CS 240 – Data Structures and Data Management

Module 9: String Matching

Mark Petrick, Éric Schost

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2024

version 2024−07−03 12:49

Outline

9 String Matching

- Introduction
- Karp-Rabin Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays
- Conclusion

Outline

9 String Matching
- Introduction
- Karp-Rabin Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays
- Conclusion

Pattern Matching Introduction

- Search for a string (pattern) in a large body of text. Useful for
  - Information Retrieval (text editors, search engines)
  - Bioinformatics
  - Data Mining
- $T[0..n-1]$ – The text (or haystack) being searched within

  Example: T = "`Where is he?`"
- $P[0..m-1]$ – The pattern (or needle) being searched for

  Example: P_1 = "`he`" P_2 = "`who`"
- **occurrence**: index i such that $T[i..i{+}m{-}1] = P$, i.e.,

$$P[j] = T[i+j] \qquad \text{for} \quad 0 \le j \le m-1$$

- Convention: return smallest such i (leftmost occurrence)
- If P does not occur in T, return `FAIL`

Pattern Matching Observation

Recall:

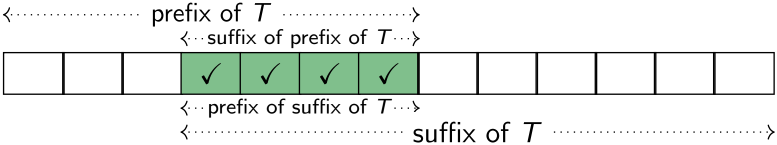
- **Substring** $T[i..j]$ for $0 \leq i \leq j+1 \leq n$: a string of length $j - i + 1$ which consists of characters $T[i], \ldots T[j]$ in order.
- **Prefix** of T: a substring $T[0..i-1]$ of T for some $0 \leq i \leq n$.
- **Suffix** of T: a substring $T[i..n-1]$ of T for some $0 \leq i \leq n$.
- The **empty string** Λ is also considered a substring, prefix and suffix.

Observe: P occurs in T

$\Leftrightarrow$ P is a substring of T.

$\Leftrightarrow$ P is a suffix of some prefix of T.

$\Leftrightarrow$ P is a prefix of some suffix of T.

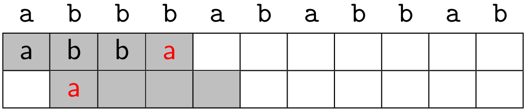


General Idea of Algorithms

Pattern matching algorithms consist of guesses and checks:

- A **guess** is a position g such that P might start at $T[i]$. Valid guesses (initially) are $0 \leq g \leq n - m$.
- A **check** of a guess is a single position j with $0 \leq j < m$ where we compare $T[g + j]$ to $P[j]$.
- We do *strncmp* to compare a guess to P. This uses m checks in the worst-case, but may use (many) fewer checks if there is a *mismatch*.

We will diagram a single run of any pattern matching algorithm by a matrix of checks, where each row represents a single guess (shaded gray).



a	b	b	b	a	b	a	b	b	a	b
a	b	b	a							
	a									

Brute-force Algorithm

Idea: Check every possible guess.

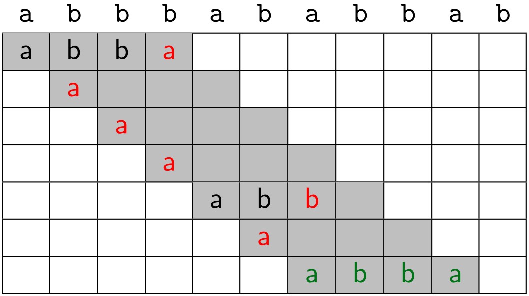
```
Bruteforce::pattern-matching($T[0..n-1], P[0..m-1]$)
$T$: String of length $n$ (text), $P$: String of length $m$ (pattern)
for $g \leftarrow 0$ to $n-m$ do          // $g$: index of guess
    if strncmp($T, P, g, m$) $= 0$
        return "found at guess $g$"
return FAIL
```

Note: *strncmp* takes $\Theta(m)$ time.

```
strncmp($T, P, g \leftarrow 0, m$))
// Compare $m$ chars of $T$ and $P$, starting at $T[g]$
for $j \leftarrow 0$ to $m-1$ do
    if $T[g+j]$ is before $P[j]$ in $\Sigma$ then return -1
    if $T[g+j]$ is after $P[j]$ in $\Sigma$ then return 1
return 0
```

Brute-Force Example

- Example: $T =$ `abbbababbab`, $P =$ `abba`



a	b	b	b	a	b	a	b	b	a	b
a	b	b	a							
	a									
		a								
			a							
				a	b	b				
					a					
						a	b	b	a	

- What is the worst possible input?

Brute-Force Example

- Example: $T =$ `abbbababbab`, $P =$ `abba`

a	b	b	b	a	b	a	b	b	a	b
a	b	b	a							
	a									
		a								
			a							
				a	b	b				
					a					
						a	b	b	a	

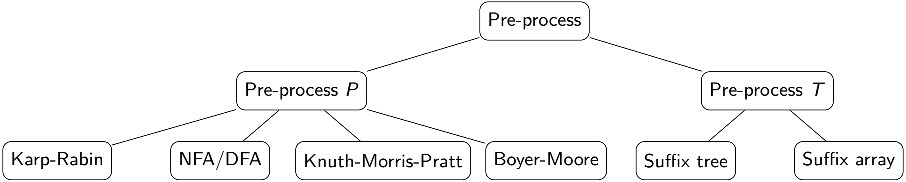
- What is the worst possible input? $P = a^{m-1}b,\ T = a^n$
- Worst case performance $\Theta((n-m+1)\cdot m)$
- This is too slow (quadratic if $m \approx n/2$).

How to improve?

General idea of **preprocessing**: Do work on some parts of input beforehand, so that the actual **query** (with rest of input) then goes faster.

For pattern matching, we have two options:

- Do preprocessing on the pattern P
  - We eliminate guesses based on characters we have seen.
- Do preprocessing on the text T
  - We create a data structure to find matches easily.



Pre-process

Pre-process P

Pre-process T

Karp-Rabin

NFA/DFA

Knuth-Morris-Pratt

Boyer-Moore

Suffix tree

Suffix array

Outline

9 String Matching

- Introduction
- Karp-Rabin Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays
- Conclusion

Karp-Rabin Fingerprint Algorithm – Idea

Idea: Use **fingerprints** to eliminate guesses

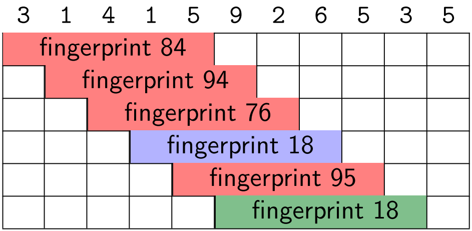
- Need function $h : \{\text{strings of length } m\} \rightarrow \{0, \ldots, M{-}1\}$
  (Call these 'hash-function' and 'table-size', but there is no dictionary here)
- **Insight:** If $h(P) \neq h(\,T[g..g{+}m{-}1])$ then guess g cannot work

Example: $\Sigma = \{0{-}9\}$, $P = 9\ 2\ 6\ 5\ 3$, $T = 3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3\ 5$

- Use standard hash-function for words, with $R = |\Sigma|$ and $M = 97$:

$$h(x_0 \ldots x_4) = \big(x_0x_1x_2x_3x_4\big)_{10} \bmod 97$$

- Pre-compute $h(P) = 92653 \bmod 97 = 18$.



no *strncmp* needed

no *strncmp* needed

no *strncmp* needed

do *strncmp*, false positive

no *strncmp* needed

do *strncmp*, found

Karp-Rabin Fingerprint Algorithm – First Attempt

```
Karp-Rabin-Simple::pattern-matching(T, P)
$h_P \leftarrow h(P[0..m-1)])$
for $g \leftarrow 0$ to $n - m$
    $h_T \leftarrow h(T[g..g+m-1])$        // not constant time
    if $h_T = h_P$
        if strncmp$(T, P, g, m) = 0$
            return "found at guess $g$"
return FAIL
```

- Never misses a match: $h(T[g..g+m-1]) \neq h(P) \Rightarrow$ guess g is not P
- $h(T[g..g+m-1])$ depends on m characters, so naive computation takes $\Theta(m)$ time per guess
- Running time is $\Theta(mn)$ if P is not in T. Can we improve this?

Karp-Rabin Fingerprint Algorithm – Fast Update

Idea: Consecutive guesses share $m-1$ characters
$\Rightarrow$ for suitable hash-functions, can compute next fingerprint from previous

Example: $15926 = (41592 - 4 \cdot 10\,000) \cdot 10 + 6$

$$\underbrace{15926 \bmod 97}_{h(15926)} = \Big((\underbrace{41592 \bmod 97}_{\text{previous fingerprint}} -4 \cdot \underbrace{10000 \bmod 97}_{9 \text{ (pre-computed)}} \Big) \cdot 10 + 6) \bmod 97$$

$$= \Big((76 - 4 \cdot 9) \cdot 10 + 6 \Big) \bmod 97 = 18$$

- So pre-compute $R^{m-1} \bmod M$ (here $10000 \bmod 97 = 9$)
- Compute leftmost fingerprint
- Use previous fingerprint to compute next fingerprint in $O(1)$ time
- Run-time: $O(m + n + m \cdot \#\{\text{false positives}\})$

Karp-Rabin Fingerprint Algorithm – Conclusion

```
Karp-Rabin::pattern-matching($T, P$)  // rolling hash-function
    $M \leftarrow$ suitable prime number
    $h_P \leftarrow h(P[0..m-1)])$
    $s \leftarrow R^{m-1} \bmod M$
    $h_T \leftarrow h(T[0..m-1)])$
    for $g \leftarrow 0$ to $n - m$
        if $h_T = h_P$
            if strncmp($T, P, g, m$) $= 0$ return "found at guess $g$"
        if $g < n - m$ // compute fingerprint for next guess
            $h_T \leftarrow ((h_T - T[g] \cdot s) \cdot R + T[g+m]) \bmod M$
    return "FAIL"
```

- Choose "table size" M to be random prime in $\{2, \ldots, mn^2\}$
- Can show: Then $P(\text{at least one false positive}) \in O(\frac{1}{n})$
- Expected time $O(m+n)$, worst-luck time $O(m \cdot n)$ (extremely unlikely)
- Improvement: reset M after a false positive

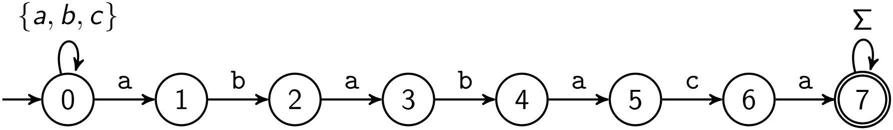
Outline

9 String Matching

- Introduction
- Karp-Rabin Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays
- Conclusion

String Matching with Finite Automata

Example: Automaton for the pattern $P =$ `ababaca`

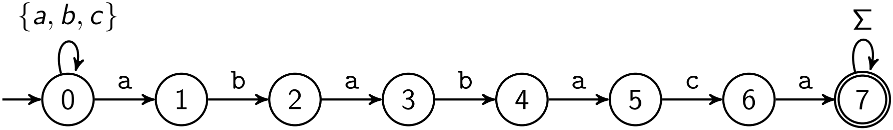


$\left(\right.$ You should be familiar with:

- finite automaton, DFA, NFA, converting NFA to DFA
- transition function, states, start state, accepting states $\left. \right)$

String Matching with Finite Automata

Example: Automaton for the pattern $P =$ `ababaca`

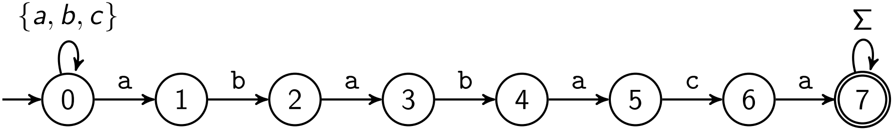


$\left(\begin{array}{l}\text{You should be familiar with:} \\ \bullet\ \text{finite automaton, DFA, NFA, converting NFA to DFA} \\ \bullet\ \text{transition function, states, start state, accepting states}\end{array}\right)$

- This is a **N**on-deterministic **F**inite **A**utomaton
- **Forward-arc** $\textcircled{j} \longrightarrow \textcircled{\scriptstyle j+1}$ labelled with $P[j]$
- State j expresses "we have j leftmost characters of P'
- NFA accepts T if and only if T contains P

String Matching with Finite Automata

Example: Automaton for the pattern $P = \texttt{ababaca}$



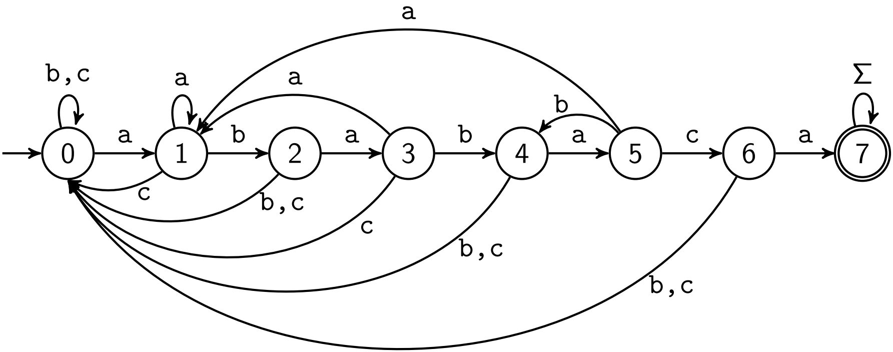
$$\left(\begin{array}{l}\text{You should be familiar with:}\\ \quad\bullet\ \text{finite automaton, DFA, NFA, converting NFA to DFA}\\ \quad\bullet\ \text{transition function, states, start state, accepting states}\end{array}\right)$$

- This is a **N**on-deterministic **F**inite **A**utomaton
- **Forward-arc** $\textcircled{j} \longrightarrow \textcircled{j{+}1}$ labelled with $P[j]$
- State j expresses "we have j leftmost characters of P'
- NFA accepts T if and only if T contains P

But evaluating NFAs is very slow.

String matching with DFA

Can show: There exists an equivalent **D**eterministic **F**inite **A**utomaton:



- Same states, forward-arcs, start state, accepting states.
- Easy to test whether P is in T.
- But how do we find the backward-arcs?

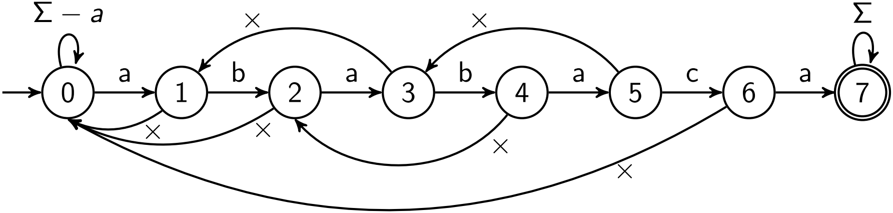
(We will not give the details of this since there is an even better automaton.)

Outline

9 String Matching

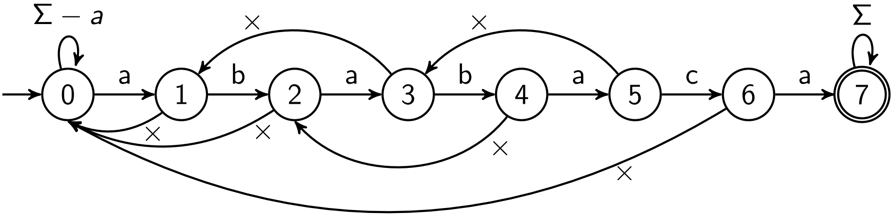
- Introduction
- Karp-Rabin Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays
- Conclusion

Knuth-Morris-Pratt Motivation



- Same states, forward-arcs, start state, accepting states.
- Use a new type of transition × (*"failure"*) but stay deterministic:
    - One per state $1, \ldots, m-1$, use it only if no other transition fits.
    - Does not consume a character.

Knuth-Morris-Pratt Motivation



- Same states, forward-arcs, start state, accepting states.
- Use a new type of transition × (*"failure"*) but stay deterministic:
  - One per state $1, \ldots, m-1$, use it only if no other transition fits.
  - Does not consume a character.
- We will (later) determine failure-arcs such that the automaton accepts T if and only if T contains `ababaca`
- Store the failure-arcs in an array $F[0..m-1]$ (index off by one!):

j	0	1	2	3	4	5	6
failure arc from $\textcircled{j}$ to	NA	0	0	1	2	3	0
$F[j]$	0	0	1	2	3	0	?

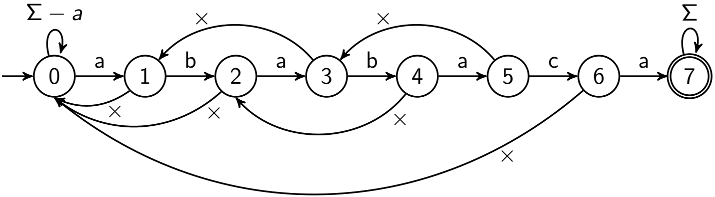
Knuth-Morris-Pratt Algorithm

There is no need to build an automaton; ‘parsing’ can be described with variables and failure-array F.

```
KMP::pattern-matching(T, P)
$F \leftarrow$ compute-failure-array$(P)$
$i \leftarrow 0$                          // character of T to parse
$j \leftarrow 0$                          // current state
while $i < n$ do
// inv:  $P[0..j-1]$ is a suffix of $T[0..i-1]$
        if $P[j] = T[i]$
                if $j = m - 1$ then return “found at guess $i - m + 1$”
                else                                        // forward-arc
                        $i \leftarrow i + 1$
                        $j \leftarrow j + 1$
        else // next character is mismatch
                if $j > 0$ then $j \leftarrow F[j-1]$             // failure-arc
                else $i \leftarrow i + 1$                          // loop at 0
return FAIL
```

String matching with KMP – Example

Example: T = abababbabaca, P = ababaca

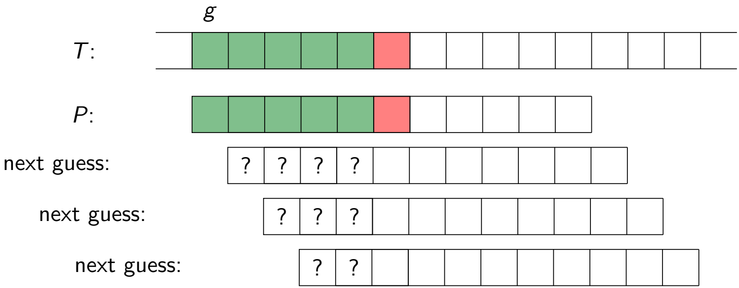


T :	a	b	a	b	a	b	b	c	a	b	a	b	a	c	a
	a	b	a	b	a	×									
			(a)	(b)	(a)	b	×								
					(a)	(b)	×								
							×								
								×							
									a	b	a	b	a	c	a
state :	1	2	3	4	5	3, 4	2, 0	0	1	2	3	4	5	6	7

(after reading this character)

String matching with KMP – Failure-function

Assume that we reach a mismatch (say at guess g):

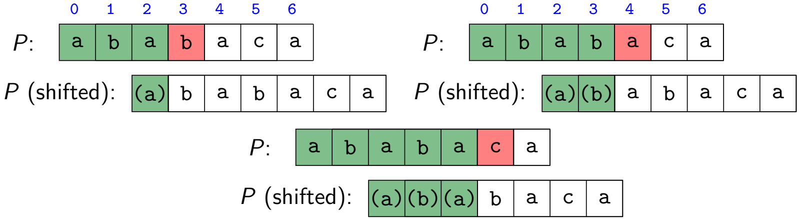


- Consider guesses at index $g+1, g+2, \ldots$. Could they match?
- The matched characters will rule out many of these guesses.
- We want the leftmost guess that cannot be ruled out.

- **Note:** This depends *only* on P, and not on T.
  In particular it can be *pre-computed*.

String matching with KMP – Failure-function

- Consider again the example P = ababaca.



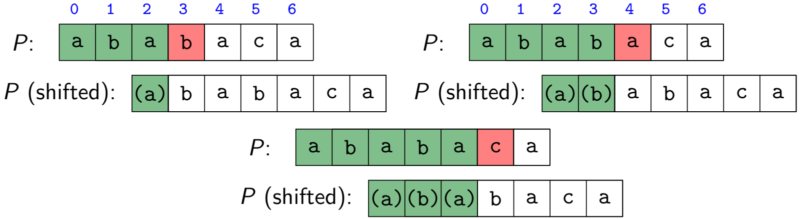
	0	1	2	3	4	5	6	
P:	a	b	a	b	a	c	a	
P (shifted):		(a)	b	a	b	a	c	a

	0	1	2	3	4	5	6		
P:	a	b	a	b	a	c	a		
P (shifted):			(a)	(b)	a	b	a	c	a

P:	a	b	a	b	a	c	a			
P (shifted):			(a)	(b)	(a)	b	a	c	a	

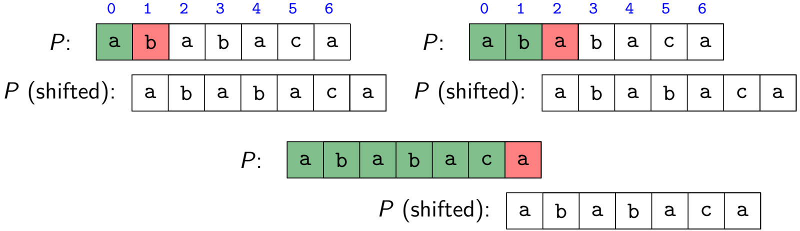
String matching with KMP – Failure-function

- Consider again the example $P =$ ababaca.



0 1 2 3 4 5 6
P: a b a b a c a
P (shifted): (a) b a b a c a
0 1 2 3 4 5 6
P: a b a b a c a
P (shifted): (a) (b) a b a c a
P: a b a b a c a
P (shifted): (a) (b) (a) b a c a

- Sometimes nothing fits. Then shift past matched part.



0 1 2 3 4 5 6
P: a b a b a c a
P (shifted): a b a b a c a
0 1 2 3 4 5 6
P: a b a b a c a
P (shifted): a b a b a c a
P: a b a b a c a
P (shifted): a b a b a c a

- Store in $F[\cdot]$ how many characters are matched in new shift.

String matching with KMP – Failure function

- **Definition:** $F[j]$ = number of re-used characters if $P[0..j]$ matched
- For $P =$ `ababaca`, we get

j	0	1	2	3	4	5	6
$F[j]$	0	0	1	2	3	0	?

(This matches exactly the failure-arcs in KMP-automaton.)

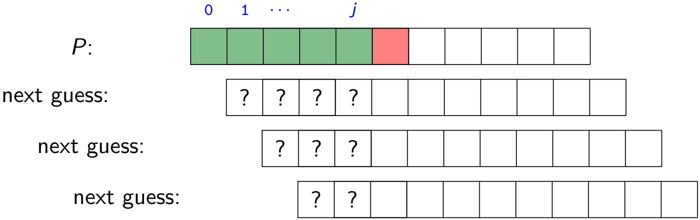
String matching with KMP – Failure function

- **Definition:** $F[j]$ = number of re-used characters if $P[0..j]$ matched
- For P = `ababaca`, we get

j	0	1	2	3	4	5	6
$F[j]$	0	0	1	2	3	0	?

(This matches exactly the failure-arcs in KMP-automaton.)

- In general: We must find a long prefix of P that is a suffix of $P[0..j]$ (except it should not be **all** of $P[0..j]$)



- Equivalently: We must find a long prefix of P that is a suffix of $P[1..j]$

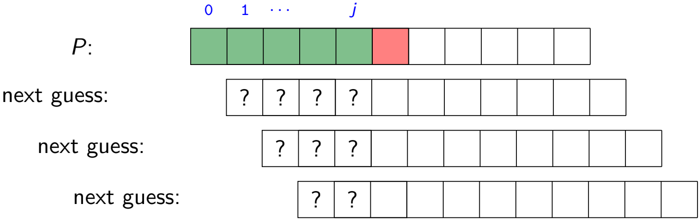
String matching with KMP – Failure function

- **Definition:** $F[j]$ = number of re-used characters if $P[0..j]$ matched
- For P = ababaca, we get

j	0	1	2	3	4	5	6
$F[j]$	0	0	1	2	3	0	?

(This matches exactly the failure-arcs in KMP-automaton.)

- In general: We must find a long prefix of P that is a suffix of $P[0..j]$ (except it should not be **all** of $P[0..j]$)



- Equivalently: We must find a long prefix of P that is a suffix of $P[1..j]$

Result: $F[j]$ = length of the longest prefix of P that is a suffix of $P[1..j]$.

KMP Failure Array – Easy Computation

$F[j] =$ **length of the longest prefix of** P **that is a suffix of** $P[1..j]$**.**

Write down all prefixes (including empty word Λ).
Then for $j \in \{0, \ldots, m-1\}$ and each prefix of P
check whether the prefix is a suffix of $P[1..j]$.

j	$P[1..j]$	Prefixes of P	longest	$F[j]$
0	Λ	Λ, a, ab, aba, abab, ababa, ...	Λ	0
1	b	Λ, a, ab, aba, abab, ababa, ...	Λ	0
2	ba	Λ, a, ab, aba, abab, ababa, ...	a	1
3	bab	Λ, a, ab, aba, abab, ababa, ...	ab	2
4	baba	Λ, a, ab, aba, abab, ababa, ...	aba	3
5	babac	Λ, a, ab, aba, abab, ababa, ...	Λ	0
6	babaca	Λ, a, ab, aba, abab, ababa, ...	a	1

($F[m-1]$ is not needed for KMP automaton, but useful elsewhere)

This can clearly be computed in $O(m^3)$ time, but we can do better!

KMP Failure Array – Fast Computation

$F[q-1]$ is maximum ℓ such that $P[0..\ell-1]$ is a suffix of $P[1..q-1]$.

(For easier comparison, we have substituted $q \leftarrow j+1$.)

Idea: This is same as loop-invariant for KMP if we parse $P[1..q-1]$.

KMP Failure Array – Fast Computation

$F[q-1]$ is maximum ℓ such that $P[0..\ell-1]$ is a suffix of $P[1..q-1]$.

(For easier comparison, we have substituted $q \leftarrow j+1$.)

Idea: This is same as loop-invariant for KMP if we parse $P[1..q-1]$.

```
KMP::compute-failure-array(P)
Initialize array $F$ as all-0
$q \leftarrow 1$          // index of $P[1..m-1]$ to parse
$\ell \leftarrow 0$          // current state
while $j < m$ do
// inv:  $P[0..\ell-1]$ equals last $\ell$ characters of $P[1..q-1]$
    $F[q-1] \leftarrow \max\{F[q-1], \ell\}$
    if $P[q] = P[\ell]$
        $\ell \leftarrow \ell + 1$
        $q \leftarrow q + 1$
    else if $\ell > 0$ then $\ell \leftarrow F[\ell - 1]$
    else $q \leftarrow q + 1$
$F[m-1] \leftarrow \ell$
```

Note: $\ell < q$ at all times, so needed failure-arcs are already computed.

KMP Runtime

Parsing text T with $|T| = n$:

- Run-time is proportional to the number of arcs followed.
- Every loop and forward-arc consumes a character of T.
  So this happens at most n times
- For every failure-arc (leads left) there was a forward-arc that we followed earlier $\rightsquigarrow$ happens at most n times

KMP Runtime

Parsing text T with $|T| = n$:

- Run-time is proportional to the number of arcs followed.
- Every loop and forward-arc consumes a character of T.
  So this happens at most n times
- For every failure-arc (leads left) there was a forward-arc that we followed earlier $\rightsquigarrow$ happens at most n times

So the main routine (without *compute-failure-array*) takes $O(n)$ time.

KMP Runtime

Parsing text T with $|T| = n$:

- Run-time is proportional to the number of arcs followed.
- Every loop and forward-arc consumes a character of T.
  So this happens at most n times
- For every failure-arc (leads left) there was a forward-arc that we followed earlier $\rightsquigarrow$ happens at most n times

So the main routine (without *compute-failure-array*) takes $O(n)$ time.

compute-failure-array parses a text of length $m-1 \rightsquigarrow O(m)$ time.

KMP Runtime

Parsing text T with $|T| = n$:

- Run-time is proportional to the number of arcs followed.
- Every loop and forward-arc consumes a character of T.
  So this happens at most n times
- For every failure-arc (leads left) there was a forward-arc that we followed earlier $\rightsquigarrow$ happens at most n times

So the main routine (without *compute-failure-array*) takes $O(n)$ time.

compute-failure-array parses a text of length $m{-}1 \rightsquigarrow O(m)$ time.

Result: Pattern matching with Knuth-Morris-Pratt has $O(n+m)$ worst-case run-time.

KMP Runtime

Parsing text T with $|T| = n$:

- Run-time is proportional to the number of arcs followed.
- Every loop and forward-arc consumes a character of T.
  So this happens at most n times
- For every failure-arc (leads left) there was a forward-arc that we followed earlier $\rightsquigarrow$ happens at most n times

So the main routine (without *compute-failure-array*) takes $O(n)$ time.

compute-failure-array parses a text of length $m-1 \rightsquigarrow O(m)$ time.

Result: Pattern matching with Knuth-Morris-Pratt has $O(n+m)$ worst-case run-time.

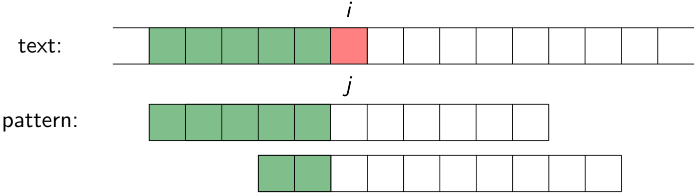
But we can do *even* better!

Outline

9 String Matching
- Introduction
- Karp-Rabin Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays
- Conclusion

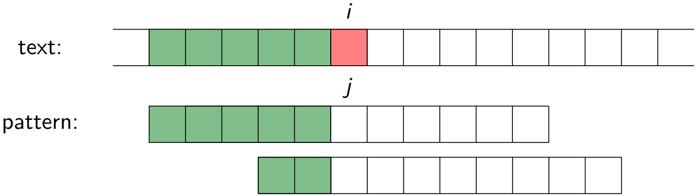
Towards the Boyer-Moore Algorithm

Recall: KMP eliminates guesses based on matched part of P.



Towards the Boyer-Moore Algorithm

Recall: KMP eliminates guesses based on matched part of P.

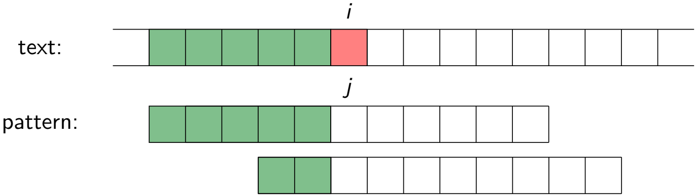


Boyer-Moore exploits *two* insights:

- Eliminate guesses based on matched part of P. (**good suffix heuristic**)—very similar to KMP.
- Eliminate guesses based on mismatched characters of T (**bad character jumps**)—this is new.

Towards the Boyer-Moore Algorithm

Recall: KMP eliminates guesses based on matched part of P.



Boyer-Moore exploits *two* insights:

- Eliminate guesses based on matched part of P.
  (**good suffix heuristic**)—very similar to KMP.
- Eliminate guesses based on mismatched characters of T
  (**bad character jumps**)—this is new.

The second insight turns out to be very helpful, and leads to fastest pattern matching on English text as long as we search *backwards*.

Forward-searching vs. reverse-searching

P: aldo
T: whereiswaldo

Forward-searching:

w	h	e	r	e	i	s	w	a	l	d	o
▨	▨	▨	▨								

Reverse-searching:

w	h	e	r	e	i	s	w	a	l	d	o
▨	▨	▨	▨								

Forward-searching vs. reverse-searching

P: aldo
T: whereiswaldo

Forward-searching:

w	h	e	r	e	i	s	w	a	l	d	o
a											

- `w` does not occur in P.
  $\Rightarrow$ shift pattern past `w`.

Reverse-searching:

w	h	e	**r**	e	i	s	w	a	l	d	o
			o								

- `r` does not occur in P.
  $\Rightarrow$ shift pattern past `r`.

Forward-searching vs. reverse-searching

P: aldo
T: whereiswaldo

Forward-searching:

w	**h**	e	r	e	i	s	w	a	l	d	o
▨ a	▨	▨	▨								
	▨ a	▨	▨	▨							

- `w` does not occur in *P*.
  ⇒ shift pattern past `w`.
- `h` does not occur in *P*.
  ⇒ shift pattern past `h`.

Reverse-searching:

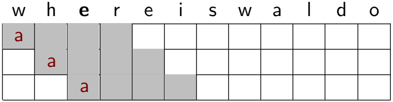
w	h	e	r	e	i	s	**w**	a	l	d	o
▨	▨	▨	▨ o								
				▨	▨	▨	▨ o				

- `r` does not occur in *P*.
  ⇒ shift pattern past `r`.
- `w` does not occur in *P*.
  ⇒ shift pattern past `w`.

Forward-searching vs. reverse-searching

P: aldo
T: whereiswaldo

Forward-searching:

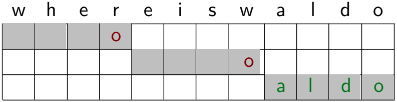


w	h	**e**	r	e	i	s	w	a	l	d	o
a											
	a										
		a									

- w does not occur in P.
  ⇒ shift pattern past w.
- h does not occur in P.
  ⇒ shift pattern past h.

With forward-searching, fewer guesses are ruled out.

Reverse-searching:



w	h	e	r	e	i	s	w	a	l	d	o
			o								
							o				
								a	l	d	o

- r does not occur in P.
  ⇒ shift pattern past r.
- w does not occur in P.
  ⇒ shift pattern past w.

This *bad character heuristic* works well with reverse-searching.

Bad character heuristic details

P: paper

T:	f	e	e	d	a	l	l	p	o	o	r	p	a	r	r	o	t	s

Bad character heuristic details

P : p a p e r

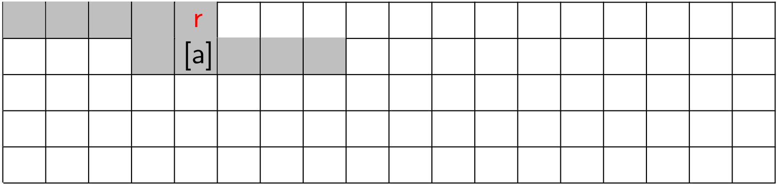
f	e	e	d	a	l	l	p	o	o	r	p	a	r	r	o	t	s
				r													

(1) Mismatched character in the text is a

Bad character heuristic details

P : p a p e r

T : f e e d **a** l l p o o r p a r r o t s



(1) Mismatched character in the text is `a`

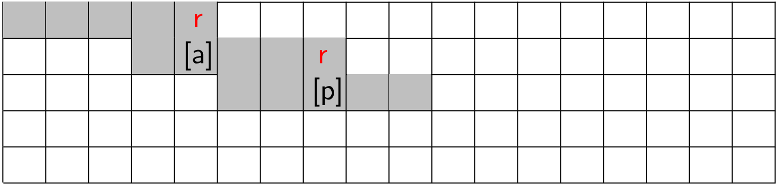
Shift the guess until `a` in P aligns with `a` in T

- All skipped guessed are impossible since they do not match `a`

Bad character heuristic details

P : p a p e r

T : f e e d a l l **p** o o r p a r r o t s



(1) Mismatched character in the text is `a`

Shift the guess until `a` in P aligns with `a` in T

- All skipped guessed are impossible since they do not match `a`

(2) Shift the guess until *last* `p` in P aligns with `p` in T

- Use “last” since we cannot rule out this guess.

Bad character heuristic details

P : p a p e r

T :	f	e	e	d	a	l	l	p	o	**o**	r	p	a	r	r	o	t	s
	▨	▨	▨	▨	r													
				▨	[a]	▨	▨	r										
						▨	▨	[p]	▨	r								
											▨	▨	▨	▨	▨			

(1) Mismatched character in the text is a

Shift the guess until a in *P* aligns with a in *T*

- All skipped guessed are impossible since they do not match a

(2) Shift the guess until *last* p in *P* aligns with p in *T*

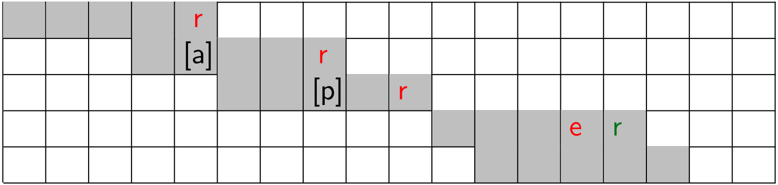
- Use "last" since we cannot rule out this guess.

(3) As before, shift completely past o since o is not in *P*.

Bad character heuristic details

P : p a p e r

T : f e e d a l l p o o r p a **r** r o t s



(1) Mismatched character in the text is `a`

Shift the guess until `a` in P aligns with `a` in T

- All skipped guessed are impossible since they do not match `a`

(2) Shift the guess until *last* `p` in P aligns with `p` in T

- Use "last" since we cannot rule out this guess.

(3) As before, shift completely past `o` since `o` is not in P.

(4) The shift that aligns with `r` has already been ruled out.

- Bad character heuristic not helpful, shift guess right by one unit.

Bad character heuristic details

P : p a p e r

T :	f	e	e	d	a	l	l	p	o	o	r	p	a	r	r	**o**	t	s
	▨	▨	▨	▨	r													
				▨	[a]	▨	▨	r										
						▨	▨	[p]	▨	r								
											▨	▨	▨	e	r			
												▨	▨	▨	▨	r		

(1) Mismatched character in the text is a

Shift the guess until a in P aligns with a in T

- All skipped guessed are impossible since they do not match a

(2) Shift the guess until *last* p in P aligns with p in T

- Use "last" since we cannot rule out this guess.

(3) As before, shift completely past o since o is not in P.

(4) The shift that aligns with r has already been ruled out.

- Bad character heuristic not helpful, shift guess right by one unit.

(5) Shift completely past o → out of bounds.

Boyer-Moore Algorithm – incomplete

```
Boyer-Moore::pattern-matching($T, P$)
$i \leftarrow m-1$,    $j \leftarrow m-1$
while $i < n$ and $j \geq 0$ do
        // current guess begins at index $i-j$
        if $T[i] = P[j]$
                $i \leftarrow i-1$          // go backwards
                $j \leftarrow j-1$
        else
                $i \leftarrow$ ???
                $j \leftarrow m-1$          // restart from right end
if $j = -1$ return "found at $T[i{+}1..i{+}m]$"
else return FAIL
```

Two steps missing:

- Need to pre-compute for all characters where they are in P.
- Need to determine how to do the update i at a mismatch.

Helper-Array for Bad Character Heuristic

- Build the helper-array L mapping Σ to integers
- $L[c]$ is the largest index i such that $P[i] = c$

Pattern:

0	1	2	3	4
p	a	p	e	r

Helper-array:

char	p	a	e	r	all others
$L[\cdot]$	2	1	3	4	?

Helper-Array for Bad Character Heuristic

- Build the helper-array L mapping Σ to integers
- $L[c]$ is the largest index i such that $P[i] = c$

Pattern:

0	1	2	3	4
p	a	p	e	r

Helper-array:

char	p	a	e	r	all others
$L[\cdot]$	2	1	3	4	?

- What value should be used if c not in P?
  - We want to shift past c entirely.
  - Equivalently view this as 'c is to the left of P'
  - Equivalently: c is at $P[-1]$, so set $L[c] = -1$

Helper-Array for Bad Character Heuristic

- Build the helper-array L mapping Σ to integers
- $L[c]$ is the largest index i such that $P[i] = c$

Pattern:

0	1	2	3	4
p	a	p	e	r

Helper-array:

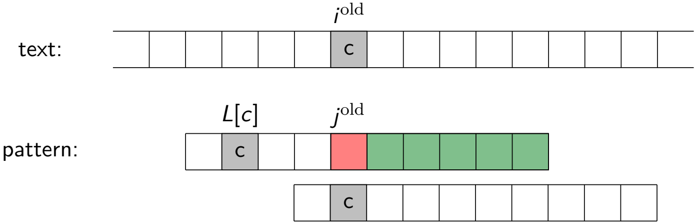
char	p	a	e	r	all others
$L[\cdot]$	2	1	3	4	?

- What value should be used if c not in P?
  - We want to shift past c entirely.
  - Equivalently view this as 'c is to the left of P'
  - Equivalently: c is at $P[-1]$, so set $L[c] = -1$
- We can build this in time $O(m + |\Sigma|)$ with simple for-loop

```
BoyerMoore::bad-character-helper-array($P[0..m-1]$)
    initialize array $L$ indexed by $\Sigma$ with all $-1$
    for $j \leftarrow 0$ to $m-1$ do $L[P[j]] \leftarrow j$
    return $L$
```

Bad character heuristic – update formula

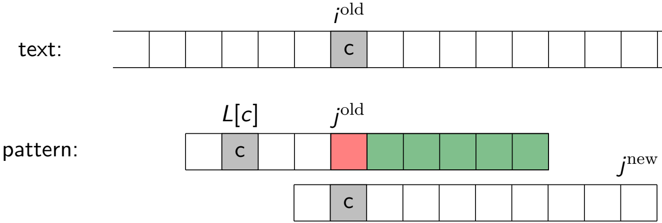
“Good” case: $L[c] < j$, so c is left of $P[j]$.



Want: i^{new} = index in T that corresponds to j^{new}.

Bad character heuristic – update formula

“Good” case: $L[c] < j$, so c is left of $P[j]$.

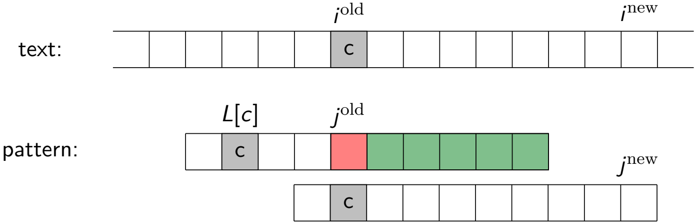


Want: i^{new} = index in T that corresponds to j^{new}.

- Δ_1 = amount that we should shift $= j^{\text{old}} - L[c]$

Bad character heuristic – update formula

"Good" case: $L[c] < j$, so c is left of $P[j]$.

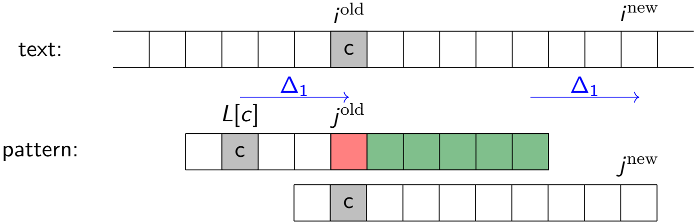


Want: i^{new} = index in T that corresponds to j^{new}.

- Δ_1 = amount that we should shift = $j^{\text{old}} - L[c]$
- Δ_2 = how much we had compared = $(m-1) - j^{\text{old}}$

Bad character heuristic – update formula

"Good" case: $L[c] < j$, so c is left of $P[j]$.



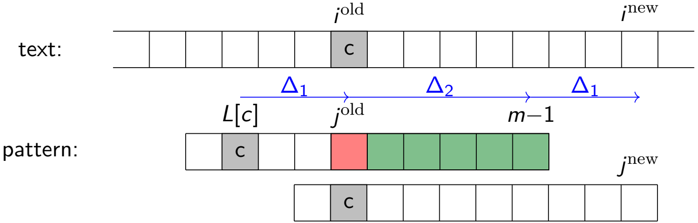
Want: i^{new} = index in T that corresponds to j^{new}.

- Δ_1 = amount that we should shift $= j^{\text{old}} - L[c]$
- Δ_2 = how much we had compared $= (m-1) - j^{\text{old}}$
- $i^{\text{new}} = i^{\text{old}} + \Delta_2 + \Delta_1 = i^{\text{old}} + (m-1) - L[c]$

$$\boxed{= i^{\text{old}} + (m-1) - \min\left\{L[c], j^{\text{old}}-1\right\}}$$

Bad character heuristic – update formula

“Good” case: $L[c] < j$, so c is left of $P[j]$.



Want: i^{new} = index in T that corresponds to j^{new}.

- Δ_1 = amount that we should shift = $j^{\text{old}} - L[c]$
- Δ_2 = how much we had compared = $(m-1) - j^{\text{old}}$
- $i^{\text{new}} = i^{\text{old}} + \Delta_2 + \Delta_1 = i^{\text{old}} + (m-1) - L[c]$

$$\boxed{= i^{\text{old}} + (m-1) - \min\left\{L[c], j^{\text{old}}-1\right\}}$$

Can show: The same formula also holds for the other cases.

Boyer-Moore Algorithm

```
Boyer-Moore::pattern-matching($T, P$)        // simplified version
$L \leftarrow$ bad-character-helper-array($P$)
$i \leftarrow m-1$,    $j \leftarrow m-1$
while $i < n$ and $j \geq 0$ do
    if $T[i] = P[j]$
        $i \leftarrow i-1$
        $j \leftarrow j-1$
    else
        $i \leftarrow i + m-1 - \min\{L[T[i]], j-1\}$
        $j \leftarrow m-1$
if $j = -1$ return "found at $T[i+1..i+m]$"
else return FAIL
```

For *full* Boyer-Moore algorithm:

- precompute helper-array G for good-suffix heuristic from P
- update-formula becomes $i \leftarrow i + m-1 - \min\{L[T[i]], G[j]\}$

Good Suffix Heuristic

Doing examples is easy, but computing G is complicated (no details).

P :	G	C	G	C	T	A	G	C										
T :	G	C	G	C	T	G	G	C	C	A	G	C	G	C	T	A	G	C
						A	G	C										

Good Suffix Heuristic

Doing examples is easy, but computing G is complicated (no details).

P :	G	C	G	C	T	A	G	C										
T :	G	C	G	C	T	G	G	C	C	A	G	C	G	C	T	A	G	C
						A	G	C										

Do smallest shift so that matched text GC fits the new guess.

							(G)	(C)										

Good Suffix Heuristic

Doing examples is easy, but computing G is complicated (no details).

P :	G	C	G	C	T	A	G	C										
T :	G	C	G	C	T	G	G	C	C	A	G	C	G	C	T	A	G	C

					A	G	C										

Do smallest shift so that matched text GC fits the new guess.

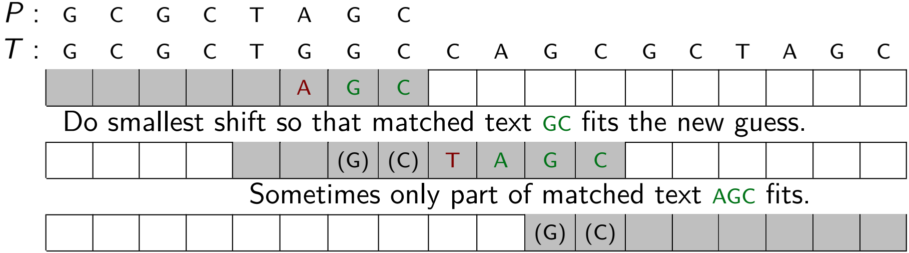
						(G)	(C)	T	A	G	C						

Sometimes only part of matched text AGC fits.

										(G)	(C)						

Good Suffix Heuristic

Doing examples is easy, but computing G is complicated (no details).



P : G C G C T A G C

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

A G C

Do smallest shift so that matched text GC fits the new guess.

(G) (C) T A G C

Sometimes only part of matched text AGC fits.

(G) (C)

Summary:

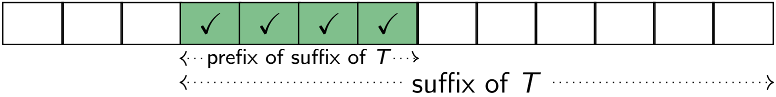
- Boyer-Moore performs very well (even without good suffix heuristic).
- On typical *English text* Boyer-Moore looks at only ≈ 25% of T
- Worst-case run-time for is $O(mn)$, but in practice much faster.
  [There are ways to ensure $O(n)$ run-time. No details.]

Outline

9 String Matching
- Introduction
- Karp-Rabin Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays
- Conclusion

Tries of Suffixes and Suffix Trees

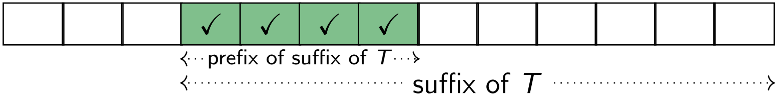
Recall: P occurs in $T \Leftrightarrow P$ is a prefix of some suffix of T.



- **Idea:** Build a data structure that stores all suffixes of T.
  - So we preprocess the text T rather than the pattern P
  - This is useful if we want to search for many patterns P within the same fixed text T.
- Naive idea: Store the suffixes in a trie.
  - $|T| = n \Rightarrow$ the $n+1$ suffixes together have $\binom{n+1}{2} \in \Theta(n^2)$ characters
  - This wastes space

Tries of Suffixes and Suffix Trees

Recall: P occurs in $T \Leftrightarrow P$ is a prefix of some suffix of T.



prefix of suffix of T

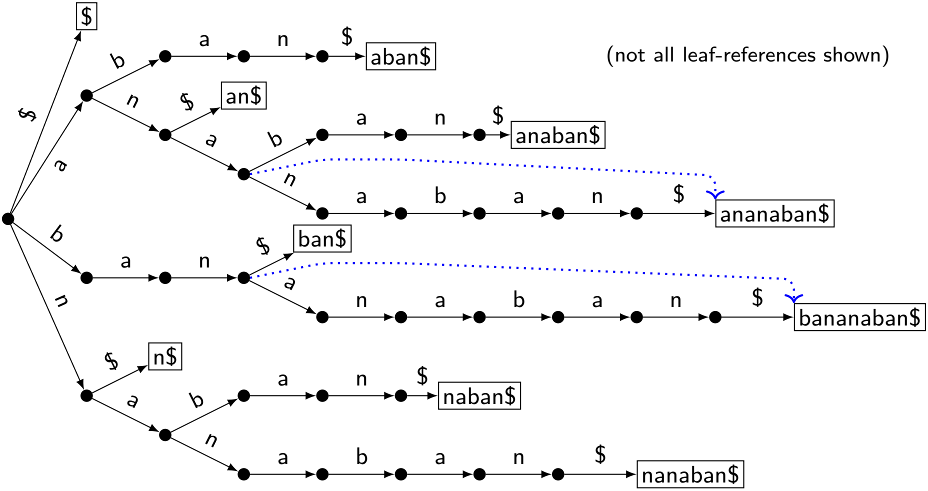
suffix of T

- **Idea:** Build a data structure that stores all suffixes of T.
  - So we preprocess the text T rather than the pattern P
  - This is useful if we want to search for many patterns P within the same fixed text T.
- Naive idea: Store the suffixes in a trie.
  - $|T| = n \Rightarrow$ the $n+1$ suffixes together have $\binom{n+1}{2} \in \Theta(n^2)$ characters
  - This wastes space
- **Suffix tree** saves space in multiple ways:
  - Store suffixes implicitly via indices into T.
  - Use a compressed trie.
  - Then the space is $O(n)$ since we store $n+1$ words.

Trie of suffixes: Example

$T =$ bananaban has suffixes

{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n, Λ}

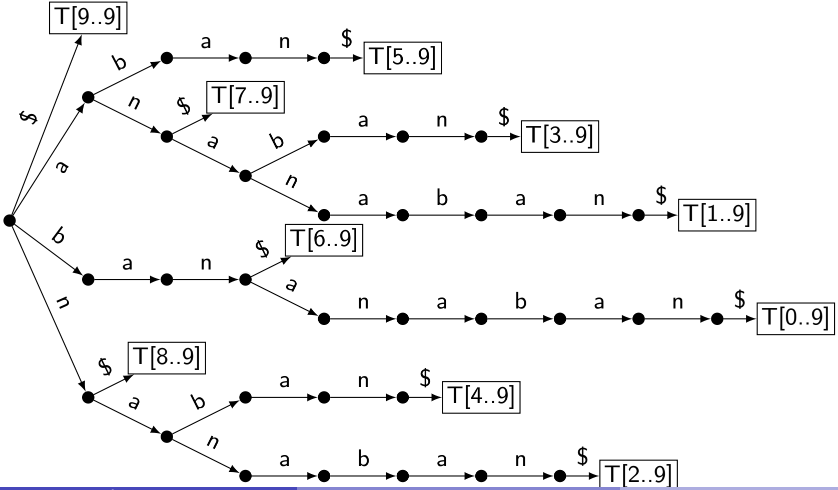


(not all leaf-references shown)

Tries of suffixes

Store suffixes via indices:

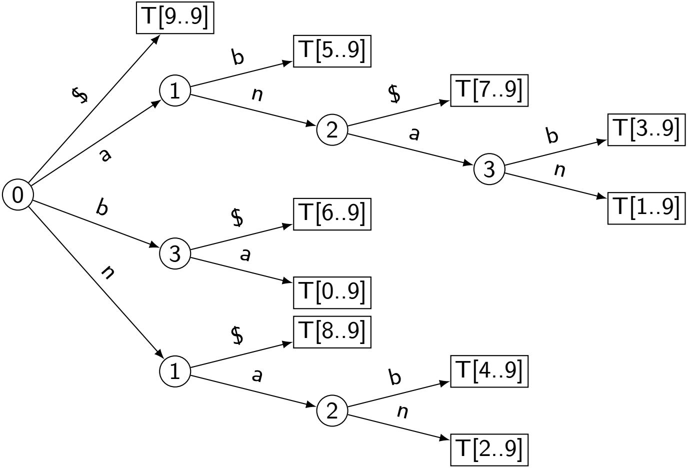
	0	1	2	3	4	5	6	7	8	9
$T =$	b	a	n	a	n	a	b	a	n	\$



Suffix tree

Suffix tree: Compressed trie of suffixes where leaves store indices.

	0	1	2	3	4	5	6	7	8	9
$T =$	b	a	n	a	n	a	b	a	n	\$



More on Suffix Trees

Pattern Matching:

- *prefix-search* for P in compressed trie.
- This returns longest word with prefix P, hence leftmost occurrence.
- Run-time: $O(|\Sigma|m)$.

Building:

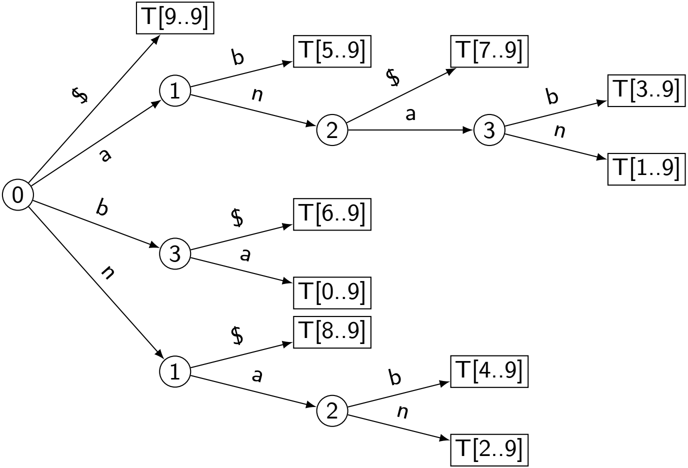
- Text T has n characters and $n+1$ suffixes
- We can build the suffix tree by inserting each suffix of T into a compressed trie. This takes time $\Theta(|\Sigma|n^2)$.
- There *is* a way to build a suffix tree of T in $\Theta(|\Sigma|n)$ time. This is quite complicated and beyond the scope of the course.

Summary: Theoretically good, but construction is slow or complicated, and lots of space-overhead $\rightsquigarrow$ rarely used.

Pattern Matching in Suffix Tree: Example 1

	0	1	2	3	4	5	6	7	8	9
$T =$	b	a	n	a	n	a	b	a	n	$

	0	1	2
$P =$	a	n	n



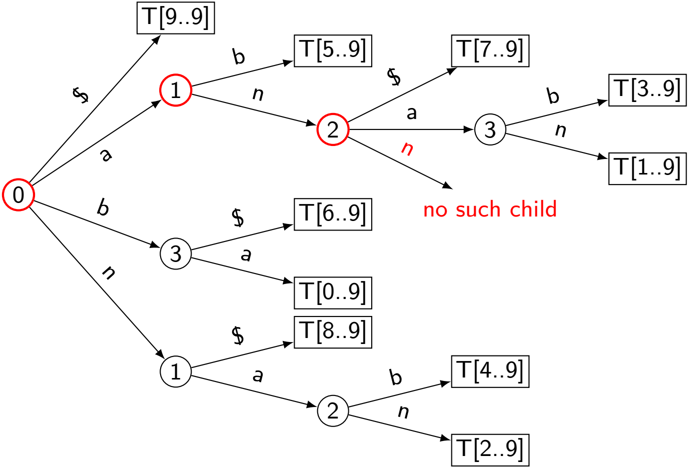
Pattern Matching in Suffix Tree: Example 1

$T =$

0	1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	b	a	n	$

$P =$

0	1	2
a	n	n



If 'no such child' before we reach end of P: `FAIL`

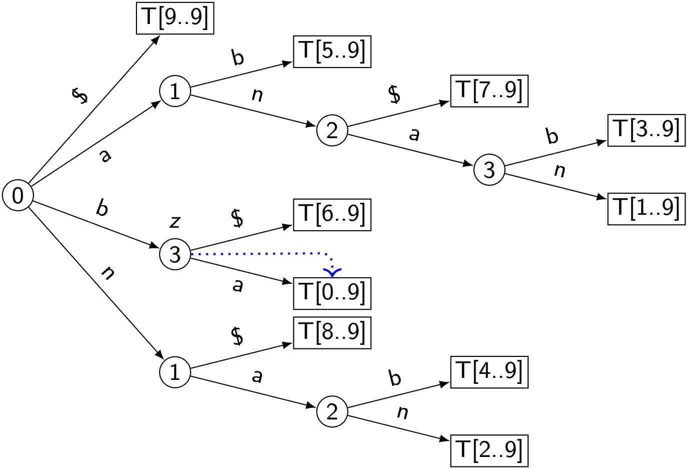
Pattern Matching in Suffix Tree: Example 2

$T =$

0	1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	b	a	n	$

$P =$

0	1
b	e

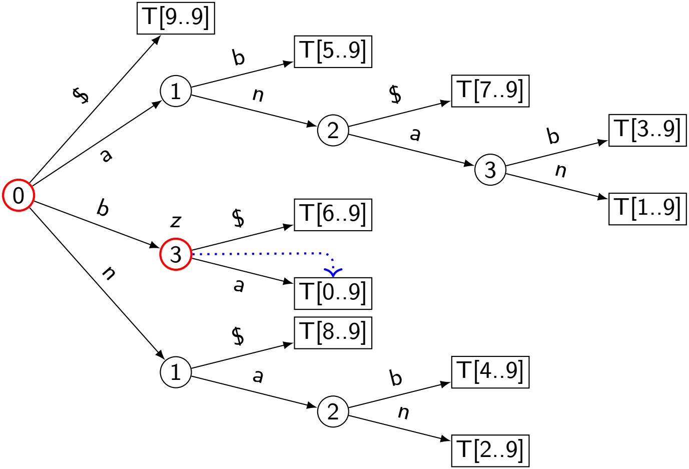


If we reach node z at end of P: Compare P to z.*leaf*.

Pattern Matching in Suffix Tree: Example 2

	0	1	2	3	4	5	6	7	8	9
$T =$	b	a	n	a	n	a	b	a	n	\$

	0	1
$P =$	b	e



,

If we reach node z at end of P: Compare P to $z.$*leaf*.

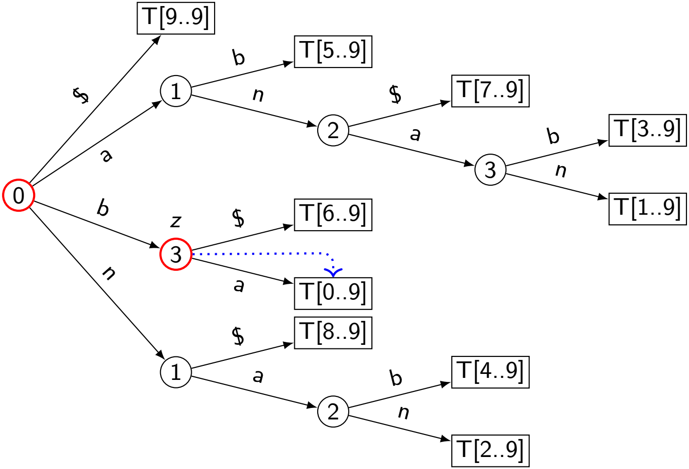
Pattern Matching in Suffix Tree: Example 2

$T =$

0	1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	b	a	n	$

$P =$

0	1
b	e



,,

If we reach node z at end of P: Compare P to z.*leaf*.

Outline

9 String Matching

- Introduction
- Karp-Rabin Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays
- Conclusion

Suffix Arrays

- Relatively recent development (popularized in the 1990s)
- Sacrifice some performance for simplicity:
  - Slightly slower (by a log-factor) than suffix trees.
  - Much easier to build.
  - Much simpler pattern matching.
  - Very little space; only one array.

Idea:

- Store suffixes implicitly (by storing start-indices)
- Store *sorting permutation* of the suffixes of T.

Suffix Array Example

	0	1	2	3	4	5	6	7	8
Text T:	b	a	n	a	n	a	b	a	n

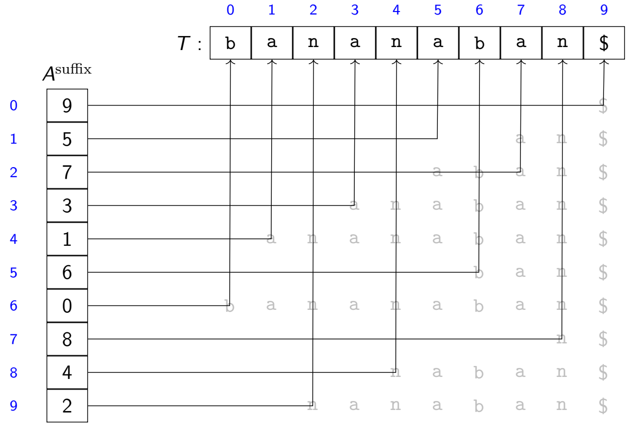
i	suffix $T[i..n]$
0	bananaban$
1	ananaban$
2	nanaban$
3	anaban$
4	naban$
5	aban$
6	ban$
7	an$
8	n$
9	$

→

sort lexicographically

j	$A^{\text{suffix}}[j]$	
0	9	$
1	5	aban$
2	7	an$
3	3	anaban$
4	1	ananaban$
5	6	ban$
6	0	bananaban$
7	8	n$
8	4	naban$
9	2	nanaban$

Suffix array



	0	1	2	3	4	5	6	7	8	9
T :	b	a	n	a	n	a	b	a	n	$

	A^{suffix}	
0	9	$
1	5	a n $
2	7	a b a n $
3	3	a n a b a n $
4	1	a n a n a b a n $
5	6	b a n $
6	0	b a n a n a b a n $
7	8	n $
8	4	n a b a n $
9	2	n a n a b a n $

We do *not* store the suffixes, but they are easy to retrieve if needed.

Suffix Array Construction

- Easy to construct using *MSD-Radix-Sort*.
  - Pad suffixes with trailing $ to achieve equal length.
  - Fast in practice; suffixes are unlikely to share many leading characters.
  - But worst-case run-time is $\Theta(n^2)$
    - ★ n rounds of recursions (have n chars)
    - ★ Each round takes $\Theta(n)$ time (bucket-sort)

Suffix Array Construction

- Easy to construct using *MSD-Radix-Sort*.
  - Pad suffixes with trailing \$ to achieve equal length.
  - Fast in practice; suffixes are unlikely to share many leading characters.
  - But worst-case run-time is $\Theta(n^2)$
    - ★ n rounds of recursions (have n chars)
    - ★ Each round takes $\Theta(n)$ time (bucket-sort)
- **Idea:** We do not need n rounds!

  (
  - Consider sub-array after one round.
  - These have same leading char. Ties are broken by rest of words.
  - But rest of words are also suffixes ⇝ sorted elsewhere
  - We can double length of sorted part every round.

)

  - $O(\log n)$ rounds enough ⇒ $O(n \log n)$ **run-time**
  - You do not need to know details (⇝ cs482).
- Construction-algorithm: MSD-radix-sort plus some bookkeeping
  - A bit complicated to explain but easy to implement

Pattern matching in suffix arrays

- Suffix array stores suffixes (implicitly) in sorted order.
- **Idea:** apply binary search!

P =ban:

	j	$A^{\text{suffix}}[j]$	$T[A^{\text{suffix}}[j]..n-1]$
$\ell \rightarrow$	0	9	\$
	1	5	aban\$
	2	7	an\$
	3	3	anaban\$
$\nu \rightarrow$	4	1	ananaban\$
	5	6	ban\$
	6	0	bananaban\$
	7	8	n\$
	8	4	naban\$
$r \rightarrow$	9	2	nanaban\$

Pattern matching in suffix arrays

- Suffix array stores suffixes (implicitly) in sorted order.
- **Idea:** apply binary search!

P =ban:

	j	$A^{\text{suffix}}[j]$	$T[\,A^{\text{suffix}}[j]..n{-}1]$
	0	9	\$
	1	5	aban\$
	2	7	an\$
	3	3	anaban\$
	4	1	ananaban\$
$\ell \rightarrow$	5	6	ban\$
	6	0	bananaban\$
$\nu \rightarrow$	7	8	n\$
	8	4	naban\$
$r \rightarrow$	9	2	nanaban\$

Pattern matching in suffix arrays

- Suffix array stores suffixes (implicitly) in sorted order.
- **Idea:** apply binary search!

P =ban:

	j	$A^{\text{suffix}}[j]$	$T[A^{\text{suffix}}[j]..n-1]$
	0	9	$
	1	5	aban$
	2	7	an$
	3	3	anaban$
	4	1	ananaban$
$\nu=\ell \rightarrow$	5	6	ban$ found
$r \rightarrow$	6	0	bananaban$
	7	8	n$
	8	4	naban$
	9	2	nanaban$

Pattern matching in suffix arrays

- Suffix array stores suffixes (implicitly) in sorted order.
- **Idea:** apply binary search!

P =ban:

	j	$A^{\text{suffix}}[j]$	$T[A^{\text{suffix}}[j]..n-1]$
	0	9	\$
	1	5	aban\$
	2	7	an\$
	3	3	anaban\$
	4	1	ananaban\$
$\nu = \ell \rightarrow$	5	6	ban\$ found
$r \rightarrow$	6	0	bananaban\$
	7	8	n\$
	8	4	naban\$
	9	2	nanaban\$

- $O(\log n)$ comparisons.
- Each comparison is a *strncmp* of P with a suffix
- $O(m)$ time per comparison ⇒ **run-time** $O(m \log n)$

Pattern matching in suffix arrays

```
SuffixArray::pattern-matching($T, P, A^{\text{suffix}}$)
$\ell \leftarrow 0$, $r \leftarrow$ last index of $A^{\text{suffix}}$
while ($\ell \leq r$)
    $\nu \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
    $g \leftarrow A^{\text{suffix}}[\nu]$     // suffix of middle index begins at $T[g]$
    $s \leftarrow$ strncmp($T, P, g, m$)
            // Case $g + m > n$ is handled correctly if $T$ has end-sentinel
    if ($s < 0$) do $\ell \leftarrow \nu + 1$
    else if ($s > 0$) do $r \leftarrow \nu - 1$
    else return "found at guess $g$"
return FAIL
```

- Does not always return leftmost occurrence.
- Can find leftmost occurrence (and reduce run-time to $O(m + \log n)$) with further pre-computations (no details).

Outline

9 String Matching

- Introduction
- Karp-Rabin Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays
- Conclusion

String Matching Conclusion

		Preprocess P				Preprocess T	
	Brute-Force	**Karp-Rabin**	**DFA**	**Knuth-Morris-Pratt**	**Boyer-Moore**	**Suffix Tree**	**Suffix Array**
Preproc.	—	$O(m)$	$O(m\lvert\Sigma\rvert)$	$O(m)$	$O(m)$	$O(n^2\lvert\Sigma\rvert)$ $[O(n\lvert\Sigma\rvert)]$	$O(n \log n)$ $[O(n)]$
Search time	$O(nm)$	$O(n+m)$ expected	$O(n)$	$O(n)$	$O(n)$ or better	$O(m\lvert\Sigma\rvert)$	$O(m \log n)$ $[O(m + \log n)]$
Extra space	—	$O(1)$	$O(m\lvert\Sigma\rvert)$	$O(m)$	$O(m)$	$O(n)$	$O(n)$

(Some additive $|\Sigma|$-terms are not shown.)

- Our algorithms stopped once they have found one occurrence.
- Most of them can be adapted to find *all* occurrences within the same worst-case run-time.