Applied Databases

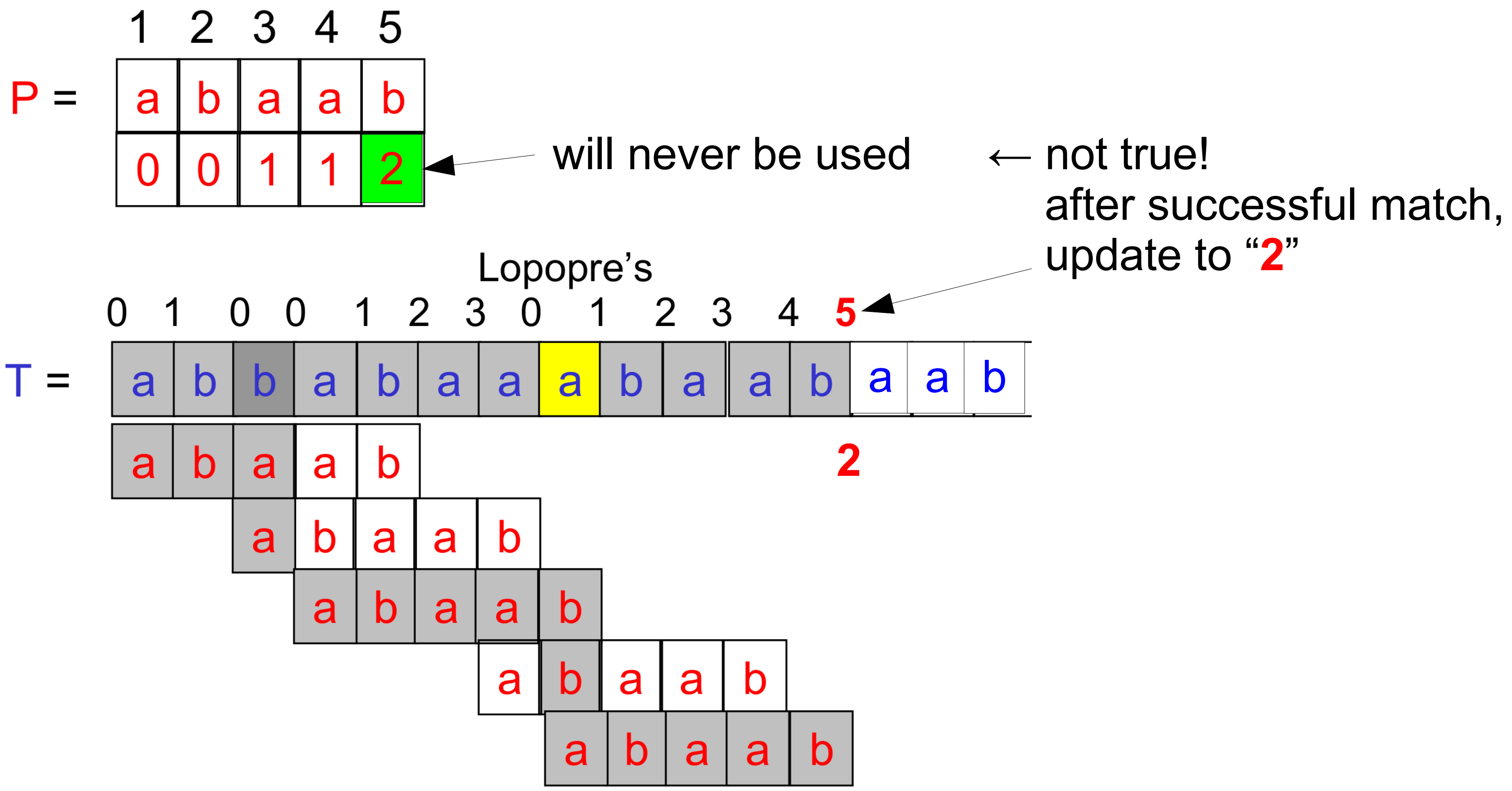
Lecture 14
Indexed String Search, Suffix Trees

Sebastian Maneth

University of Edinburgh - March 9th, 2017

Recap: Morris-Pratt (1970)

Given Pattern P, Text T, find all occurrences of P in T.



	1	2	3	4	5
P =	a	b	a	a	b
	0	0	1	1	**2**

will never be used ← not true!
after successful match,
update to "**2**"

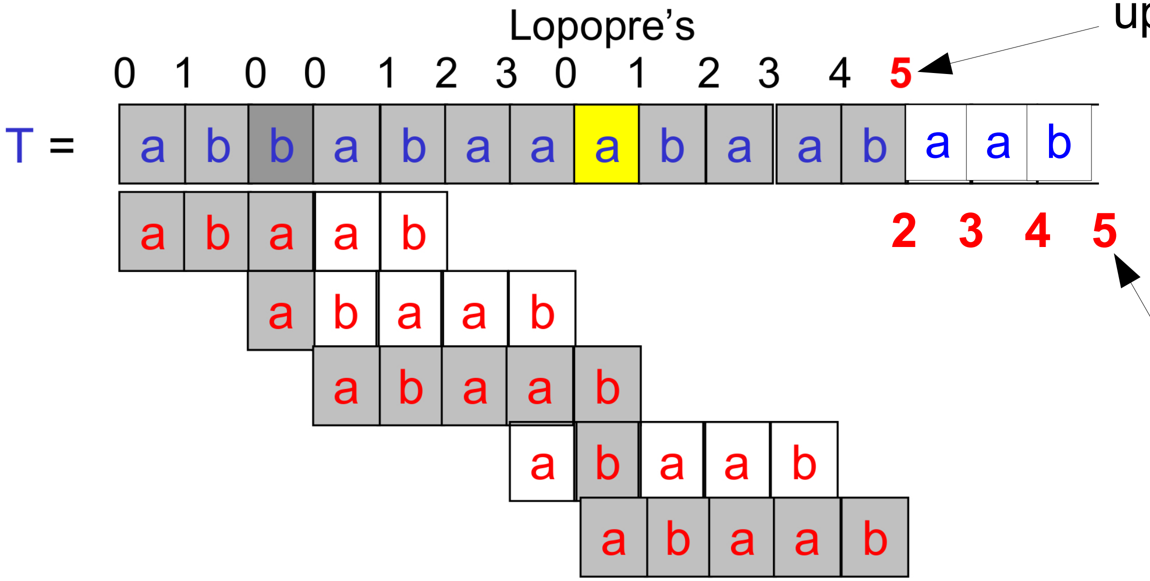
Morris-Pratt (1970)

Given Pattern P, Text T, find all occurrences of P in T.

	1	2	3	4	5
P =	a	b	a	a	b
	0	0	1	1	2

~~will never be used~~

← not true!
after successful match,
update to "**2**"



one more
successful
match!

Knuth-Morris-Pratt (1977)

P =

1	2	3	4	5
a	b	a	a	b
0	0	1	1	2

Previous table (Morris-Pratt)

-1	1	2	3	4	5
	a	b	a	a	b
	0	-1	1	0	2

Knuth-Morris-Pratt table

KMP[j] largest k>=0 such that strong_cond(j,k) holds, or -1 if such k does not exist

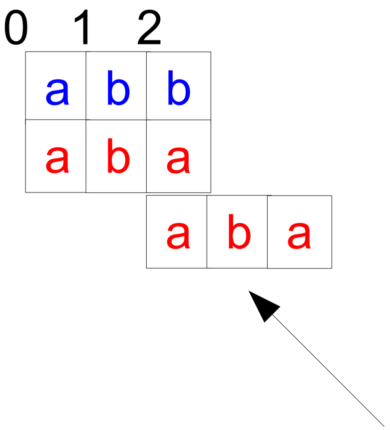
strong_cond(j,k): P[1..k] is a proper suffix of P[1..j] **and**
P[k+1] != P[j+1]

Why?

KMP

	1	2	3	4	5
P =	a	b	a	a	b
	0	0	1	1	2

-1	1	2	3	4	5
	a	b	a	a	b
	0	-1	1	0	2



this shift make **no sense**!

KMP[j] largest k>=0 such that strong_cond(j,k) holds, or -1 if such k does not exist

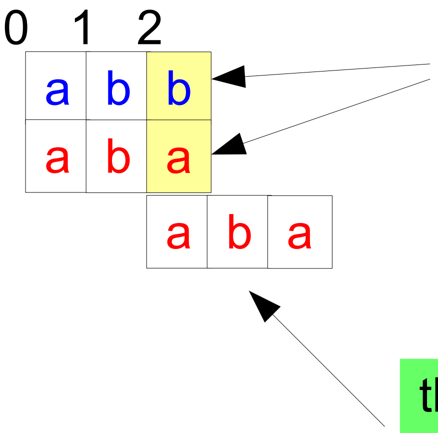
strong_cond(j,k): P[1..k] is a proper suffix of P[1..j] **and**
P[k+1] != P[j+1]

Why?
Otherwise next check will fail!!

KMP

	1	2	3	4	5
P =	a	b	a	a	b
	0	0	1	1	2

-1	1	2	3	4	5
	a	b	a	a	b
	0	-1	1	0	2



different letters!
–– but P[1] **=** P[3]
→ the shift MUST fail!

this shift make **no sense**!

KMP[j] largest k>=0 such that strong_cond(j,k) holds, or -1 if such k does not exist

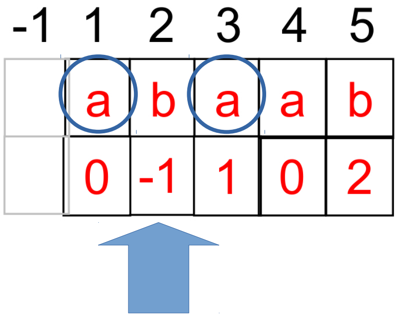
strong_cond(j,k): P[1..k] is a proper suffix of P[1..j] **and**
P[k+1] != P[j+1]

Why?
Otherwise next check will fail!!

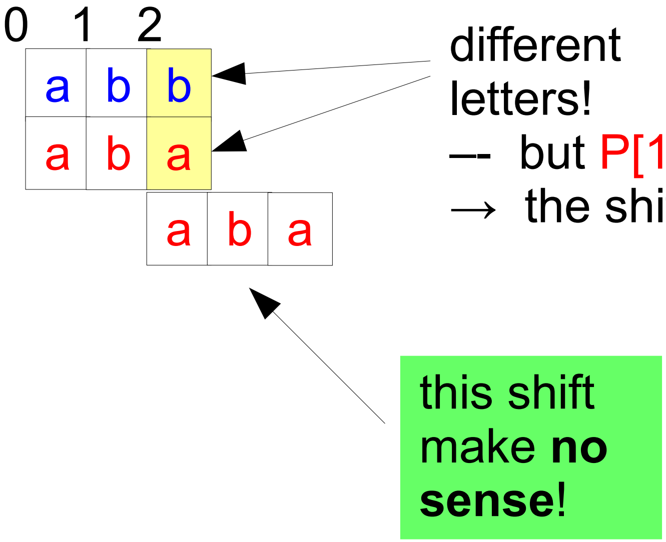
KMP

P =

1	2	3	4	5
a	b	a	a	b
0	0	1	1	2



-1	1	2	3	4	5
	a	b	a	a	b
	0	-1	1	0	2



different letters!
–- but P[1] **=** P[3]
→ the shift MUST fail!

this shift make **no sense**!

KMP[j] largest k>=0 such that strong_cond(j,k) holds, or -1 if such k does not exist

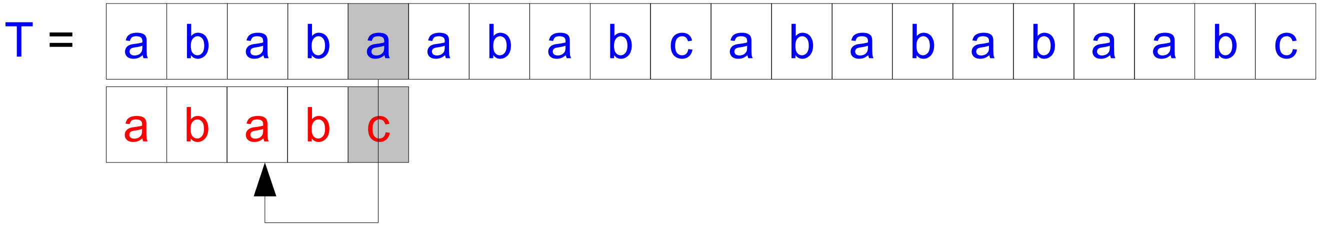
strong_cond(j,k): P[1..k] is a proper suffix of P[1..j] **and** ← (C1)
P[k+1] != P[j+1] ← (C2)

KMP[2] = 0? (C1) satisfied
(C2) **not** satisfied: P[0 + 1] = a = P[2 + 1]

→ KMP[2] = –1

Horspool = Idea 1 of Boyer-Moore

Idea 1 Match **RIGHT-TO-LEFT** in window



T = a b a b a a b a b c a b a b a b a a b c

a b a b c

R(a) = 2

For each letter z, let
R(z) = distance from right-most occurrence of z in P[1..m– 1], to the end of P
(and |P| if there is no occurrence)
R(c) = 5
R(b) = 1

Horspool (ONE RULE ONLY):
If mismatch and P[m] aligned to z in T, shift pattern to the RIGHT by R(z).

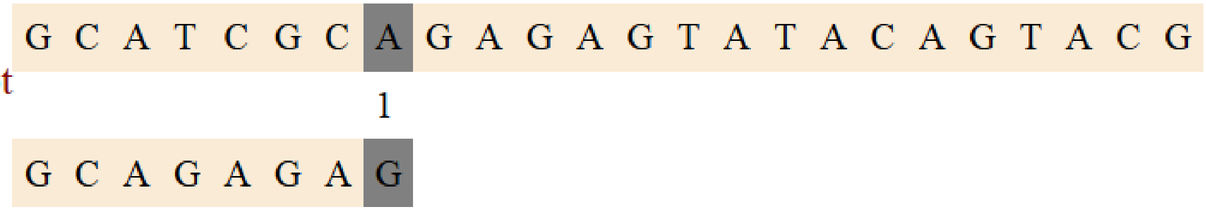
P =

G	C	A	G	A	G	A	G

A	C	G	T
1	6	2	8

= R(z)-table

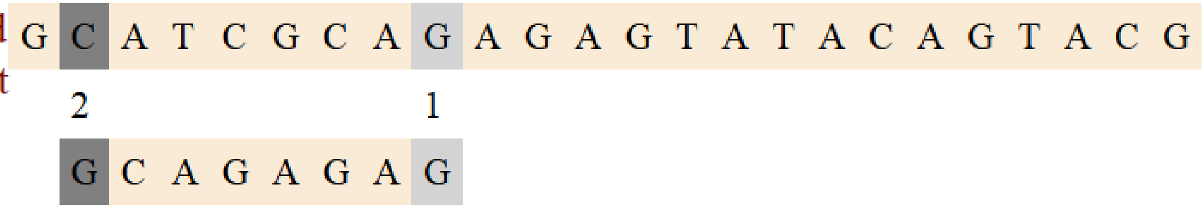
First attempt



G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
							1																
G	C	A	G	A	G	A	G																

Shift by: 1 (*bmBc*[A])

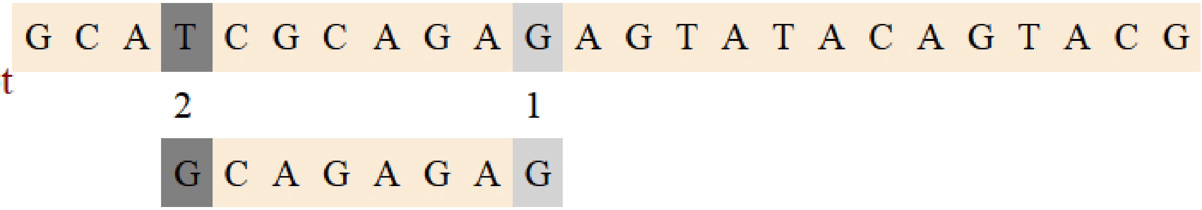
Second attempt



G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
	2							1															
	G	C	A	G	A	G	A	G															

Shift by: 2 (*bmBc*[G])

Third attempt



G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
			2							1													
			G	C	A	G	A	G	A	G													

Shift by: 2 (*bmBc*[G])

Fourth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
					2	3	4	5	6	7	8	1											
					G	C	A	G	A	G	A	G											

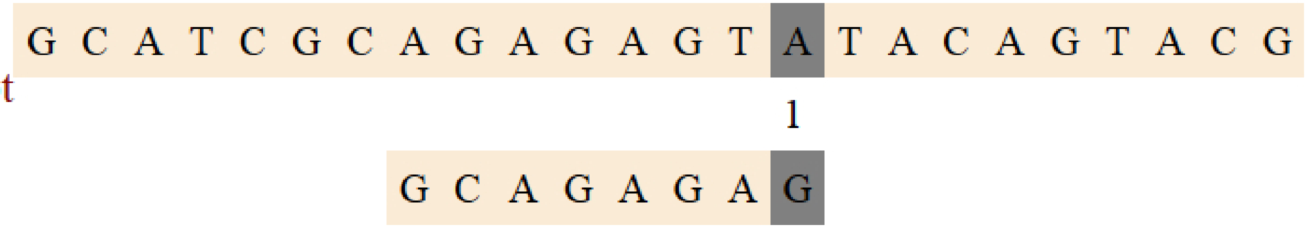
P =

G	C	A	G	A	G	A	G

A	C	G	T
1	6	2	8

= R(z)-table

Fifth attempt



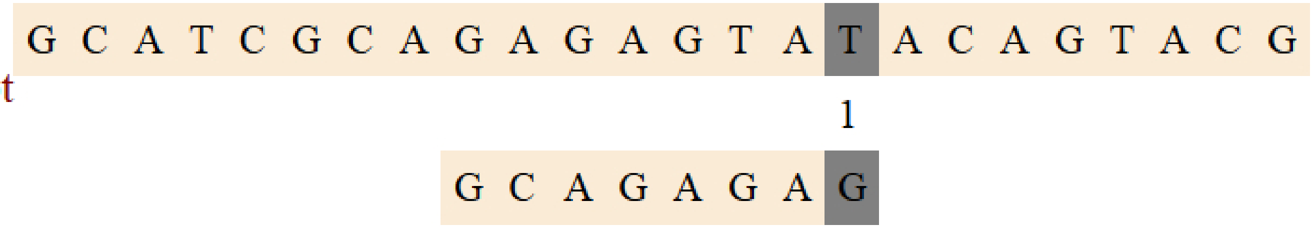
G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Shift by: 1 (*bmBc*[A])

Sixth attempt



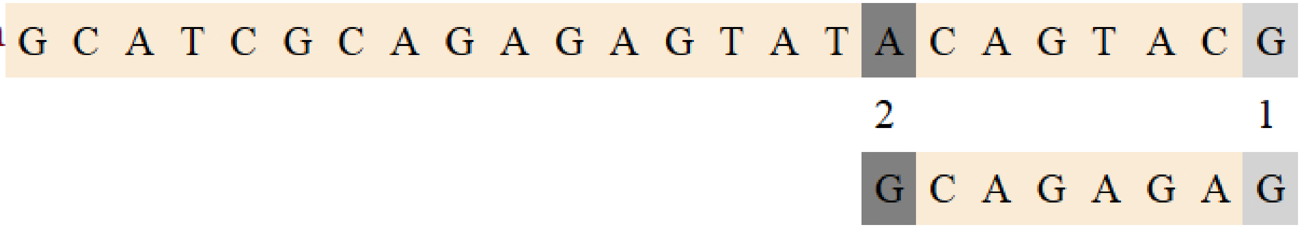
G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Shift by: 8 (*bmBc*[T])

Seventh attempt



G C A T C G C A G A G A G T A T A C A G T A C G

2 1

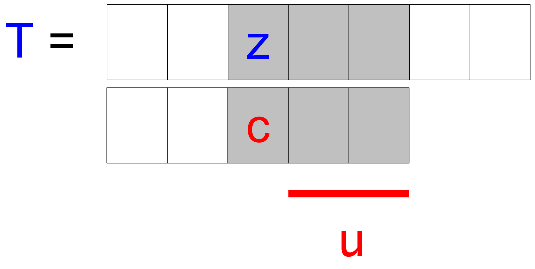
G C A G A G A G

Shift by: 2 (*bmBc*[G])

The Horspool algorithm performs 17 character comparisons on the example.

Boyer-Moore

Idea 2



```
if mismatch after cu on symbol z, then
  k = max( R(z), D(u) )
  k = min( k, |P|-L(u) )
else
    report occurrence of P
    k := |P|-L(u)
shift by k
```

maximum shift

restrict by **lospre**

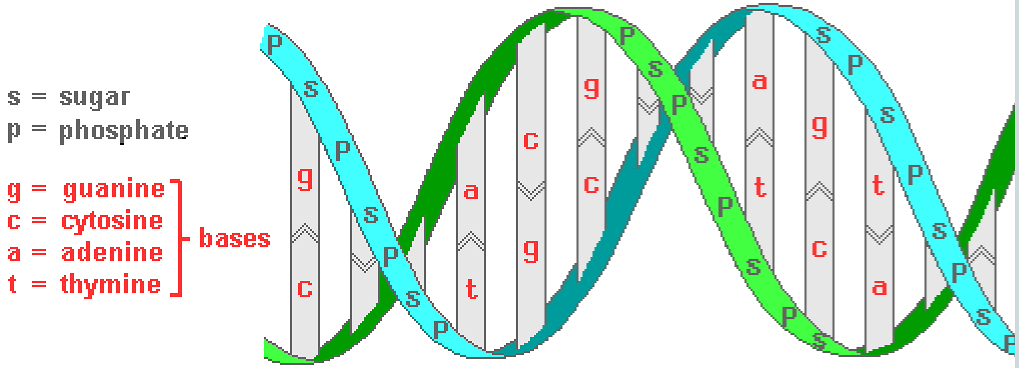
→ D(u) = distance to the next occurrence of u to the left (|P| if not exists)
→ L(u) = **lospre**(u, P)

Lecture 14
Indexed String Search

1. Suffix Trie
2. Suffix Tree
3. Suffix Tree Construction
4. Applications of Suffix Trees

String Search

- → search over DNA sequences
- → huge sequence over C, T, G A (ca. 3.2 billion)
- → no spaces, no tokens....



```
  1 GTACTCATACACACATGTGTGTGTAGGTACTCATGACTAGACACTGATGTCATACT
 57 GCTGGTAGCTCATCACTAGATCGTGACTCACGTGAGGTCGTAGTCTTGTGTACTCA
113 GTGTATCATGCAGCTCGAGTCGTGACTGACTGACTGTCTAGCTGACGTAGCTGCTC
169 GTGACAGGGAGAGAGTCTCTCTATATCATGATCTGCAGTCGGATTGTTGTGACTCA
225 AAGAGACCAGTGTCATCAGTGATGTCAGTCGTATGTACAGTCAGTTTGTCTCTCTA
281 TATCAAGTCTCTCTATATCATGATACACATGTGTGTGTAGACTAGACACTGATGTC
337 ATTGTGTACTCAGTGTCTCATCACTAGATCTCTATATCATGATCTGCAGTCGGTCT
393 CTATATCATGATCTGCAGTCGGGTCTCTCTATATCATTGTGTACTCAGTGTAGTCT
449 CTCTATATCATGATACACATGTGTGTGTAGGTCTCTCTATATCAAGTCTCTCTATA
505 TCATGATTCTCTATATCATGATCTGCAGTCGGTCTCTATATCATGATCTGCAGTCG
561 GGTCTCTCTATATCATTGTGTACTCAGTGTAGTCTCTCTATATCATGATACACATG
617 TGTGTGTAGTTGTGTACTCAGTGTAGTCTCTCTATATCATGATTCTCTATATCATG
673 ATCTGCAGTCGGGTCTCTCTATATCATTGTGTACTCAGTGTAGTCTCTCTATATCA
729 TGATACACATGTGTGTGTAGTTGTGTACTCAGTGTTCTCTATATCATGATCTGCAG
785 TCGGTTGTGTACTCAGTGTAGTCTCTCTATATCATGATACACATGTGTGTGTAGTT
841 GTGTACTCAGTGTAGTCTCTCTATATCATGATGTCTCTCTATATCATTGTGTACTC
897 AGTGTAGTCTCTCTATATCATGATACACATGTGTGTGTAGTGATACACATGTGTGT
```

String Search

→ search over DNA sequences
→ huge sequence over C, T, G A (ca. 3.2 billion)
→ no spaces, no tokens....

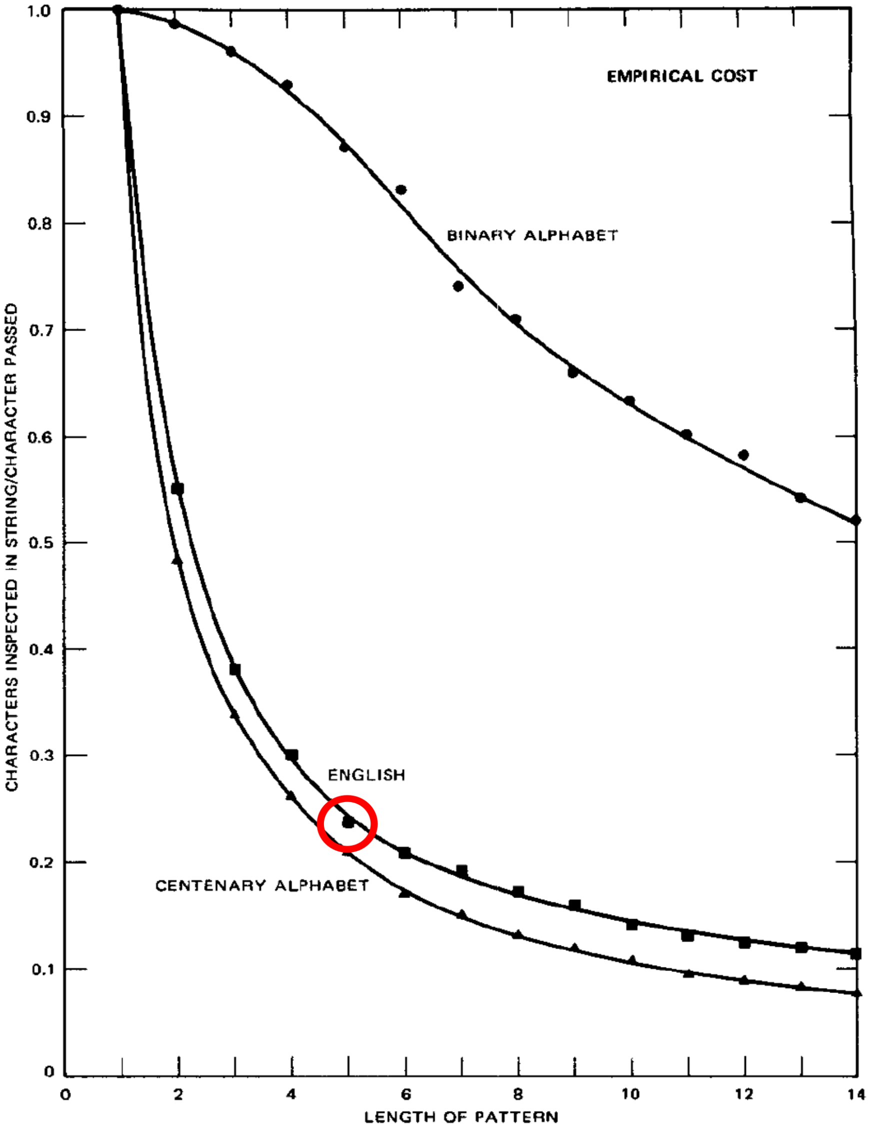
Given
– a long string T (text) of length n
– a short string P (pattern) of length m

Problem 1: find all occurrences of P in T
Problem 2: count #occurrence of P in T

Online Search O(|T|) time with O(|P|) preprocessing
E.g., using *automaton* or *KMP*

→ **sublinear time** using *Horspool* / *Boyer-Moore*
→ average time limit: O(n (log m)/m)

BM – Average Case



To find first occurrence i of an arbitrary 5-letter word in an English text Inspects on average

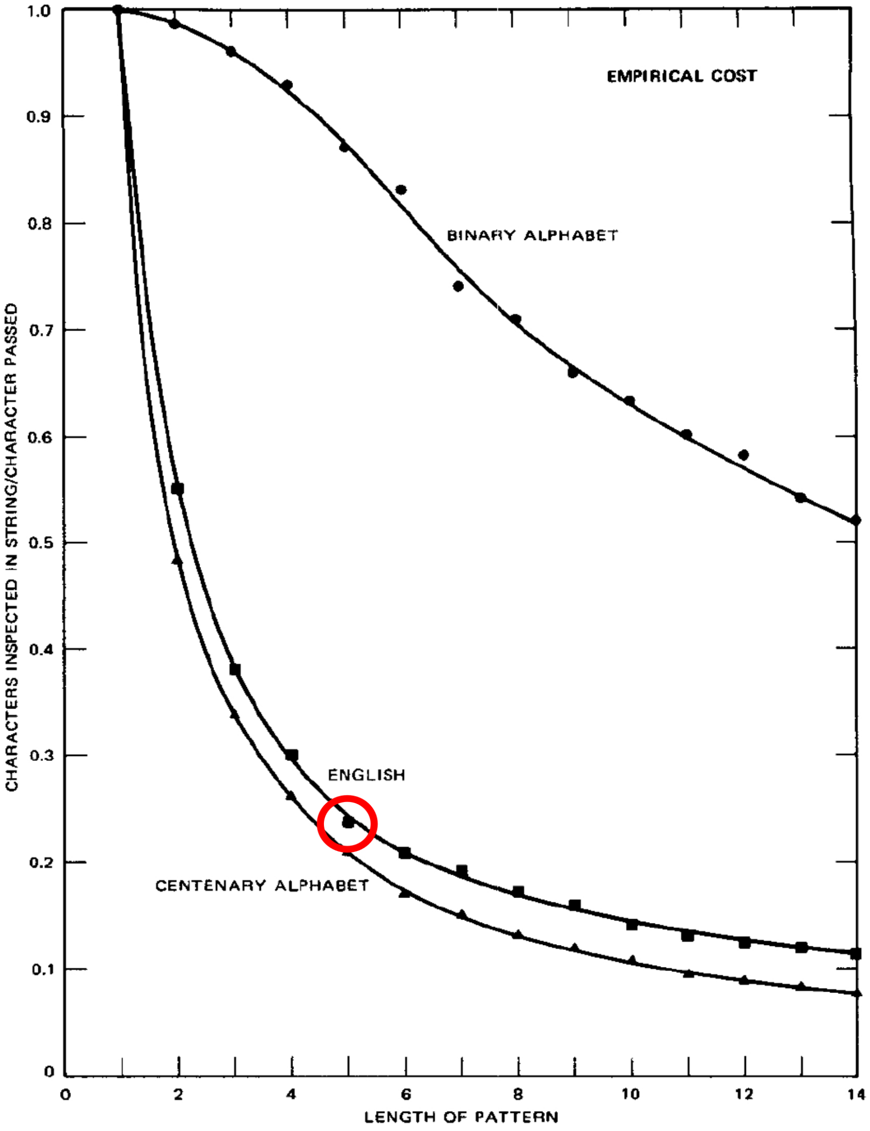
(0.25 * i)

text symbols.

→ **sublinear time** using *Horspool* / *Boyer-Moore*

→ average time **limit**: O(n (log m)/m)

BM – Average Case



To find first occurrence i of an arbitrary 5-letter word in an English text Inspects on average

(0.25 * i)

text symbols.

→ for DNA, 40% of 3.2 billion is still huge

Indexed String Search

Given
– a long string T (text)
– a short string P (pattern)   m = |P|

Problem 1   find all occurrences of P in T
Problem 2   count #occurrence of P in T

Offline Search = **Indexed Search**
= (linear time) preprocessing of T

Highlights → O(m) time   for Problem 1
→ O(m + #occ) time   for Problem 2

Indexed String Search

Given
– a long string T (text)
– a short string P (pattern) $m = |P|$

Problem 1   find all occurrences of P in T
Problem 2   count #occurrence of P in T

Offline Search = **Indexed Search**
= (linear time) preprocessing of T

Highlights
→ $O(m)$ time   for Problem 1
→ $O(m + \#occ)$ time   for Problem 2

Independent of size of text T!!!

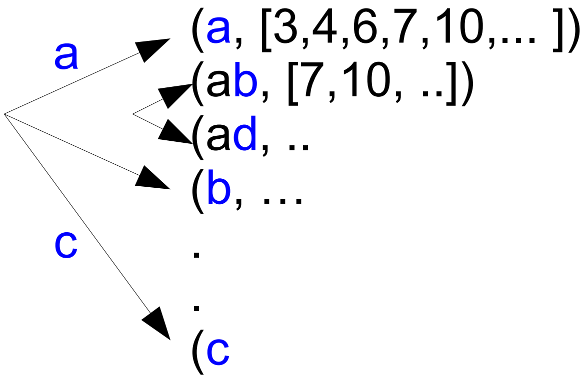
Indexed String Search

Count / Find all occurrences of P in T

Preprocessing (“indexing”) of T is permitted

Naive Solution

1. List all substrings of T, together with their occurrence lists
   (string1, [3,7,21]), (string2, [3,21]), …
2. Lexicographically sort the substrings
3. Record the beginnings of each distinct “next letter” (tree structure)



(a, [3,4,6,7,10,...])
(ab, [7,10, ..])
(ad, ..
(b, …
.
.
(c

Indexed String Search

Count / Find all occurrences of P in T

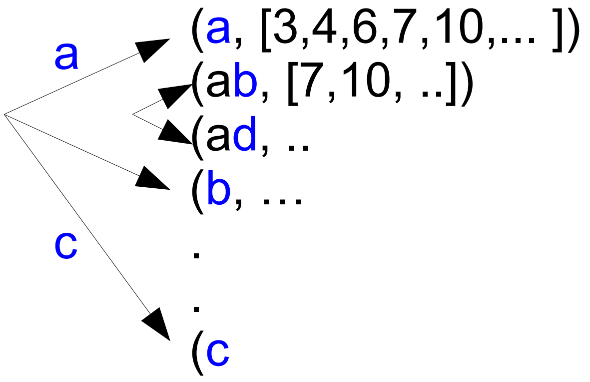
Preprocessing (“indexing”) of T is permitted

Naive Solution

1. List all substrings of T, together with their occurrence lists
   (string1, [3,7,21]), (string2, [3,21]), …

2. Lexicographically sort the substrings

3. Record the beginnings of each distinct “next letter” (tree structure)



(a, [3,4,6,7,10,...])
(ab, [7,10, ..])
(ad, ..
(b, …
.
.
(c

Search occurrences of P:

→ jump to substrings starting with letter P[1]
→ from there, jump to substrings with
next letter P[2]
Etc.
after m jumps, reach (or not) matching substring
with its occurrence list

Indexed String Search

Count / Find all occurrences of P in T

Preprocessing (“indexing”) of T is permitted

Search Time
→ O(m) [good!]

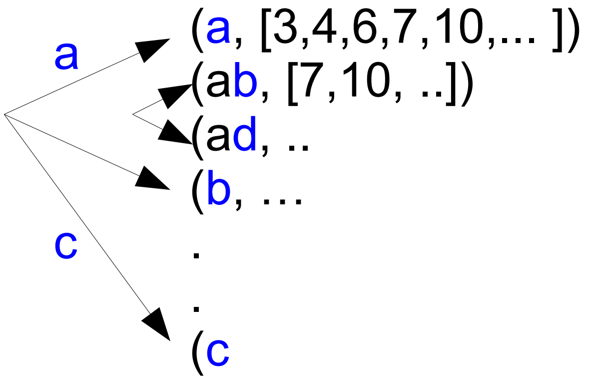
Indexing Time
→ ????

Naive Solution

1. List all substrings of T, together with their occurrence lists
(string1, [3,7,21]), (string2, [3,21]), …

2. Lexicographically sort the substrings

3. Record the beginnings of each distinct “next letter” (tree structure)



Search occurrences of P:

→ jump to substrings starting with letter P[1]
→ from there, jump to substrings with next letter P[2]

Etc.
after m jumps, reach (or not) matching substring with its occurrence list

Indexed String Search

Count / Find all occurrences of P in T

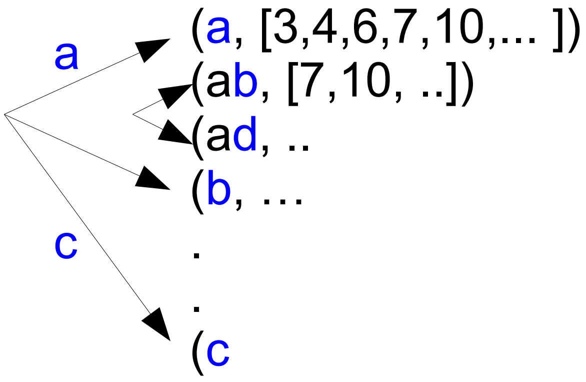
Preprocessing ("indexing") of T is permitted

Search Time
→ O(m) [good!]

Indexing Time
→ exceeds O(n^2)
(sort n^2 substrings)

Naive Solution

1. List all substrings of T, together with their occurrence lists
   (string1, [3,7,21]), (string2, [3,21]), …
2. Lexicographically sort the substrings
3. Record the beginnings of each distinct "next letter" (tree structure)



a

c

(a, [3,4,6,7,10,...])
(ab, [7,10, ..])
(ad, ..
(b, …
.
.
(c

Search occurrences of P:

→ jump to substrings starting with letter P[1]
→ from there, jump to substrings with next letter P[2]
Etc.
after m jumps, reach (or not) matching substring with its occurrence list

1. Suffix Trie

→ Idea: consider all suffixes of text T
i.e., suffix starting at position 1 (= T)
suffix starting at position 2
suffix starting at position 3
etc.

→ arrange suffixes in a “prefix tree” (trie),
with longest common prefixes shared

1. Suffix Trie

→ Idea: consider all suffixes of text T

i.e., suffix starting at position 1 (= T)
suffix starting at position 2
suffix starting at position 3
etc.

→ arrange suffixes in a “prefix tree” (trie), with longest common prefixes shared

---

→ trie datastructure: 1959 by de la Briandais

→ “trie” (Fredkin, 1961), pronounced /ˈtriː/ (as "tree")

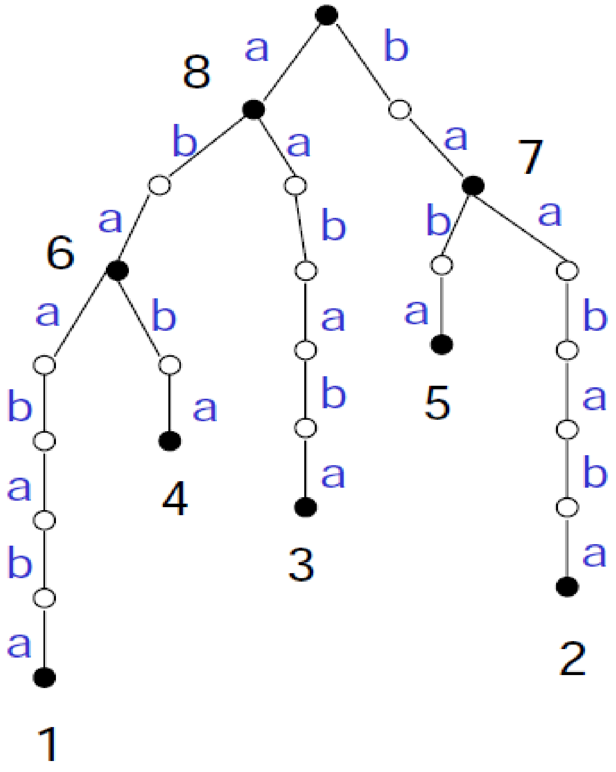
RE**TRIE**VAL

→ to distinguish from “tree” many authors say /ˈtraɪ/ (as "try")

→ aka “digital tree” or “radix tree” or “prefix tree”

1. Suffix Trie

```
     12345678
T  = abaababa
```

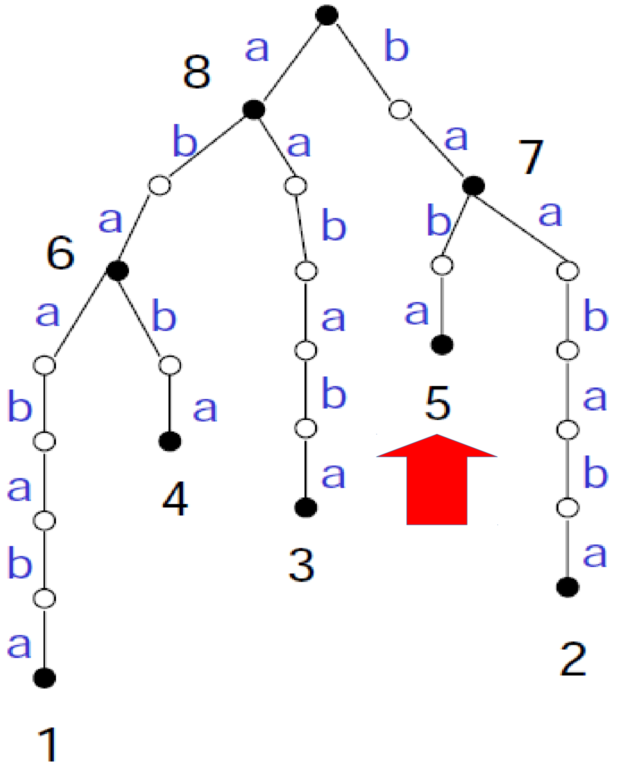


Suffixes
1 abaababa
2 baababa
3 aababa
4 ababa
5 baba
6 aba
7 ba
8 a

Trie of all suffixes of T=abaababa.

1. Suffix Trie

```
     12345678
T  = abaababa
```



Trie of all suffixes of T=abaababa.

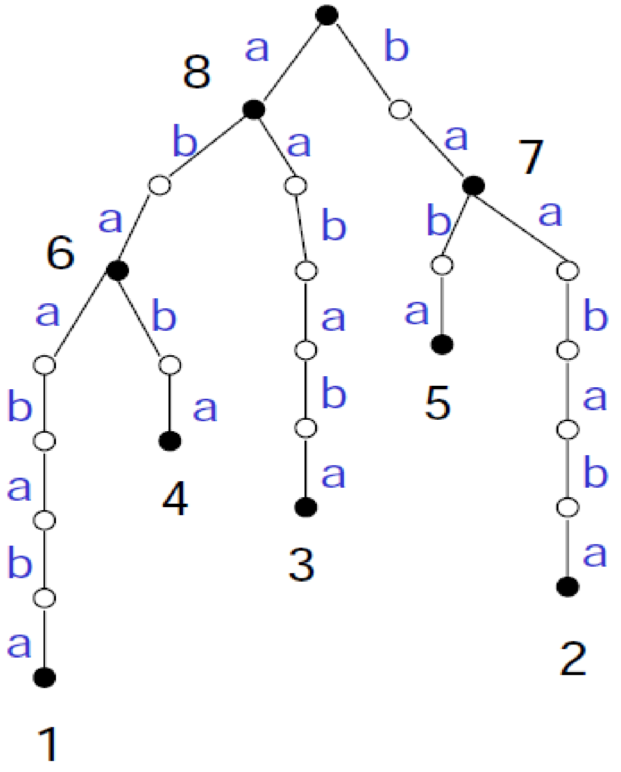
Suffixes	
1	abaababa
2	baababa
3	aababa
4	ababa
5	baba
6	aba
7	ba
8	a

→ black nodes represent suffixes

→ are labeled by the corresponding number of the suffix

1. Suffix Trie

```
    12345678
T = abaababa
```



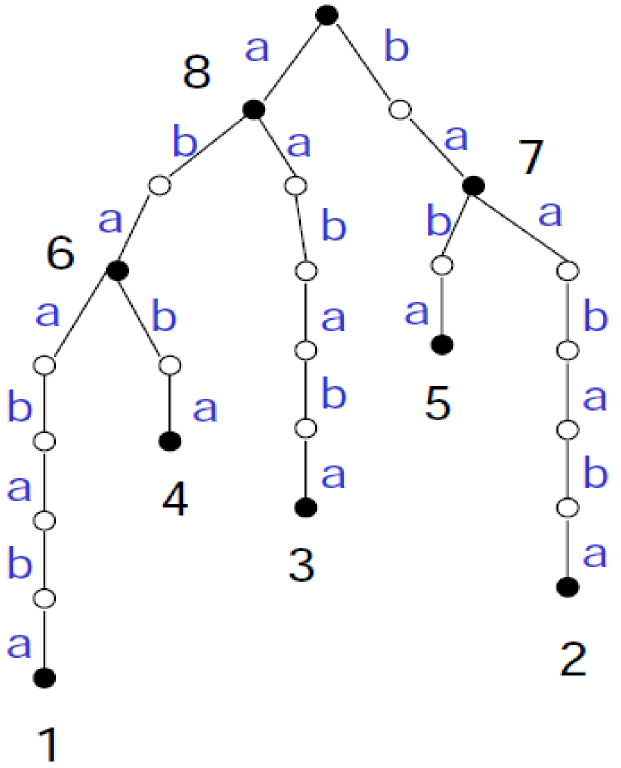
Trie of all suffixes of T=abaababa.

Suffixes	
1	abaababa
2	baababa
3	aababa
4	ababa
5	baba
6	aba
7	ba
8	a

→ how to search for all occurrences of a pattern P?

1. Suffix Trie

```
    12345678
T = abaababa
```



Trie of all suffixes of T=abaababa.

Suffixes

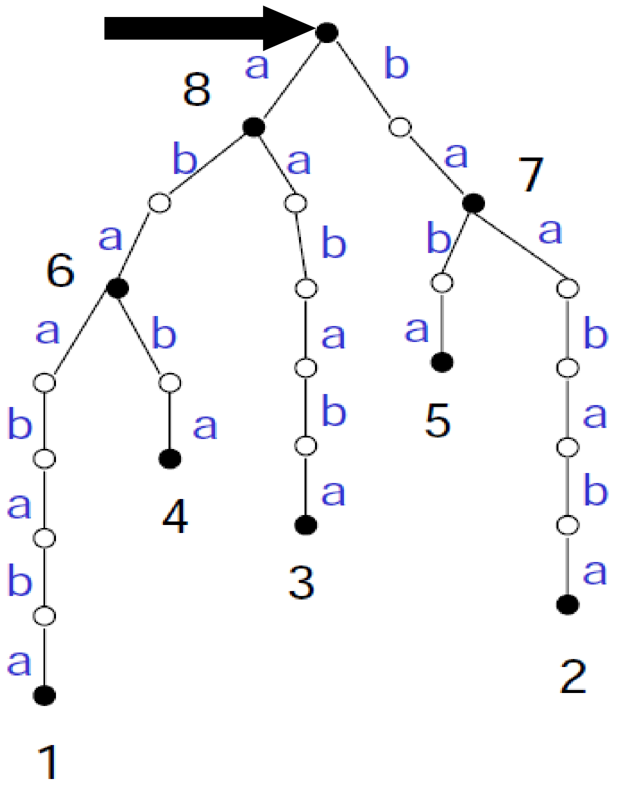
1	abaababa
2	baababa
3	aababa
4	ababa
5	baba
6	aba
7	ba
8	a

→ how to search for all occurrences of a pattern P?

→ starting at the root node follow letter-by-letter wrt P the unique edges in the trie!

1. Suffix Trie

```
    12345678
T = abaababa
```



Trie of all suffixes of T=abaababa.

Suffixes

1	abaababa
2	baababa
3	aababa
4	ababa
5	baba
6	aba
7	ba
8	a

→ how to search for all occurrences of a pattern P?

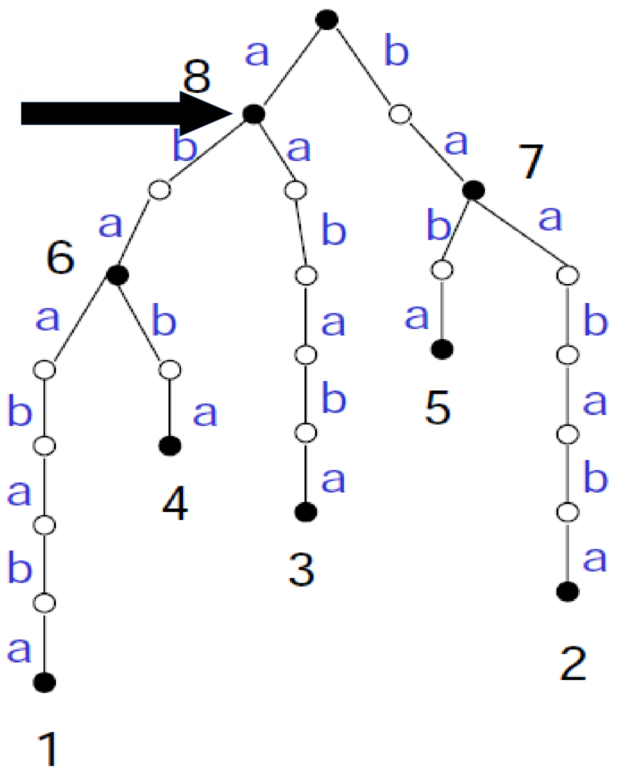
→ starting at the root node follow letter-by-letter wrt P the unique edges in the trie!

P = **a**ba

⬆

1. Suffix Trie

12345678
T = abaababa



Trie of all suffixes of T=abaababa.

	Suffixes
1	abaababa
2	baababa
3	aababa
4	ababa
5	baba
6	aba
7	ba
8	a

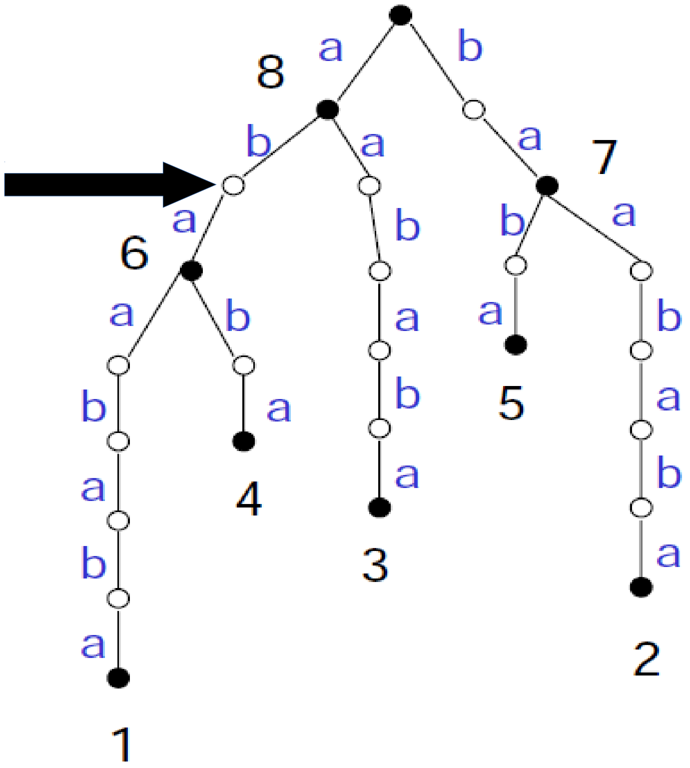
→ how to search for all occurrences of a pattern P?

→ starting at the root node follow letter-by-letter wrt P the unique edges in the trie!

P = aba

1. Suffix Trie

```
    12345678
T = abaababa
```



Trie of all suffixes of T=abaababa.

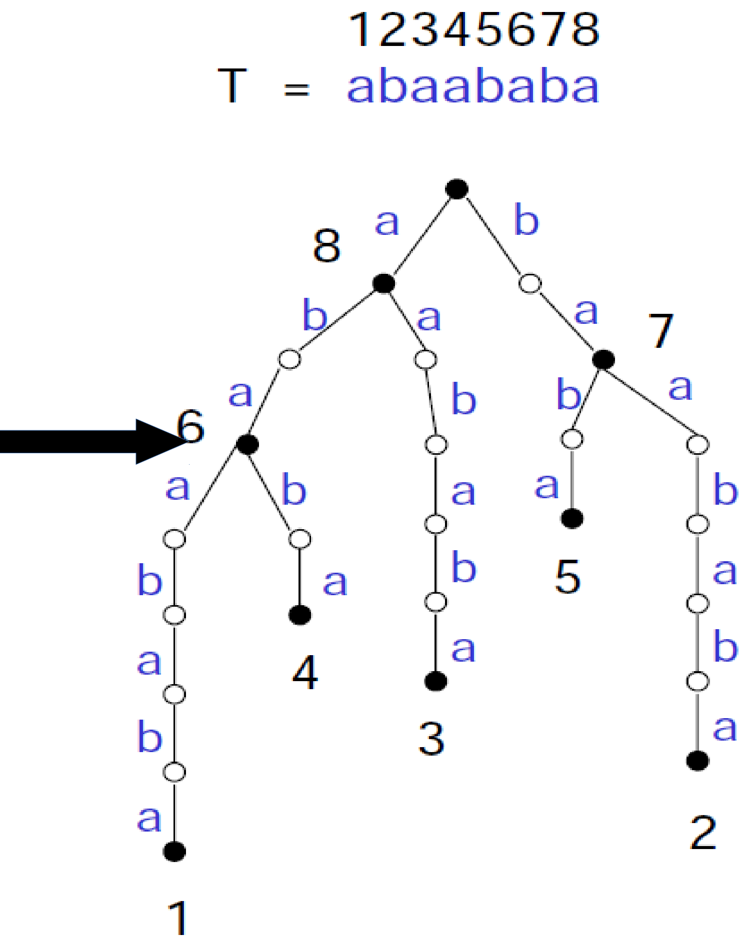
Suffixes	
1	abaababa
2	baababa
3	aababa
4	ababa
5	baba
6	aba
7	ba
8	a

→ how to search for all occurrences of a pattern P?

→ starting at the root node follow letter-by-letter wrt P the unique edges in the trie!

P = ab**a**

1. Suffix Trie



Trie of all suffixes of T=abaababa.

Suffixes	
1	abaababa
2	baababa
3	aababa
4	ababa
5	baba
6	aba
7	ba
8	a

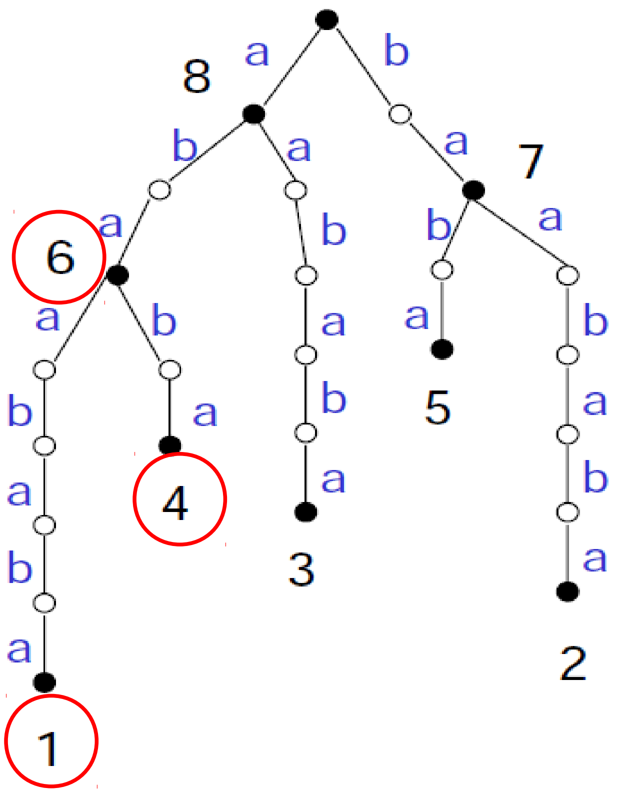
→ how to search for all occurrences of a pattern P?

→ starting at the root node follow letter-by-letter wrt P the unique edges in the trie!

P = aba

1. Suffix Trie

```
    12345678
T = abaababa
```



3 matches of P = “aba”

Suffixes
1 abaababa
2 baababa
3 aababa
4 ababa
5 baba
6 aba
7 ba
8 a

→ how to search for all occurrences of a pattern P?

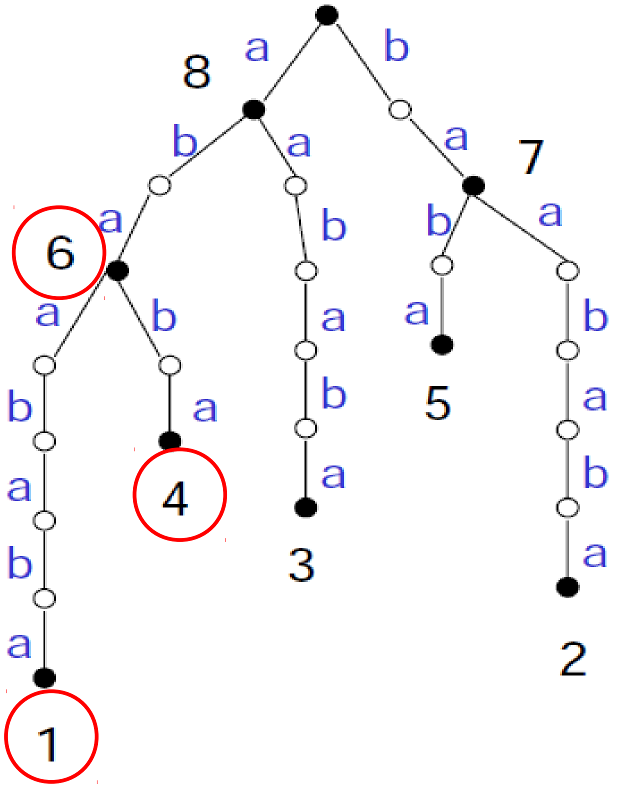
→ starting at the root node follow letter-by-letter wrt P the unique edges in the trie!

P = aba

⬆

1. Suffix Trie

12345678
T = abaababa



3 matches of P = “aba”

Suffixes
1 abaababa
2 baababa
3 aababa
4 ababa
5 baba
6 aba
7 ba
8 a

→ O(m) **count time**

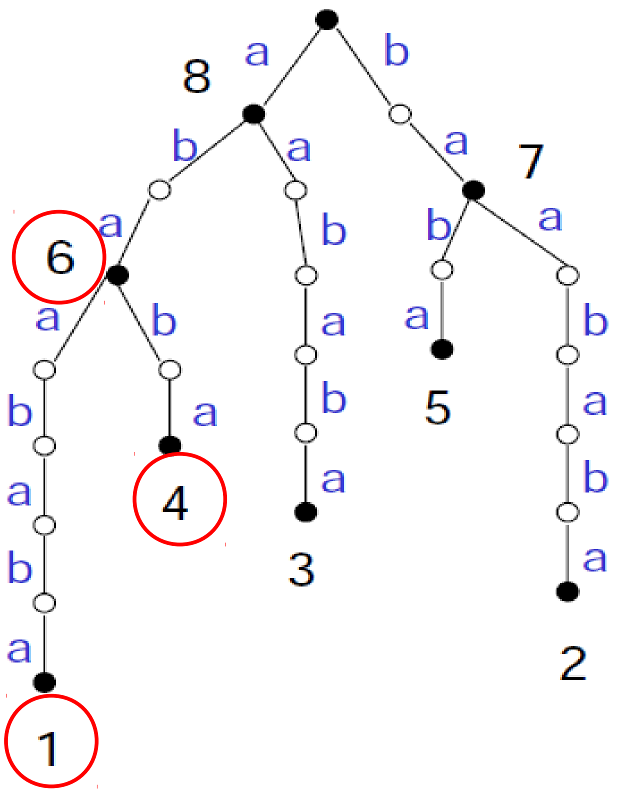
If we can count #black nodes of a subtree in constant time.

→ O(m + #occ) **retrieval time**

If we can iterate through the leaves of a subtree with constant delay

1. Suffix Trie

```
     12345678
T  = abaababa
```



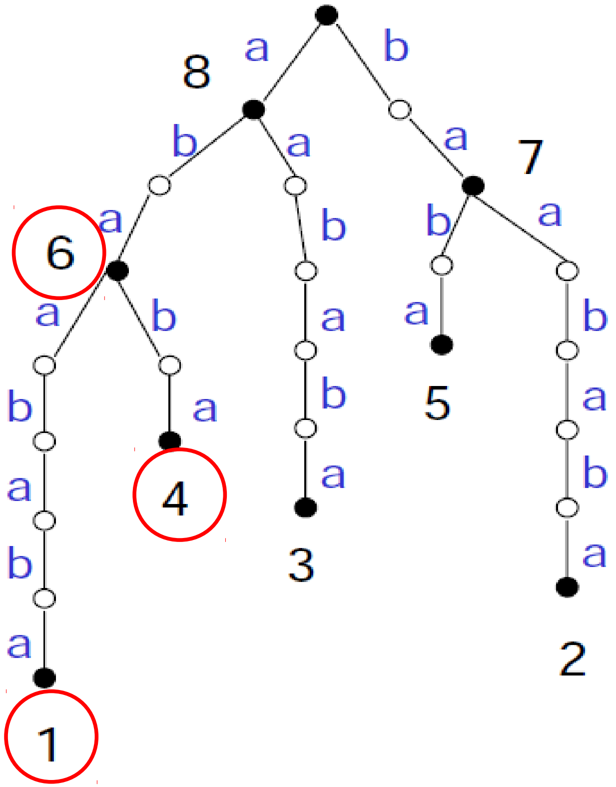
3 matches of P = “aba”

Suffixes
1 abaababa
2 baababa
3 aababa
4 ababa
5 baba
6 aba
7 ba
8 a

→ **Indexing time?**

1. Suffix Trie

```
     12345678
T  = abaababa
```



3 matches of P = “aba”

Suffixes
1 abaababa
2 baababa
3 aababa
4 ababa
5 baba
6 aba
7 ba
8 a

→ **Indexing time?**

No sorting, but

→ still quadratic in n, i.e., O(n^2) :-(

→ the size (#nodes) of trie is O(n^2)

END
Lecture 14