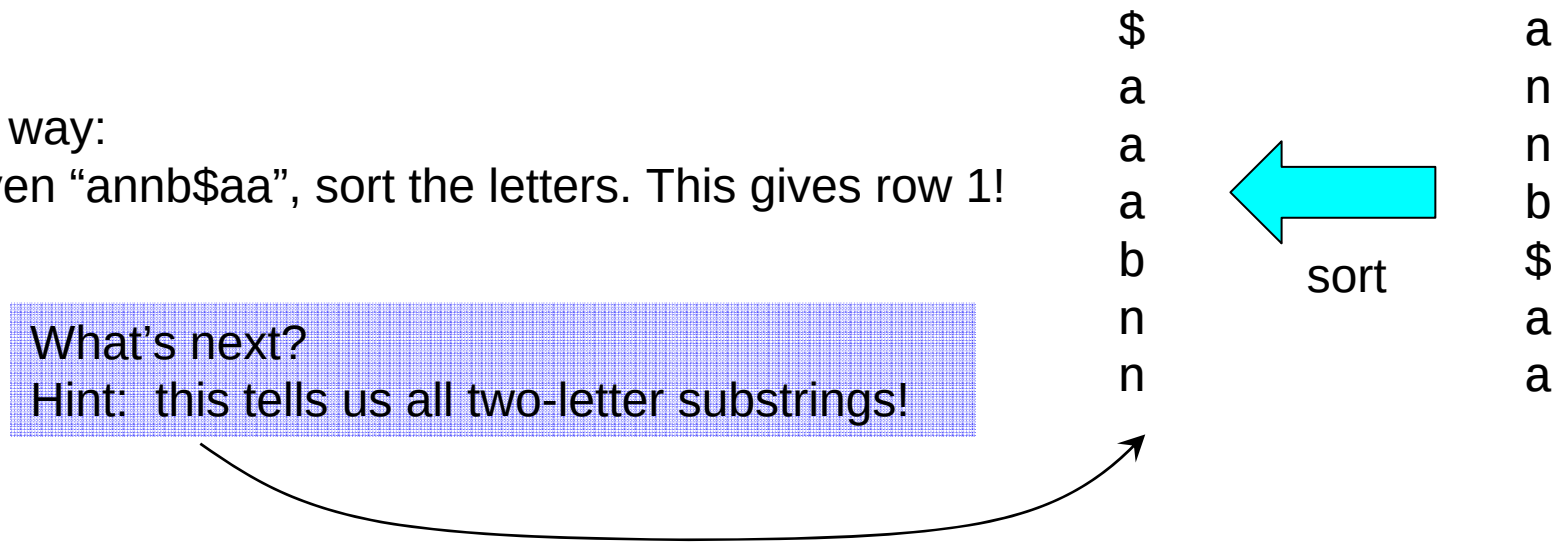
# The Burrows-Wheeler Transform

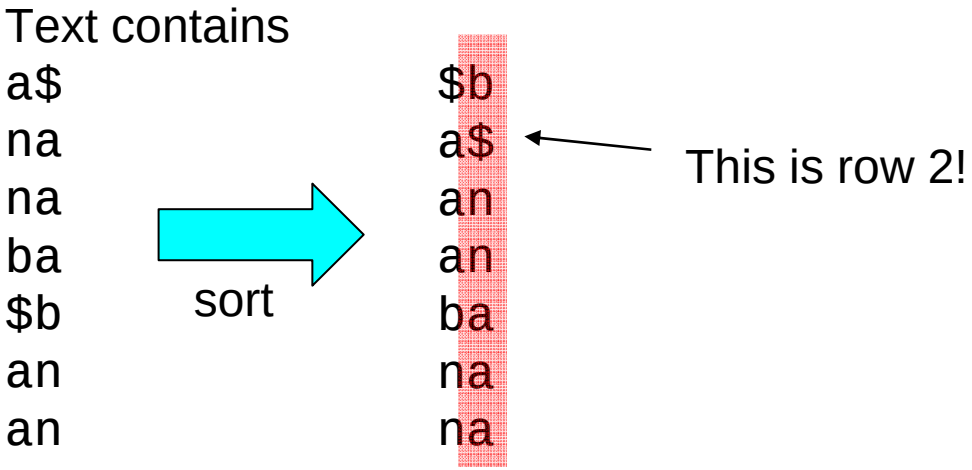
But, we can reconstruct T from the last row!! ☺

**How?**

Naïve way:

1. given “annb$aa”, sort the letters. This gives row 1!



# The Burrows-Wheeler Transform

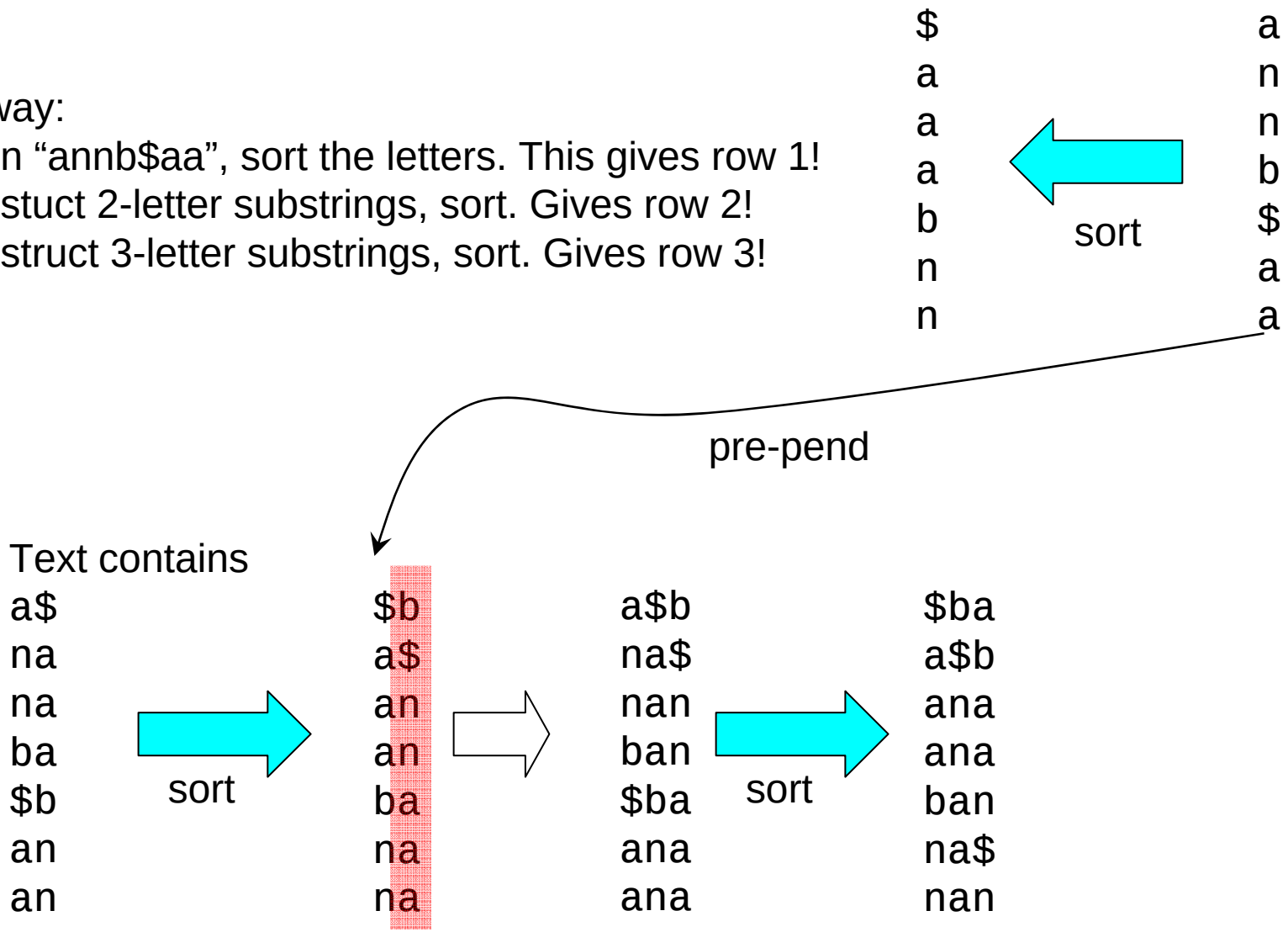
But, we can reconstruct T from the last row!! ☺

**How?**

Naïve way:

1. given “annb$aa”, sort the letters. This gives row 1!
2. Constuct 2-letter substrings, sort. Gives row 2!
3. Construct 3-letter substrings, sort. Gives row 3!

etc



# The Burrows-Wheeler Transform

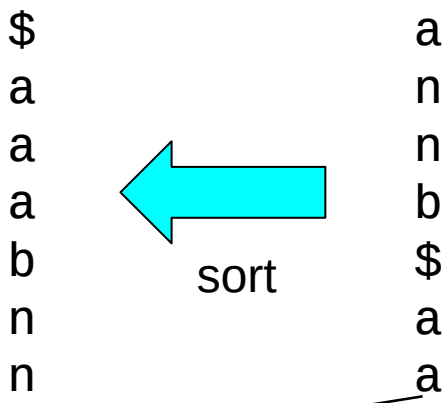
But, we can reconstruct T from the last row!! ☺

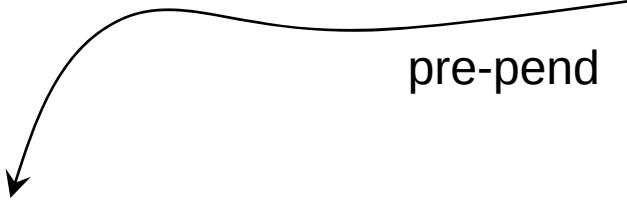
**How?**

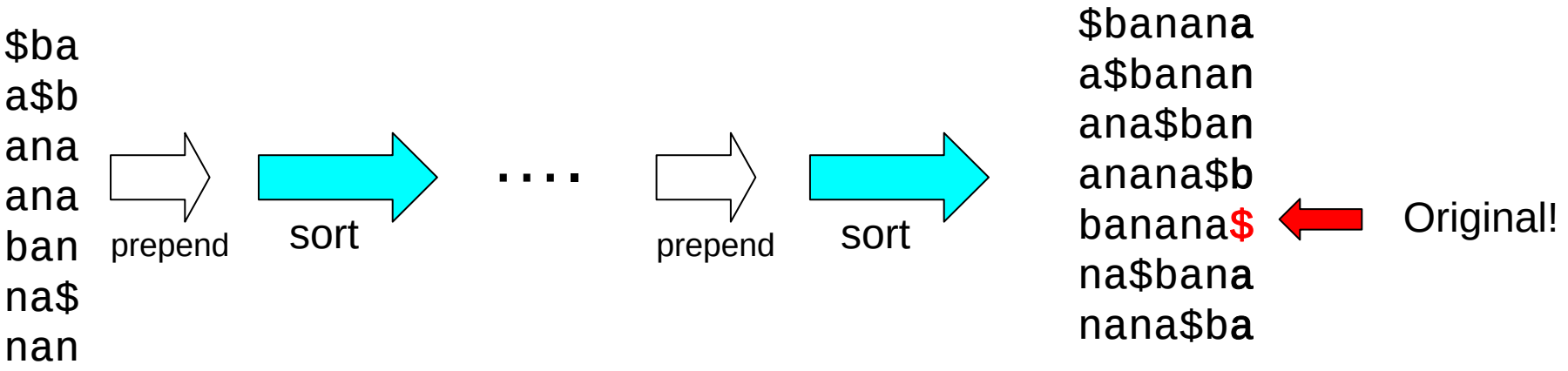
Naïve way:

1. given “annb$aa”, sort the letters. This gives row 1!
2. Constuct 2-letter substrings, sort. Gives row 2!
3. Construct 3-letter substrings, sort. Gives row 3!

etc



# BWT: Better Decompression

→ In a real implementation we may NOT construct all cyclic shifts and sort… (because that takes quadratic time!!)

→ Same for decompression. May not do it the naïve way!

---

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

```
C $ a b n
  0 1 4 5
```

LF-mapping

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

O(log S) time
using *wavelet tree*

Retrieving T: start from end marker, read backwards (by applying LF)

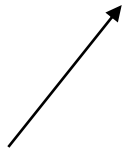
e.g.: LF(5)=1, LF(1)=2, LF(2)=6, LF(6)=3, LF(3)=7, LF(7)=4

L[.]= $ a n a n a b

# Backward Search

Here comes the ***magic***: we are now able to count the number of occurrences of a substring of length m, only in time O(m log S)!

S = size of alphabet



This is what makes fast keyword Search a la Google possible!

Search time is INDEPENDENT of the size of the text!!

# Backward Search

Here comes the ***magic***: we are now able to count the number of occurrences of a substring of length m, only in time O(m log S)!

S = size of alphabet

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

```
C $ a b n
  0 1 4 5
```

LF-mapping

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

O(log S) time using *wavelet tree*

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

Initial range: [sp,ep] with sp=C[P[m]]+1 and ep=C[P[m]+1]

Then [s,e] with

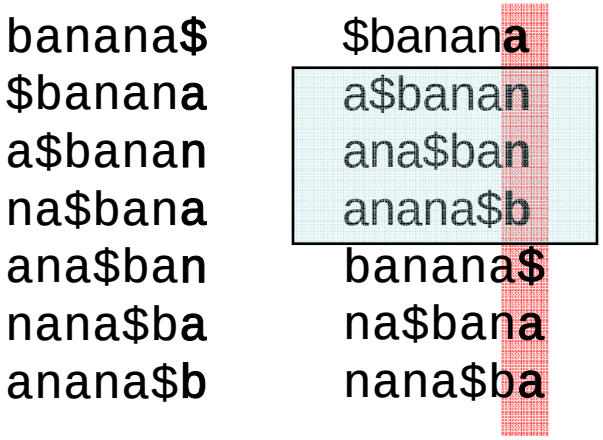
$s = C[P[i]] + rank_{L[i]}(L,sp-1) + 1$

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

# Backward Search

Here comes the ***magic***: we are now able to count the number of occurrences of a substring of length m, only in time O(m log S)!

S = size of alphabet




| C | $ | a | b | n |
|---|---|---|---|---|
| | 0 | 1 | 4 | 5 |

```
      123
P =   ana
```

[sp,ep] = [2,4]

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

➔ Initial range: [sp,ep] with sp=C[P[m]]+1 and ep=C[P[m]+1]

Then [s,e] with

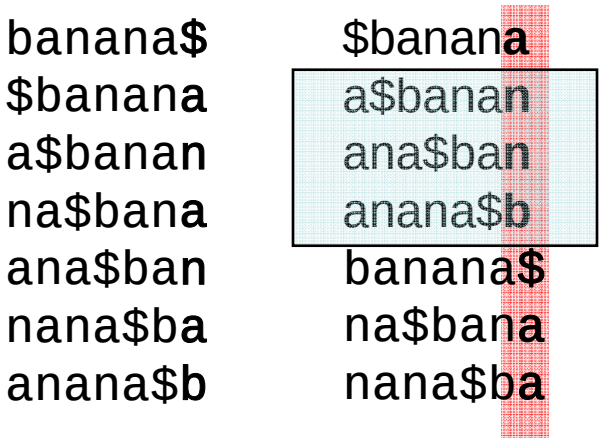
$s = C[P[i]] + rank_{L[i]}(L,sp-1) + 1$

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

# Backward Search

Here comes the ***magic***: we are now able to count the number of occurrences of a substring of length m, only in time O(m log S)!

S = size of alphabet




| C | $ | a | b | n |
|---|---|---|---|---|
| | 0 | 1 | 4 | 5 |

```
      123
P =   ana
```

[sp,ep] = [2,4]

s=C["n"] + $rank_n(L,1)$ +1
= 5 + 0 + 1 = 6

e= 5 + $rank_n(L,4)$
= 5 + 2 = 7

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

Then [s,e] with

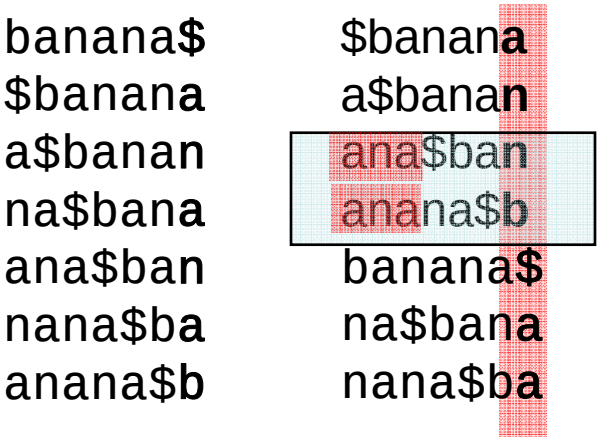
$s = C[P[i]] + rank_{L[i]}(L,sp-1) + 1$

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

# Backward Search

Here comes the ***magic***: we are now able to count the number of occurrences of a substring of length m, only in time O(m log S)!

S = size of alphabet




| C | $ | a | b | n |
|---|---|---|---|---|
| | 0 | 1 | 4 | 5 |

P = ana (1 2 3)

[sp,ep] = [2,4]

sp=6

ep=7

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

= 1 + 1 + 1 = 3

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

e = 1 + rank_a(L,7) =

1 + 3 = 4

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

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

**Done!**

[3,4]=final range

➔ 2 Occs of "ana"

# Backward Search

Here comes the ***magic***: we are now able to count the number of occurrences of a substring of length m, only in time O(m log S)!

S = size of alphabet

**Backward search** for Pattern P[1]..P[m] **Counting**: O(m log S) time

**Locating**
If every $l=\log^{1+\text{epsilon}} n$ position is sampled
then O(l log S) per occurrence,
by backward traversal using LF.

# Real Performance

```
/* In order : IsContains, Timing of IsContains, GlobalCount, Timing of
GlobalCount, CountContains, time of CountContains,
time of Full Report Contains */
-----------------
Sampling rate 64
"Bakst": 1, 0.038, 1, 0.004, 1, 0.04, 0.012, max_mem = 61
"ruminants": 1, 0.04, 22, 0.009, 19, 2.281, 1.588, max_mem = 61
"morphine": 1, 0.026, 392, 0.009, 144, 29.924, 32.668, max_mem = 61
"AUSTRALIA": 1, 0.028, 438, 0.009, 438, 4.616, 4.457, max_mem = 61
"molecule": 1, 0.051, 1472, 0.008, 966, 128.28, 122.014, max_mem = 61
"brain": 1, 0.02, 2685, 0.005, 1493, 218.462, 215.196, max_mem = 61
"human": 1, 0.019, 6897, 0.005, 4690, 553.496, 548.009, max_mem = 62
"blood": 1, 0.018, 10402, 0.005, 8534, 401.214, 399.674, max_mem = 62
"from": 1, 0.016, 20859, 0.004, 12073, 1722.95, 1717.83, max_mem = 62
"with": 1, 0.016, 63332, 0.004, 22974, 5084.14, 5083.77, max_mem = 63
" in": 1, 0.014, 238638, 0.003, 42586, 19641.8, 19630.3, max_mem = 64
"a": 1, 0.001, 2932251, 0, 595716, 189299, 188377, max_mem = 93
"\n": 1, 0.001, 9730750, 0.001, 5870474, 132780, 132241, max_mem = 86
```

use
naïve

CountContains/FullContains on naïve text: ca. **2700ms**

# Real Performance

```
/* In order : IsContains, Timing of IsContains, GlobalCount, Timing of
GlobalCount, CountContains, time of CountContains,
time of Full Report Contains */
-----------------
Sampling rate 5
"Bakst": 1, 0.038, 1, 0.005, 1, 0.049, 0.013, max_mem = 100
"ruminants": 1, 0.038, 22, 0.01, 19, 0.156, 0.086, max_mem = 100
"morphine": 1, 0.027, 392, 0.009, 144, 1.718, 1.357, max_mem = 100
"AUSTRALIA": 1, 0.098, 438, 0.009, 438, 4.145, 3.942, max_mem = 100
"molecule": 1, 0.029, 1472, 0.009, 966, 6.247, 5.853, max_mem = 101
"brain": 1, 0.019, 2685, 0.006, 1493, 12.24, 11.588, max_mem = 101
"human": 1, 0.018, 6897, 0.005, 4690, 25.403, 27.344, max_mem = 101
"blood": 1, 0.026, 10402, 0.005, 8534, 77.175, 73.613, max_mem = 101
"from": 1, 0.016, 20859, 0.003, 12073, 84.012, 78.663, max_mem = 101
"with": 1, 0.015, 63332, 0.004, 22974, 242.834, 235.043, max_mem = 102
" in": 1, 0.012, 238638, 0.002, 42586, 1105.6, 1091.43, max_mem = 103
"b": 1, 0, 411409, 0.001, 135307, 1779.27, 1762.62, max_mem = 108
"g": 1, 0.001, 748326, 0, 320440, 3411.65, 3378.85, max_mem = 119
""a": 1, 0, 2932251, 0, 595716, 13183.4, 13173.4, max_mem = 133
"\n": 1, 0.001, 9730750, 0.001, 5870474, 87770.9, 88230.4, max_mem = 126
```

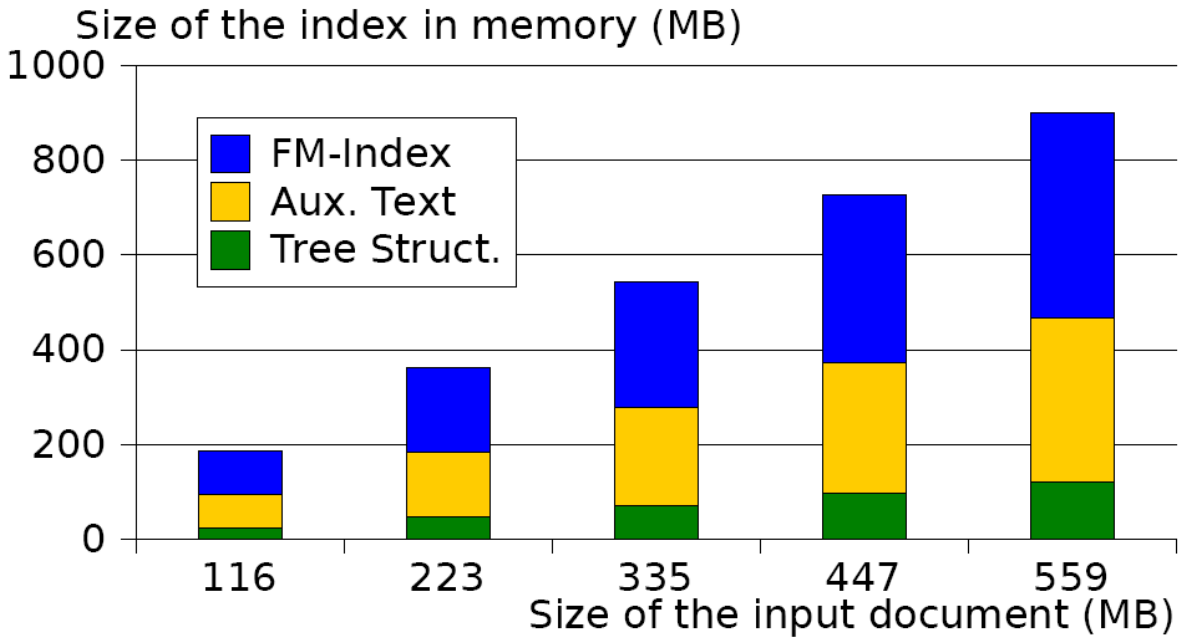
use naïve

CountContains/FullContains on naïve text: ca. **2700ms**

# Construction Time

**XMark data** 174 different element labels
Max Depth: 14, Average Depth: 9.6

| | |
|---|---|
| 116MB XMark 6,074,297 nodes | Text: 7min 18s **TOTAL= 9min 20s** |
| 559MB XMark 29,239,763 nodes | Text: 38min 45s **TOTAL= 53min 25s** |
| 1GB XMark 58,472,941 nodes | Text: 1h 24min **TOTAL= 1h 55min** |



# ☺ Advertisement ☺

New course, will be first offered in Session 1 of 2011.

COMP9319 -- Web Data Compression and Search (PG, UOC: 6)

---

Contents

**Data Compression :** (a) Adaptive Coding, Information Theory
(b) Text Compression (ZIP, GZIP, BZIP, etc)
(c) Burrows-Wheeler Transform and Backward Search
(d) XML Compression

**Search:** (a) Indexing
(b) Pattern Matching and Regular Expression Search
(c) Distributed Querying
(d) Fast Index Construction
(e) Implementation
If time allows: Streaming Algorithms, On-Line Data Analytics

The lecture materials will be complemented by projects and assignments.

# END
# Lecture 13 and of the course.

→ Thanks for your attention and hard work.
→ Hopefully you have enjoyed the lecture.
→ Good luck and all the best with
the exam on June 12th.