Algorithms on Strings

Advanced Programming and Algorithmic Design

Alberto Casagrande
Email: `acasagrande@units.it`

a.a. 2018/2019

String-Matching

Basic Definitions and Properties

An alphabet is a set of symbols e.g., $\{0, 1\}$ or $\{a, \ldots, z\}$

A string $S[1 \ldots |S|]$ is a finite sequence of symbols in an alphabet

Σ^* is the set of all strings built on Σ

ϵ is the empty string and belongs to Σ^*

E.g.,

"This is a string" or "This_is,a+string" or ϵ

Basic Definitions and Properties (Cont'd)

If $x \in \Sigma^*$ and $y \in \Sigma^*$, then $xy \in \Sigma^*$ is their concatenation

If $y = xw$:

- x is a prefix of y and we write $x \sqsubset y$
- w is a suffix of y and we write $x \sqsupset y$

If $x \in \Sigma^*$ and $q \in \mathbb{N}$, x_q will be the x's prefix of length q

Basic Definitions and Properties (Cont'd)

If $x \in \Sigma^*$ and $y \in \Sigma^*$, then $xy \in \Sigma^*$ is their concatenation

If $y = xw$:

- x is a prefix of y and we write $x \sqsubset y$
- w is a suffix of y and we write $x \sqsupset y$

If $x \in \Sigma^*$ and $q \in \mathbb{N}$, x_q will be the x's prefix of length q

Lemma (Overlapping-suffix lemma)

Let x, y, and w s.t. $x \sqsupset w$ and $y \sqsupset w$.

- *if $|x| > |y|$, then $y \sqsupset x$*
- *if $|x| = |y|$, then $y = x$*

Basic Definitions and Properties (Cont'd)

Given :

- a finite alphabet Σ

c	a	b	a	b	b

Basic Definitions and Properties (Cont'd)

Given :

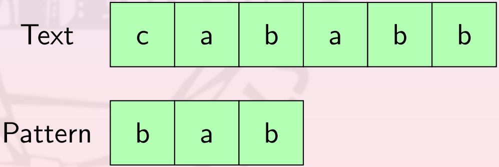
- a finite alphabet Σ
- a text $T[1 \ldots n]$

Text	c	a	b	a	b	b

Basic Definitions and Properties (Cont’d)

Given :

- a finite alphabet Σ
- a text $T[1 \ldots n]$
- a pattern $P[1 \ldots m]$ with $m \leq n$



Text	c	a	b	a	b	b
Pattern	b	a	b			

Basic Definitions and Properties (Cont'd)

Given :

- a finite alphabet Σ
- a text $T[1 \ldots n]$
- a pattern $P[1 \ldots m]$ with $m \leq n$

P occurs with shift s in T means $T[s+1 \ldots s+m] = P$

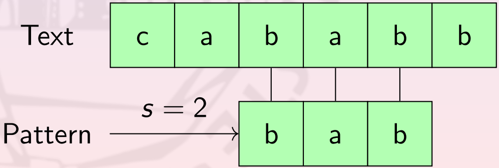
Text	c	a	b	a	b	b
Pattern	b	a	b			

Basic Definitions and Properties (Cont'd)

Given :

- a finite alphabet Σ
- a text $T[1 \dots n]$
- a pattern $P[1 \dots m]$ with $m \leq n$

P occurs with shift s in T means $T[s+1 \dots s+m] = P$

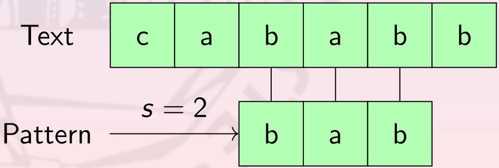


Basic Definitions and Properties (Cont'd)

Given :

- a finite alphabet Σ
- a text $T[1 \ldots n]$
- a pattern $P[1 \ldots m]$ with $m \leq n$

P occurs with shift s in T means $T[s+1 \ldots s+m] = P$



Text c a b a b b

$s = 2$

Pattern b a b

If P occurs with shift s in T, then s is a valid shift

The String-Matching Problem

Requires to find all the valid shifts for P in T

The String-Matching Problem

Requires to find all the valid shifts for P in T

E.g., For

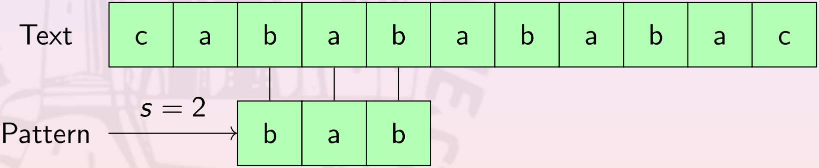
Text	c	a	b	a	b	a	b	a	b	a	c
Pattern	b	a	b								

We get $\{$

The String-Matching Problem

Requires to find all the valid shifts for P in T

E.g., For



Text c a b a b a b a b a c

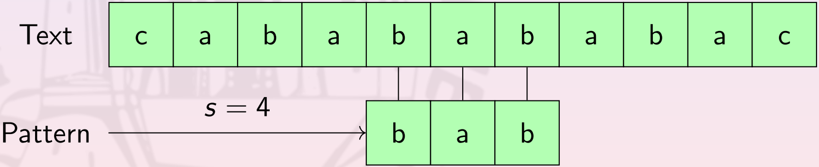
Pattern $\xrightarrow{s=2}$ b a b

We get $\{2,$

The String-Matching Problem

Requires to find all the valid shifts for P in T

E.g., For



Text

c	a	b	a	b	a	b	a	b	a	c

Pattern $s = 4$

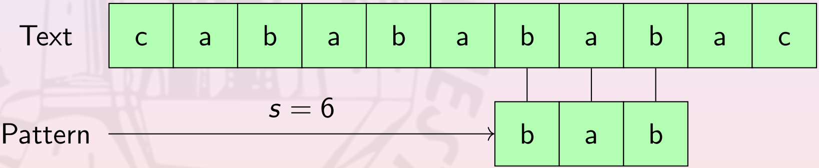
b	a	b

We get $\{2, 4,$

The String-Matching Problem

Requires to find all the valid shifts for P in T

E.g., For



Text: c a b a b a b a b a c

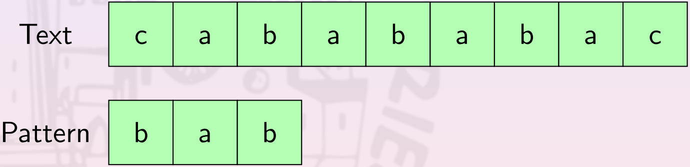
Pattern: b a b

$s = 6$

We get $\{2, 4, 6\}$

A Naïve Solution to String-Matching Problem

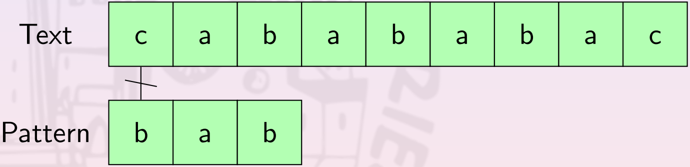
To solve the problem, we may try all the possible shifts for P in T



Text	c	a	b	a	b	a	b	a	c
Pattern	b	a	b						

A Naïve Solution to String-Matching Problem

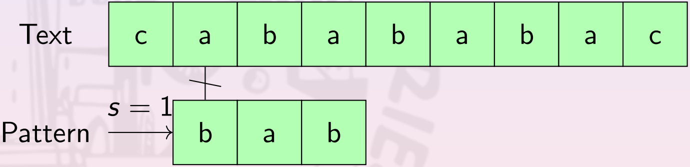
To solve the problem, we may try all the possible shifts for P in T



Text	c	a	b	a	b	a	b	a	c
Pattern	b	a	b						

A Naïve Solution to String-Matching Problem

To solve the problem, we may try all the possible shifts for P in T

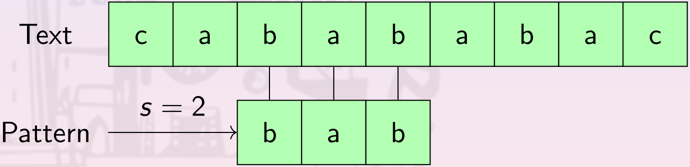


Text: c a b a b a b a c

Pattern $s = 1$: b a b

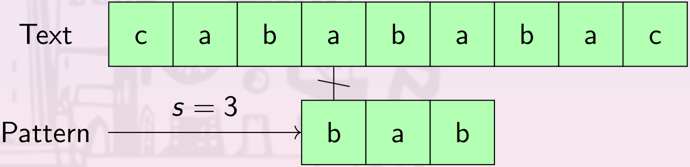
A Naïve Solution to String-Matching Problem

To solve the problem, we may try all the possible shifts for P in T



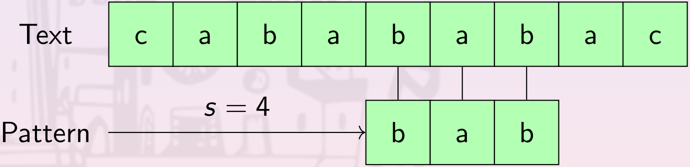
A Naïve Solution to String-Matching Problem

To solve the problem, we may try all the possible shifts for P in T



A Naïve Solution to String-Matching Problem

To solve the problem, we may try all the possible shifts for P in T



Text

c	a	b	a	b	a	b	a	c

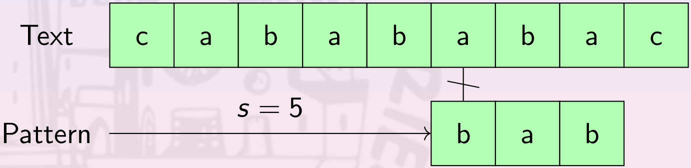
$s = 4$

Pattern

b	a	b

A Naïve Solution to String-Matching Problem

To solve the problem, we may try all the possible shifts for P in T



Text

c	a	b	a	b	a	b	a	c

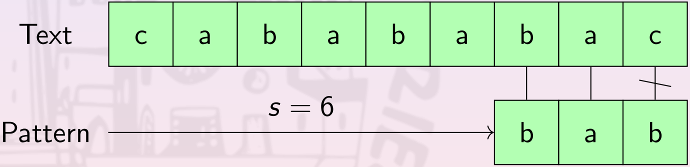
$s = 5$

Pattern

b	a	b

A Naïve Solution to String-Matching Problem

To solve the problem, we may try all the possible shifts for P in T



Text c a b a b a b a c

$s = 6$

Pattern b a b

Naïve Solution: Pseudo-Code

```
def NAIVE_STRING_MATCHING(T, P):
  valid ← []
  for s ← 1 upto |T|−|P|+1:
    i ← 1
    while i ≤ |P| and T[i+s] = P[i]:
      i ← i+1
    endwhile

    if i > |P|:
      valid.append(s)
    endif
  endfor

  return valid
enddef
```

Naïve Solution: Complexity

A match is tested for all the possible $|T| - |P| + 1$ shifts

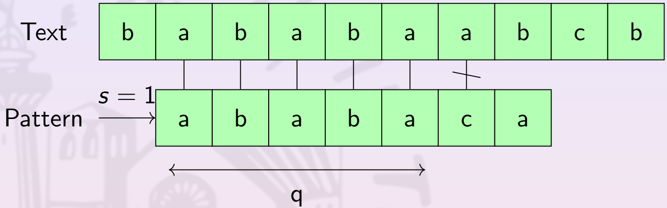
Each match test costs $O(|P|)$

Since $|P| \leq |T|$, the overall complexity is $O(|P| * |T|)$

E.g., to face a worst-case-scenario consider:

Text	a	a	a	a	a	a	a	a	a
Pattern	a	a	a	a	b				

A Better Idea



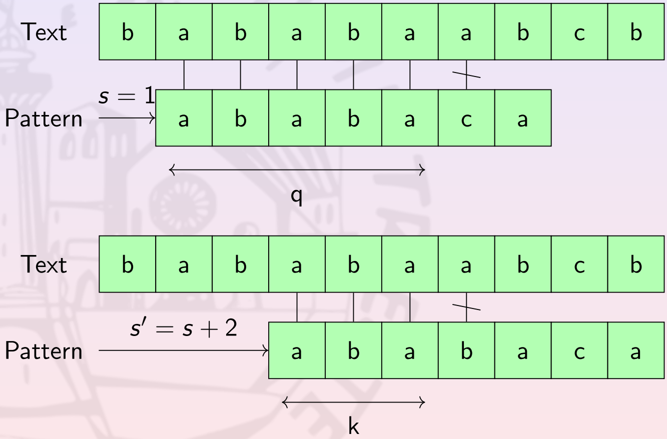
Text b a b a b a a b c b

$s = 1$

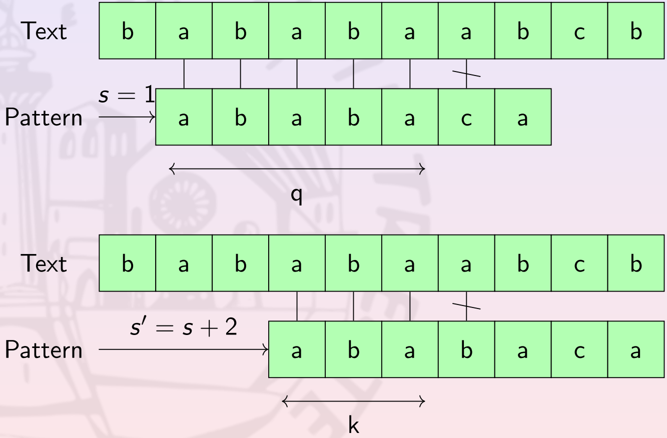
Pattern a b a b a c a

q

A Better Idea



A Better Idea



Text: b a b a b a a b c b

$s = 1$

Pattern: a b a b a c a

q

Text: b a b a b a a b c b

$s' = s + 2$

Pattern: a b a b a c a

k

Thus, $P_k \sqsupset P_q$ beacuse $P_q \sqsupset T[2..q+1]$ and $P_k \sqsupset T[2..q+1]$

The Prefix Function

The prefix function for P is defined as

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

P_5	a	b	a	b	a

The Prefix Function

The prefix function for P is defined as

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

$\pi[q]$ is the longest prefix of P that is a proper suffix of P_q

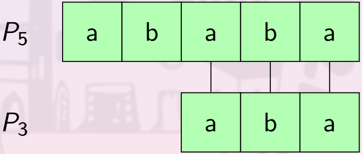
P_5	a	b	a	b	a

The Prefix Function

The prefix function for P is defined as

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

$\pi[q]$ is the longest prefix of P that is a proper suffix of P_q



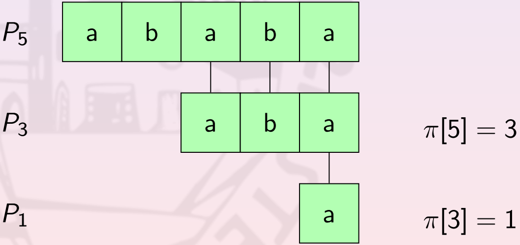
$\pi[5] = 3$

The Prefix Function

The prefix function for P is defined as

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

$\pi[q]$ is the longest prefix of P that is a proper suffix of P_q



P_5 a b a b a

P_3 a b a $\qquad \pi[5] = 3$

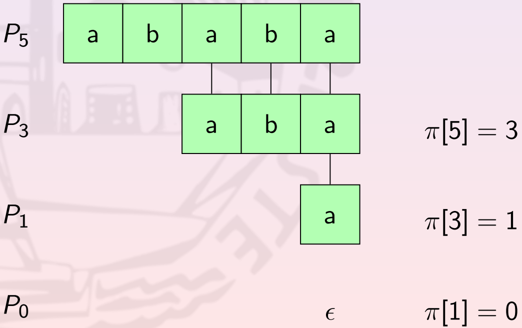
P_1 a $\qquad \pi[3] = 1$

The Prefix Function

The prefix function for P is defined as

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

$\pi[q]$ is the longest prefix of P that is a proper suffix of P_q



P_5 a b a b a

P_3 a b a $\pi[5] = 3$

P_1 a $\pi[3] = 1$

P_0 ϵ $\pi[1] = 0$

Computing the Prefix Function

Let $\pi^*[q]$ be $\{\pi[q], \pi^2[q], \ldots, \pi^{(t)}[q]\}$

Lemma (Prefix-function iteration lemma)

$\pi^*[q] = \{k : k < q \textit{ and } P_k \sqsupset P_q\}$

Lemma

If $\pi[q] > 0$, *then* $\pi[q] - 1 \in \pi^*[q-1]$

Let E_q be $\{k \in \pi^*[q] : P[k+1] = P[q+1]\}$

Theorem

$$\pi[q] = \begin{cases} 0 & \textit{if } E_{q-1} = \emptyset \\ 1 + \max\{k \in E_{q-1}\} & \textit{otherwise} \end{cases}$$

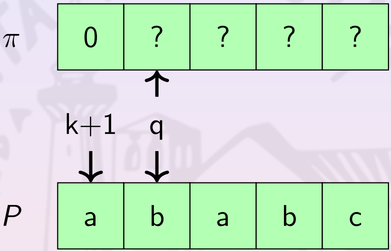
Computing the Prefix Function: Pseudo-Code

```
def COMPUTE_PREFIX_FUNCTION(P):
   $\pi \leftarrow$ INIT_ARRAY(|P|)
   $\pi$[1] $\leftarrow$ 0
   k $\leftarrow$ 0
   for q $\leftarrow$ 2 upto |P|:
      while k > 0 and P[k+1] $\neq$ P[q]:
         k = $\pi$[k]
      endwhile

      if P[k+1] = P[q]:
         k = k + 1
      $\pi$[q] $\leftarrow$ k
   endfor

   return $\pi$
enddef
```

Computing the Prefix Function: an Example



π	0	?	?	?	?
P	a	b	a	b	c

k+1 q

At the begin of each **for**-loop iteration, $k = \pi[q-1]$

P_k is the largest proper suffix of P_{q-1} which is also a prefix for it i.e., $P_k \sqsupset P_{q-1}$ and $P_k \sqsubset P_{q-1}$

The initialization sets

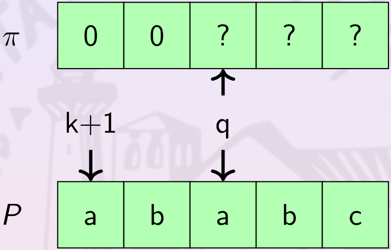
- $\pi[1] = 0$
- $q = 2$
- $k = 0$

Then no **while**-loop iterations

Since $P[k+1] \neq P[q]$, $P_{k+1} \not\sqsupset P_q$ and k is not updated

$\pi[q] \leftarrow 0$

Computing the Prefix Function: an Example



π	0	0	?	?	?
P	a	b	a	b	c

At the begin of each **for**-loop iteration, $k = \pi[q-1]$

P_k is the largest proper suffix of P_{q-1} which is also a prefix for it i.e., $P_k \sqsupset P_{q-1}$ and $P_k \sqsubset P_{q-1}$

When $q = 3$:

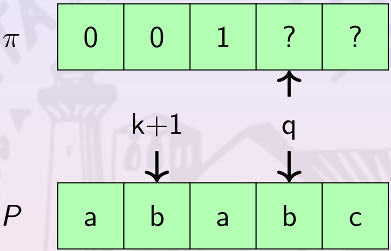
- $k = 0$
- $P[k+1] = P[q]$

Then no **while**-loop iterations

Since $P[k+1] = P[q]$, $P_{k+1} \sqsupset P_q$ and k is updated to 1

$\pi[q] \leftarrow 1$

Computing the Prefix Function: an Example



π	0	0	1	?	?
P	a	b	a	b	c

At the begin of each **for**-loop iteration, $k = \pi[q-1]$

P_k is the largest proper suffix of P_{q-1} which is also a prefix for it i.e., $P_k \sqsupset P_{q-1}$ and $P_k \sqsubset P_{q-1}$

When $q = 4$:

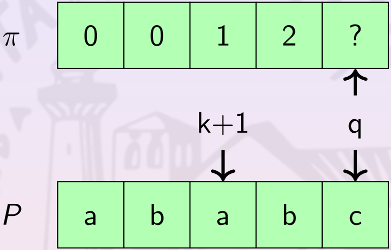
- $k = 1$
- $P[k+1] = P[q]$

Then no **while**-loop iterations

Since $P[k+1] = P[q]$, $P_{k+1} \sqsupset P_q$ and k is updated to 2

$\pi[q] \leftarrow 2$

Computing the Prefix Function: an Example



π	0	0	1	2	?
P	a	b	a	b	c

At the begin of each **for**-loop iteration, $k = \pi[q-1]$

P_k is the largest proper suffix of P_{q-1} which is also a prefix for it i.e., $P_k \sqsupset P_{q-1}$ and $P_k \sqsubset P_{q-1}$

When $q = 5$:

- $k = 2$
- $P[k+1] \neq P[q]$

Since $P[k+1] \neq P[q]$, the 2nd largest prefix-suffix P_{q-1} is computed i.e., $\pi[k]$ and k is updated to 0

Since $P[k+1] \neq P[q]$, k is not updated

$\pi[q] \leftarrow 0$

Computing the Prefix Function: an Example

π	0	0	1	2	0
P	a	b	a	b	c

At the begin of each **for**-loop iteration, $k = \pi[q-1]$

P_k is the largest proper suffix of P_{q-1} which is also a prefix for it i.e., $P_k \sqsupset P_{q-1}$ and $P_k \sqsubset P_{q-1}$

The Prefix Function: Complexity

The **while**-loop condition holds only if $k > 0$

However, each iteration of the **while**-loop decreases k

k is initialized to 0 and is increased in the **for**-loop

So, the **while**-loop can be repeated $|P| - 1$ times at most

The overall asymptotic complexity is $\Theta(|P|)$

The Knuth-Morris-Pratt Algorithm

Once a mismatch has been identified after q matches

The algorithm uses the prefix function to avoid $\pi[q]$ useless character comparisons

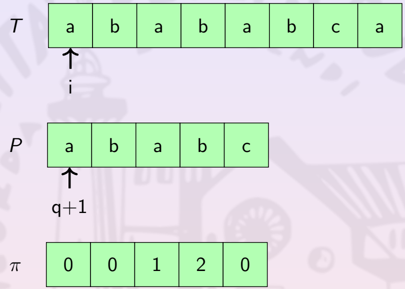
The Knuth-Morris-Pratt Algorithm: Pseudo-Code

```
def KMP(T,P):
  valid = []
  $\pi \leftarrow$ COMPUTE_PREFIX_FUNCTION(P)
  q $\leftarrow$ 0
  for i $\leftarrow$ 1 upto |T|:
    while q > 0 and P[q+1] $\neq$ T[i]:
      q = $\pi$[q]
    endwhile

    if P[q+1] = T[i]:
      q = q + 1
    if q = |P|:
      valid.append(i−q+1)
      q = $\pi$[q]
    endif
  endfor

  return valid
enddef
```

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a
P	a	b	a	b	c			
π	0	0	1	2	0			

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

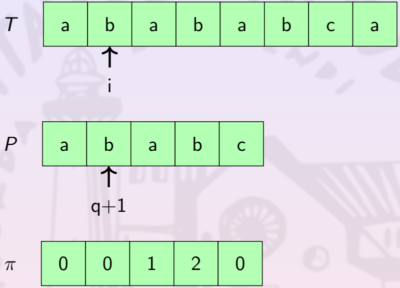
The initialization sets

- $i = 1$
- $q = 0$

Then no **while**-loop iterations

Since $P[q+1] = T[i]$, $P_{q+1} \sqsupset T_i$ and q is updated to 1

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a

P	a	b	a	b	c

π	0	0	1	2	0

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

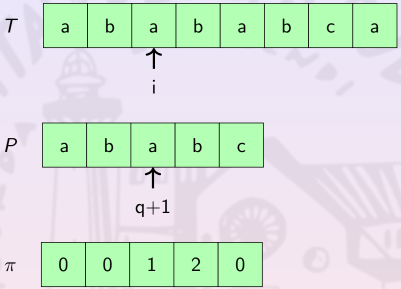
When $i = 2$:

- $q = 1$
- $P[q+1] = T[i]$

Then no **while**-loop iterations

Since $P[q+1] = T[i]$, $P_{q+1} \sqsupset T_i$ and q is updated to 2

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a
			↑ i					
P	a	b	a	b	c			
			↑ q+1					
π	0	0	1	2	0			

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

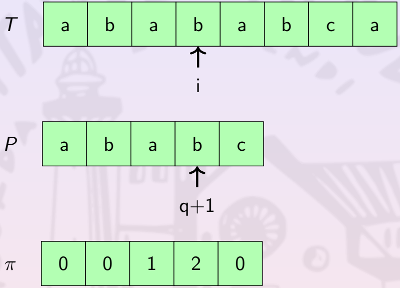
When $i = 3$:

- $q = 2$
- $P[q+1] = T[i]$

Then no **while**-loop iterations

Since $P[q+1] = T[i]$, $P_{q+1} \sqsupset T_i$ and q is updated to 3

The Knuth-Morris-Pratt Algorithm: an Example



T								
	a	b	a	b	a	b	c	a

P					
	a	b	a	b	c

π					
	0	0	1	2	0

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

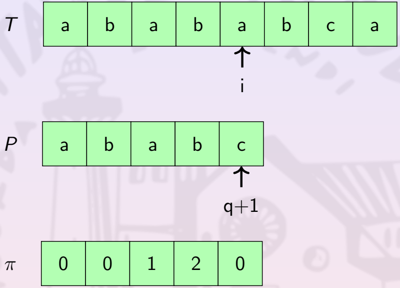
When $i = 4$:

- $q = 3$
- $P[q+1] = P[i]$

Then no **while**-loop iterations

Since $P[q+1] = T[i]$, $P_{q+1} \sqsupset T_i$ and q is updated to 4

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a
P	a	b	a	b	c			
π	0	0	1	2	0			

When $i = 5$:

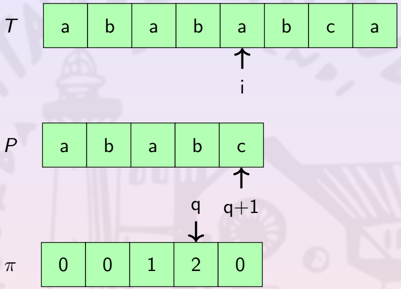
- $q = 4$
- $P[q+1] \neq T[i]$

Since $P[q+1] \neq T[i]$,

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a
P	a	b	a	b	c			
π	0	0	1	2	0			

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

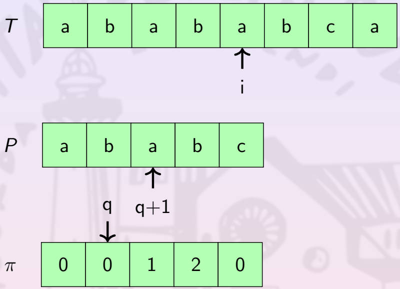
P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

When $i = 5$:

- $q = 4$
- $P[q+1] \neq T[i]$

Since $P[q+1] \neq T[i]$, the 2nd largest prefix-suffix P_q is computed i.e., $\pi[q]$ and

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a
P	a	b	a	b	c			
π	0	0	1	2	0			

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

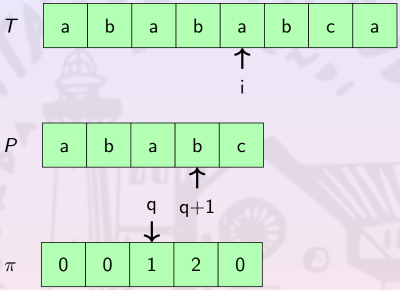
P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

When $i = 5$:

- $q = 4$
- $P[q+1] \neq T[i]$

Since $P[q+1] \neq T[i]$, the 2nd largest prefix-suffix P_q is computed i.e., $\pi[q]$ and q is updated to 2

The Knuth-Morris-Pratt Algorithm: an Example



	1	2	3	4	5	6	7	8
T	a	b	a	b	a	b	c	a
P	a	b	a	b	c			
π	0	0	1	2	0			

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

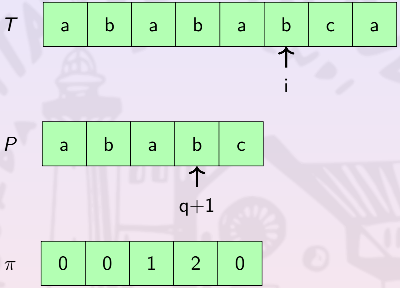
When $i = 5$:

- $q = 4$
- $P[q+1] \neq T[i]$

Since $P[q+1] \neq T[i]$, the 2nd largest prefix-suffix P_q is computed i.e., $\pi[q]$ and q is updated to 2

Since $P[q+1] = T[i]$, q is updated to 3

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a

P	a	b	a	b	c

π	0	0	1	2	0

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

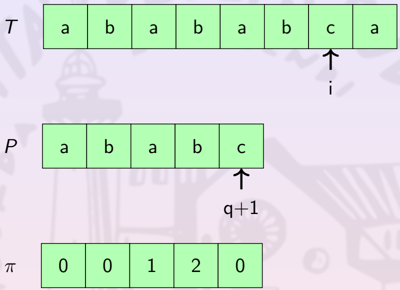
When $i = 6$:

- $q = 3$
- $P[q+1] = T[i]$

Then no **while**-loop iterations

Since $P[q+1] = T[i]$, $P_{q+1} \sqsupset T_i$ and q is updated to 4

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a
							i	

P	a	b	a	b	c
					q+1

π	0	0	1	2	0

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

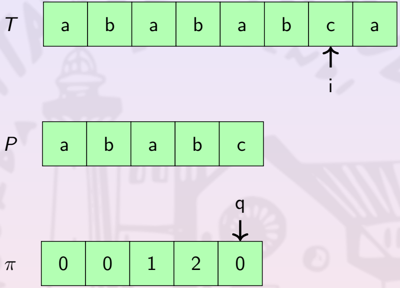
When $i = 7$:

- $q = 4$
- $P[q+1] = T[i]$

Then no **while**-loop iterations

Since $P[q+1] = T[i]$, $P_{q+1} \sqsupset T_i$ and

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a
P	a	b	a	b	c			
π	0	0	1	2	0			

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

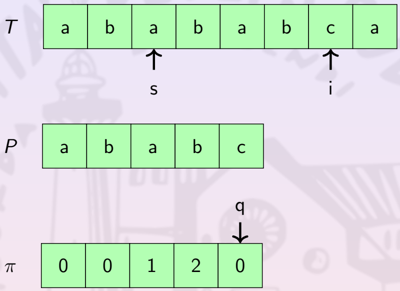
When $i = 7$:

- $q = 4$
- $P[q+1] = T[i]$

Then no **while**-loop iterations

Since $P[q+1] = T[i]$, $P_{q+1} \sqsupset T_i$ and q is updated to 5

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a
P	a	b	a	b	c			
π	0	0	1	2	0			

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

When $i = 7$:

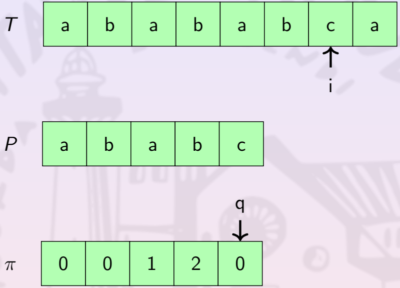
- $q = 4$
- $P[q+1] = T[i]$

Then no **while**-loop iterations

Since $P[q+1] = T[i]$, $P_{q+1} \sqsupset T_i$ and q is updated to 5

Since $q = |P|$, $s = i - q + 1$ is a valid shift

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a
P	a	b	a	b	c			
π	0	0	1	2	0			

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

When $i = 7$:

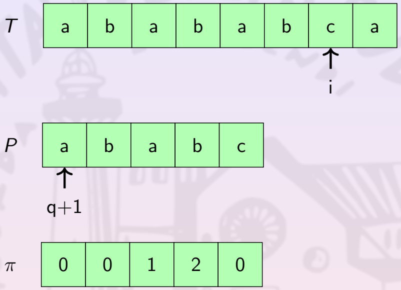
- $q = 4$
- $P[q+1] = T[i]$

Then no **while**-loop iterations

Since $P[q+1] = T[i]$, $P_{q+1} \sqsupset T_i$ and q is updated to 5

Since $q = |P|$, $s = i - q + 1$ is a valid shift and q is updated to $\pi[q]$

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a

P	a	b	a	b	c
π	0	0	1	2	0

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

When $i = 7$:

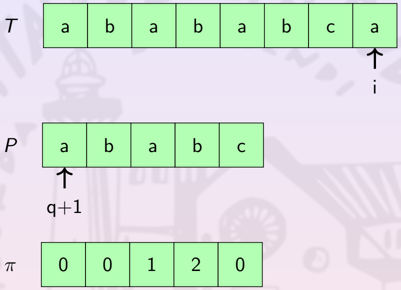
- $q = 4$
- $P[q+1] = T[i]$

Then no **while**-loop iterations

Since $P[q+1] = T[i]$, $P_{q+1} \sqsupset T_i$ and q is updated to 5

Since $q = |P|$, $s = i - q + 1$ is a valid shift and q is updated to $\pi[q] = 0$

The Knuth-Morris-Pratt Algorithm: an Example



T	a	b	a	b	a	b	c	a

P	a	b	a	b	c

π	0	0	1	2	0

At the begin of each **for**-loop iteration, if $i > 1$, then $q = \pi[i-1]$

P_q is the largest proper suffix of P_{i-1} which is also a prefix for it i.e., $P_q \sqsupset P_{i-1}$ and $P_q \sqsubset P_{i-1}$

When $i = 8$:

- $q = 0$
- $P[q+1] = T[i]$

Then no **while**-loop iterations

Since $P[q+1] = T[i]$, $P_{q+1} \sqsupset T_i$ and q is updated to 1

The Knuth-Morris-Pratt Algorithm: Complexity

As for the prefix function computation, the **while**-loop condition holds only if $q > 0$

However, each iteration of the **while**-loop decreases q

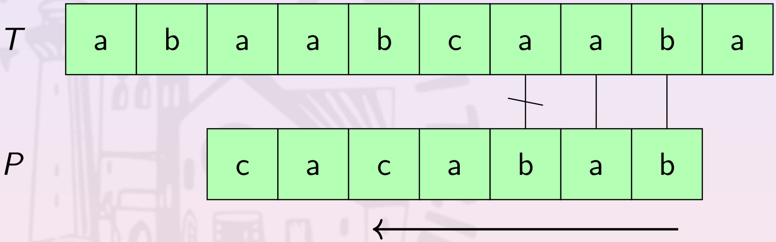
q is initialized to 0 and is increased in the **for**-loop

So, the **while**-loop can be repeated $|T|$ times at most

The overall asymptotic complexity is $\Theta(|P| + |T|)$

The Boyer-Moore-Galil Algorithm

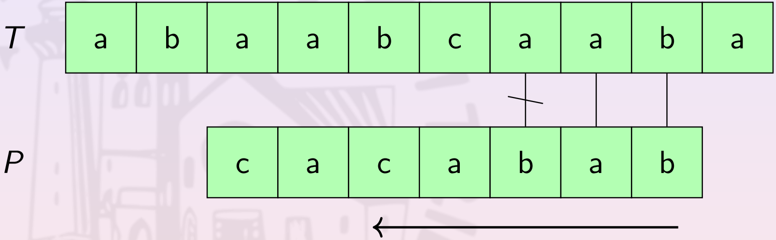
Matches the pattern backward



T	a	b	a	a	b	c	a	a	b	a
P			c	a	c	a	b	a	b	

The Boyer-Moore-Galil Algorithm

Matches the pattern backward



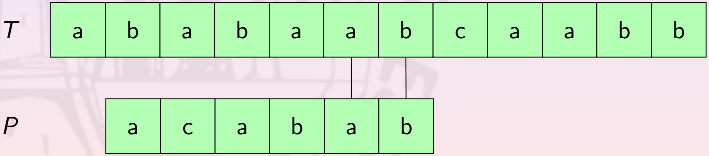
T	a	b	a	a	b	c	a	a	b	a
P			c	a	c	a	b	a	b	

Uses 3 main ingredients:

- **good-suffix rule**
- **bad-character rule**
- **Galil's rule**

The Good-Suffix Rules

If $P[i \dots |P|] = T[i + j \dots |P| + j]$ and

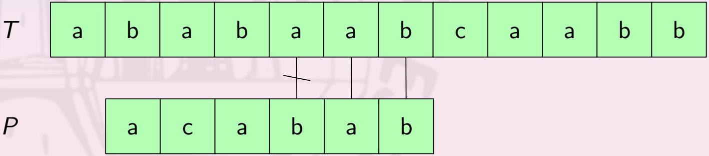


T a b a b a a b c a a b b

P a c a b a b

The Good-Suffix Rules

If $P[i \dots |P|] = T[i+j \dots |P|+j]$ and $P[i-1] \neq T[i+j-1]$



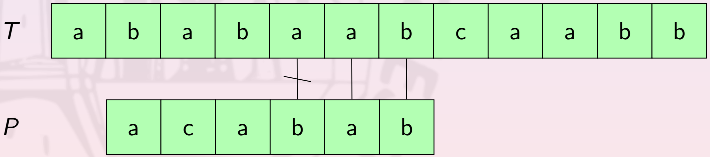
T a b a b a a b c a a b b

P a c a b a b

The Good-Suffix Rules

If $P[i \ldots |P|] = T[i+j \ldots |P|+j]$ and $P[i-1] \neq T[i+j-1]$

- align $T[i+j \ldots |P|+j]$ to its rightmost occurrence in P with a preceding character $\neq P[i-1]$



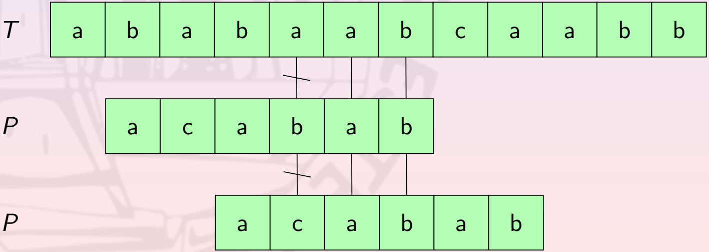
T: a b a b a a b c a a b b

P: a c a b a b

The Good-Suffix Rules

If $P[i \ldots |P|] = T[i+j \ldots |P|+j]$ and $P[i-1] \neq T[i+j-1]$

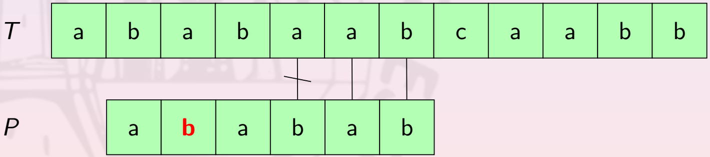
- align $T[i+j \ldots |P|+j]$ to its rightmost occurrence in P with a preceding character $\neq P[i-1]$



The Good-Suffix Rules

If $P[i \ldots |P|] = T[i+j \ldots |P|+j]$ and $P[i-1] \neq T[i+j-1]$

- align $T[i+j \ldots |P|+j]$ to its rightmost occurrence in P with a preceding character $\neq P[i-1]$
- if not exists,

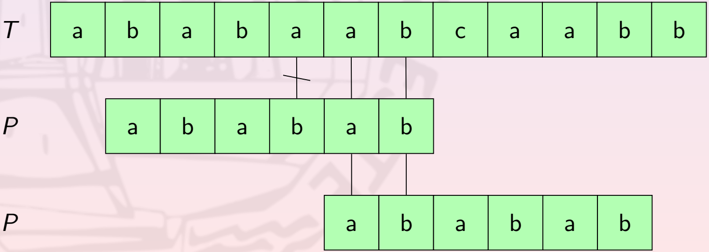


T	a	b	a	b	a	a	b	c	a	a	b	b
P		a	**b**	a	b	a	b					

The Good-Suffix Rules

If $P[i \ldots |P|] = T[i+j \ldots |P|+j]$ and $P[i-1] \neq T[i+j-1]$

- align $T[i+j \ldots |P|+j]$ to its rightmost occurrence in P with a preceding character $\neq P[i-1]$
- if not exists, align the longest $P_q \sqsubset P$ to $T[|P|+j-q \ldots |P|+j]$



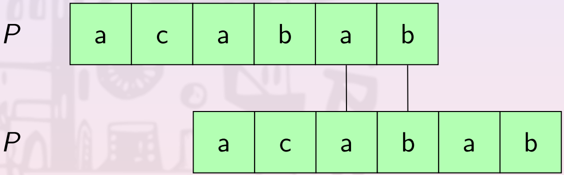
The Good-Suffix Rules: Computing it

It is almost like π^{-1} on the reversed pattern P^{-1}

P	a	c	a	b	a	b

The Good-Suffix Rules: Computing it

It is almost like π^{-1} on the reversed pattern P^{-1}



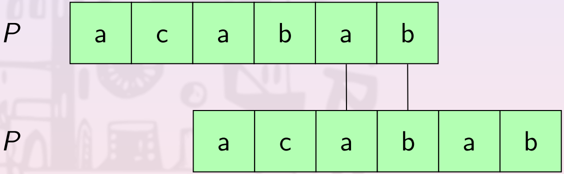
P a c a b a b

P a c a b a b

$P_2^{-1} \sqsupset P_4^{-1}$

The Good-Suffix Rules: Computing it

It is almost like π^{-1} on the reversed pattern P^{-1}



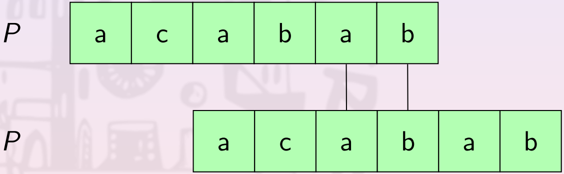
P a c a b a b

P a c a b a b

$P_2^{-1} \sqsupset P_4^{-1}$, $\pi[4] = 2$

The Good-Suffix Rules: Computing it

It is almost like π^{-1} on the reversed pattern P^{-1}



P a c a b a b

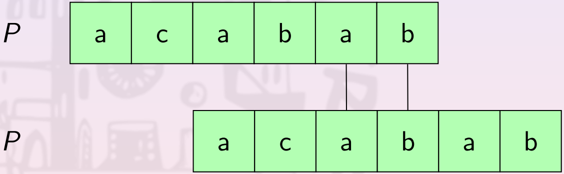
P a c a b a b

$P_2^{-1} \sqsupset P_4^{-1}$, $\pi[4] = 2$, and $\pi^{-1}[2] = 4$

The Good-Suffix Rules: Computing it

It is almost like π^{-1} on the reversed pattern P^{-1}

But $P^{-1}[q+1] \neq P^{-1}[\pi[q]+1]$ must be guaranteed



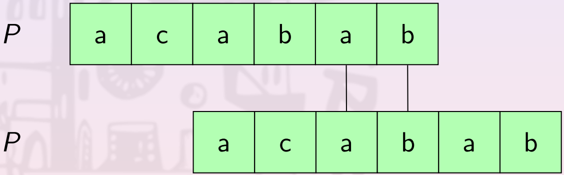
P	a	c	a	b	a	b		
P			a	c	a	b	a	b

$P_2^{-1} \sqsupset P_4^{-1}$, $\pi[4] = 2$, and $\pi^{-1}[2] = 4$

The Good-Suffix Rules: Computing it

It is almost like π^{-1} on the reversed pattern P^{-1}

But $P^{-1}[q+1] \neq P^{-1}[\pi[q]+1]$ must be guaranteed



P a c a b a b

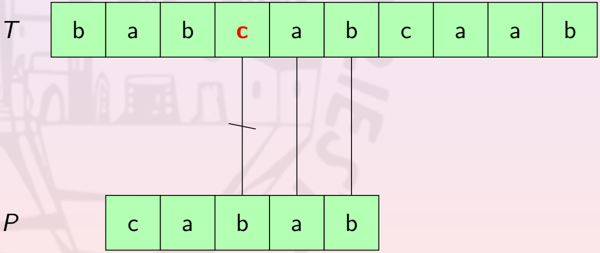
P a c a b a b

$P_2^{-1} \sqsupset P_4^{-1}$, $\pi[4] = 2$, and $\pi^{-1}[2] = 4$

You can guess a complexity $\Theta(|P|)$ to compute it

The Bad-Character Rules

If $P[i] \neq T[i+j]$

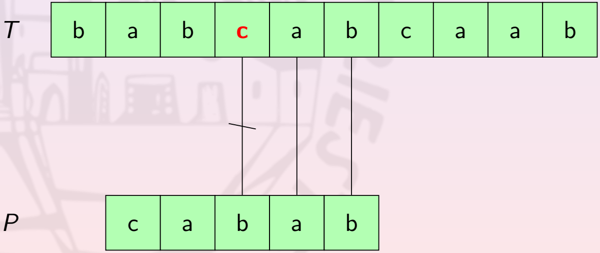


T	b	a	b	c	a	b	c	a	a	b
P		c	a	b	a	b				

The Bad-Character Rules

If $P[i] \neq T[i+j]$

- align $T[i+j]$ to its rightmost occurrence in P



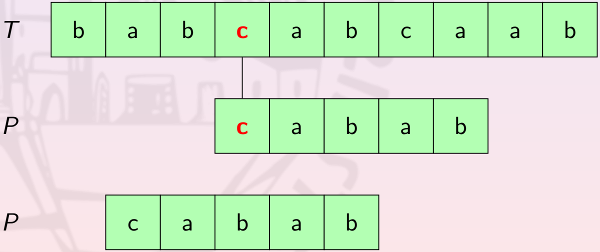
T b a b c a b c a a b

P c a b a b

The Bad-Character Rules

If $P[i] \neq T[i+j]$

- align $T[i+j]$ to its rightmost occurrence in P



T b a b c a b c a a b

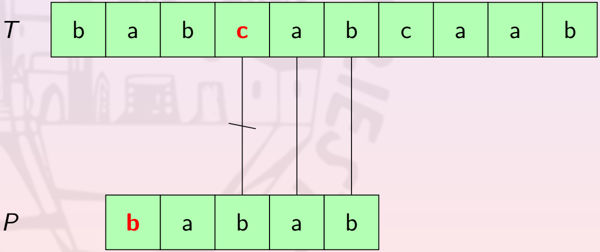
P c a b a b

P c a b a b

The Bad-Character Rules

If $P[i] \neq T[i+j]$

- align $T[i+j]$ to its rightmost occurrence in P
- if not exists,



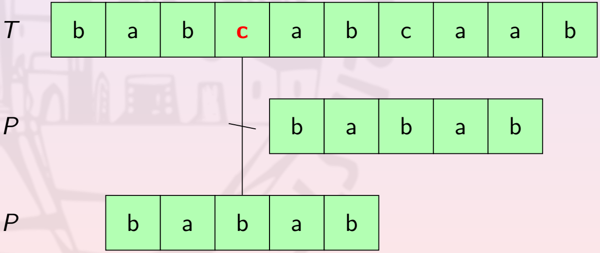
T: b a b c a b c a a b

P: b a b a b

The Bad-Character Rules

If $P[i] \neq T[i+j]$

- align $T[i+j]$ to its rightmost occurrence in P
- if not exists, align $P[1]$ to T[i+j+1]



T	b	a	b	c	a	b	c	a	a	b
P					b	a	b	a	b	
P		b	a	b	a	b				

The Bad-Character Rules: Computing It

- initialize an array C s.t. $|C| = |\Sigma|$

The Bad-Character Rules: Computing It

- initialize an array C s.t. $|C| = |\Sigma|$
- $C[a] \leftarrow |P|$ for each $a \in \Sigma$

The Bad-Character Rules: Computing It

- initialize an array C s.t. $|C| = |\Sigma|$
- $C[a] \leftarrow |P|$ for each $a \in \Sigma$
- $C[P[i]] \leftarrow |P| - i$ for each $i \in [1 \ldots |P|]$

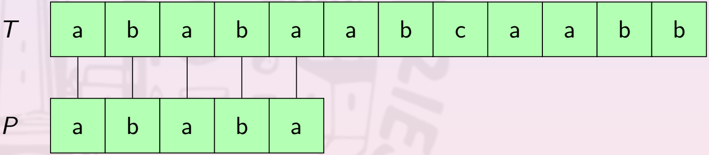
The Bad-Character Rules: Computing It

- initialize an array C s.t. $|C| = |\Sigma|$
- $C[a] \leftarrow |P|$ for each $a \in \Sigma$
- $C[P[i]] \leftarrow |P| - i$ for each $i \in [1 \ldots |P|]$

The complexity is $\Theta(|P| + |\Sigma|)$

The Galil's Rules

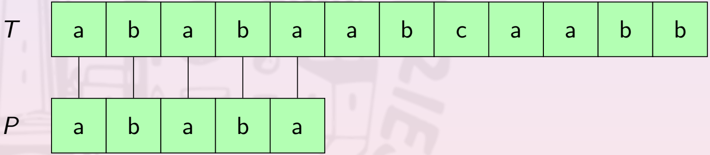
If a valid match has been discovered



T	a	b	a	b	a	a	b	c	a	a	b	b
P	a	b	a	b	a							

The Galil's Rules

If a valid match has been discovered and P is k-periodic

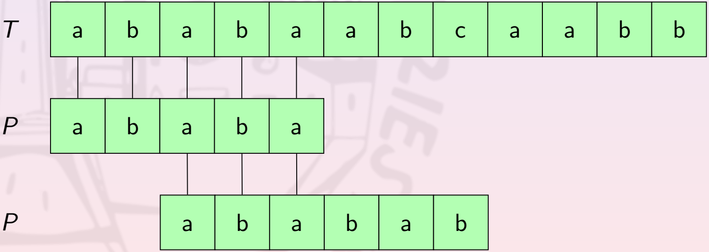


T	a	b	a	b	a	a	b	c	a	a	b	b
P	a	b	a	b	a							

The Galil's Rules

If a valid match has been discovered and P is k-periodic

P is shifted forward by k and $|P| - k$ comparisons avoided



T	a	b	a	b	a	a	b	c	a	a	b	b
P	a	b	a	b	a							
P			a	b	a	b	a	b				

The Boyer-Moore-Galil's Algorithm

- try to match P on T backward
- if a mismatch is found, then select the largest shift among those suggested by the good-suffix and the bad-character rules
- if a valid shift is found, apply the Galil's rules or revert to the mismatch case

The Boyer-Moore-Galil's Algorithm

- try to match P on T backward
- if a mismatch is found, then select the largest shift among those suggested by the good-suffix and the bad-character rules
- if a valid shift is found, apply the Galil's rules or revert to the mismatch case

The Boyer-Moore-Galil's Algorithm

- try to match P on T backward
- if a mismatch is found, then select the largest shift among those suggested by the good-suffix and the bad-character rules
- if a valid shift is found, apply the Galil's rules or revert to the mismatch case

The Boyer-Moore-Galil's Algorithm

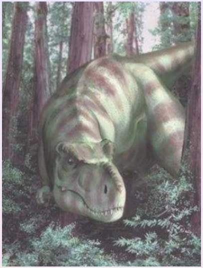
- try to match P on T backward
- if a mismatch is found, then select the largest shift among those suggested by the good-suffix and the bad-character rules
- if a valid shift is found, apply the Galil's rules or revert to the mismatch case

The overall asymptotic complexity is $O(|P| + |T|)$

In an average scenario is sub-linear w.r.t $|T|$.

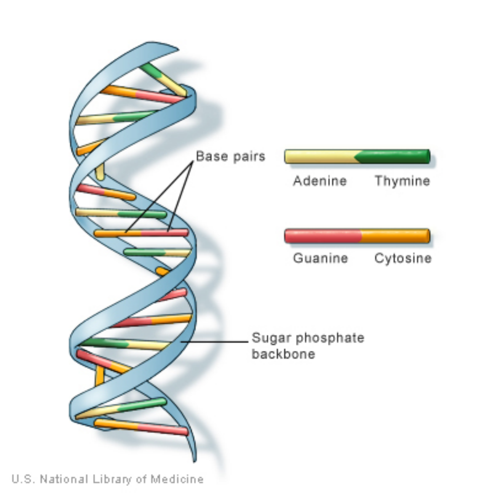
Multiple Patterns String Matching

Life’s Code



Life’s Code

All life forms share the same *code*: the DNA.



Why Studing DNA is Interesting?

- forecast/cure diseases
- threat genetic conditions

“Reading” DNA

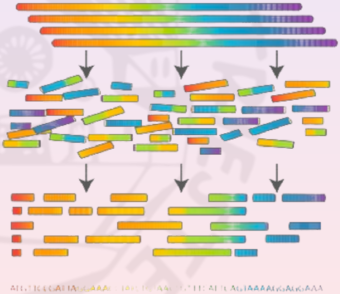
Sequencers are machines to read DNA molecules

But they cannot (yet) accurately read a full DNA molecule

The longer the reading, the higher the probability of errors

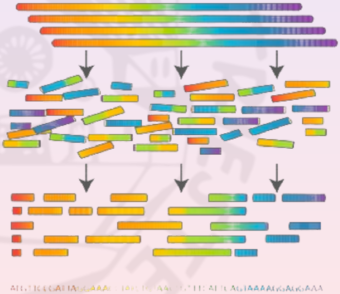
Sequencing and Assembling DNA

- DNA is fragmented in relative short pieces (about 800bps)
- the fragments are sequenced
- the sequencer reads are assembled like a 1-D puzzle



Sequencing and Assembling DNA

- DNA is fragmented in relative short pieces (about 800bps)
- the fragments are sequenced
- the sequencer reads are assembled like a 1-D puzzle



Still slow and expensive due to fragment lengths

Re-sequencing and Aligning

Should we repeat the process of each individual? No

- DNA is fragmented in smaller pieces (about 100bps)
- the fragments are sequenced
- the sequencer reads are aligned over the reference genome

Re-sequencing and Aligning

Should we repeat the process of each individual? No

- DNA is fragmented in smaller pieces (about 100bps)
- the fragments are sequenced
- the sequencer reads are aligned over the reference genome

Fast and cheap due to small fragment size

The Multiple Patterns Single Text Matching Problem

We have

- a text T
- a large set of patterns $\mathcal{P} = \{P_1, \ldots, P_l\}$

The Multiple Patterns Single Text Matching Problem

We have

- a text T
- a large set of patterns $\mathcal{P} = \{P_1, \ldots, P_l\}$

We want to find a valid shift for each P_i

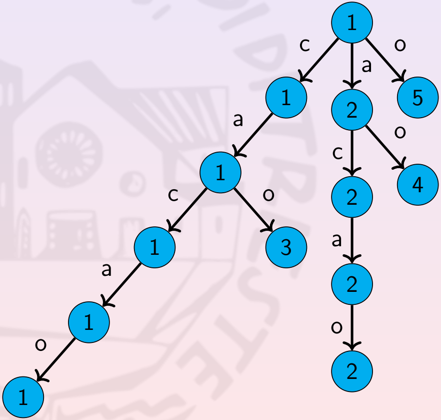
A Naïve Solution

For each P_i, compute BOYER_MOORE_GALIL(T, P_i)

Complexity: $O\left(|T| * \sum_{i=1}^{l} |P_i|\right)$

A Tree-Based Solution

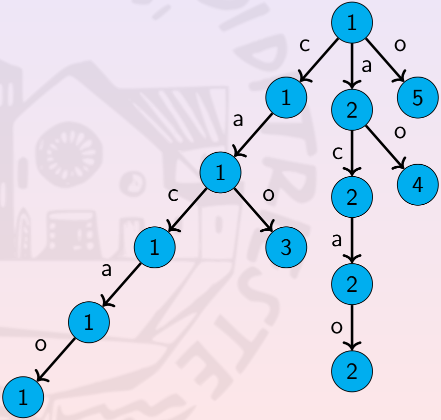
E.g. $T = cacao$



A Tree-Based Solution

E.g. $T = cacao$

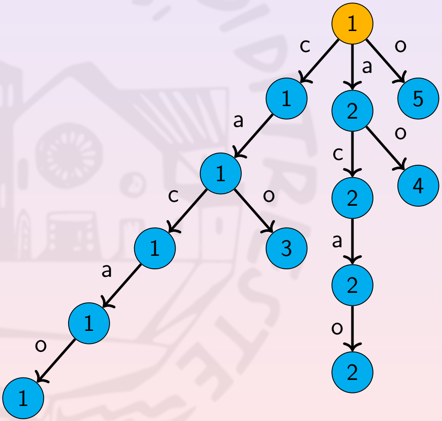
Searching for $P_1 = cao$



A Tree-Based Solution

E.g. $T = cacao$

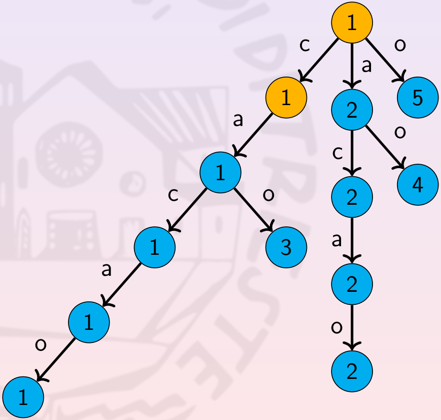
Searching for $P_1 = cao$



A Tree-Based Solution

E.g. $T = cacao$

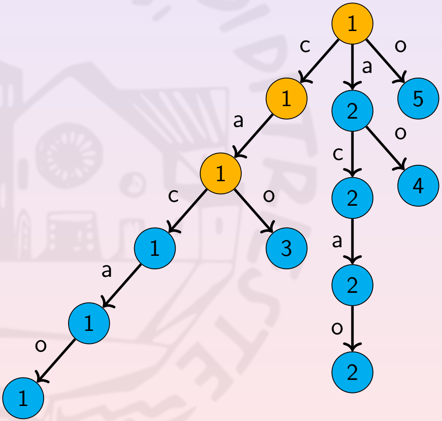
Searching for $P_1 = cao$



A Tree-Based Solution

E.g. $T = cacao$

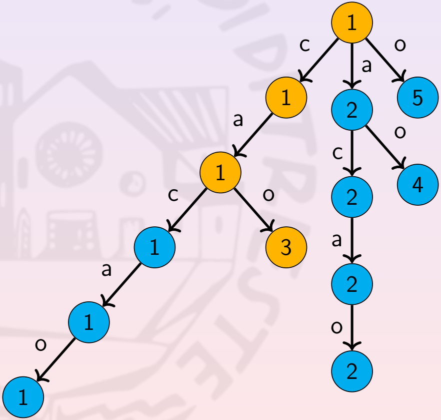
Searching for $P_1 = cao$



A Tree-Based Solution

E.g. $T = cacao$

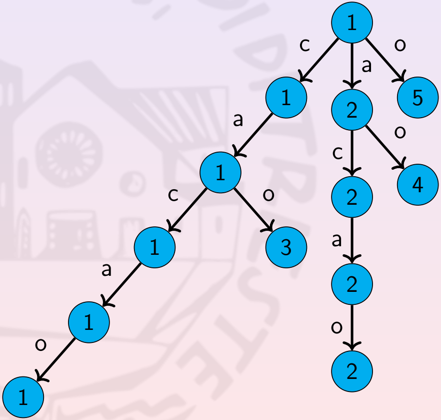
Searching for $P_1 = cao$



A Tree-Based Solution

E.g. $T = cacao$

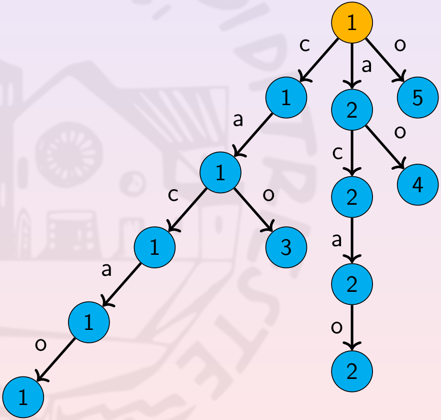
Searching for $P_1 = ac$



A Tree-Based Solution

E.g. $T = cacao$

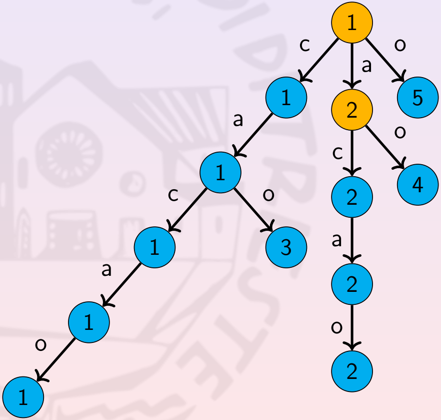
Searching for $P_1 = ac$



A Tree-Based Solution

E.g. $T = cacao$

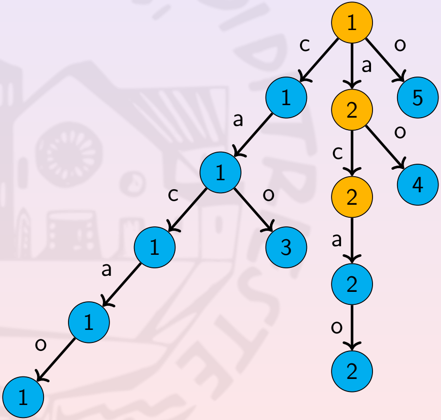
Searching for $P_1 = ac$



A Tree-Based Solution

E.g. $T = cacao$

Searching for $P_1 = ac$



Searching Time

Searching for P_i in the of T's substrings costs $\Theta\left(|P_i|\right)$

Once it has been computed, solving our problem takes time

$$\Theta\left(\sum_{i=1}^{l} |P_i|\right)$$

Searching Time

Searching for P_i in the of T's substrings costs $\Theta\left(|P_i|\right)$

Once it has been computed, solving our problem takes time

$$\Theta\left(\sum_{i=1}^{l} |P_i|\right)$$

How much does its computation cost?

Suffix Tries: A Formal Definition

Let $\sigma(T)$ be the set of all the substrings of T.

$STrie(T)$ of T is a tuple $(Q \cup \{\bot\}, \overline{\epsilon}, L, g, f)$ where:

- $Q = \{\overline{x} \mid x \in \sigma(T)\}$
- $\bot \notin Q$
- $L : Q \mapsto [1 \ldots |T|]$ is the shift label
- $g : (Q \cup \{\bot\}) \times \Sigma \mapsto Q$ is the transition function
  - $g(\overline{x}, a) = \overline{xa}$ for all $xa \in \sigma(T)$
  - $g(\bot, a) = \epsilon$ for all $a \in \Sigma$
- $f : Q \mapsto Q \cup \{\bot\}$ is the prefix function
  - $f(\overline{ax}) = \overline{x}$ for all $ax \in \sigma(T)$
  - $f(\epsilon) = \bot$

Suffix Tries: A Formal Definition

Let $\sigma(T)$ be the set of all the substrings of T.

$STrie(T)$ of T is a tuple $(Q \cup \{\bot\}, \overline{\epsilon}, L, g, f)$ where:

- $Q = \{\overline{x} \,|\, x \in \sigma(T)\}$
- $\bot \notin Q$
- $L : Q \mapsto [1 \ldots |T|]$ is the shift label
- $g : (Q \cup \{\bot\}) \times \Sigma \mapsto Q$ is the transition function
    - $g(\overline{x}, a) = \overline{xa}$ for all $xa \in \sigma(T)$
    - $g(\bot, a) = \epsilon$ for all $a \in \Sigma$
- $f : Q \mapsto Q \cup \{\bot\}$ is the prefix function
    - $f(\overline{ax}) = \overline{x}$ for all $ax \in \sigma(T)$
    - $f(\overline{\epsilon}) = \bot$

Suffix Tries: A Formal Definition

Let $\sigma(T)$ be the set of all the substrings of T.

$STrie(T)$ of T is a tuple $(Q \cup \{\bot\}, \overline{\epsilon}, L, g, f)$ where:

- $Q = \{\overline{x} \mid x \in \sigma(T)\}$
- $\bot \notin Q$
- $L : Q \mapsto [1 \ldots |T|]$ is the shift label
- $g : (Q \cup \{\bot\}) \times \Sigma \mapsto Q$ is the transition function
  - $g(\overline{x}, a) = \overline{xa}$ for all $xa \in \sigma(T)$
  - $g(\bot, a) = \epsilon$ for all $a \in \Sigma$
- $f : Q \mapsto Q \cup \{\bot\}$ is the prefix function
  - $f(\overline{ax}) = \overline{x}$ for all $ax \in \sigma(T)$
  - $f(\epsilon) = \bot$

Suffix Tries: A Formal Definition

Let $\sigma(T)$ be the set of all the substrings of T.

$STrie(T)$ of T is a tuple $(Q \cup \{\bot\}, \overline{\epsilon}, L, g, f)$ where:

- $Q = \{\overline{x} \mid x \in \sigma(T)\}$
- $\bot \notin Q$
- $L : Q \mapsto [1 \ldots |T|]$ is the shift label
- $g : (Q \cup \{\bot\}) \times \Sigma \mapsto Q$ is the transition function
    - $g(\overline{x}, a) = \overline{xa}$ for all $xa \in \sigma(T)$
    - $g(\bot, a) = \epsilon$ for all $a \in \Sigma$
- $f : Q \mapsto Q \cup \{\bot\}$ is the prefix function
    - $f(\overline{ax}) = \overline{x}$ for all $ax \in \sigma(T)$
    - $f(\overline{\epsilon}) = \bot$

Suffix Tries: A Formal Definition

Let $\sigma(T)$ be the set of all the substrings of T.

$STrie(T)$ of T is a tuple $(Q \cup \{\perp\}, \overline{\epsilon}, L, g, f)$ where:

- $Q = \{\overline{x} \mid x \in \sigma(T)\}$
- $\perp \notin Q$
- $L : Q \mapsto [1 \ldots |T|]$ is the shift label
- $g : (Q \cup \{\perp\}) \times \Sigma \mapsto Q$ is the transition function
    - $g(\overline{x}, a) = \overline{xa}$ for all $xa \in \sigma(T)$
    - $g(\perp, a) = \epsilon$ for all $a \in \Sigma$
- $f : Q \mapsto Q \cup \{\perp\}$ is the prefix function
    - $f(\overline{ax}) = \overline{x}$ for all $ax \in \sigma(T)$
    - $f(\overline{\epsilon}) = \perp$

Suffix Tries: A Formal Definition

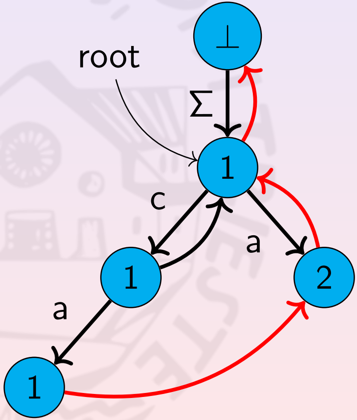
Let $\sigma(T)$ be the set of all the substrings of T.

$STrie(T)$ of T is a tuple $(Q \cup \{\perp\}, \overline{\epsilon}, L, g, f)$ where:

- $Q = \{\overline{x} \mid x \in \sigma(T)\}$
- $\perp \notin Q$
- $L : Q \mapsto [1 \ldots |T|]$ is the shift label
- $g : (Q \cup \{\perp\}) \times \Sigma \mapsto Q$ is the transition function
    - $g(\overline{x}, a) = \overline{xa}$ for all $xa \in \sigma(T)$
    - $g(\perp, a) = \epsilon$ for all $a \in \Sigma$
- $f : Q \mapsto Q \cup \{\perp\}$ is the prefix function
    - $f(\overline{ax}) = \overline{x}$ for all $ax \in \sigma(T)$
    - $f(\overline{\epsilon}) = \perp$

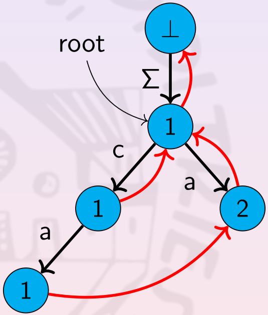
Suffix Tries: An Example

E.g. $T = ca$



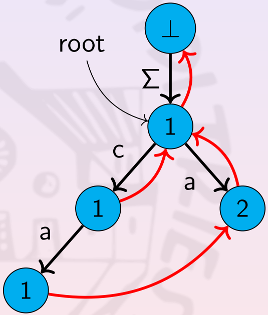
Growing Tries by Appending Characters

E.g. $T = ca$



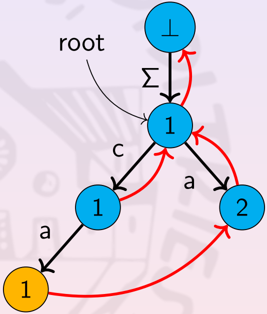
Growing Tries by Appending Characters

E.g. $T = ca\mathbf{c}$



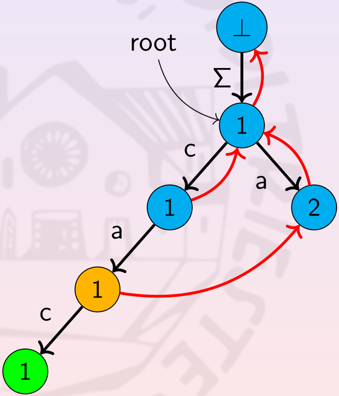
Growing Tries by Appending Characters

E.g. $T = ca\mathbf{c}$



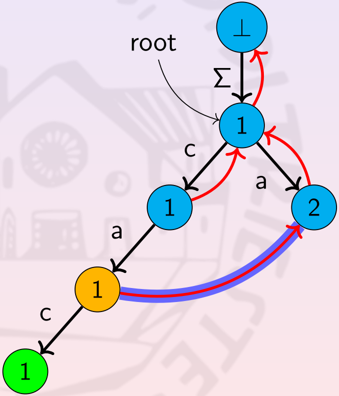
Growing Tries by Appending Characters

E.g. $T = ca\mathbf{c}$



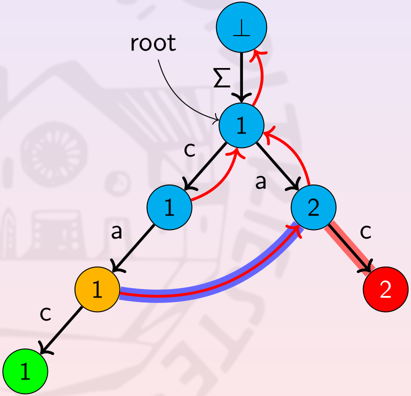
Growing Tries by Appending Characters

E.g. $T = cac$



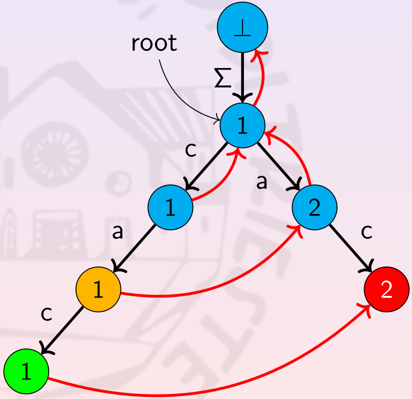
Growing Tries by Appending Characters

E.g. $T = ca\mathbf{c}$



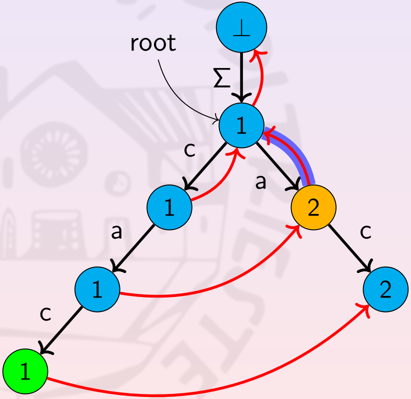
Growing Tries by Appending Characters

E.g. $T = ca\mathbf{c}$



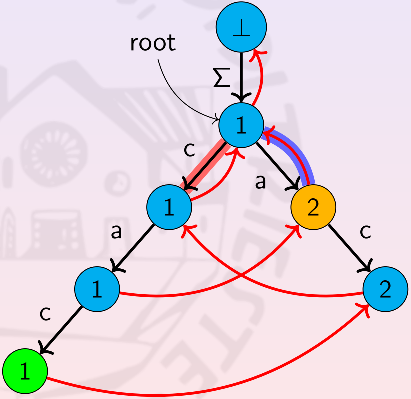
Growing Tries by Appending Characters

E.g. $T = ca\mathbf{c}$



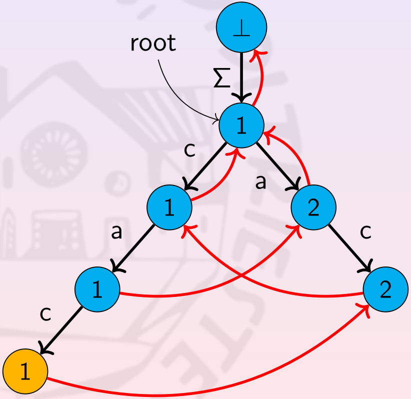
Growing Tries by Appending Characters

E.g. $T = ca\mathbf{c}$



Growing Tries by Appending Characters

E.g. $T = cac$



Boundary Path

Let T^i be $T[1 \dots i]$

The boundary path of $STrie(T^i)$ is the sequence

$$\overline{T^i} = s_1, s_2, \dots, s_{i+1} = \perp$$

where $s_k = f^k(\overline{T^i})$

Boundary Path

Let T^i be $T[1 \ldots i]$

The boundary path of $STrie(T^i)$ is the sequence

$$\overline{T^i} = s_1, s_2, \ldots, s_{i+1} = \perp$$

where $s_k = f^k(\overline{T^i})$

The active point is the first s_j that is a leaf

The end point is the first $s_{j'}$ having a $T[i+1]$-transition

How the Algorithm Works

It adds a $T[i+1]$-transition from s_h for all $h \in [1, j' - 1]$

If $h \in [1, j-1]$, then it extends a branch

If $h \in [j, j'-1]$, then it creates a new branch

Building a Suffix Trie: Pseudo-Code

```
def UPDATE_STRIE(S, T, i, top):  # top is the node corresponding to T[1
  r ← top
  old_s ← None
  while S.g(r, T[i]) = None:
    s ← CREATE_NEW_NODE()

    S.add_node(s)
    S.g(r, T[i]) ← s

    if old_s ≠ None:
      S.f(old_s) ← s
    endif
    old_s ← s

    r ← S.f(r)
  endwhile
  f(old_s) ← S.g(r, T[i])

  return S.g(top, T[i])
enddef
```

Building a Suffix Trie: Complexity

- each node is visited at most twice
- constant steps per node
- $|Q| = |\sigma(T)|$

Building $STrie(T)$ costs $\Theta(\sigma(T))$

Building a Suffix Trie: Complexity

- each node is visited at most twice
- constant steps per node
- $|Q| = |\sigma(T)|$

Building $STrie(T)$ costs $\Theta(\sigma(T))$

Lemma

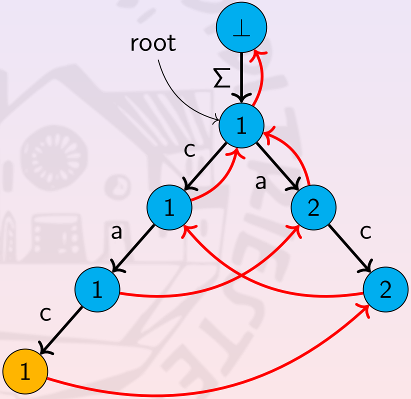
$|\sigma(T)| \in O(|T|^2)$ *(e.g., $a^n b^n$)*

Theorem

Building a $STrie(T)$ costs $O(|T|^2)$

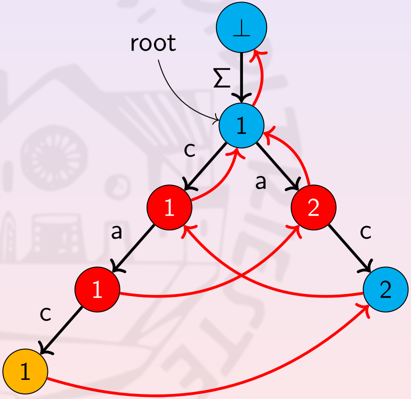
Reducing Complexity

Suffix tries are redundant



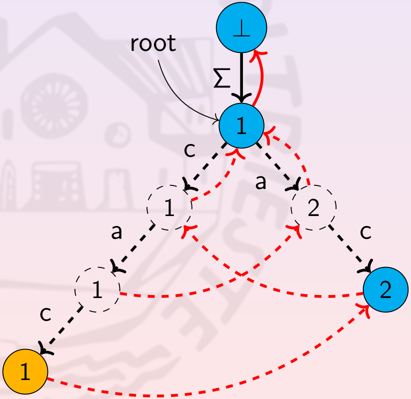
Reducing Complexity

Suffix tries are redundant



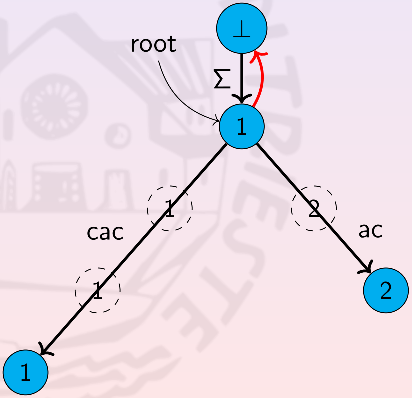
Reducing Suffix Trie Redundancy: Suffix Trees

- Q' containing branching nodes Q_b and leaves Q_l
- $g' : ((Q_b \cup \{\perp\}) \times \Sigma^*) \mapsto Q'$
- $f' : Q_b \mapsto Q_b$



Reducing Suffix Trie Redundancy: Suffix Trees

- Q' containing branching nodes Q_b and leaves Q_l
- $g' : ((Q_b \cup \{\perp\}) \times \Sigma^*) \mapsto Q'$
- $f' : Q_b \mapsto Q_b$



Counting Nodes

- the leaves represent some of the suffixes of T and they are at most $|T|$
- all the internal nodes are branching and they are at most $|T|$-1

These kind of trees has $\Theta(|T|)$ nodes

Substrings to Indexes Intervals

To save space g' labels are represented as T-index intervals

E.g., if $T = cacao$, then

- cao is represented by $[3,5]$
- $g'(\overline{ca},[3,5]) = \overline{cao}$

Substrings to Indexes Intervals

To save space g' labels are represented as T-index intervals

E.g., if $T = cacao$, then

- cao is represented by $[3, 5]$
- $g'(\overline{ca}, [3, 5]) = \overline{cao}$

If $\Sigma = \{a_1, \ldots, a_k\}$, then

- the string a_i labeling $(\perp, \overline{\epsilon})$ is encoded as $[-i, -i]$
- $g'(\perp, [-i, -i]) = \overline{\epsilon}$ for all $i \in [1, k]$

Substrings to Indexes Intervals

To save space g' labels are represented as T-index intervals

E.g., if $T = cacao$, then

- cao is represented by $[3,5]$
- $g'(\overline{ca}, [3,5]) = \overline{cao}$

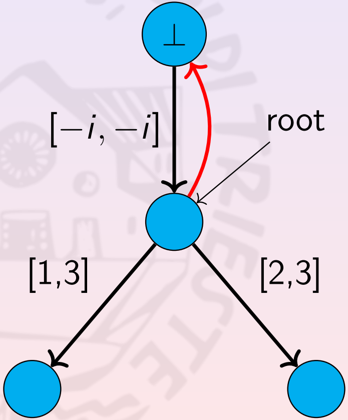
If $\Sigma = \{a_1, \ldots, a_k\}$, then

- the string a_i labeling $(\perp, \overline{\epsilon})$ is encoded as $[-i, -i]$
- $g'(\perp, [-i, -i]) = \overline{\epsilon}$ for all $i \in [1, k]$

We can also avoid L: look at the last matching label to infer shifts

A Suffix Tree Example

E.g. $T = cac$



Implicit and Explicit Nodes

Not all the node of the suffix tries are explicitly represented

We can represent implicit nodes by reference pairs *explicit node/substring*

E.g., if $T = cac$, then $\overline{ca}$ is encoded as $(\overline{\epsilon}, [1, 2])$

Implicit and Explicit Nodes

Not all the node of the suffix tries are explicitly represented

We can represent implicit nodes by reference pairs *explicit node/substring*

E.g., if $T = cac$, then $\overline{ca}$ is encoded as $(\overline{\epsilon}, [1, 2])$

Also explicit nodes can be represented by reference pairs as (x, ϵ)

(x, ϵ) is encoded as $(x, [p + 1, p])$

Implicit and Explicit Nodes

Not all the node of the suffix tries are explicitly represented

We can represent implicit nodes by reference pairs *explicit node/substring*

E.g., if $T = cac$, then $\overline{ca}$ is encoded as $(\overline{\epsilon}, [1, 2])$

Also explicit nodes can be represented by reference pairs as (x, ϵ)

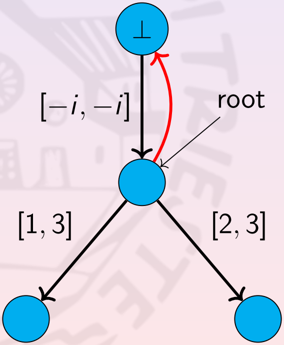
(x, ϵ) is encoded as $(x, [p+1, p])$

If x is the closed ancestor of (x, w), then (x, w) is canonical

Branch Extensions in Suffix Trees

Branch extensions is avoided by labeling transition to leaf as $[h, \infty]$

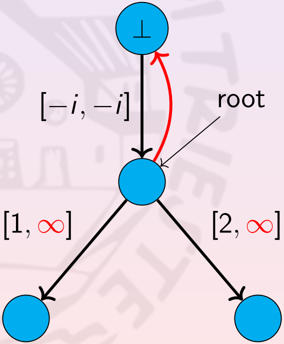
$T = cac$



Branch Extensions in Suffix Trees

Branch extensions is avoided by labeling transition to leaf as $[h, \infty]$

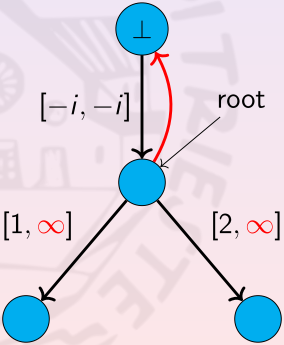
$T = cac$



Branch Extensions in Suffix Trees

Branch extensions is avoided by labeling transition to leaf as $[h, \infty]$

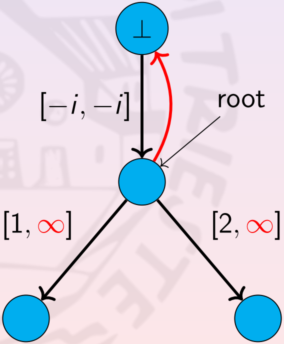
$T = caca$



Branch Extensions in Suffix Trees

Branch extensions is avoided by labeling transition to leaf as $[h, \infty]$

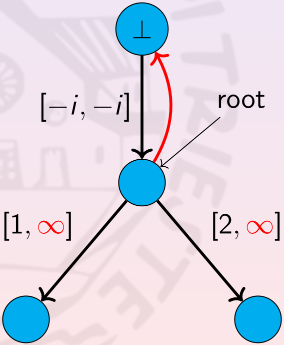
$T = caca{\color{red}c}$



Branch Extensions in Suffix Trees

Branch extensions is avoided by labeling transition to leaf as $[h, \infty]$

$T = cacaca$

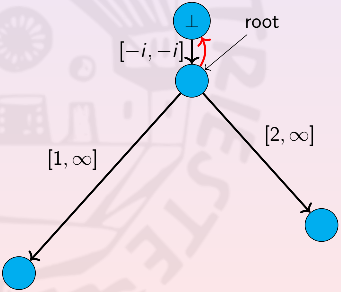


Branching in Suffix Trees: Explicit a Node

s_j has a canonical reference pair $(s, [k, i])$

If it is implicit, it should become explicit

$T = cac$

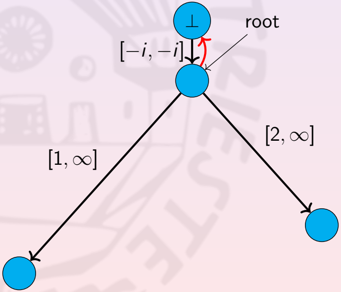


Branching in Suffix Trees: Explicit a Node

s_j has a canonical reference pair $(s, [k, i])$

If it is implicit, it should become explicit

$T = caco$

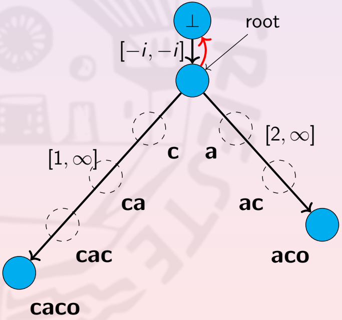


Branching in Suffix Trees: Explicit a Node

s_j has a canonical reference pair $(s, [k, i])$

If it is implicit, it should become explicit

$T = caco$

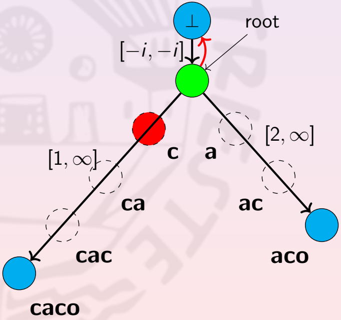


Branching in Suffix Trees: Explicit a Node

s_j has a canonical reference pair $(s, [k, i])$

If it is implicit, it should become explicit

$T = caco$

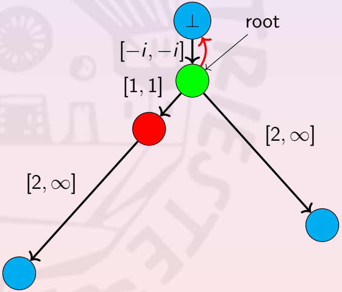


Branching in Suffix Trees: Explicit a Node

s_j has a canonical reference pair $(s, [k, i])$

If it is implicit, it should become explicit

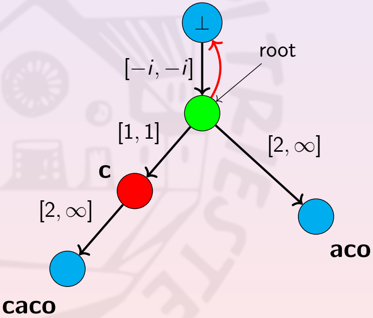
$T = caco$



Branching in Suffix Trees: Adding a Branch

If s_j is explicit, add a new branch labeled $[i+1,\infty]$

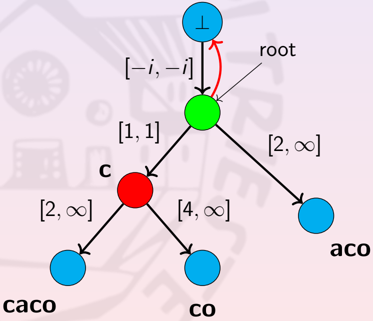
$T = caco$



Branching in Suffix Trees: Adding a Branch

If s_j is explicit, add a new branch labeled $[i+1, \infty]$

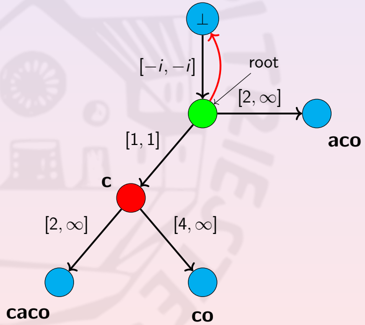
$T = caco$



Following the Boundary Path

If $(s, [k, i])$ is on boundary path, the next node is $(f'(s), [k, i])$

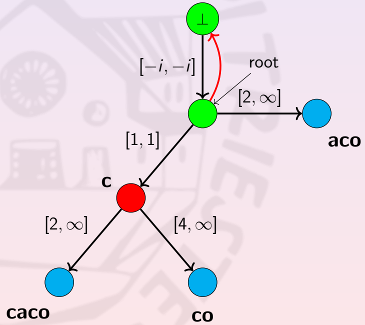
$T = caco$



Following the Boundary Path

If $(s, [k, i])$ is on boundary path, the next node is $(f'(s), [k, i])$

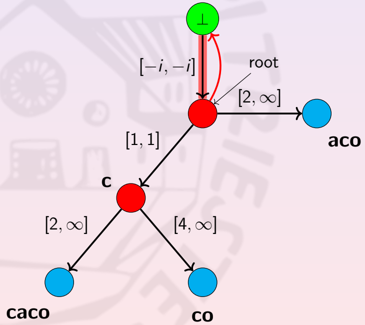
$T = caco$



Following the Boundary Path

If $(s, [k, i])$ is on boundary path, the next node is $(f'(s), [k, i])$

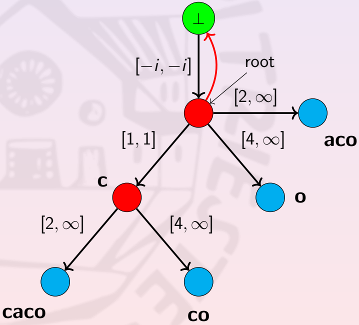
$T = caco$



Following the Boundary Path

If $(s, [k, i])$ is on boundary path, the next node is $(f'(s), [k, i])$

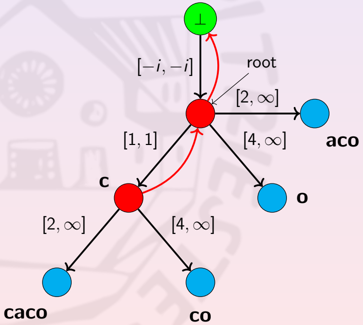
$T = caco$



Following the Boundary Path

If $(s, [k, i])$ is on boundary path, the next node is $(f'(s), [k, i])$

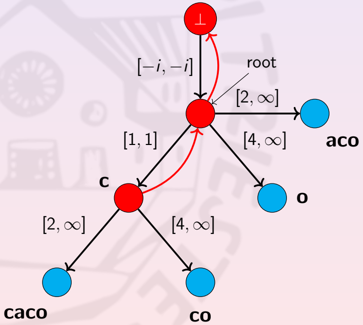
$T = caco$



Following the Boundary Path

If $(s, [k, i])$ is on boundary path, the next node is $(f'(s), [k, i])$

$T = caco$



Canonizing Nodes on Boundary Paths

$(s', [k, i])$, where $s' = f'(s)$, may be non-canonical

Before any other procedure, we must canonize it.

Canonizing Nodes on Boundary Paths

$(s', [k, i])$, where $s' = f'(s)$, may be non-canonical

Before any other procedure, we must canonize it.

Let $[k', i']$ the label of the $T[k]$-transition from s'

Canonizing Nodes on Boundary Paths

$(s', [k, i])$, where $s' = f'(s)$, may be non-canonical

Before any other procedure, we must canonize it.

Let $[k', i']$ the label of the $T[k]$-transition from s'

- if $[k, i]$ is shorter than $[k', i']$, it is canonical

Canonizing Nodes on Boundary Paths

$(s', [k, i])$, where $s' = f'(s)$, may be non-canonical

Before any other procedure, we must canonize it.

Let $[k', i']$ the label of the $T[k]$-transition from s'

- if $[k, i]$ is shorter than $[k', i']$, it is canonical
- otherwise replace:
  - s' with $g'(s', [k', i'])$
  - $[k, i]$ with $[k + (i' - k') + 1, i]$

  and repeat

Finding Next Active Point

$\overline{T[j \ldots i]}$ is the active point of $STree(T^i)$ iff $T[j \ldots i]$ is the longest suffix of T^i that occurs twice

Finding Next Active Point

$\overline{T[j \dots i]}$ is the active point of $STree(T^i)$ iff $T[j \dots i]$ is the longest suffix of T^i that occurs twice

$\overline{T[j' \dots i]}$ is the end point of $STree(T^i)$ iff $T[j \dots i]$ is the longest suffix of T^i s.t. $T[j \dots i+1]$ is a substring of T^i

Finding Next Active Point

$\overline{T[j \ldots i]}$ is the active point of $STree(T^i)$ iff $T[j \ldots i]$ is the longest suffix of T^i that occurs twice

$\overline{T[j' \ldots i]}$ is the end point of $STree(T^i)$ iff $T[j \ldots i]$ is the longest suffix of T^i s.t. $T[j \ldots i+1]$ is a substring of T^i

Theorem

If $(s, [k, i])$ is the end point of $STree(T^i)$, then $(s, [k, i+1])$ is the active point of $STree(T^{i+1})$.

Building the Suffix Tree: Pseudo-Code

```
def UPDATE_STREE(root,T,s,k,i): # (s,[k,i−i]) is the canonical
                                # reference to active point
  old_r ← root

  (end_point,r) ← TEST_AND_SPLIT((s,[k,i−1]), T, i)
  while not end_point:
    s ← CREATE_NEW_BRANCH(r,i)

    ADD_A_SUFFIX_TRANSITION(root,old_r,r)
    old_r ← r

    (s,k) ← CANONIZE((s.f,[k,i−1]))
    (end_point,r) ← TEST_AND_SPLIT((s,[k,i−1]), T, i)
  endwhile
  ADD_A_SUFFIX_TRANSITION(root,old_r,r)

  return (s,k)
enddef
```

Building the Suffix Tree: Pseudo-Code

```
def CREATE_NEW_BRANCH(r, i):
  s $\leftarrow$ CREATE_NEW_NODE()

  # add a T[i]-transition from r to s
  s.g[T[i]] $\leftarrow$ (r,[i, $\infty$])

  return s
enddef

def ADD_A_SUFFIX_TRANSITION(root, s, r):
  # if s = root, then s.f = BOT
  if s $\neq$ root:
    s.f $\leftarrow$ r
  endif
endif
```

Building the Suffix Tree: Pseudo-Code

```
def TEST_AND_SPLIT((s,[k,p]), T, i):
  if k > p: # if (s,[k,p]) is explicit
    return HANDLE_EXPLICIT_NODE(s, T, i)
  endif:

  # (s,[k,p]) is implicit
  return HANDLE_IMPLICIT_NODE((s,[k,p]), T, i)
enddef

def HANDLE_EXPLICIT_NODE(s, T, i):
  if s.g[T[i]] ≠ NIL: # s has a T[i]−transition
    return (False,s)
  endif

  return (True,s)
enddef
```

Building the Suffix Tree: Pseudo-Code

```
def HANDLE_IMPLICIT_NODE((s,[k,p]), T, i):
  (dst,[sk,sp]) ← s.g[T[k]] # get T[k]−transition's
                            # dst and label

  rk ← sk+p−k+1

  if T[i] = T[rk]:
    return (True,s)
  endif

  r ← CREATE_NEW_NODE()

  # split the T[k]−transition in (s,r) and (r,dst)
  s.g[T[sk]] ← (r,[sk,rk−1])
  r.g[T[rk]] ← (dst,[rk,sp])

  return (False,r)
enddef
```

Building the Suffix Tree: Pseudo-Code

```
def CANONIZE((s,[k,p])):
  if $k > p$: # if (s,[k,p]) is explicit
    return (s,k)
  endif
  (dst, [sk,sp]) $\leftarrow$ s.g[T[k]] # get T[k]-transition's
                                      # dst and label
  while $sp-sk \leq p-k$:
    $k \leftarrow k+sp-sk+1$
    $s \leftarrow dst$

    if $k \leq p$:
      (dst,[sk,sp]) $\leftarrow$ s.g[T[k]]
    endif
  endwhile

  return (s,k)
enddef
```

Building the Suffix Tree: Pseudo-Code

```
def BUILD_STREE(T, Sigma):
  root ← CREATE_NEW_NODE()
  BOT ← CREATE_NEW_NODE()

  root.f ← BOT
  for i←1 upto |Sigma|:
    ai ← Sigma[i]
    BOT.g[ai] ← (root,[−i,−i])
  endfor

  k ← 1
  s ← root
  for i←1 upto |T|:
      (s,k) ← UPDATE_STREE(root,T,s,k,i)
      (s,k) ← CANONIZE((s,[k,i]))
  endfor

  return root
enddef
```

Building the Suffix Tree: Complexity

The algorithm is split into two components:

1. The `canonize` calls:

2. The remaining computation:

Building the Suffix Tree: Complexity

The algorithm is split into two components:

1. The `canonize` calls:

   Each iteration of the `while`-loop increase k in $(s, [k, p])$ by at least 1

   However, p is increased by 1 at each iteration of the main procedure's `for`-loop

   `canonize` takes time $O(|T|)$ in total
2. The remaining computation:

Building the Suffix Tree: Complexity

The algorithm is split into two components:

1. The `canonize` calls:

   Each iteration of the `while`-loop increase k in $(s, [k, p])$ by at least 1

   However, p is increased by 1 at each iteration of the main procedure's `for`-loop

   `canonize` takes time $O(|T|)$ in total
2. The remaining computation:

   Consists in many suffix links and one T[i]-transition crossing

   If r_i is the active point for $STree(T^i)$, the algorithm visits $depth(r_{i-1}) - depth(r_i) + 2$.

   In total $depth(r_0) - depth(r_{|T|}) + 2|T| \in Theta(|T|)$

Suffix Trees: Conclusions

- Solving the MPSM problem takes time

$$\Theta(|T|)$$

Suffix Trees: Conclusions

- Solving the MPSM problem takes time

$$\Theta(|T|) + \Theta(\sum_{i=1}^{l} |P_i|)$$

Suffix Trees: Conclusions

- Solving the MPSM problem takes time

$$\Theta(|T|) + \Theta(\sum_{i=1}^{l} |P_i|)$$

- Suffix trees take space $\Theta(|T|)$

Suffix Trees: Conclusions

- Solving the MPSM problem takes time

$$\Theta(|T|) + \Theta(\sum_{i=1}^{l} |P_i|)$$

- Suffix trees take space $\Theta(|T|)$

However,

- it handles only one of the pattern valid shifts
- it requires up $(2 + 1 + |\Sigma|) * |T|$ words of RAM (non- feasable for genome)

Suffix Trees: Conclusions

- Solving the MPSM problem takes time

$$\Theta(|T|) + \Theta(\sum_{i=1}^{l} |P_i|)$$

- Suffix trees take space $\Theta(|T|)$

However,

- it handles only one of the pattern valid shifts
- it requires up $(2 + 1 + |\Sigma|) * |T|$ words of RAM (non- feasable for genome)

A possible alternative is Suffix Arrays