CS 240E – Data Structures and Data Management (Enriched)

Module 9: String Matching

Therese Biedl
Based on lecture notes by many previous cs240 instructors

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

Winter 2025

version *2025–03–11 14:24*

Outline

9 String Matching

- Introduction
- Karp-Rabin Algorithm
- Skip-heuristics
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays

Outline

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.,

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

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

Pattern Matching Observation

Recall:

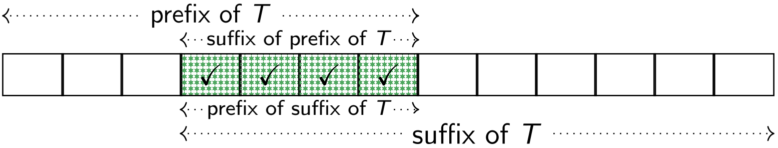
- **Substring** $T[i..j]$ for $0 \le i \le j{+}1 \le n$: a string of length $j - i + 1$ which consists of characters $T[i], \ldots T[j]$ in order.
- **Prefix** of T: a substring $T[0..i{-}1]$ of T for some $0 \le i \le n$.
- **Suffix** of T: a substring $T[i..n{-}1]$ of T for some $0 \le i \le n$.
- 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 $T[g]$. 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*.

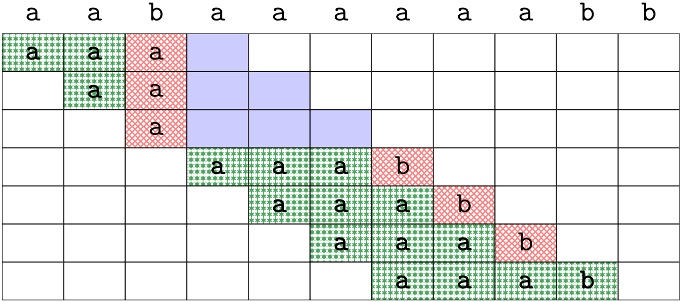
We will diagram a single run of any pattern matching algorithm by a matrix of checks, where each row represents a single guess (shaded gray).

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

Brute-force idea: Check every possible guess.

Brute-Force Example

- Example: $T =$ abbbababbab, $P =$ aaab



- What is the worst possible input?

Brute-Force Example

- Example: T = `abbbababbab`, P = `aaab`

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

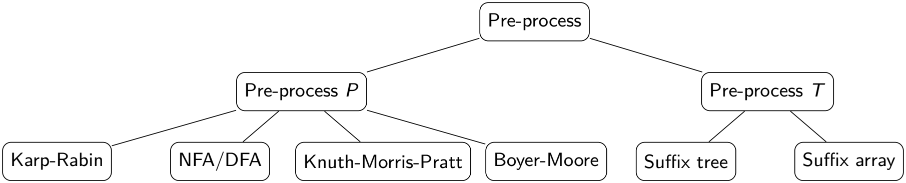
- What is the worst possible input? $P = a^{m-1}b,\ T = a^n$
- Worst case performance $\Theta((n-m+1)\cdot m)$
- This is too slow (quadratic if $m \approx n/2$).

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.



Outline

9 String Matching

- Introduction
- Karp-Rabin Algorithm
- Skip-heuristics
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays

Karp-Rabin Fingerprint Algorithm – Idea

Idea: Use **fingerprints** to eliminate guesses

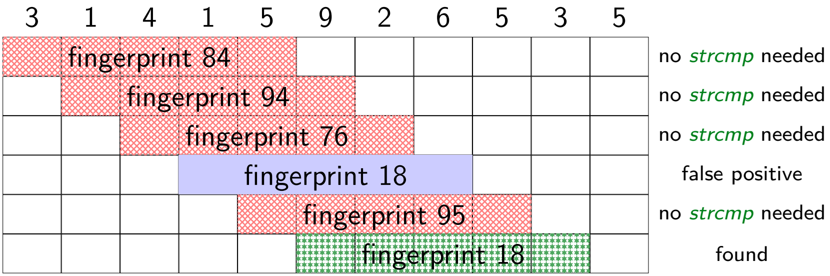
- Need function $h : \{\text{strings of length } m\} \to \{0, \ldots, M-1\}$
  (Call these 'hash-function' and 'table-size', but there is no dictionary here)
- **Insight:** If $h(P) \neq h(T[g..g+m-1])$ then guess g cannot work

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 $M = 97$:

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

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



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 ← suitable prime number
    h_P ← h(P[0..m−1)])
    s ← R^(m−1) mod M
    h_T ← h(T[0..m−1)])
    for g ← 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 ← ((h_T − T[g] · s) · R + T[g+m]) mod 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
- Skip-heuristics
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays

Skip-heuristics

We now make the brute-force algorithm smarter in a different way.

- Exploit information gained during *strncmp* to rule out guesses

P : a b a b a c c

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

a	b	a	b												

Skip-heuristics

We now make the brute-force algorithm smarter in a different way.

- Exploit information gained during *strncmp* to rule out guesses

P : a b a b a c c

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

a b a b

- **Good prefix**: The matched prefix of P (here aba).

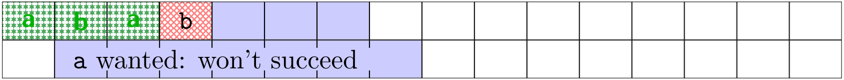
Skip-heuristics

We now make the brute-force algorithm smarter in a different way.

- Exploit information gained during *strncmp* to rule out guesses

P : a b a b a c c

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

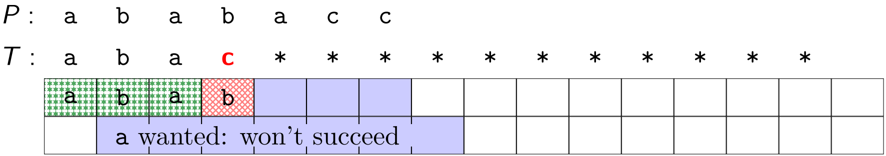


- **Good prefix**: The matched prefix of P (here aba).
  New guess must match aligned characters.

Skip-heuristics

We now make the brute-force algorithm smarter in a different way.

- Exploit information gained during *strncmp* to rule out guesses



- **Good prefix**: The matched prefix of P (here aba).
  New guess must match aligned characters.
- **Bad T-character**: The mismatched character of T (here c).

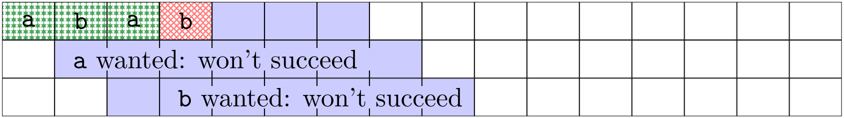
Skip-heuristics

We now make the brute-force algorithm smarter in a different way.

- Exploit information gained during *strncmp* to rule out guesses

P : a b a b a c c

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



- **Good prefix**: The matched prefix of P (here aba).
  New guess must match aligned characters.
- **Bad T-character**: The mismatched character of T (here c).
  New guess must match it.

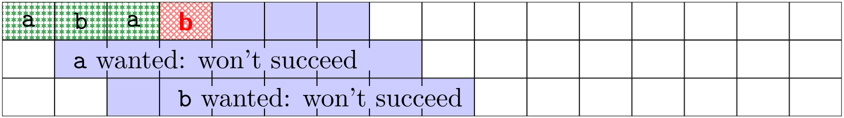
Skip-heuristics

We now make the brute-force algorithm smarter in a different way.

- Exploit information gained during *strncmp* to rule out guesses

P : a b a b a c c

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



- **Good prefix**: The matched prefix of P (here aba).
  New guess must match aligned characters.
- **Bad T-character**: The mismatched character of T (here c).
  New guess must match it.
- **Bad P-character**: The mismatched character of P (here b).
  New guess must mismatch it. (Implied by bad-T-character heuristic.)

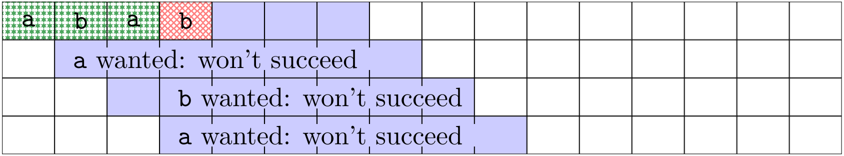
Skip-heuristics

We now make the brute-force algorithm smarter in a different way.

- Exploit information gained during *strncmp* to rule out guesses

P : a b a b a c c

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



- **Good prefix**: The matched prefix of P (here `aba`).
  New guess must match aligned characters.
- **Bad T-character**: The mismatched character of T (here `c`).
  New guess must match it.
- **Bad P-character**: The mismatched character of P (here `b`).
  New guess must mismatch it. (Implied by bad-T-character heuristic.)

Skip-heuristics

Any subset of the three heuristics gives a pattern-matching algorithm:
Do brute-force matching, except skip all guesses that can be ruled out.

Crucial: For all three heuristics, the guesses to skip depend only on

- the pattern P,
- the index j such that $P[0..j-1]$ was matched (the good suffix),
- the bad-T-character c,
- the bad-P-character $P[j]$.

They does *not* depend on text T, and therefore can be *pre-computed*.

Skip-heuristics

Any subset of the three heuristics gives a pattern-matching algorithm:
Do brute-force matching, except skip all guesses that can be ruled out.

Crucial: For all three heuristics, the guesses to skip depend only on

- the pattern P,
- the index j such that $P[0..j-1]$ was matched (the good suffix),
- the bad-T-character c,
- the bad-P-character $P[j]$.

They does *not* depend on text T, and therefore can be *pre-computed*.

First idea: Do pattern matching with *all* skip-heuristics.
Presumably this will skip many guesses $\rightsquigarrow$ fast in practice?

Skip-heuristics

Any subset of the three heuristics gives a pattern-matching algorithm:
Do brute-force matching, except skip all guesses that can be ruled out.

Crucial: For all three heuristics, the guesses to skip depend only on

- the pattern P,
- the index j such that $P[0..j-1]$ was matched (the good suffix),
- the bad-T-character c,
- the bad-P-character $P[j]$.

They does *not* depend on text T, and therefore can be *pre-computed*.

First idea: Do pattern matching with *all* skip-heuristics.
Presumably this will skip many guesses $\rightsquigarrow$ fast in practice?

No! The pre-computation is too slow. (Course notes have details.)

Outline

9 String Matching
- Introduction
- Karp-Rabin Algorithm
- Skip-heuristics
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays

Knuth-Morris-Pratt algorithm, incomplete

Surprisingly, using *only* the good-prefix heuristic works well enough.
This is the idea for **Knuth-Morris-Pratt** (KMP) pattern matching.

```
KMP::pattern-matching(T, P)
F ← compute and store failure-array, using only P
i ← 0, j ← 0                  // currently compare T[i] to P[j]
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                          // check next character
            i ← i + 1, j ← j + 1
    else                              // bad T-character is T[i]
        j ← [...] // read from F and old j
        i ← [...] // depends on j-update
return FAIL
```

Observe: j is always the number of matched characters of P.

Knuth-Morris-Pratt example

Example: Search for $P =$ `ababaca`. We first mismatch at $j = 5$.

a	b	a	b	a	b	b	c	a	b	a	b	a	c	a
a	b	a	b	a	×									

Knuth-Morris-Pratt example

Example: Search for $P =$ `ababaca`. We first mismatch at $j = 5$.

a	b	a	b	a	b	b	c	a	b	a	b	a	c	a
a	b	a	b	a	×									
		(a)	(b)	(a)										

- The good-prefix heuristic rules out one guess.
  In new guess we have three matched characters. $j^{\text{new}} = 3$, $i^{\text{new}} = i^{\text{old}}$.

Knuth-Morris-Pratt example

Example: Search for $P =$ `ababaca`. We first mismatch at $j = 5$.

a	b	a	b	a	b	b	c	a	b	a	b	a	c	a
a	b	a	b	a	×									
		(a)	(b)	(a)	b	×								

- The good-prefix heuristic rules out one guess.
  In new guess we have three matched characters. $j^{\text{new}} = 3$, $i^{\text{new}} = i^{\text{old}}$.
- We match a character, but then have a mismatch at $j = 4$.

Knuth-Morris-Pratt example

Example: Search for $P =$ `ababaca`. We first mismatch at $j = 5$.

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)									

- The good-prefix heuristic rules out one guess.
  In new guess we have three matched characters. $j^{\text{new}} = 3$, $i^{\text{new}} = i^{\text{old}}$.
- We match a character, but then have a mismatch at $j = 4$.
- In new guess we have two matched characters. $j^{\text{new}} = 2$, $i^{\text{new}} = i^{\text{old}}$.

Knuth-Morris-Pratt example

Example: Search for $P =$ `ababaca`. We first mismatch at $j = 5$.

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)	×								

- The good-prefix heuristic rules out one guess.
  In new guess we have three matched characters. $j^{\text{new}} = 3$, $i^{\text{new}} = i^{\text{old}}$.
- We match a character, but then have a mismatch at $j = 4$.
- In new guess we have two matched characters. $j^{\text{new}} = 2$, $i^{\text{new}} = i^{\text{old}}$.
  But then we immediately mismatch with $j = 2$.

Knuth-Morris-Pratt example

Example: Search for $P =$ `ababaca`. We first mismatch at $j = 5$.

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)	×								

- The good-prefix heuristic rules out one guess.
  In new guess we have three matched characters. $j^{\text{new}} = 3$, $i^{\text{new}} = i^{\text{old}}$.
- We match a character, but then have a mismatch at $j = 4$.
- In new guess we have two matched characters. $j^{\text{new}} = 2$, $i^{\text{new}} = i^{\text{old}}$.
  But then we immediately mismatch with $j = 2$.
- Nothing matches the good suffix. $j^{\text{new}} = 0$.

Knuth-Morris-Pratt example

Example: Search for $P =$ `ababaca`. We first mismatch at $j = 5$.

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)	×								
						×								

- The good-prefix heuristic rules out one guess.
  In new guess we have three matched characters. $j^{\text{new}} = 3$, $i^{\text{new}} = i^{\text{old}}$.
- We match a character, but then have a mismatch at $j = 4$.
- In new guess we have two matched characters. $j^{\text{new}} = 2$, $i^{\text{new}} = i^{\text{old}}$.
  But then we immediately mismatch with $j = 2$.
- Nothing matches the good suffix. $j^{\text{new}} = 0$.
- We still have a mismatch at $j = 0$. Increase i.

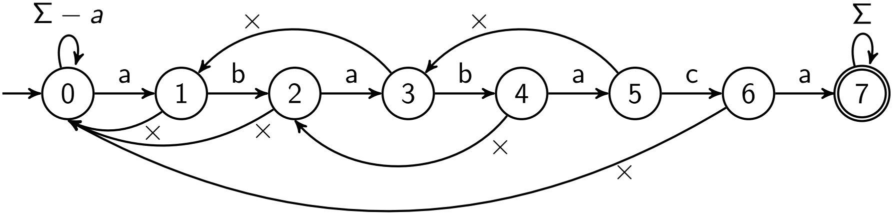
Knuth-Morris-Pratt algorithm, complete

Precompute $F[J]$ = new j to use if the current good prefix was $F[0..J]$.

```
KMP::pattern-matching(T, P)
    F ← compute-failure-array(P)
    i ← 0, j ← 0                  // currently compare T[i] to P[j]
    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                          // check next character
                i ← i + 1, j ← j + 1
        else                              // bad T-character is T[i]
            if j = 0 then i ← i + 1
            else j ← F[j − 1]
```

Knuth-Morris-Pratt Automaton

The Knuth-Morris-Pratt algorithm can also be described as an automaton. Define a state for the current value of j = length of good prefix.



This uses an unusual type of transition $\times$ (*"failure"*):

- Used only if no other transition fits.
- Does not consume a character.
- For $j = 1, \ldots, m-1$, the failure arc from j leads to $F[j-1]$ (because at this point the good prefix was $P[0..j-1]$)

String matching with KMP – Failure-function

- To compute F, re-use as much of good prefix as possible.

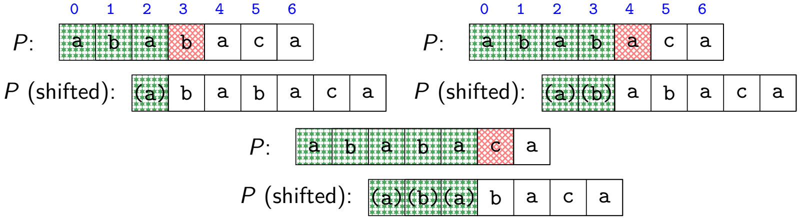
	0	1	2	3	4	5	6		
P:	a	b	a	b	a	c	a		
P (shifted):			(a)	b	a	b	a	c	a

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

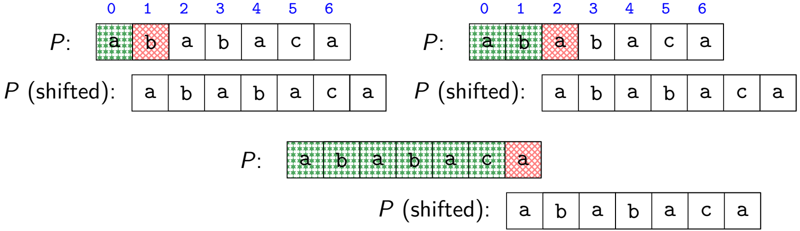
P:	a	b	a	b	a	c	a		
P (shifted):			(a)	(b)	(a)	b	a	c	a

String matching with KMP – Failure-function

- To compute F, re-use as much of good prefix as possible.



- Sometimes nothing fits. Then shift past good prefix.



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

String matching with KMP – Failure function

- $F[J]$ = number of re-used characters if good prefix was $P[0..J]$
- For $P =$ `abababaca`, we get

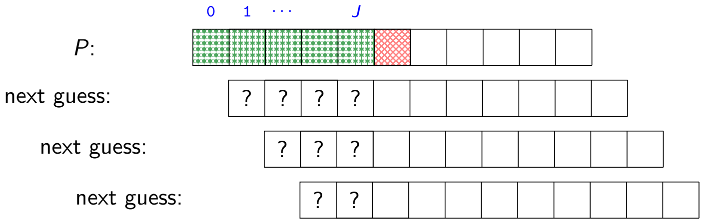
J	0	1	2	3	4	5	6
$F[J]$	0	0	1	2	3	0	?

String matching with KMP – Failure function

- $F[J]$ = number of re-used characters if good prefix was $P[0..J]$
- For $P =$ `ababaca`, we get

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

- 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 $P[0..J]$)



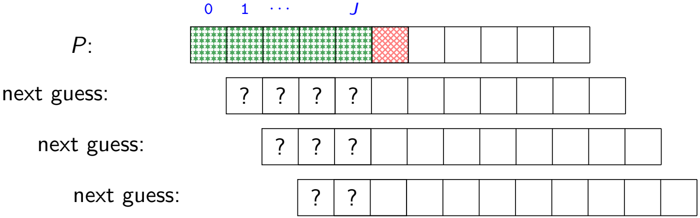
- Equivalently: Find longest prefix of P that is a suffix of $P[1..J]$

String matching with KMP – Failure function

- $F[J]$ = number of re-used characters if good prefix was $P[0..J]$
- For P = ababaca, we get

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

- 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: Find longest 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 algorithm, but useful elsewhere)

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

KMP Failure Array – Fast Computation

- Recall: "$F[J] =$ maximal ℓ: $\{P[0..\ell-1]$ is a suffix of $P[1..J]\}$."
- Loop invariant: "j maximal: $P[0..j-1]$ is a suffix of $T[0..i-1]$."

Idea: Run *KMP::pattern-matching* on input $P[1..m-1]$.
Update F whenever we enter loop.

KMP Failure Array – Fast Computation

- Recall: "$F[J]$ = maximal ℓ: $\{P[0..\ell-1]$ is a suffix of $P[1..J]\}$."
- Loop invariant: "j maximal: $P[0..j-1]$ is a suffix of $T[0..i-1]$."

Idea: Run *KMP::pattern-matching* on input $P[1..m-1]$.
Update F whenever we enter loop.

KMP::compute-failure-array(P)

```
Initialize array F as all-0
i ← 1, j ← 0                // currently compare P[i] to P[j]
while i < m do
// inv:  P[0..j−1] is a suffix of P[1..i−1]
    F[i − 1] ← max{F[i − 1], j}
    if P[j] = P[i]
        i ← i + 1, j ← j + 1
    else
        if j = 0 then i ← i + 1
        else j ← F[j − 1]
```

Note: $j < i$ at all times, so needed F-entries are already computed.

KMP Runtime

Consider the main routine *KMP::pattern-matching*:

- How often does the **while** loop execute?
  - i need not increase, j can increase or decrease.
  - Not even obviously finite. What is getting bigger?

KMP Runtime

Consider the main routine *KMP::pattern-matching*:

- How often does the **while** loop execute?
  - i need not increase, j can increase or decrease.
  - Not even obviously finite. What is getting bigger?

- **Idea:** Consider function $2i - j$. Initially this is 0.
- In each iteration that does not exit, there are three possibilities:
  1. i and j both increase by $1 \Rightarrow 2i - j$ increases
  2. $j = 0$ unchanged, i increases $\Rightarrow 2i - j$ increases
  3. j decreases ($F[j-1] < j$), i unchanged $\Rightarrow 2i - j$ increases

KMP Runtime

Consider the main routine *KMP::pattern-matching*:

- How often does the **while** loop execute?
  - i need not increase, j can increase or decrease.
  - Not even obviously finite. What is getting bigger?

- **Idea:** Consider function $2i - j$. Initially this is 0.
- In each iteration that does not exit, there are three possibilities:
  1. i and j both increase by $1 \Rightarrow 2i - j$ increases
  2. $j = 0$ unchanged, i increases $\Rightarrow 2i - j$ increases
  3. j decreases ($F[j-1] < j$), i unchanged $\Rightarrow 2i - j$ increases

- $i \leq n$ and $j \geq 0$ throughout, therefore $2i - j \leq 2n$.
- So no more than $2n$ iterations of the while loop.
  The main routine (without *compute-failure-array*) takes $O(n)$ time.

KMP Runtime

Consider the main routine *KMP::pattern-matching*:

- How often does the **while** loop execute?
  - i need not increase, j can increase or decrease.
  - Not even obviously finite. What is getting bigger?

- **Idea:** Consider function $2i - j$. Initially this is 0.
- In each iteration that does not exit, there are three possibilities:
  1. i and j both increase by $1 \Rightarrow 2i - j$ increases
  2. $j = 0$ unchanged, i increases $\Rightarrow 2i - j$ increases
  3. j decreases ($F[j-1] < j$), i unchanged $\Rightarrow 2i - j$ increases

- $i \leq n$ and $j \geq 0$ throughout, therefore $2i - j \leq 2n$.
- So no more than $2n$ iterations of the while loop.
  The main routine (without *compute-failure-array*) takes $O(n)$ time.

Similarly: *compute-failure-array* takes $O(m)$ time.

Result: KMP pattern matching has $O(n+m)$ *worst-case* run-time.

KMP failure function – improvement

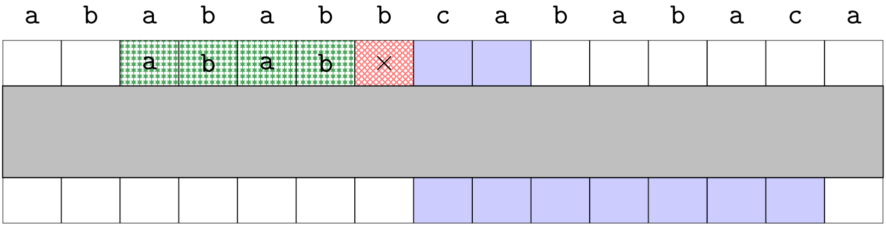
We can define an even better failure-function F^+:

a	b	a	b	a	b	b	c	a	b	a	b	a	c	a
		a	b	a	b	×								
				(a)	(b)	×								
						×								

- We had a mismatch when we wanted an a.
- The next guess wants a in the same place.
- This can never work! Skip all guesses that have a in this place.

KMP failure function – improvement

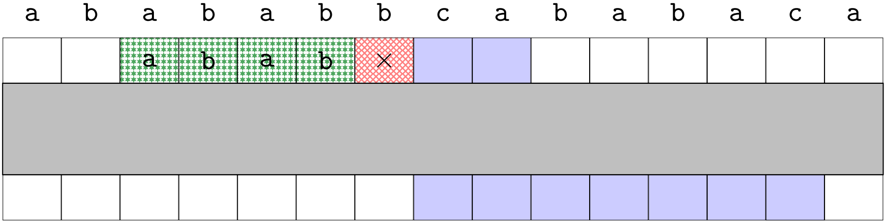
We can define an even better failure-function F^+:



- We had a mismatch when we wanted an a.
- The next guess wants a in the same place.
- This can never work! Skip all guesses that have a in this place.

KMP failure function – improvement

We can define an even better failure-function F^+:



- We had a mismatch when we wanted an a.
- The next guess wants a in the same place.
- This can never work! Skip all guesses that have a in this place.

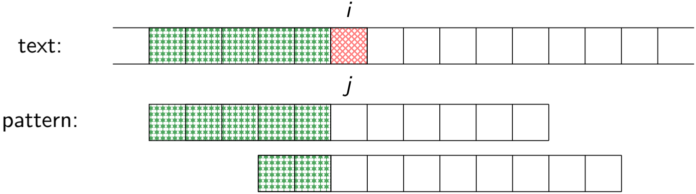
Easy to compute from F in $O(m)$ time (except for annoying ± 1):

$$F^+[J] = \begin{cases} F[J] & \text{if } P[J+1] \neq P[F[J]] \text{ or } F[J]=0 \\ F^+[F[J]-1] & \text{otherwise} \end{cases}$$

Outline

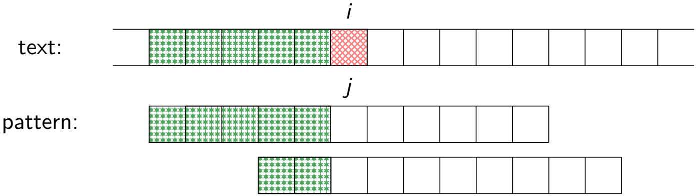
Towards the Boyer-Moore Algorithm

Recall: KMP eliminates guesses based on good-prefix heuristic.



Towards the Boyer-Moore Algorithm

Recall: KMP eliminates guesses based on good-prefix heuristic.

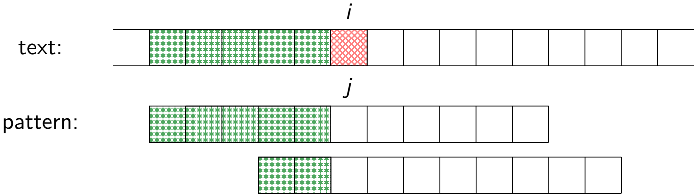


Boyer-Moore uses *two* skip-heuristics:

- Eliminate guesses based on matched characters. Now called **good suffix heuristic**. Very similar to KMP.
- Use weak version of bad-T-char heuristics called **last-occurrence heuristic**—this is new.

Towards the Boyer-Moore Algorithm

Recall: KMP eliminates guesses based on good-prefix heuristic.



Boyer-Moore uses *two* skip-heuristics:

- Eliminate guesses based on matched characters. Now called **good suffix heuristic**. Very similar to KMP.
- Use weak version of bad-T-char heuristics called **last-occurrence heuristic**—this is new.

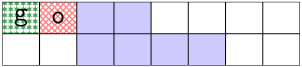
The second heuristic turns out to be very helpful, and leads to fastest pattern matching on English text as long as we search *backwards*.

Forward-searching vs. reverse-searching

Forward-searching:

P : g o o d

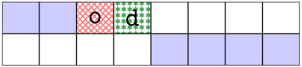
T : g r a d i e n t



Reverse-order searching:

P : g o o d

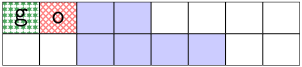
T : g r a d i e n t



Forward-searching vs. reverse-searching

Forward-searching:

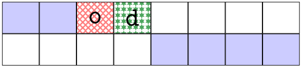
P : `g o o d`
T : `g r a d i e n t`



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

Reverse-order searching:

P : `g o o d`
T : `g r a d i e n t`

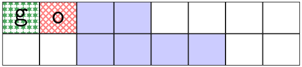


- o does not occur in P.
- d cannot be matched again
  $\Rightarrow$ shift pattern past d.

Forward-searching vs. reverse-searching

Forward-searching:

P : g o o d
T : g r a d i e n t

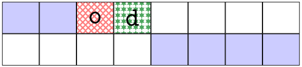


- o does not occur in P.
  $\Rightarrow$ shift pattern past o.

At most $j-1$ guesses ruled out after j checks.

Reverse-order searching:

P : g o o d
T : g r a d i e n t



- o does not occur in P.
- d cannot be matched again
  $\Rightarrow$ shift pattern past d.

Sometimes rule out $m-1$ guesses even after only one check

Reverse-order searching typically eliminates more guesses.

Last-occurrence 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													

Last-occurrence 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													

(1) Bad T-character is a.

Last-occurrence 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

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

(1) Bad T-character 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

Last-occurrence 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) Bad T-character 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 bad T-character p

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

Last-occurrence 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								
									(o)				e	r			

(1) Bad T-character 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 bad T-character `p`
- Use "last" since we cannot rule out this guess.

(3) Shift completely past `o` since `o` is not in P.

Last-occurrence 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								
									(o)				e	r			
															r		

(1) Bad T-character 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 bad T-character p
- Use “last” since we cannot rule out this guess.

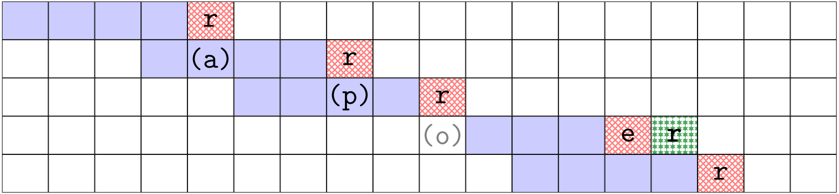
(3) Shift completely past o since o is not in P.

(4) The guess that aligns rightmost r of P has already been ruled out.
- Simply shift one unit to the right.

Last-occurrence 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) Bad T-character 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 bad T-character `p`
- Use "last" since we cannot rule out this guess.

(3) Shift completely past `o` since `o` is not in P.

(4) The guess that aligns rightmost `r` of P has already been ruled out.
- Simply shift one unit to the right.

(5) Shift completely past `o` $\rightarrow$ out of bounds.

Boyer-Moore Algorithm – incomplete

Boyer-Moore::pattern-matching(T, P)

```
L ← [...] // pre-computation
i ← m − 1, j ← m − 1  // currently compare T[i] to P[j]
while i < n do
        // inv: current guess begins at index i − j
        if P[j] = T[i]
            if j = 0 then return "found at guess i − m + 1"
            else                          // go backwards
                i ← i − 1, j ← j − 1
        else
            i ← [...] // read from L and T[i]
            j ← m − 1           // restart from right end
return FAIL
```

Two steps missing:

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

Helper-Array for Last-Occurrence 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 Last-Occurrence 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 Last-Occurrence 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:

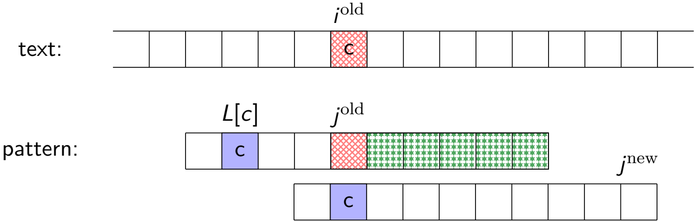
char	p	a	e	r	all others
$L[\cdot]$	2	1	3	4	?

- 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::last-occurrence-array($P[0..m-1]$)
1. initialize array L indexed by Σ with all -1
2. **for** $j \leftarrow 0$ **to** $m-1$ **do** $L[P[j]] \leftarrow j$
3. **return** L

Last-occurrence heuristic – update formula

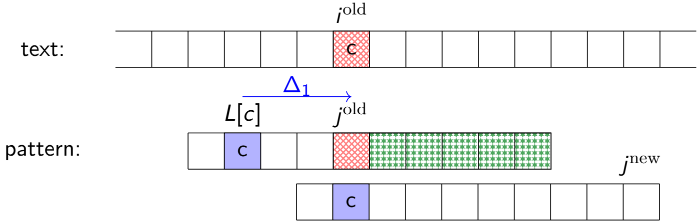
"Good" case: $L[c] < j$, so c is left of $P[j]$.



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

Last-occurrence heuristic – update formula

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

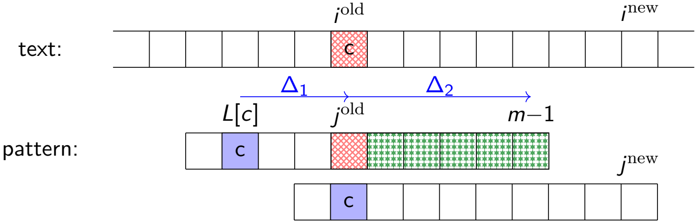


Want: i^{new} = index in T that corresponds to j^{new}.

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

Last-occurrence heuristic – update formula

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

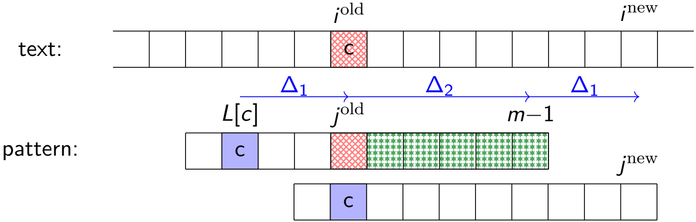


Want: i^{new} = index in T that corresponds to j^{new}.

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

Last-occurrence heuristic – update formula

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



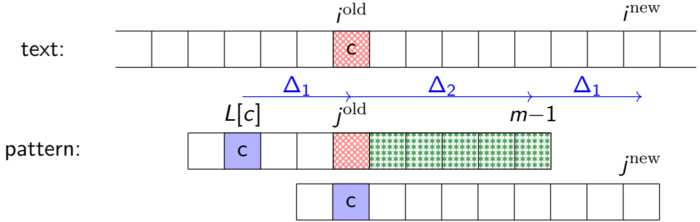
Want: i^{new} = index in T that corresponds to j^{new}.

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

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

Last-occurrence 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 the guess $= j^{\text{old}} - L[c]$
- 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 ← last-occurrence-array(P)
i ← m − 1, j ← m − 1  // currently compare T[i] to P[j]
while i < n do
        // inv: current guess begins at index i − j
        if P[j] = T[i]
            if j = 0 then return "found at guess i − m + 1"
            else                              // go backwards
                i ← i − 1, j ← j − 1
        else
            i ← i + m−1 − min{L[T[i]], j−1}
            j ← m − 1            // restart from right end
return FAIL
```

For *full* Boyer-Moore algorithm:

- precompuate 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: Example

Want: new guess agrees with successfully matched previous characters.

(Same as good prefix heuristic, but renamed due to backward searching.)

P :	b	a	a	b	a	b	a										
T :	a	n	a	a	a	b	a	n	a	i	n	b	a	b	a	l	a
				n	a	b	a										

- Do smallest shift so that aba fits in the new guess.

Good suffix heuristic: Example

Want: new guess agrees with successfully matched previous characters.

(Same as good prefix heuristic, but renamed due to backward searching.)

P :	b	a	a	b	a	b	a										
T :	a	n	a	a	a	b	a	n	a	i	n	b	a	b	a	l	a
				n	a	b	a										
					(a)	(b)	(a)	b	a								

- Do smallest shift so that aba fits in the new guess.
- Do smallest shift so that a fits in the new guess.

Good suffix heuristic: Example

Want: new guess agrees with successfully matched previous characters.

(Same as good prefix heuristic, but renamed due to backward searching.)

P:	b	a	a	b	a	b	a										
T:	a	n	a	a	a	b	a	n	a	i	n	b	a	b	a	l	a
				n	a	b	a										
					(a)	(b)	(a)	b	a								
									(a)		a						

- Do smallest shift so that aba fits in the new guess.
- Do smallest shift so that a fits in the new guess.
- No suffix matched. Shift over by one (or by last-char heuristic)

Good suffix heuristic: Example

Want: new guess agrees with successfully matched previous characters.

(Same as good prefix heuristic, but renamed due to backward searching.)

P:	b	a	a	b	a	b	a										
T:	a	n	a	a	a	b	a	n	a	i	n	b	a	b	a	l	a
				n	a	b	a										
					(a)	(b)	(a)	b	a								
									(a)		a						
												a					

- Do smallest shift so that aba fits in the new guess.
- Do smallest shift so that a fits in the new guess.
- No suffix matched. Shift over by one (or by last-char heuristic)

Good suffix heuristic: Example

Want: new guess agrees with successfully matched previous characters.

(Same as good prefix heuristic, but renamed due to backward searching.)

P :	b	a	a	b	a	b	a										
T :	a	n	a	a	a	b	a	n	a	i	n	b	a	b	a	l	a
				n	a	b	a										
					(a)	(b)	(a)	b	a								
									(a)		a						
												a					
											a	b	a				

- Do smallest shift so that aba fits in the new guess.
- Do smallest shift so that a fits in the new guess.
- No suffix matched. Shift over by one (or by last-char heuristic)

Good suffix heuristic: Example

Want: new guess agrees with successfully matched previous characters.

(Same as good prefix heuristic, but renamed due to backward searching.)

P:	b	a	a	b	a	b	a										
T:	a	n	a	a	a	b	a	n	a	i	n	b	a	b	a	l	a
				n	a	b	a										
					(a)	(b)	(a)	b	a								
									(a)		a						
												a					
											a	b	a				
											a	b	a	b	a		

- Do smallest shift so that aba fits in the new guess.
- Do smallest shift so that a fits in the new guess.
- No suffix matched. Shift over by one (or by last-char heuristic)
- What if we cannot match the entire good suffix?

Good suffix array - if matched part doesn't repeat

P : b a a b a b a

T : a n a i n b a b a l a n d i n g

				a	b	a	b	a							

- Cannot match all of baba

Good suffix array - if matched part doesn't repeat

P:	b	a	a	b	a	b	a									
T:	a	n	a	i	n	b	a	b	a	l	a	n	d	i	n	g
					a	b	a	b	a							
								(b)	(a)							

- Cannot match all of baba
- But ba fits a prefix of P $\rightsquigarrow$ shift to that guess

Good suffix array - if matched part doesn't repeat

P:	b	a	a	b	a	b	a									
T:	a	n	a	i	n	b	a	b	a	l	a	n	d	i	n	g
					a	b	a	b	a							
								(b)	(a)							

- Cannot match all of baba
- But ba fits a prefix of P $\rightsquigarrow$ shift to that guess
- Generally: Re-use longest suffix of matched part that fits a prefix of P

Good suffix array - if matched part doesn't repeat

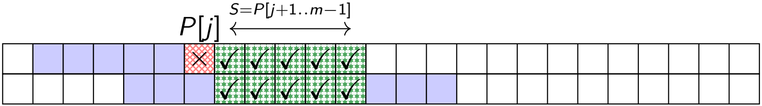
P:	b	a	a	b	a	b	a									
T:	a	n	a	i	n	b	a	b	a	l	a	n	d	i	n	g
					a	b	a	b	a							
								(b)	(a)							

- Cannot match all of baba
- But ba fits a prefix of $P \rightsquigarrow$ shift to that guess
- Generally: Re-use longest suffix of matched part that fits a prefix of P
- If nothing fits: Shift guess all the way past previous guess.

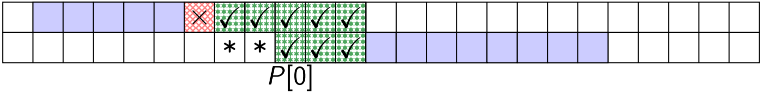
P:	c	a	a	b	a	b	a									
T:	a	n	a	i	n	b	a	b	a	l	a	n	d	i	n	g
					a	b	a	b	a							

Definition of good suffix array

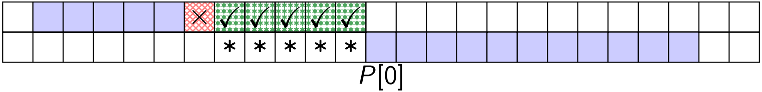
- Assume search failed at $P[j]$, but had matched $P[j+1..m-1] =: S$
- Case 0: S empty: shift by 1
- Case 1: S appears as substring of P elsewhere



- Case 2: A suffix of Q is a prefix of P.

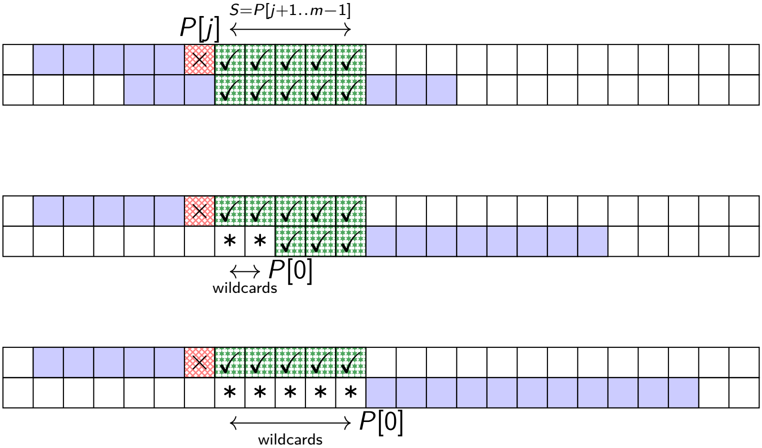


- Case 3: Neither (i.e., only empty suffix fits).



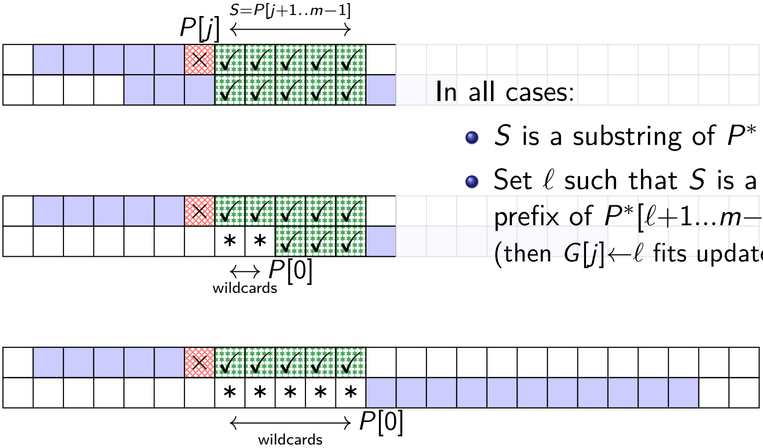
Definition of good suffix array

- Can unify all three cases into one!
- Let P^* be P with m *wildcards* attached in front.



Definition of good suffix array

- Can unify all three cases into one!
- Let P^* be P with m *wildcards* attached in front.

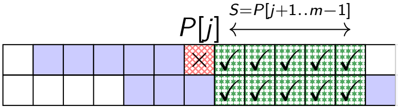


In all cases:

- S is a substring of P^*
- Set ℓ such that S is a prefix of $P^*[\ell{+}1...m{-}1]$ (then $G[j]{\leftarrow}\ell$ fits update-formula)

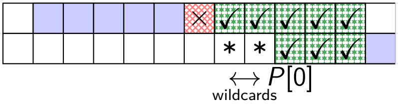
Definition of good suffix array

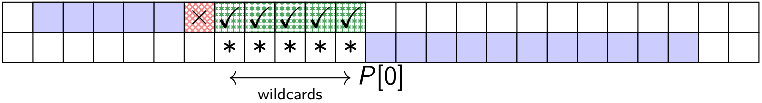
- Can unify all three cases into one!
- Let P^* be P with m *wildcards* attached in front.



In all cases:

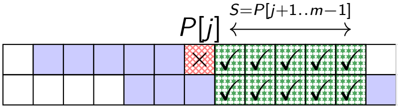
- S is a substring of P^*
- Set ℓ such that S is a prefix of $P^*[\ell{+}1...m{-}1]$ (then $G[j]{\leftarrow}\ell$ fits update-formula)
- Want $\ell \neq j$ so that we shift guess





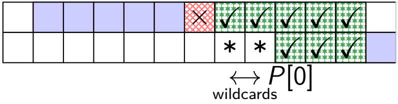
Definition of good suffix array

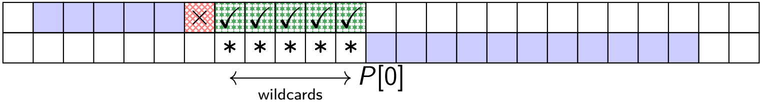
- Can unify all three cases into one!
- Let P^* be P with m *wildcards* attached in front.



In all cases:

- S is a substring of P^*
- Set ℓ such that S is a prefix of $P^*[\ell+1...m-1]$
  (then $G[j] \leftarrow \ell$ fits update-formula)
- Want $\ell \neq j$ so that we shift guess





$$G[j] = \max_{\ell \neq j} \quad P[j+1..m-1] \textbf{ is a prefix of } P^*[\ell+1..m-1]$$

Good Suffix Array Computation - human

$$G[j] \;=\; \max_{\ell \neq j} \quad P[j{+}1..m{-}1] \text{ is a prefix of } P^*[\ell{+}1..m{-}1]$$
$$\;=\; \max_{\ell} \quad P[j{+}1..m{-}1] \text{ is a prefix of } P^*[\ell{+}1..m{-}2]$$

	$P =$ baababa	$P^*[-m..m{-}2]$														
		-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5		
j	$S{=}P[j{+}1..m{-}1]$	*	*	*	*	*	*	*	b	a	a	b	a	b	$\ell+1$	$G[j]$
5	a												a		4	3
4	ba											b	a		3	2
3	aba										a	b	a		2	1
2	baba						b	a	b	a					-2	-3
1	ababa					a	b	a	b	a					-3	-4
0	aababa				a	a	b	a	b	a					-4	-5

- Easy to compute in polynomial time:
- Feels a lot like failure-array (with some prefix/suffix reversed)
- **Can show** (no details): Can compute G in $O(m)$ time.

Boyer-Moore Summary

Boyer-Moore::pattern-matching(T, P) `// full version`

$L \leftarrow$ *last-occurrence-array*(Σ), $G \leftarrow$ *good-suffix-array*(P)
$i \leftarrow m-1, j \leftarrow m-1$ `// currently compare` $T[i]$ `to` $P[j]$
while $i < n$ **do**
  `// inv: current guess begins at index` $i - j$
  if $P[j] = T[i]$
    if $j = 0$ **then return** "found at guess $i - m + 1$"
    else $i \leftarrow i - 1, j \leftarrow j - 1$
  else
    $i \leftarrow i + m{-}1 - \min\{L[T[i]], G[j]\}$
    $i \leftarrow \max\{1, j - L[T[i]]\}$
    $j \leftarrow m - 1$ `// restart from right end`
return `FAIL`

Boyer-Moore Summary

Boyer-Moore::pattern-matching(T, P) `// full version`

$L \leftarrow$ *last-occurrence-array*(Σ), $G \leftarrow$ *good-suffix-array*(P)

$i \leftarrow m-1, j \leftarrow m-1$ `// currently compare` $T[i]$ `to` $P[j]$

while $i < n$ **do**

// `// inv: current guess begins at index` $i - j$

  if $P[j] = T[i]$

    if $j = 0$ **then return** "found at guess $i - m + 1$"

    else $i \leftarrow i - 1, j \leftarrow j - 1$

  else

    $i \leftarrow i + m - 1 - \min\{L[T[i]], G[j]\}$

    $i \leftarrow \max\{1, j - L[T[i]]\}$

    $j \leftarrow m - 1$ `// restart from right end`

return `FAIL`

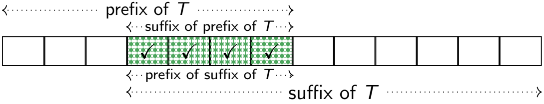
- Boyer-Moore performs very well (even the simplified version).
- On typical *English text* Boyer-Moore looks at only $\approx 25\%$ of T
- Worst-case run-time for is $O(mn)$, but in practice much faster.

  [There are ways to ensure $O(n)$ run-time. No details.]

Outline

Tries of Suffixes and Suffix Trees

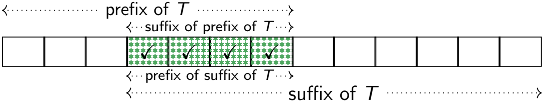
Recall: P occurs in $T \Leftrightarrow P$ is a prefix of some suffix of T.



- **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 different 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.

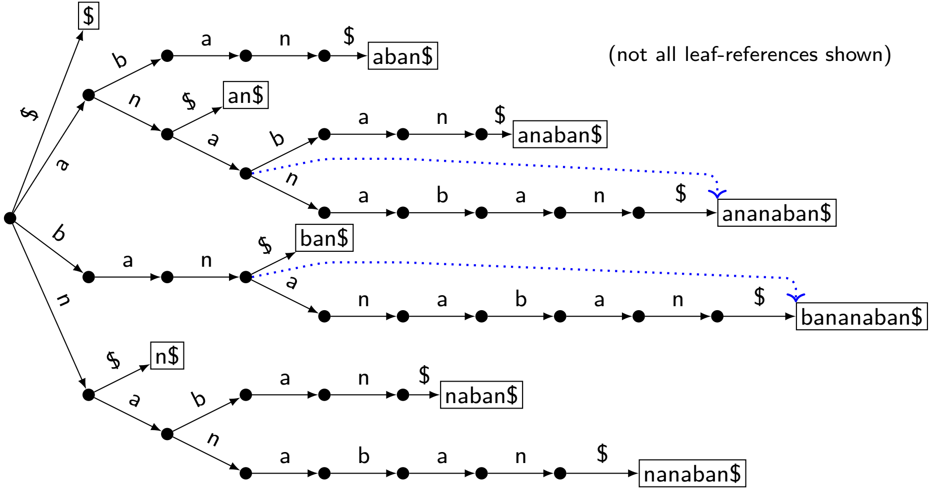


- **Idea:** Build a data structure that stores all suffixes of T.
  - So we preprocess the text T rather than the pattern P
  - This is useful if we want to search for many different patterns P within the same fixed text T.
- Naive idea: Store the suffixes in a trie.
  - $|T| = n \Rightarrow$ the $n+1$ 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}

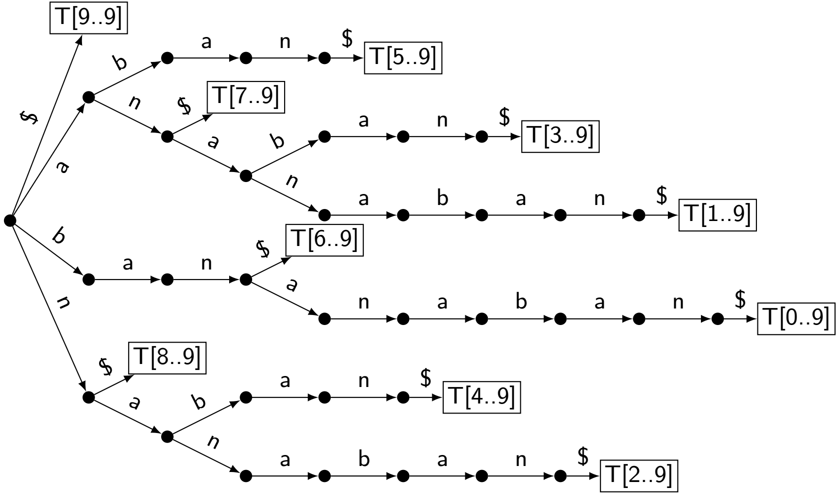


Tries of suffixes

Store suffixes via indices:

$T =$

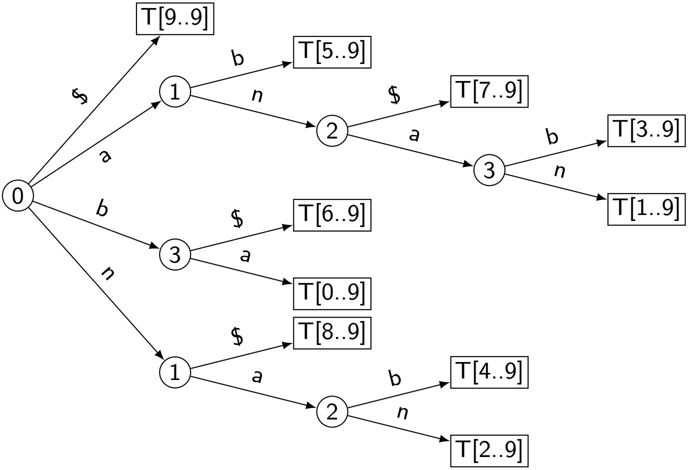
0	1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	b	a	n	$



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.

Outline

9. String Matching
   - Introduction
   - Karp-Rabin Algorithm
   - Skip-heuristics
   - Knuth-Morris-Pratt algorithm
   - Boyer-Moore Algorithm
   - Suffix Trees
   - Suffix Arrays

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

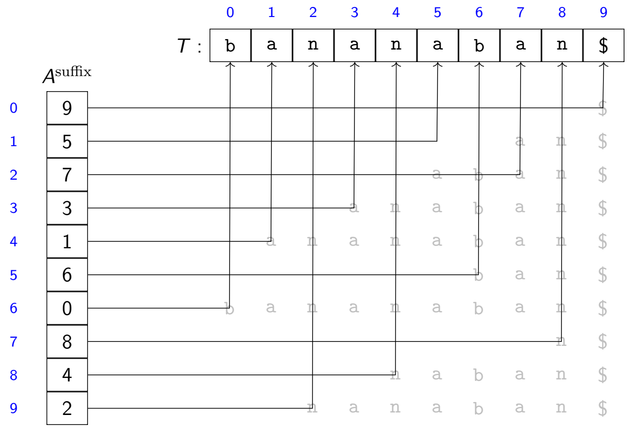
i	suffix $T[i..n]$
0	bananaban$
1	ananaban$
2	nanaban$
3	anaban$
4	naban$
5	aban$
6	ban$
7	an$
8	n$
9	$

$\longrightarrow$

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



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 $\rightsquigarrow$ sorted elsewhere
  - We can double length of sorted part every round.

)

  - $O(\log n)$ rounds enough $\Rightarrow$ $O(n \log n)$ **run-time**
  - You do not need to know details ($\rightsquigarrow$ 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 $\Rightarrow$ **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^{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).

String Matching Conclusion

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

(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.