CS 240 – Data Structures and Data Management

Module 9: String Matching

O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2024

Outline

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

Pattern Matching Definitions

- Search for a string (pattern) in a large body of text
- $T[0 \ldots n-1]$ text (or haystack) being searched
- $P[0 \ldots m-1]$ pattern (or needle) being searched for
- Strings over alphabet Σ
- Convention: return the first occurrence of P in T
- Example

  $T =$ L i t t l e p i g l e t s c o o k e d f o r m o t h e r p i g

  7 + 0   7 + 1   7 + 2

  $P =$ p i g

  $n = 36, \; m = 3, \; i = 7$

  - return smallest i such that

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

- If P does not occur in T, return FAIL
- Applications
  - information retrieval (text editors, search engines), bioinformatics, data mining

More Definitions

antidisestablishmentarianism

- Substring $T[i \ldots j]$ $0 \leq i \leq j+1 \leq n$ is a string consisting of characters $T[i], T[i+1], \ldots, T[j]$
  - length is $j-i+1$
  - empty string included: $T[j+1 \ldots j]$
- Prefix of T is a substring $T[0 \ldots i-1]$ of T for some $0 \leq i \leq n$
  - empty prefix included: $T[0 \ldots -1]$
- Suffix of T is a substring $T[i \ldots n-1]$ of T for some $0 \leq i \leq n$
  - empty suffix included: $T[n \ldots n-1]$
- The empty substring is usually denoted by Λ

General Idea of Algorithms

guess at $i = 0$ | guess at $i = 1$ | … | guess at $i = 6$ | guess at $i = 7$

guess at $i = 0$	guess at $i = 1$	...	guess at $i = 6$	guess at $i = 7$
abbbababbab	a**bbba**babbab		abbbab**abba**b	abbaba**bbab**
abba	**abba**		**abba**	**abba**

check at $j = 0$ check at $j = 1$

- Pattern matching algorithms consist of guesses and checks
  - a **guess** is a position i such that P might start at $T[i]$
  - valid guesses (initially) are $0 \leq i \leq n - m$
  - a **check** of a guess is a single position j with $0 \leq j < m$ where we compare $T[i + j]$ to $P[j]$
    - must perform m checks of a single correct guess
  - may make fewer checks of an incorrect guess

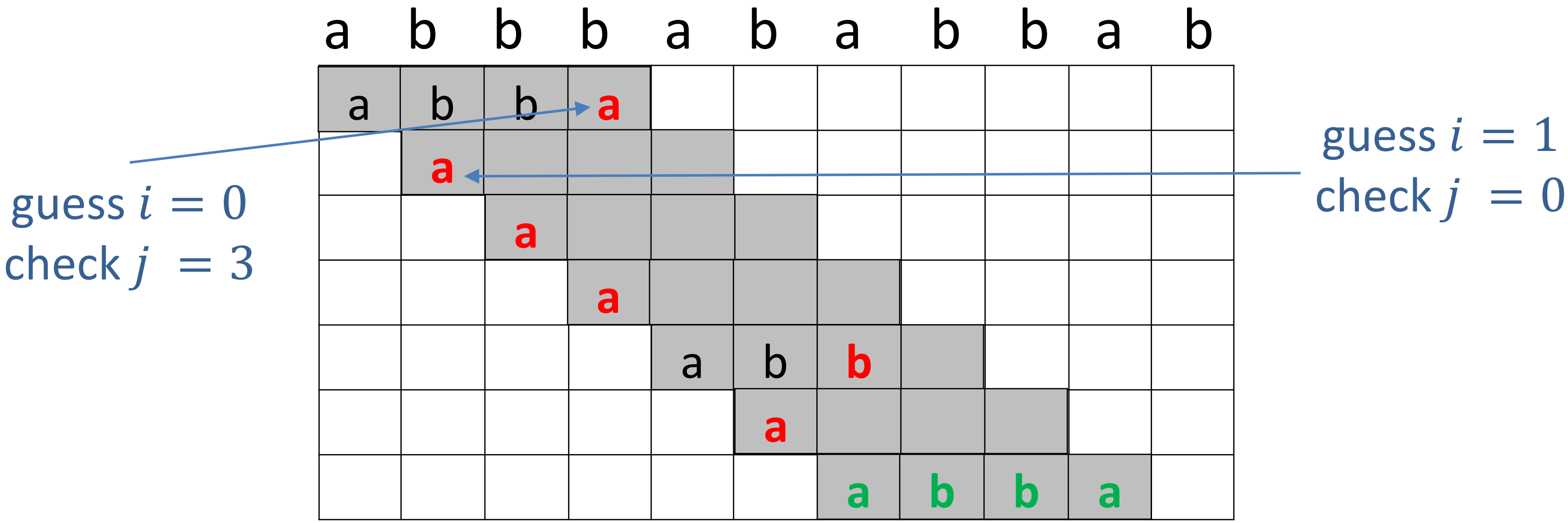
Diagrams for Matching

- Diagram single run of pattern matching algorithm by matrix of checks
    - each row represents a single guess

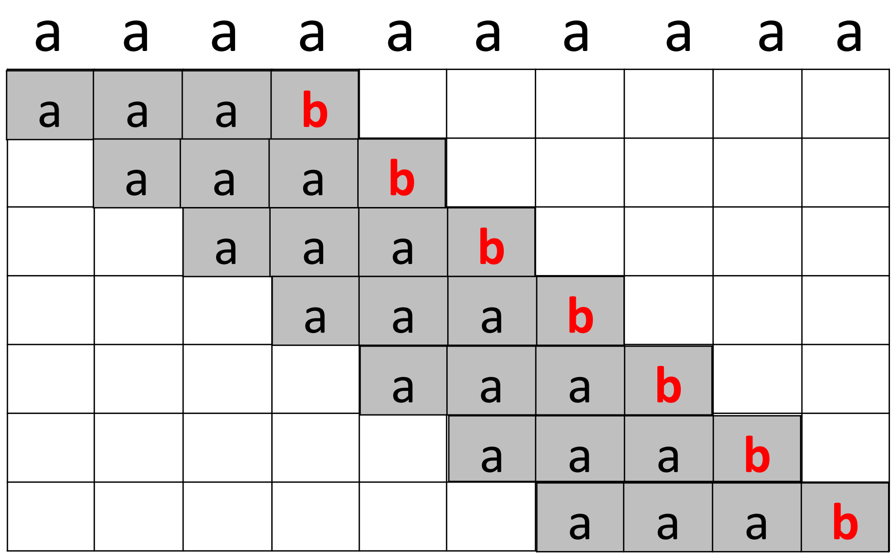
a	b	b	b	a	b	a	b	b	a	b
a	b	b	a							

Brute-Force Algorithm: Example

Example: T = abbbababbab, P = abba



Brute-Force Algorithm: Running Time



- Worst possible input
  - $P = \underbrace{a \dots a}_{m-1 \text{ times}} b, \; T = \underbrace{aaaaaaa \dots aaaaaaa}_{n \text{ times}}$
- Perform $(n-m+1)m$ checks, which is $\Theta((n-m)m)$
  - small m, say $m = 5$ runtime is $\Theta(n)$
  - medium m, say $m = n/2$: runtime is $\Theta(n^2)$
  - large m, say $m = n - 5$: runtime is $\Theta(n)$

Brute-force Algorithm

- Checks every possible guess

> *Bruteforce::PatternMatching*$(T\,[0..n-1], P[0..m\,-\,1])$
> T : String of length n (text), P: String of length m (pattern)
> **for** $i \leftarrow 0$ **to** $n - m$ **do**
> **if** *strcmp*$(T,\ P, i,\ m) = 0$
> **return** "found at guess i"
> **return** FAIL

- Note: *strcmp* takes $\Theta(m)$ time

> *strcmp*$(T\,, P, i \leftarrow 0, m \leftarrow P.size())$
> **for** $j \leftarrow 0$ **to** $m - 1$ **do**
> **if** $T\,[i + j]$ is before $P[j]$ in Σ **then return** -1
> **if** $T\,[i + j]$ is after $P[j]$ in Σ **then return** 1
> **return** 0

Improvement via Preprocesing

- Preprocessing: do work on some parts of the input *before* pattern matching begins, so that pattern matching goes faster
- Two preprocessing options for pattern matching
    1. Do preprocessing on pattern P
        - eliminate guesses based on preprocessing
            - **Karp-Rabin**
            - **KMP**
            - **Boyer-Moore**
    2. Do preprocessing on text T
        - create a data structure to find matches easily
            - **Suffix-tree**
            - **Suffix-arrays**

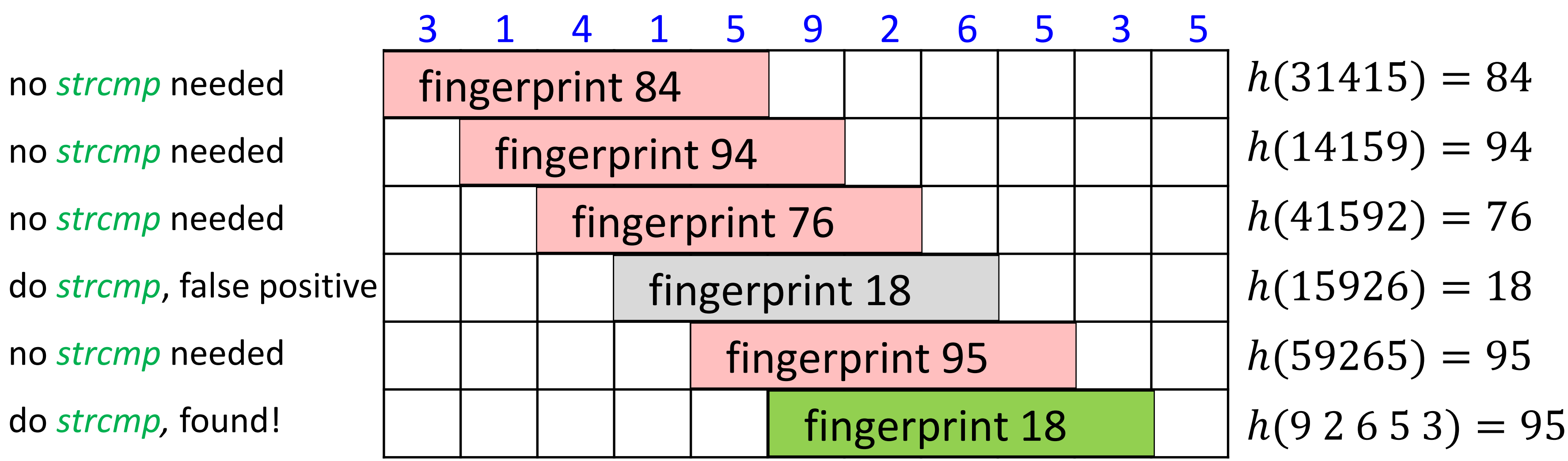
Outline

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

Karp-Rabin Fingerprint Algorithm: Idea

- **Idea:** use hash values (called fingerprints) to eliminate guesses faster
  - need function h: {strings of length m} $\longrightarrow \{0, \dots, M-1\}$
    - call these hash-function and table-size, but there is no dictionary here
  - insight: if $h(P) \neq h(guess)$ then guess cannot work
    - if $h(P) = h(guess)$ verify with *strcmp* that pattern actually matches text
  - 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 $M = 97$
  - precompute $h(P) = h(9\ 2\ 6\ 5\ 3)$

$$= (9 \cdot 10^4 + 2 \cdot 10^3 + 6 \cdot 10^2 + 5 \cdot 10^1 + 3)\ mod\ 97 \ = 92653\ mod\ 97 = 18$$

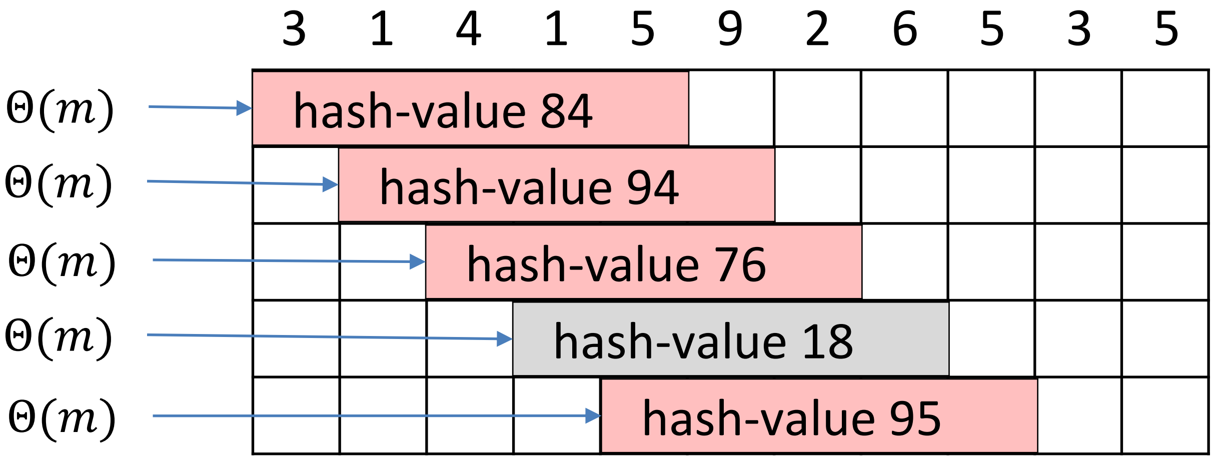


Karp-Rabin Fingerprint Algorithm – First Attempt

Karp-Rabin-Simple::patternMatching(T, P)
    $h_P \leftarrow h(P[0..m-1)])$
    for $i \leftarrow 0$ to $n - m$
        $h_T \leftarrow h(T\,[i...i+m-1])$ $\Theta(m)$
        if $h_T = h_P$
            if *strcmp*$(T\,, P, i, m) = 0$
                return "found at guess i"
    return FAIL

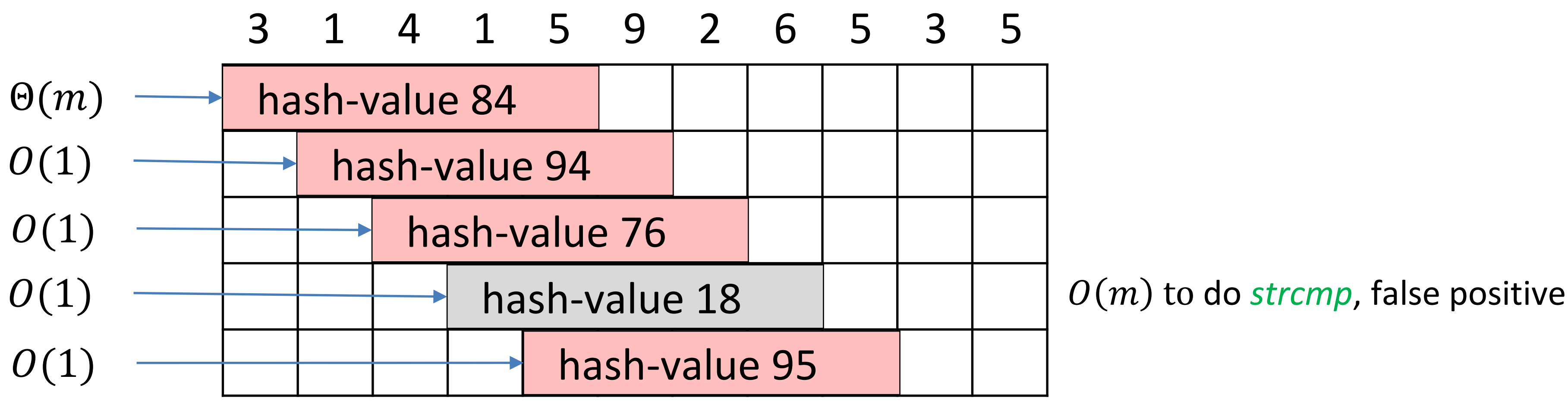
- Algorithm correctness: match is not missed
  - $h(T\,[i..i+m-1]) \neq h(P) \Rightarrow$ guess i is not P
- What about running time?

Karp-Rabin Fingerprint Algorithm: First Attempt



- For each guess, $\Theta(m)$ time to compute hash value
  - since $h(T[i \ldots i + m - 1])$ depends on all m characters
  - worse than brute-force!
    - it is possible for brute force matching to use less than $\Theta(m)$ per guess, as it stops at the first mismatched character
- $n - m + 1$ guesses in text to check
- Total time is $\Theta(mn)$ if pattern not in text
  - how can we improve this?

Karp-Rabin Fingerprint Algorithm: Idea



- Idea: compute next hash from previous one in $O(1)$ time
- $O(n)$ guesses in text to check
- $\Theta(m)$ to compute the first hash value
- $O(1)$ to compute all other hash values
- $O(n + m + m \cdot \{\#\text{false positive}\})$ time
    - recall that we still need to check if the pattern actually matches text whenever hash value of text is equal to the hash value of pattern
    - if hash function is good, then whenever hash values are equal, pattern most likely matches the text

Karp-Rabin Fingerprint Algorithm – Fast Rehash

- Can update fingerprint from previous one in $O(1)$ time for some hash functions
- **Example:** T = 4 1 5 9 2 6 5 3 5
- Initialization of the algorithm
  1. compute first fingerprint: $h(41592) = 41592 \ mod \ 97 = 76$
  2. also pre-compute $R^{m-1} \text{mod } M$ (here $10000 \ mod \ 97 = 9$)
- Main loop: repeatedly compute next hash from the previous one
- Example: from $41592 \ mod \ 97$ compute $15926 \ mod \ 97$
  - get rid of the old first digit and add new last digit

$$41592 \xrightarrow{-4 \cdot 10000} 1592 \xrightarrow{\times 10} 15920 \xrightarrow{+6} 15926$$

- Algebraically,

$$\big(41592 - (4 \cdot 10000)\big) \cdot 10 + 6 = 15926$$

Karp-Rabin Fingerprint Algorithm – Fast Rehash

- Can update fingerprint from previous one in $O(1)$ time for some hash functions
- **Example:** T = 4 1 5 9 2 6 5 3 5
- Initialization of the algorithm
  1. compute first fingerprint: $h(41592) = 41592 \ mod \ 97 = 76$
  2. also pre-compute $R^{m-1} \bmod M$ (here $10000 \ mod \ 97 = 9$)
- Main loop: repeatedly compute next hash from the previous one
- Example: from $\underline{41592} \ mod \ 97$ compute $\underline{15926} \ mod \ 97$

$$\big(41592 - (4 \cdot 10000)\big) \cdot 10 + 6 = 15926$$

$$\big(\big(41592 - (4 \cdot 10000)\big) \cdot 10 + 6\big) \ mod \ 97 = 15926 \ mod \ 97$$

$$\Big(\big(\underbrace{41592 \ mod \ 97}_{\text{previous hash}} - \big(4 \cdot (\underbrace{10000 \ mod \ 97}_{\text{precomputed}})\big)\big) \cdot 10 + 6\Big) \ mod \ 97 = 15926 \ mod \ 97$$

$$\underbrace{\Big(\big(76 - (4 \cdot 9)\big) \cdot 10 + 6\Big) mod \ 97}_{\text{constant number of operations, independent of } m} = 15926 \ mod \ 97$$

Karp-Rabin Fingerprint Algorithm – Conclusion

Karp-Rabin-RollingHash::PatternMatching(T, P)

$M \leftarrow$ suitable prime number
$h_P \leftarrow h(P[0 \ldots m-1)])$
$h_T \leftarrow h(T\,[0..m-1)])$
$s \leftarrow R^{m-1} mod\ M$
for $i \leftarrow 0$ to $n-m$
  if $h_T = h_P$
    if *strcmp*$(T, P, i, m) = 0$
      return "found at guess i"
  if $i < n-m$ // compute fingerprint for next guess
    $h_T \leftarrow \big((h_T - T[i] \cdot s) \cdot R + T\,[i+m]\big) mod\ M$
return FAIL

- Choose "table size" M at random to be prime in $\{2, \ldots, mn^2\}$
- Analysis specific to the hash function in this pseudo-code
  - can show that expected running time is $O(m+n)$
  - $\Theta(mn)$ worst-case, but this extremely is unlikely
  - improvement: reset M after false positive

Outline

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

Knuth-Morris-Pratt (KMP) Overview

- KMP starts out similar to Brute-Force pattern matching

$P = \ ababaca$

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

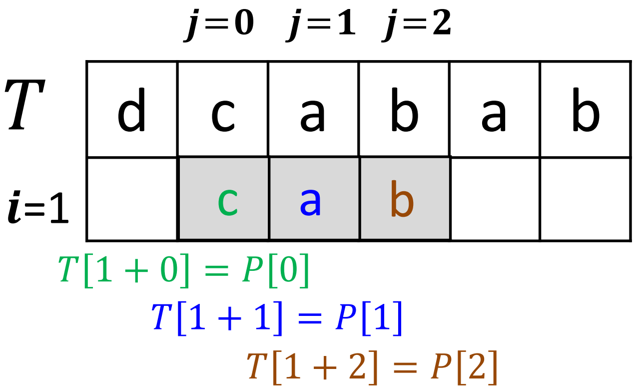
mismatch at the first pattern letter, discard current guess and move on to the next guess

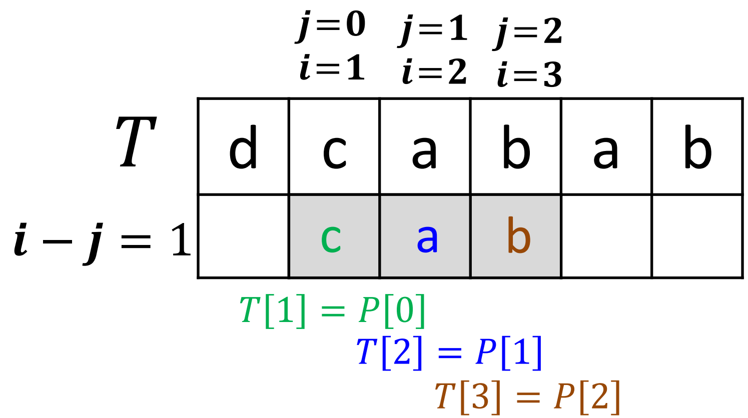
letter matches the text, move on to the next check

mismatch at pattern letter which is not the first pattern letter: do something smarter than brute-force

Knuth-Morris-Pratt (KMP) Indexing

$$P = cad$$





- Brute-force indexing
  - indexes i and j
  - j is the position in the pattern
  - i is current guess
  - check: $T[i + j] = P[j]$
- KMP indexing
  - indexes i and j
  - j is the position in the pattern
  - i is text position where check happens
  - check: $T[i] = P[j]$
  - current guess is $i - j$

Knuth-Morris-Pratt (KMP) Derivation

$P = \ ababaca$

$j=0$
$i=0$

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

- KMP starts similar to brute force pattern matching
  - maintain variables i and j
    - j is the position in the pattern
    - i is the position in the text where we do the check
    - check is performed by determining if $T[i] = P[j]$
      - current guess is $i - j$

- Begin matching with $i = 0, \ j = 0$
- If $T[i] \neq P[j]$ and $j = 0$, shift pattern by 1, same action as in brute-force
  - $i = i + 1$
  - j is unchanged
    - old guess: $i - j$, new guess: $i + 1 - j$
      - new guess increases by 1, i.e. pattern shifts by 1

Knuth-Morris-Pratt Motivation

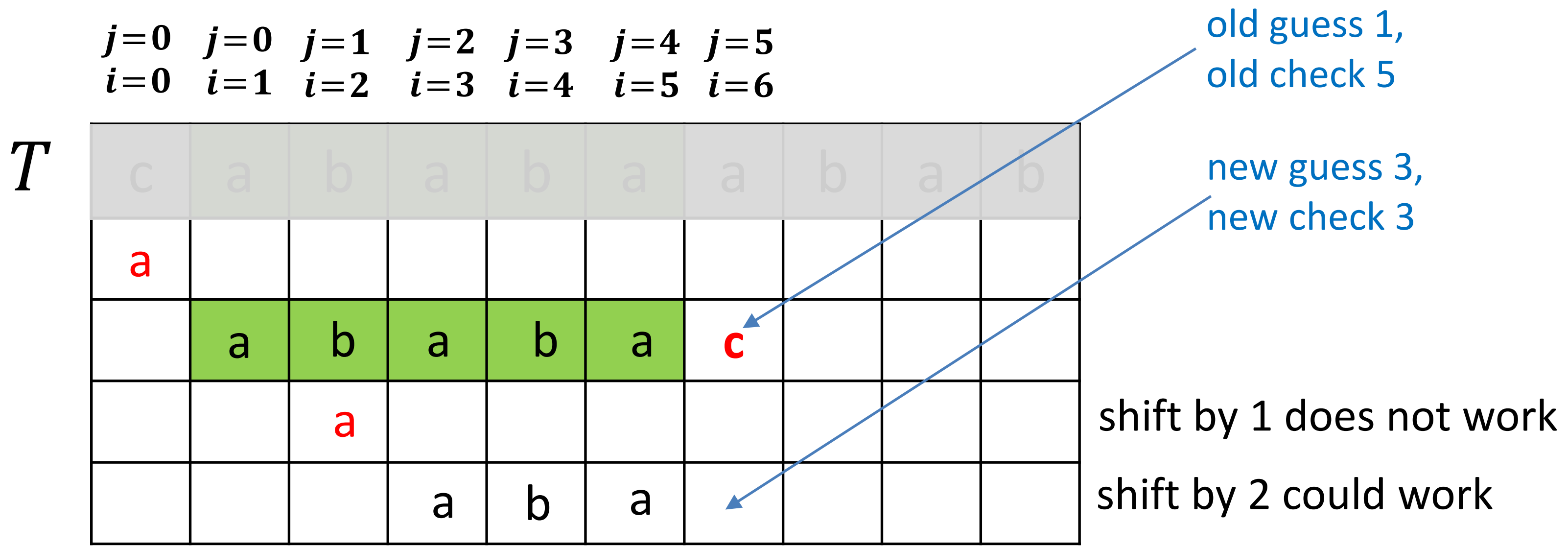
$P = \ ababaca$

	$j=0$ $i=0$	$j=0$ $i=1$	$j=1$ $i=2$	$j=2$ $i=3$	$j=3$ $i=4$	$j=4$ $i=5$	$j=5$ $i=6$			
T	c	a	b	a	b	a	a	b	a	b
	a									
		a	b	a	b	a	c			

- When $T[i] = P[j]$, the action is to check the next letter, as in brute-force
  - $i = i + 1$
  - $j = j + 1$
  - guess was: $i - j$, and it stays the same: $(i + 1) - (j + 1) = i - j$
    - pattern is not shifted
- Failure at text position $i = 6$, pattern position $j = 5$
- When failure is at pattern position $j > 0$, do something smarter than brute force

Knuth-Morris-Pratt Motivation

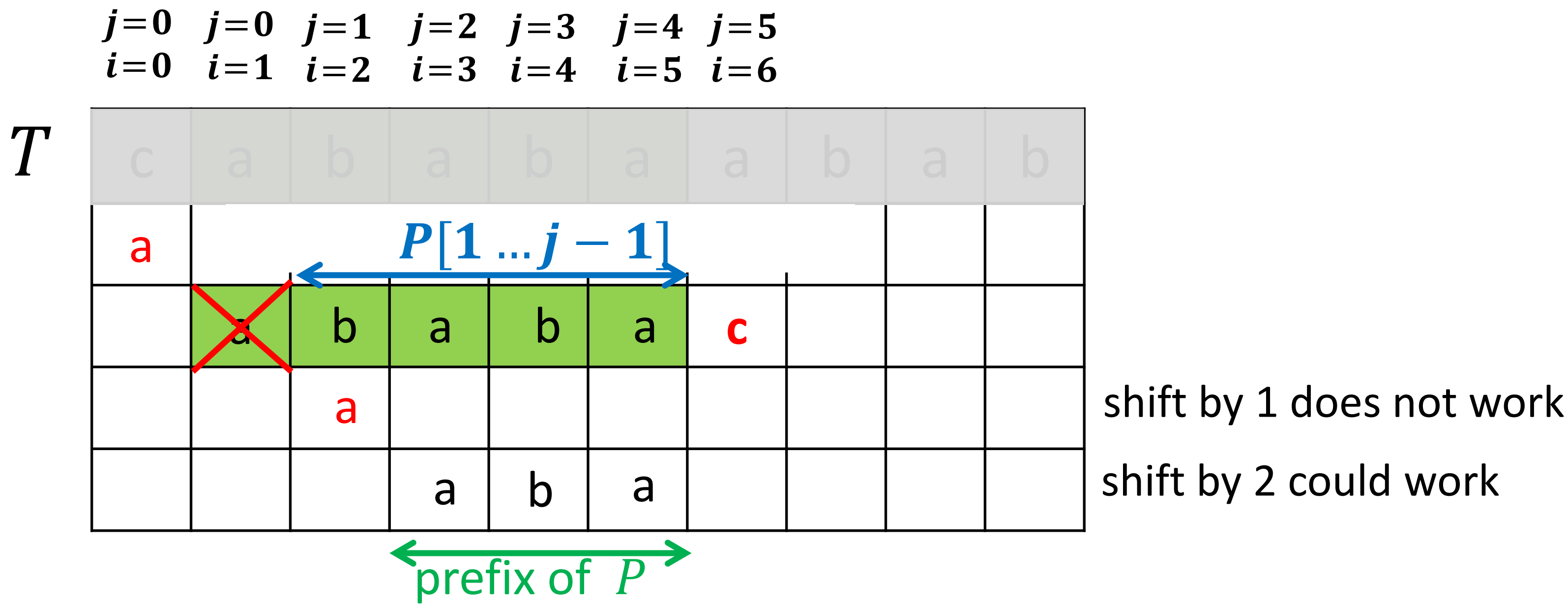
$P = \ ababaca$



- When failure is at pattern position $j > 0$, do something smarter than brute force
- Prior to $j = 5$, pattern and text are equal
  - find how to shift pattern looking only at pattern
- If failure at $j = 5$, shift pattern by 2 **and** start matching with new $j = 3$
  - i **stays the same** because we will be trying to match the same text letter
    - old guess was $i - 5$, the new guess is $i - 3$, so guess increased by 2
  - we skipped one guess and 3 character checks
  - can precompute the action of ‘shift by 2 and skip 3 characters’ before matching even begins, from the pattern, as we do not need text for this computation

Knuth-Morris-Pratt Motivation

$P = ababaca$



- If failure at $j = 5$: continue matching with the same i and new $j = 3$
  - precomputed from pattern before matching begins
- Rule for determining new j
  - find longest suffix of $P[1 \ldots j-1]$ which is also prefix of P
  - call a suffix of P valid if it is a prefix of P
  - new $j =$ length of the longest valid suffix of $P[1 \ldots j-1]$

KMP Failure Array Computation: Slow

- **Rule**: if failure at pattern index $j > 0$, continue matching with the same i and new $j =$ the length of the longest valid suffix of $P[1 \ldots j-1]$
- Store the length of the longest valid suffix of $P[1 \ldots j]$ in $F[j]$
- If failure at pattern index $j > 0$, new $j = F[j-1]$
- Important for efficiency: $F[j] \leq j$
- $P = ababaca$

F	0	1	2	3	4	5	6
	0	0	1				

- $j = 0$
  - $P[1 \ldots 0] =$ "", $P = ababaca$, longest valid suffix is ""
    - $F[0] = 0$ for any pattern
- $j = 1$
  - $P[1 \ldots 1] = b$, $P = ababaca$, longest valid suffix is ""
- $j = 2$
  - $P[1 \ldots 2] = ba$, $P = ababaca$, longest valid suffix is a

KMP Failure Array Computation: Slow

- Store the length of the longest valid suffix of $P[1 \dots j]$ in $F[j]$

F	0	1	2	3	4	5	6
	0	0	1	2	3	0	1

- $j = 3$
  - $P[1 \dots 3] = bab$, $P = ababaca$, longest valid suffix is ab
- $j = 4$
  - $P[1 \dots 4] = baba$, $P = ababaca$, longest valid suffix is aba
- $j = 5$
  - $P[1 \dots 5] = babac$, $P = ababaca$, longest valid suffix is “”
- $j = 6$
  - $P[1 \dots 6] = babaca$, $P = ababaca$, longest valid suffix is a
- Failure array is precomputed before matching starts
  - straightforward computation is $O(m^3)$ time

for $j = 0$ to $m - 1$ // go over all positions in the failure array
    for $i = 1$ to j // go over all suffixes of $P[1 \dots j]$
        for $k = 1$ to i // compare next suffix to prefix of P

String matching with KMP: Example

- $T = cababab cabababaca, P = ababaca$

F

0	1	2	3	4	5	6
0	0	1	2	3	0	1

$\boldsymbol{i=0}$
$\boldsymbol{j=0}$

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

rule 1

if $T[i] = P[j]$
- $i = i + 1$
- $j = j + 1$

rule 2

if $T[i] \neq P[j]$ and $j > 0$
- i unchanged
- $j = F[j - 1]$

rule 3

if $T[i] \neq P[j]$ and $j = 0$
- $i = i + 1$
- j is unchanged

String matching with KMP: Example

- $T = cababab cabababaca$, $P = ababaca$ — i.e. $T = cabababcababaca$, $P = ababaca$

F

0	1	2	3	4	5	6
0	0	1	2	3	0	1

	$j=0$, $i=0$	$j=0$, $i=1$	$j=1$, $i=2$	$j=2$, $i=3$	$j=3$, $i=4$	$j=4$, $i=5$	$j=3$ ~~$j=5$~~, $i=6$	$j=0$ ~~$j=2$~~ ~~$j=4$~~, $i=7$	$j=0$, $i=8$	$j=1$, $i=9$	$j=2$, $i=10$	$j=3$, $i=11$	$j=4$, $i=12$	$j=5$, $i=13$	$j=6$, $i=14$	
T:	c	a	b	a	b	a	b	c	a	b	a	b	a	c	a	
P:	***a***															
		a	b	a	b	a	**c**									new $j = 3$
				(a)	(b)	(a)	b	**a**								new $j = 2$
						(a)	(b)	**a**								new $j = 0$
								a								
									a	b	a	b	a	c	a	match!

if $T[i] = P[j]$
- $i = i + 1$
- $j = j + 1$

if $T[i] \neq P[j]$ and $j > 0$
- i unchanged
- $j = F[j-1]$

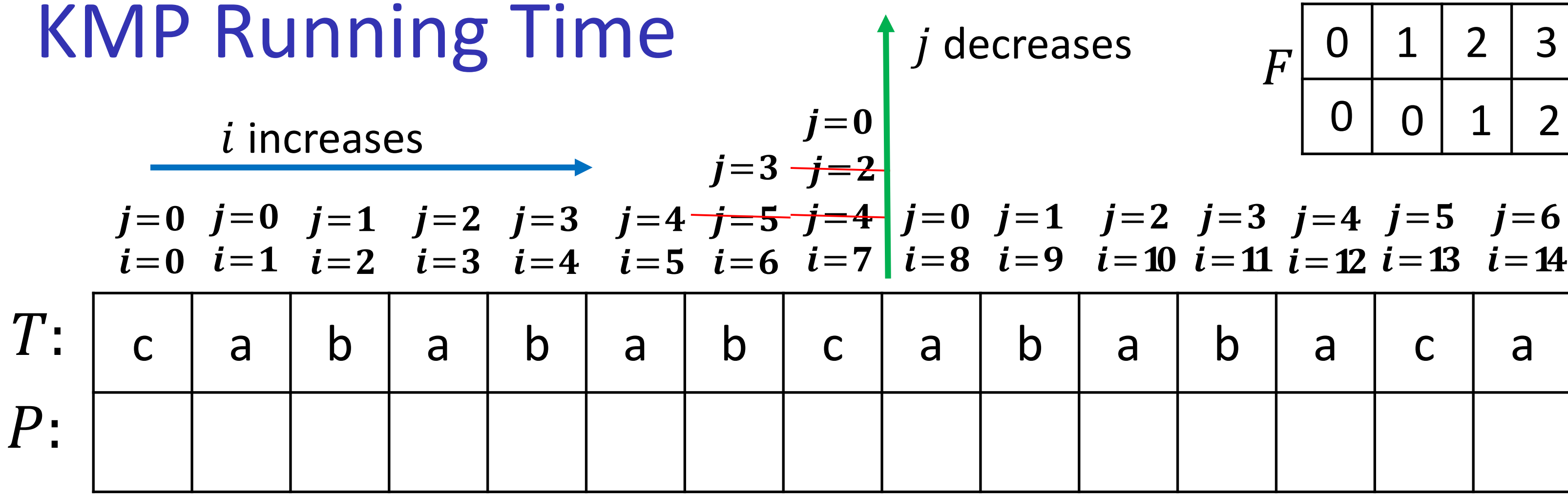
if $T[i] \neq P[j]$ and $j = 0$
- $i = i + 1$
- j is unchanged

Knuth-Morris-Pratt Algorithm

```
KMP::pattern-matching(T, P)
    F ← compute-failure-array(P)
    i ← 0 // current character of T
    j ← 0 // current character of P
    while i < n do
        if P[j] = T[i]
            if j = m − 1
                return "found at guess i − m + 1"
                // guess is equal to i − j
            else // rule 1
                i ← i + 1
                j ← j + 1
        else // P[j] ≠ T[i]
            if j > 0
                j ← F[j − 1] // rule 2
            else
                i ← i + 1 // rule 3
    return FAIL
```

KMP Running Time

F	0	1	2	3	4	5	6
	0	0	1	2	3	0	1



if $T[i] = P[j]$
- $i = i + 1$
- $j = j + 1$

if $T[i] \neq P[j]$ and $j > 0$
- i unchanged
- $j = F[j-1]$
- j decreases

if $T[i] \neq P[j]$ and $j = 0$
- $i = i + 1$
- j is unchanged

- For now, ignore the cost of computing failure array, will account for it later
- Have horizontal and vertical iterations
- At most n horizontal iterations
- i can increase at most n times $\rightarrow$ j can increase at most n times
- Total number of decreases of $j \leq$ total number of increases of $j \leq n$
- At most n vertical iterations
- Each iteration is $O(1)$, at most $2n$ iterations, total runtime is is $O(n)$

Fast Computation of F

- Failure array F
  - $F[0] = 0$, no need to compute
  - for $j > 0$, $F[j]$ = length of the longest suffix of $P[1...j]$ which is also prefix of P
    - i.e. $F[j]$ = longest valid suffix of $P[1 \dots j]$
- Crucial fact: after processing T, final value of j is longest valid suffix of T

$P = ababaca$

$j=0$ $i=0$ | $j=0$ $i=1$ | $j=1$ $i=2$ | $j=2$ $i=3$ | $j=3$ $i=4$

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

- Use the crucial fact for computation of F
  - match $T = P[1 \dots 1]$ with P, and set $F[1]$ = final j
  - match $T = P[1 \dots 2]$ with P, and set $F[2]$ = final j
  - …
  - match $T = P[1 \dots m-1]$ with P, and set $F[m-1]$ = final j
  - but first, let us rename variable j as l (only for failure array computation)
    - since j is already used for $T = P[1 \dots j]$

indexed by j
$j = 1 \dots m$

Fast Computation of F

- Failure array F
  - $F[0] = 0$, no need to compute
  - for $j > 0$, $F[j]$ = length of the longest suffix of $P[1...j]$ which is also prefix of P
    - i.e. $F[j]$ = longest valid suffix of $P[1 \ldots j]$
- Crucial fact: after processing T, final value of l is longest valid suffix of T

$P = ababaca$

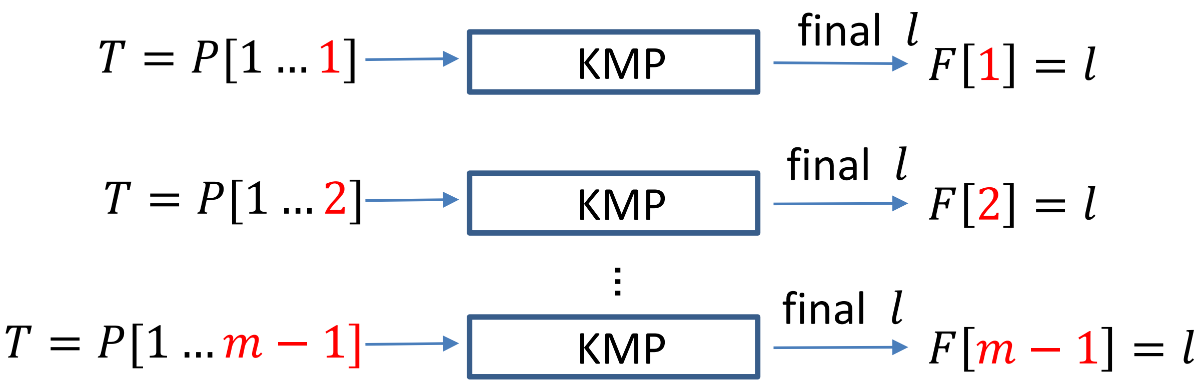
	$l=0$ $i=0$	$l=0$ $i=1$	$l=1$ $i=2$	$l=2$ $i=3$	$l=3$ $i=4$
T:	c	a	b	a	
P:	a				
		a	b	a	

- Use the crucial fact for computation of F
  - match $T = P[1 \ldots 1]$ with P, and set $F[1]$ = final l
  - match $T = P[1 \ldots 2]$ with P, and set $F[2]$ = final l
  - …
  - match $T = P[1 \ldots m-1]$ with P, and set $F[m-1]$ = final l

Fast Computation of F

- $P = ababaca$
- Useful fact
  - after processing T, final value of l is longest valid suffix of T
- Failure array F
  - for $j > 0$, $F[j] =$ length of the longest valid suffix of $P[1...j]$
- Big idea

	$l=0$, $i=0$	$l=0$, $i=1$	$l=1$, $i=2$	$l=2$, $i=3$	$l=3$, $i=4$
T:	c	a	b	a	
P:	a				
		a	b	a	



'chicken and egg' problem with big idea: need F to put text through KMP

Fast Computation of F: Big Idea Saved

if failure at $l > 0$, $l = F[l-1]$

- $j = 1$ $\quad T = P[1 \dots 1] \rightarrow$ KMP $\xrightarrow{\text{final } l}$ $F[1] = l$
  - start with $l = 0$
  - text has one letter, KMP can reach at most $l = 1$
  - need at most $F[0]$, and already have it as $F[0]$ is always 0
- $j = 2$ $\quad T = P[1 \dots 2] \rightarrow$ KMP $\xrightarrow{\text{final } l}$ $F[2] = l$
  - start with $l = 0$
  - text has two letters, can reach at most $l = 2$
  - need at most $F[0], F[1]$, already computed at previous iteration

⋮

- $j = m-1$ $\quad T = P[1 \dots m-1] \rightarrow$ KMP $\xrightarrow{\text{final } l}$ $F[m-1] = l$
  - start with $l = 0$
  - text has $m-1$ letters, can reach at most $l = m-1$
  - need at most $F[0], F[1], \dots, F[m-2]$, already computed at previous iterations

Fast Computation of F: Big Idea Made Bigger

$T = P[1 \ldots 1]$ → KMP → final l → $F[1] = l$

$T = P[1 \ldots 2]$ → KMP → final l → $F[2] = l$

do not start from scratch, start from where $P[1 \ldots 1]$ finished

$T = P[1 \ldots 3]$ → KMP → final l → $F[3] = l$

do not start from scratch, start from where $P[1 \ldots 2]$ finished

⋮

$T = P[1 \ldots m-1]$ → KMP → final l → $F[m-1] = l$

do not start from scratch, start from where $P[1 \ldots m-2]$ finished

- Cost of passing $P[1 \ldots 1], P[1 \ldots 2], \ldots, P[1 \ldots m-1]$ through KMP is equal to the cost of passing just $P[1 \ldots m-1]$ through KMP

Fast Computation of F

F	0	1	2	3	4	5	6
	0						

- Process $T = P[1 \dots j]$, $F[j] =$ final l
- $P = ababaca$
- Initialize $F[0] = 0$

Fast Computation of F

F	0	1	2	3	4	5	6
	0	0					

- Process $T = P[1 \dots j]$, $F[j] =$ final l
- $P = ababaca$
- $j = 1, T = P[1 \dots j] = b$

	$l=0$, $i=0$	$l=0$, $i=1$									
T:	b										
P:	$\boldsymbol{a}$										

if $T[i] = P[l]$
- $i = i + 1$
- $l = l + 1$

if $T[i] \neq P[l]$ and $l > 0$
- i unchanged
- $l = F[l - 1]$

if $T[i] \neq P[l]$ and $l = 0$
- $i = i + 1$
- l is unchanged

Fast Computation of F

F	0	1	2	3	4	5	6
	0	0	1				

- Process $T = P[1 \dots j]$, $F[j]$ = final l
- $P = ababaca$
- $j = 2, T = P[1 \dots j] = ba$

	$l=0$ $i=0$	$l=0$ $i=1$	$l=1$ $i=2$								
T:	b	a									
P:	$\boldsymbol{a}$										
		a									

if $T[i] = P[l]$
- $i = i + 1$
- $l = l + 1$

if $T[i] \neq P[l]$ and $l > 0$
- i unchanged
- $l = F[l-1]$

if $T[i] \neq P[l]$ and $l = 0$
- $i = i + 1$
- l is unchanged

Fast Computation of F

F	0	1	2	3	4	5	6
	0	0	1	2			

- Process $T = P[1 \ldots j]$, $F[j]$ = final l
- $P = ababaca$
- $j = 3, T = P[1 \ldots j] = bab$

	$l=0$ $i=0$	$l=0$ $i=1$	$l=1$ $i=2$	$l=2$ $i=3$							
T:	b	a	b								
P:	$\boldsymbol{a}$										
		a	b								

if $T[i] = P[l]$
- $i = i + 1$
- $l = l + 1$

if $T[i] \neq P[l]$ and $l > 0$
- i unchanged
- $l = F[l - 1]$

if $T[i] \neq P[l]$ and $l = 0$
- $i = i + 1$
- l is unchanged

Fast Computation of F

F	0	1	2	3	4	5	6
	0	0	1	2	3		

- Process $T = P[1 \dots j]$, $F[j] =$ final l
- $P = ababaca$
- $j = 4, T = P[1 \dots j] = baba$

	$l=0$ $i=0$	$l=0$ $i=1$	$l=1$ $i=2$	$l=2$ $i=3$	$l=3$ $i=4$						
T:	b	a	b	a							
P:	$\boldsymbol{a}$										
		a	b	a							

if $T[i] = P[l]$
- $i = i + 1$
- $l = l + 1$

if $T[i] \neq P[l]$ and $l > 0$
- i unchanged
- $l = F[l-1]$

if $T[i] \neq P[l]$ and $l = 0$
- $i = i + 1$
- l is unchanged

Fast Computation of F

F	0	1	2	3	4	5	6
	0	0	1	2	3	0	

- Process $T = P[1 \dots j]$, $F[j]$ = final l
- $P = ababaca$
- $j = 5, T = P[1 \dots j] = babac$

	$l=0$, $i=0$	$l=0$, $i=1$	$l=1$, $i=2$	$l=2$, $i=3$	$l=0$, ~~$l=1$~~, ~~$l=3$~~, $i=4$	$l=0$, $i=5$						
T:	b	a	b	a	c							
P:	a											
		a	b	a	b							new $l = 1$
				(a)	b							new $l = 0$
					a							

if $T[i] = P[l]$
- $i = i + 1$
- $l = l + 1$

if $T[i] \neq P[l]$ and $l > 0$
- i unchanged
- $l = F[l - 1]$

if $T[i] \neq P[l]$ and $l = 0$
- $i = i + 1$
- l is unchanged

Fast Computation of F

F	0	1	2	3	4	5	6
	0	0	1	2	3	0	1

- Process $T = P[1 \ldots j]$, $F[j] =$ final l
- $P = ababaca$
- $j = 6, T = P[1 \ldots j] = babaca$

	$l=0$, $i=0$	$l=0$, $i=1$	$l=1$, $i=2$	$l=2$, $i=3$	$l=0$, ~~$l=1$~~, ~~$l=3$~~, $i=4$	$l=0$, $i=5$	$l=1$, $i=6$					
T:	b	a	b	a	c	a						
P:	a											
		a	b	a	b							new $l = 1$
				(a)	b							new $l = 0$
					a							
						a						

if $T[i] = P[l]$
- $i = i + 1$
- $l = l + 1$

if $T[i] \neq P[l]$ and $l > 0$
- i unchanged
- $l = F[l-1]$

if $T[i] \neq P[l]$ and $l = 0$
- $i = i + 1$
- l is unchanged

Fast Computation of F

- Equivalent to matching $T = P[1 \dots m-1]$ with pattern P, updating $F[i+1] = l$ after letter i is processed

F	0	1	2	3	4	5	6
	0	0	1	2	3	0	1

update $F[1]$ after processing $i = 0$

	$l=0$, $i=0$	$l=0$, $i=1$	$l=1$, $i=2$	$l=2$, $i=3$	$l=0$, ~~$l=1$~~, ~~$l=3$~~, $i=4$	$l=0$, $i=5$	$l=1$, $i=6$					
T:	b	a	b	a	c	a						
P:	***a***											
		a	b	a	***b***							new $l = 1$
				(a)	***b***							new $l = 0$
					a							
						a						

if $T[i] = P[l]$
- $i = i + 1$
- $l = l + 1$

if $T[i] \neq P[l]$ and $l > 0$
- i unchanged
- $l = F[l-1]$

if $T[i] \neq P[l]$ and $l = 0$
- $i = i + 1$
- l is unchanged

Fast Computation of F

- Equivalent to matching $T = P[1 \dots m-1]$ with pattern P, updating $F[i+1] = l$ after letter i is processed

F	0	1	2	3	4	5	6
	0	0	1	2	3	0	1

update $F[2]$ after processing $i = 1$

	$l=0$ $i=0$	$l=0$ $i=1$	$l=1$ $i=2$	$l=2$ $i=3$	$l=0$ ~~$l=1$~~ ~~$l=3$~~ $i=4$	$l=0$ $i=5$	$l=1$ $i=6$					
T:	b	a	b	a	c	a						
P:	***a***											
		a	b	a	***b***							new $l = 1$
				(a)	***b***							new $l = 0$
					a							
						a						

if $T[i] = P[l]$
- $i = i + 1$
- $l = l + 1$

if $T[i] \neq P[l]$ and $l > 0$
- i unchanged
- $l = F[l-1]$

if $T[i] \neq P[l]$ and $l = 0$
- $i = i + 1$
- l is unchanged

Fast Computation of F

F	0	1	2	3	4	5	6
	0	0	1	2	3	0	1

- Equivalent to matching $T = P[1 \ldots m-1]$ with pattern P, updating $F[i+1] = l$ after letter i is processed

update $F[3]$ after processing $i = 2$

	$l=0$ $i=0$	$l=0$ $i=1$	$l=1$ $i=2$	$l=2$ $i=3$	$l=0$ ~~$l=1$~~ ~~$l=3$~~ $i=4$	$l=0$ $i=5$	$l=1$ $i=6$					
T:	b	a	b	a	c	a						
P:	***a***											
		a	b	a	***b***							new $l = 1$
				(a)	***b***							new $l = 0$
					a							
						a						

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

- $i = i + 1$
- $l = l + 1$

if $T[i] \neq P[l]$ and $l > 0$

- i unchanged
- $l = F[l-1]$

if $T[i] \neq P[l]$ and $l = 0$

- $i = i + 1$
- l is unchanged

Fast Computation of F

- Replace T by P and start i at 1
  - since do not use the first pattern letter $P = ababaca$
- Update $F[i] = l$ after letter i is processed

F	0	1	2	3	4	5	6
	0	0	1	2	3	0	1

	$l=0$, $i=1$	$l=0$, $i=2$	$l=1$, $i=3$	$l=2$, $i=4$	$l=0$, ~~$l=1$~~, ~~$l=3$~~, $i=5$	$l=0$, $i=6$	$l=1$, $i=7$					
$\boldsymbol{P}$:	b	a	b	a	c	a						
P:	$\boldsymbol{a}$											
		a	b	a	$\boldsymbol{b}$							new $l = 1$
				(a)	$\boldsymbol{b}$							new $l = 0$
					$\boldsymbol{a}$							
						a						

if $\boldsymbol{P}[i] = P[l]$
- $i = i + 1$
- $l = l + 1$

if $\boldsymbol{P}[i] \neq P[l]$ and $l > 0$
- i unchanged
- $l = F[l-1]$

if $\boldsymbol{P}[i] \neq P[l]$ and $l = 0$
- $i = i + 1$
- l is unchanged

Fast Computation of F

F							
	0	1	2	3	4	5	6
	0	0	1	2	3	0	1

- Rename i into j
  - makes it clear that we match text is $P[1 \ldots j]$ at each iteration

	$l=0$ $j=1$	$l=0$ $j=2$	$l=1$ $j=3$	$l=2$ $j=4$	$l=0$ ~~$l=1$~~ ~~$l=3$~~ $j=5$	$l=0$ $j=6$	$l=1$ $j=7$					
P:	b	a	b	a	c	a						
P:	a											
		a	b	a	b							new $l = 1$
				(a)	b							new $l = 0$
					a							
						a						

if $P[j] = P[l]$
- $j = j + 1$
- $l = l + 1$

if $P[j] \neq P[l]$ and $l > 0$
- j unchanged
- $l = F[l - 1]$

if $P[j] \neq P[l]$ and $l = 0$
- $j = j + 1$
- l is unchanged

KMP: Computing Failure Array

- Pseudocode is almost identical to *KMP*(T, P)
  - main difference: $F[j]$ gets both used and updated
- Runtime $\Theta(m)$, same analysis as for *KMP*

```
compute-failure-array(P)
P: string of length m (pattern)
    F[0] ← 0
    j ← 1 // matching P[1 ... j]
    l ← 0
    while j < m do
        if P[j] = P[l]  // rule 1
            l ← l + 1
            F[j] ← l
            j ← j + 1
        else if l > 0   // rule 2
            l ← F[l − 1]
        else            // rule 3
            F [j] ← 0   // l = 0
            j ← j + 1
```

KMP: Main Function Runtime

```
KMP::pattern-matching (T, P)
    F ← compute-failure-array(P)
    i ← 0
    j ← 0
    while i < n do
        if P[j] = T[i]
            if j = m − 1
                return "found at guess i − m + 1"
            else
                i ← i + 1
                j ← j + 1
        else // P[j] ≠ T[i]
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return FAIL
```

- KMP main function
  - *compute-failure-array* is $\Theta(m)$ time
  - The rest of KMP is $\Theta(n)$
  - Running time KMP altogether: $\Theta(n + m)$
    - which is the same as $\Theta(n)$ as $m \leq n$

Outline

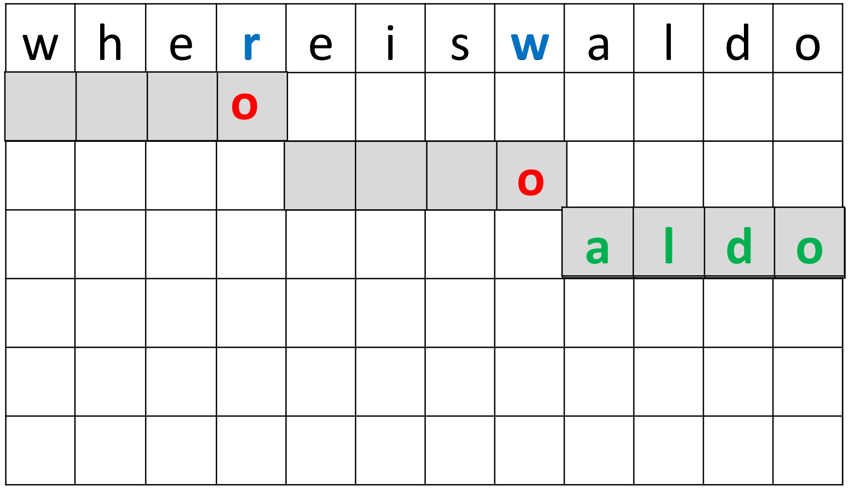
- String Matching
  - Introduction
  - Karp-Rabin Algorithm
  - Knuth-Morris-Pratt algorithm
  - Boyer-Moore Algorithm
  - Suffix Trees
  - Suffix Arrays
  - Conclusion

Boyer-Moore Algorithm Motivation

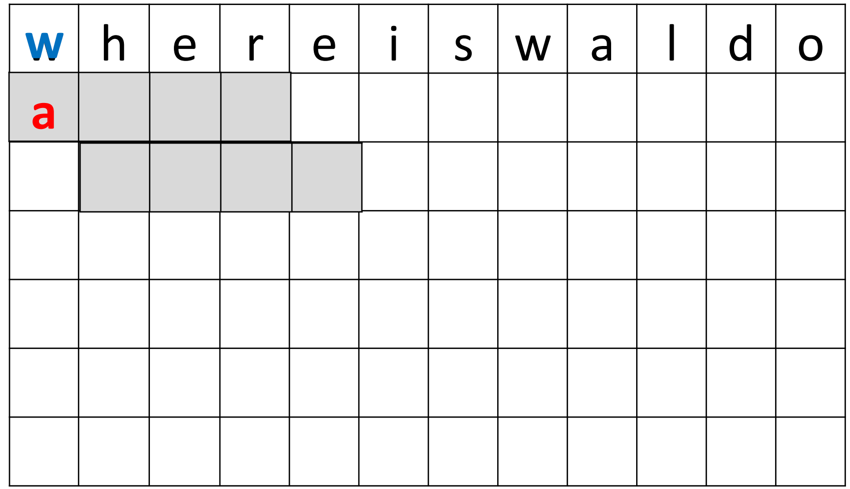
- Fastest pattern matching in practice on English Text
- Important components
  - Reverse-order searching
    - compare P with a guess moving *backwards*
- When a mismatch occurs choose the better option among the two below
  1. Bad character heuristic
     - eliminate shifts based on mismatched character of T
  2. Good suffix heuristic
     - eliminate shifts based on the matched part (i.e.) suffix of P

Reverse Searching vs. Forward Searching

T= whereiswaldo, P = aldo



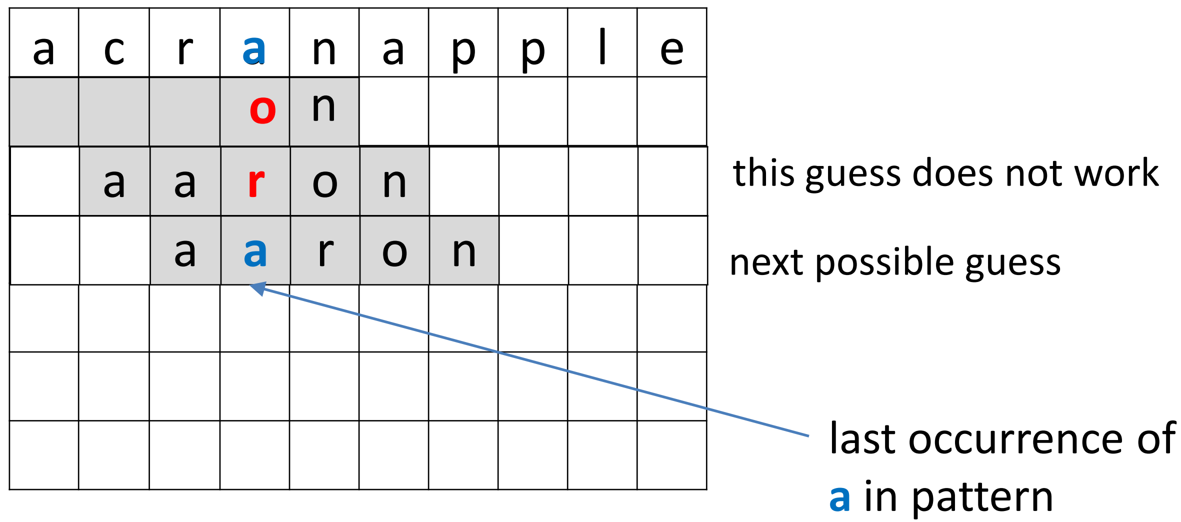
- r does not occur in P = aldo
- move pattern past r
- w does not occur in P = aldo
- move pattern past w
- bad character heuristic can rule out many guesses with reverse searching



- w does not occur in P = aldo
- move pattern past w
- shift by 1 moves pattern past w
- no guesses are ruled out
- bad character heuristic does not rule out any guesses with forward searching when the first character of the pattern is mismatched

What if Mismatched Text Character Occurs in P?

T= acranapple, P = aaron



- Mismatched character in the text is a
- Find **last** occurrence of a in P
- Move the pattern to the right until **last** a in P aligns with a in text
  - all smaller shifts are impossible since they do not match a
- Precompute last occurrence of any letter before matching starts

Bad Character Heuristic: Side Note

T= acranapple, P = aaron

a	c	r	a	n	a	p	p	l	e	
			o	n						
		a	a	r	o	n				missed valid guess
			a	a	r	o	n			also a valid guess

- If we moved until the **first** a in P aligns with a in text
  - this would give a possible guess, but misses an earlier guess which is also possible, possibly leading to a missed pattern

Bad Character Heuristic: Full Version

- Extends to the case when mismatched text character does occur in P

T= acranapple, P = aaron

a	c	r	a	n	a	p	p	l	e
			o	n					
			[a]						

- Mismatched character in the text is **a**
- Move the pattern to the right so that the last **a** in P aligns with **a** in text
- Continue matching the pattern (in reverse)

Bad Character Heuristic: Full Version

- Extends to the case when mismatched text character does occur in P

T= acranapple, P = aaron

a	c	r	a	n	a	p	p	l	e
			o	n					
			[a]			n			

- Mismatched character in the text is a
- Move the pattern to the right so that the last a in P aligns with a in text
- Continue matching the pattern (in reverse)

Bad Character Heuristic: Last Occurrence Array

- Compute the last occurrence array $L(c)$ of any character in the alphabet
  - $L(c) = -1$ if character c does not occur in P, otherwise
  - $L(c) =$ largest index j such that $P[j] = c$
- Example: P = aaron
  - initialization

$char$	a	n	o	r	all others
$L(c)$	-1	-1	-1	-1	-1

this means:

a	b	c	d	e	…	x	y	z
-1	-1	-1	-1	-1		-1	-1	-1

in actual implementation:

0	1	2	3	4	5	…	24	25
-1	-1	-1	-1	-1	-1		-1	-1

Bad Character Heuristic: Last Occurrence Array

- Compute the last occurrence array $L(c)$ of any character in the alphabet
  - $L(c) = -1$ if character c does not occur in P, otherwise
  - $L(c) =$ largest index j such that $P[j] = c$
- Example: P = aaron
  - computation

aaron

$i = 0$

$char$	a	n	o	r	all others
$L(c)$	0	-1	-1	-1	-1

L is valid for $P =$ a

Bad Character Heuristic: Last Occurrence Array

- Compute the last occurrence array $L(c)$ of any character in the alphabet
  - $L(c) = -1$ if character c does not occur in P, otherwise
  - $L(c) =$ largest index j such that $P[j] = c$
- Example: P = aaron
  - computation

aaron

$i = 1$

$char$	a	n	o	r	all others
$L(c)$	1	-1	-1	-1	-1

L is valid for $P =$ aa

Bad Character Heuristic: Last Occurrence Array

- Compute the last occurrence array $L(c)$ of any character in the alphabet
  - $L(c) = -1$ if character c does not occur in P, otherwise
  - $L(c) =$ largest index j such that $P[j] = c$
- Example: P = aaron
  - computation

$char$	a	n	o	r	all others
$L(c)$	1	-1	-1	2	-1

aaron

$i = 2$

L is valid for $P =$ aar

Bad Character Heuristic: Last Occurrence Array

- Compute the last occurrence array $L(c)$ of any character in the alphabet
  - $L(c) = -1$ if character c does not occur in P, otherwise
  - $L(c) =$ largest index j such that $P[j] = c$
- Example: P = aaron
  - computation

aaron

$i = 3$

$char$	a	n	o	r	all others
$L(c)$	1	-1	3	2	-1

L is valid for $P =$ aaro

Bad Character Heuristic: Last Occurrence Array

- Compute the last occurrence array $L(c)$ of any character in the alphabet
  - $L(c) = -1$ if character c does not occur in P, otherwise
  - $L(c) =$ largest index j such that $P[j] = c$
- Example: P = aaron
  - computation

aaron

$i = 4$

$char$	a	n	o	r	all others
$L(c)$	1	4	3	2	-1

L is valid for $P =$ aaron

- Total time is $O(m + |\sum|)$

Boyer-More Indexing

- Same as in KMP
  - maintain variables i and j
  - j is the position in the pattern
  - i is the position in the text where we do the next check
  - check is performed by determining if $T[i] = P[j]$
  - current guess is $i - j$

Bad Character Heuristic: Formula

$char$	a	n	o	r	all others
$L(c)$	1	4	3	2	-1

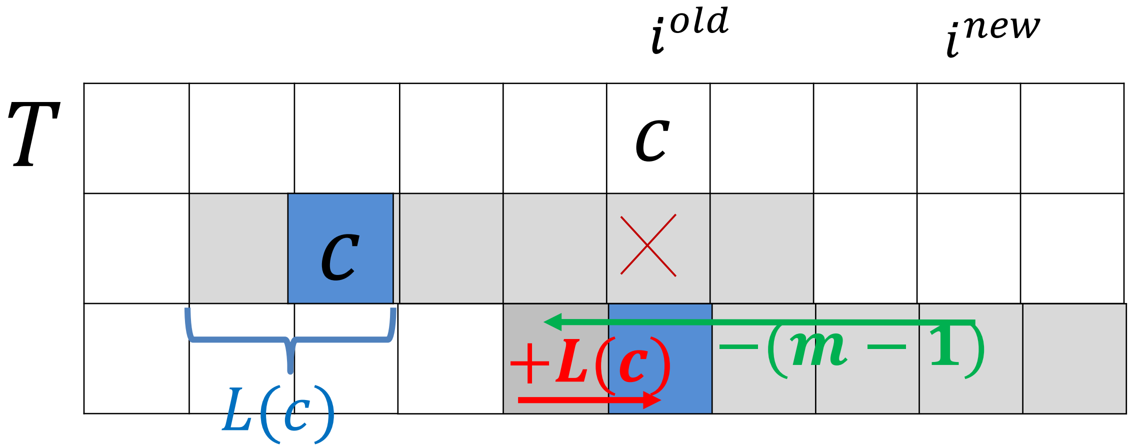
T= acranapple, P = aaron

			$j=3$ $i=3$			$j=4$ $i=6$			
a	c	r	**a**	n	a	p	p	l	e
			o	n					
			[a]			**n**			

- Let $L(c)$ be the last occurrence of character c in P
  - $L(\mathbf{a}) = 1$ in our example
- When mismatch occurs at text position i, pattern position j, update
  - $j = m - 1$
    - start matching at the end of the pattern
  - $i = i + m - 1 - L(c)$
  - for our example
    - $j = 5 - 1 = 4$
    - $i = 3 + 5 - 1 - 1 = 6$

Bad Character Heuristic: Formula Explained

- Text character is c at the mismatch position i in the text
- $i = i + m - 1 - L(c)$



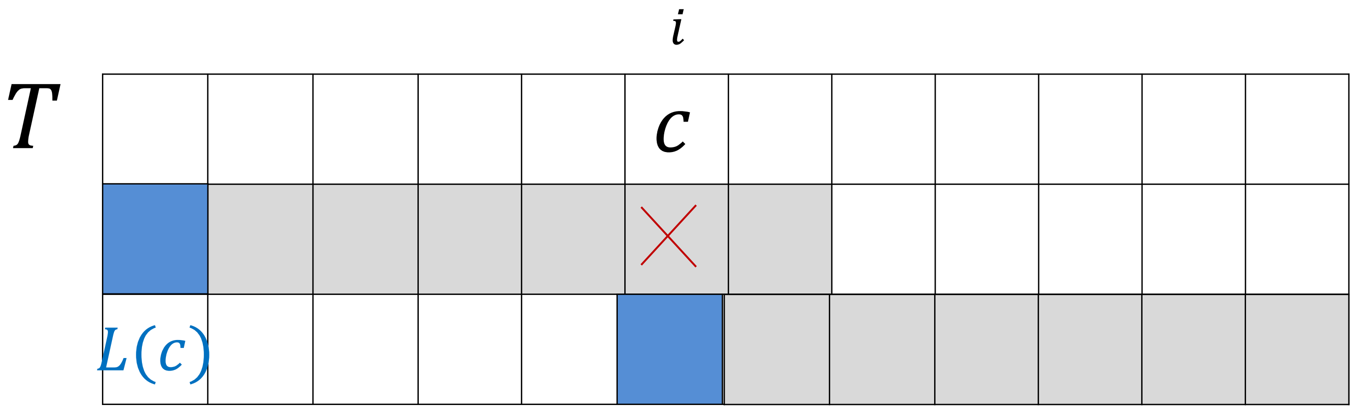
$$i^{new} - (m-1) + L(c) = i^{old}$$

$$i^{new} = i^{old} + m - 1 - L(c)$$

$$i = i + m - 1 - L(c)$$

Bad Character Heuristic: Formula Explained

- Text character is c at the mismatch position i in the text
- $i \;\; = i + m - 1 - L(c)$
- Also works if $L(c) = -1$



moves pattern completely past mismatched text character c

Bad Character Heuristic: Important Use Condition

- Text character is c at the mismatch position i in the text
  - $i \; = i + m - 1 - L(c), j = m - 1$
- Old guess: $i - j$
- New guess: $i + (m-1) - L(c) - (m-1) = i - L(c)$
- If $L(c) > j$, new guess < old guess and moves P in wrong direction, not useful
  - we already ruled that guess out, no point to come back to it
- Example: T= acranapple, P = reroa

$j=3$
$i=8$

c	a	c	r	w	a	a	p	a	a	e
				a						
								o	a	
							o	a		

$L(\mathrm{a}) = 4 > j = 3$

old guess: $i - j = 8 - 3 = 5$

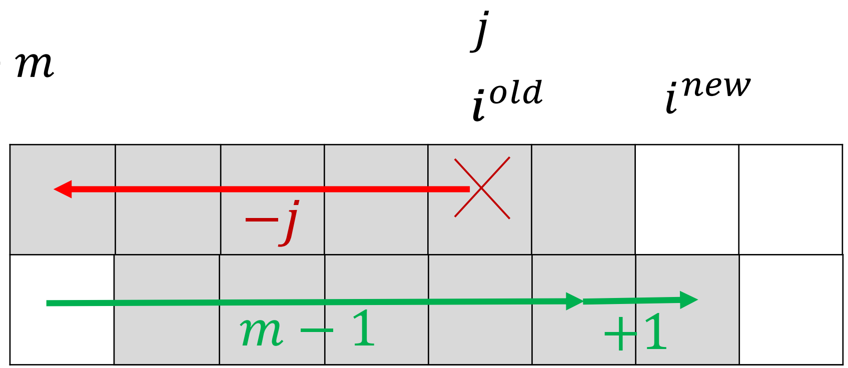
$i^{new} = 8 + 5 - 1 - 4 = 8$
$j^{new} = 5 - 1 = 4$

new guess: $i^{new} - j^{new} = 8 - 4 = 4$

- **bad character heuristic makes sense to use only if $L(c) < j$**
  - note that $L(c) \neq j$ in case of a mismatch

Bad Character Heuristic: Brute-Force Step

- If $L(c) > j$
  - pattern would move in wrong direction if used bad character heuristic
  - therefore, do brute-force step
    - $j = m - 1$
    - $i = i - j + m$



$$i^{old} \; -j \; +m - 1 \; +1 \; = i^{new}$$

$$i^{new} = i^{old} - j + m$$

$$i = i - j + m$$

Bad Character Heuristic: Unified Formula

1. If $L(c) < j$ [bad character heuristic step]
   - $j = m - 1$
   - $i = i + m - 1 - L(c)$
2. If $L(c) > j$ [brute-force step]
   - $j = m - 1$
   - $i = i - j + m$

- Unified formula for i that works in both cases

$$i = i + m - 1 - \min\{L(c), j - 1\}$$

Boyer-More Example

$char$	a	e	p	r	others
$L(c)$	1	3	2	4	-1

P = paper

$j=4, i=4$ $j=4, i=7$ $j=4, i=9$ $j=3, i=13$ $j=4, i=14$ $j=4, i=15$

T	f	e	e	d	a	l	l	p	o	o	r	p	a	r	r	o	t	s	
					r														$i=7$
					[a]			r											$i=9$
								[p]		r									$i=14$
														e	r				$i=15$
																r			$i=20$

not found!

- Unified formula for i that works in all cases

$$i = i + m - 1 - \min\{L(c), j - 1\}$$

Boyer-Moore Algorithm

BoyerMoore(T, P)

$L \leftarrow$ last occurrence array computed from P

$j \leftarrow m - 1$

$i \leftarrow m - 1$

while $i < n$ and $j \geq 0$ **do** //current guess begins at index $i - j$

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

$i \leftarrow i - 1$

$j \leftarrow j - 1$

else

$i \leftarrow i + m - 1 - \min\{L(c), j - 1\}$

$j \leftarrow m - 1$

if $j = -1$ **return** "found at guess $i + 1$" // i moved one position to

// the left of the first char in T

else return FAIL

Good Suffix Heuristic

- Idea is similar to KMP, but applied to the suffix, since matching backwards

P = onobobo

$j=3$
$i=3$

T	o	n	o	o	o	b	o	o	o	i	b	b	o	u	n	d	a	r	y
				b	o	b	o												
			o	n	o	b	o	b	o										

- Text has letters obo
- Do the smallest move so that obo fits
- Can precompute this from the pattern itself, before matching starts
  - ‘if failure at $j = 3$, shift pattern by 2’
- Continue matching from the end of the new shift
- Will not study the precise way to do it

Boyer-Moore Summary

- Boyer-Moore performs very well, even when using only bad character heuristic
- Worst case run time is $O(nm)$ with bad character heuristic only, but in practice much faster
- On typical English text, Boyer-Moore looks only at ≈25% of text T
- With good suffix heuristic, can ensure $O(n + m + |\Sigma|)$ run time
    - no details

Outline

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

Suffix Tree: Trie of Suffixes

- What if we search for many patterns P within the same fixed text T?
- Idea: preprocess the text T rather than pattern P
- Observation: P is a substring of T if and only if P is a prefix of some suffix of T
- Example: $P =$ ish

prefix

$T =$ establishment

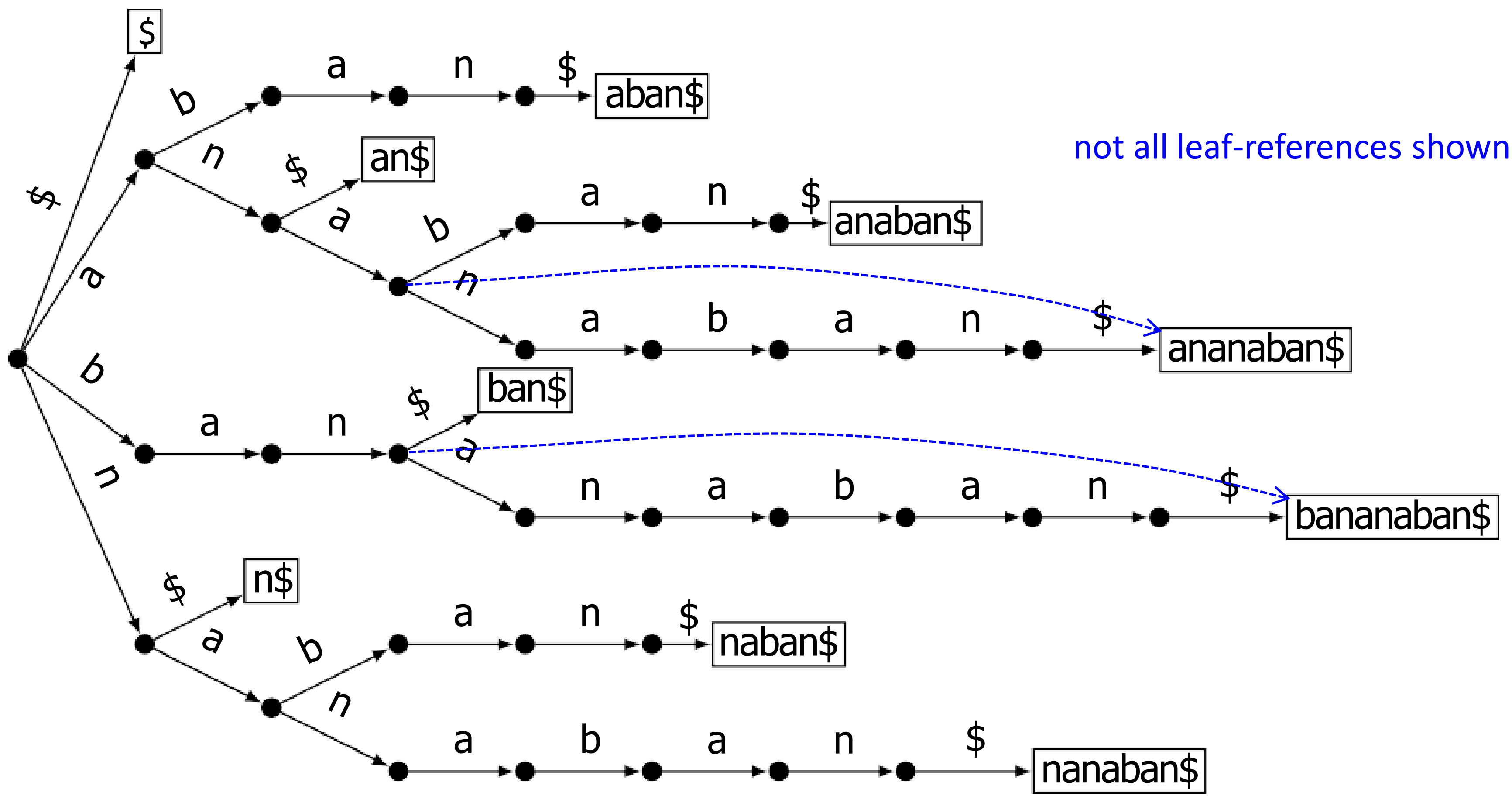
suffix

- Naïve idea: store all suffixes of T in a trie
  - if $|T| = n$, then $n+1$ suffixes together have $0+1+2+\cdots+n \in \Theta(n^2)$ characters
    - wastes space
- **Suffix tree** saves space in multiple ways
  - store suffixes implicitly via indices into T
  - use compressed trie
  - $O(n)$ space since we store $n+1$ suffixes (words)

Trie of suffixes: Example

- *T* = bananaban

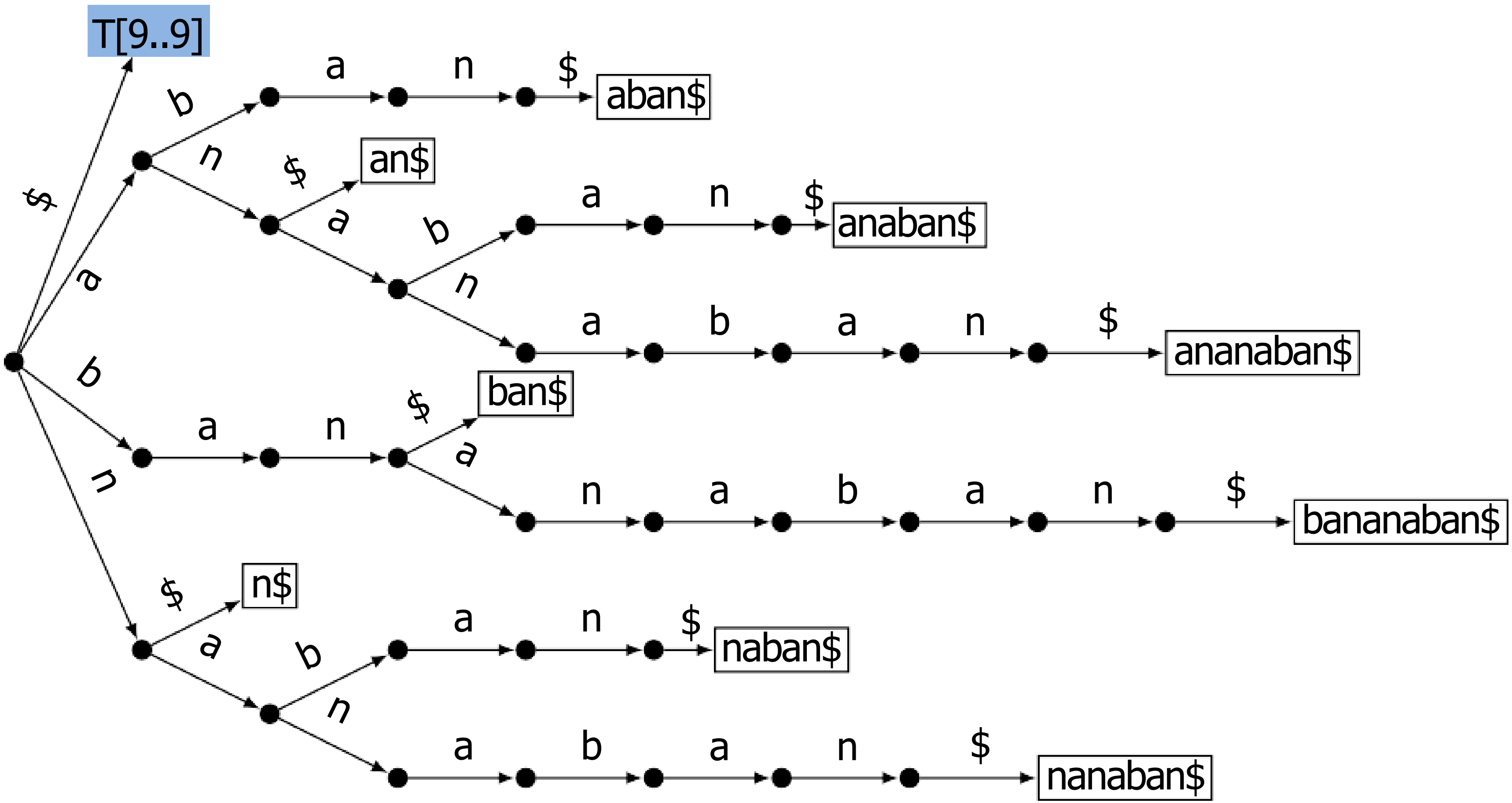
  Suffixes = {bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n, Λ}
- Convenient to order children alphabetically



Trie of suffixes: Example

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

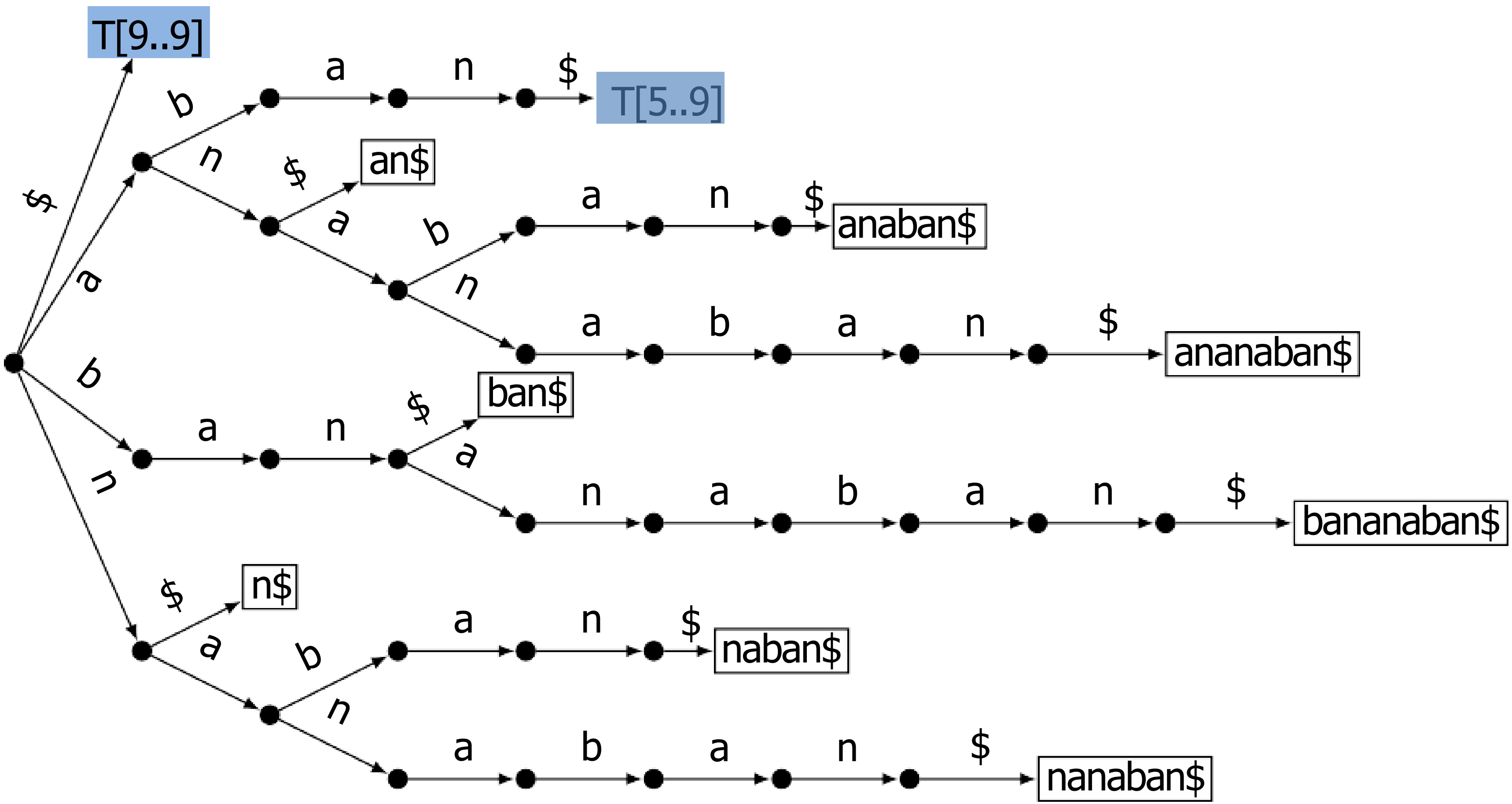
- Store suffixes via indices



Trie of suffixes: Example

- Store suffixes via indices

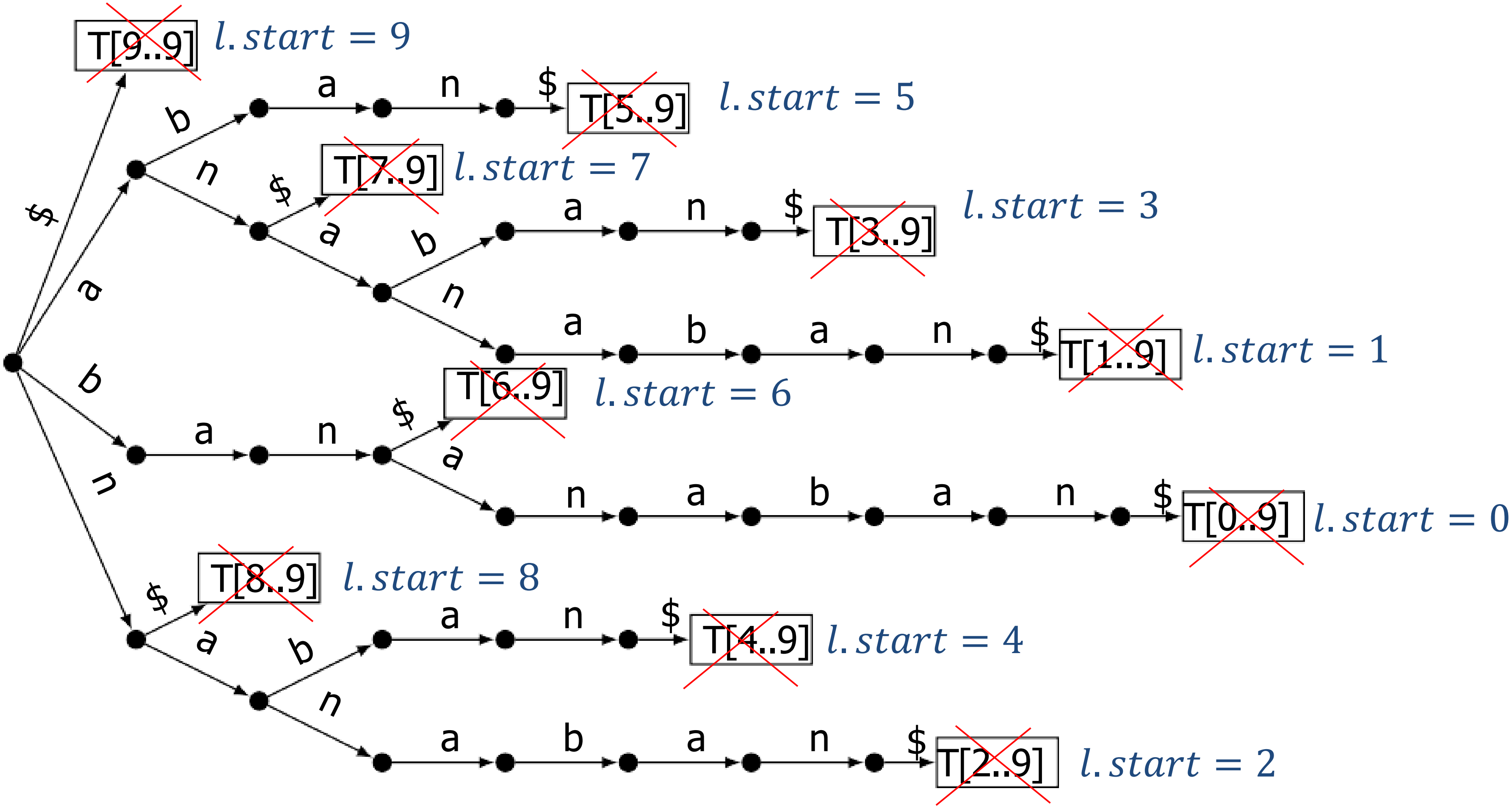
	0	1	2	3	4	5	6	7	8	9
T =	b	a	n	a	n	a	b	a	n	$



Tries of suffixes

- In actual implementation, each leaf l stores the start of its suffix in variable $l.start$

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

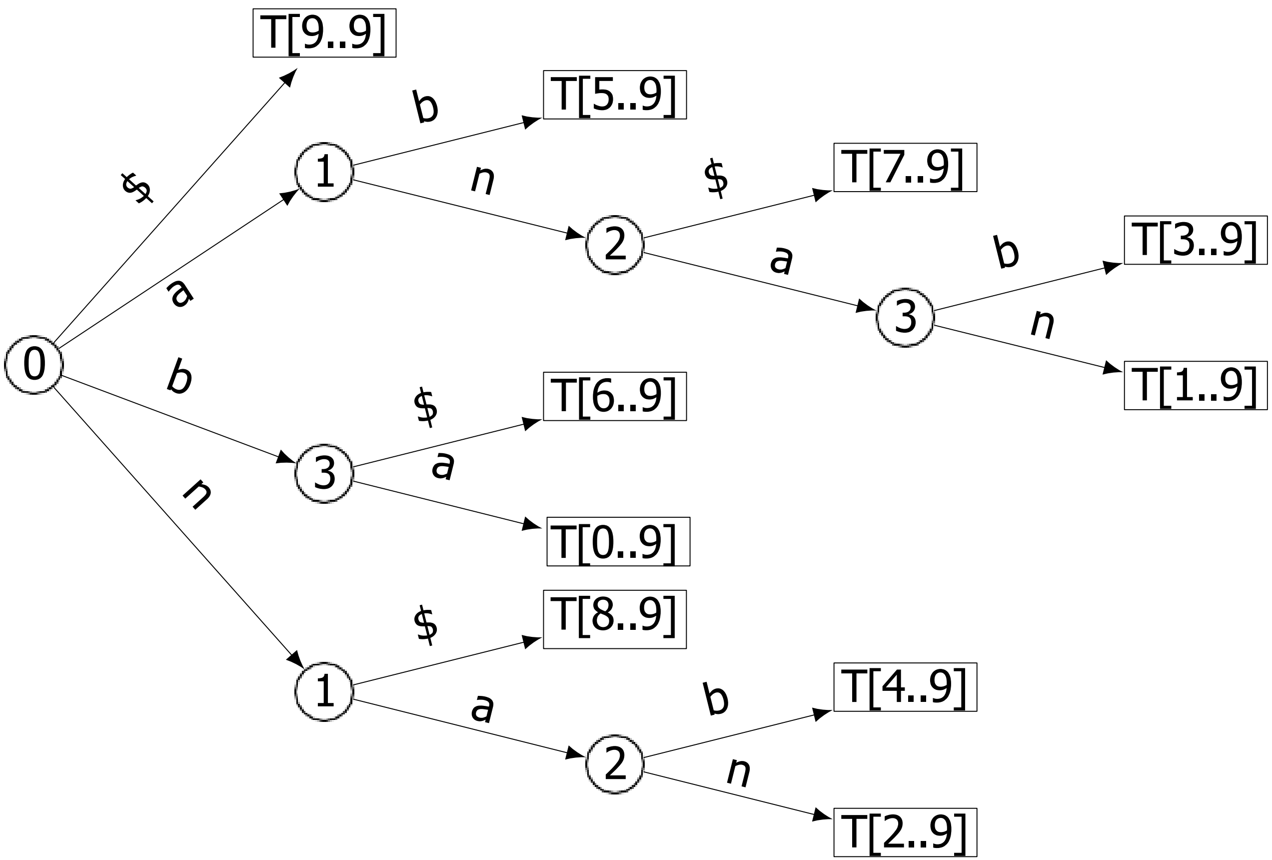


Suffix tree

T =

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

- Compress trie of suffixes to get suffix tree



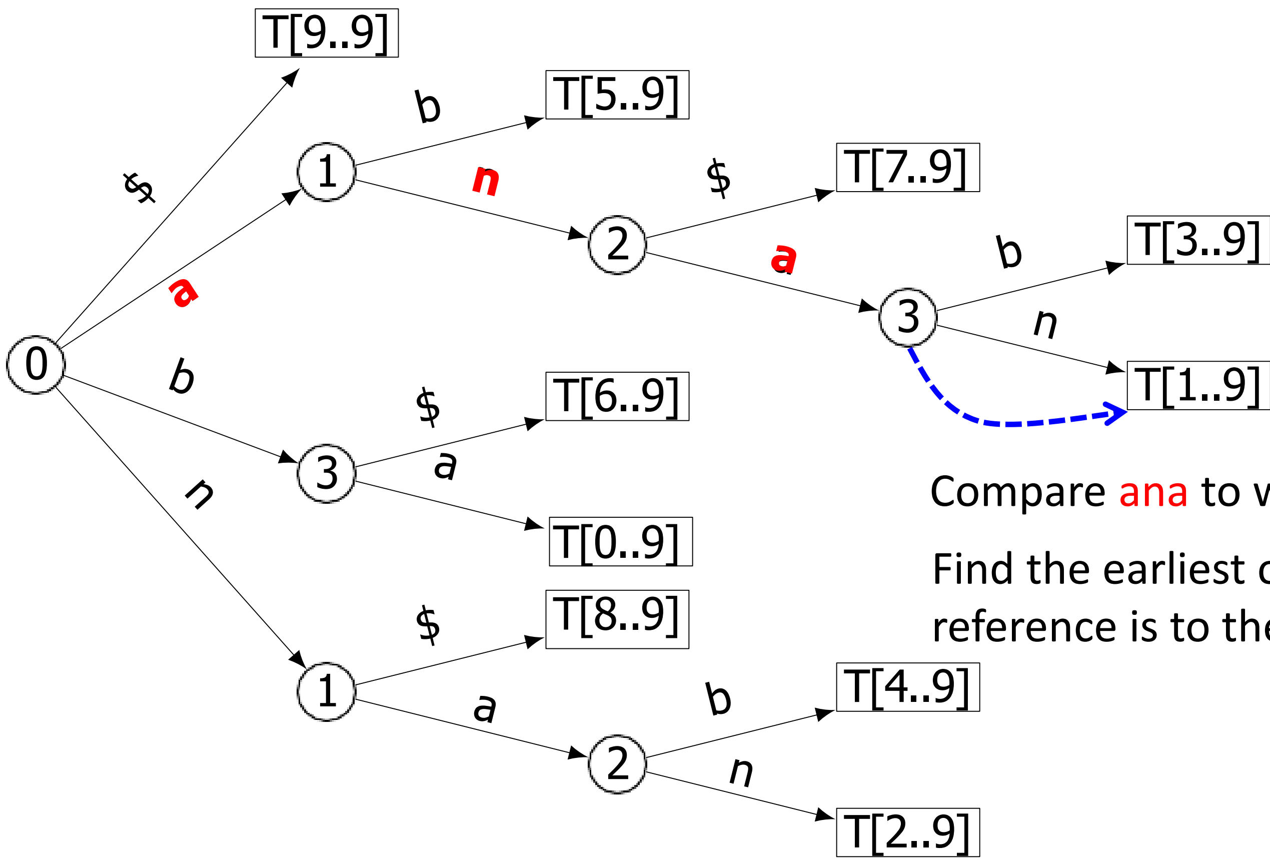
Suffix Tree Search

T =

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

found!

- If P occurs in the text, it is a prefix of one (or more) strings stored in the trie
- To search for a pattern, use prefix search on tries
- Example: search for ana



Compare ana to what is stored in T[1..3]

Find the earliest occurrence, since leaf reference is to the longest suffix

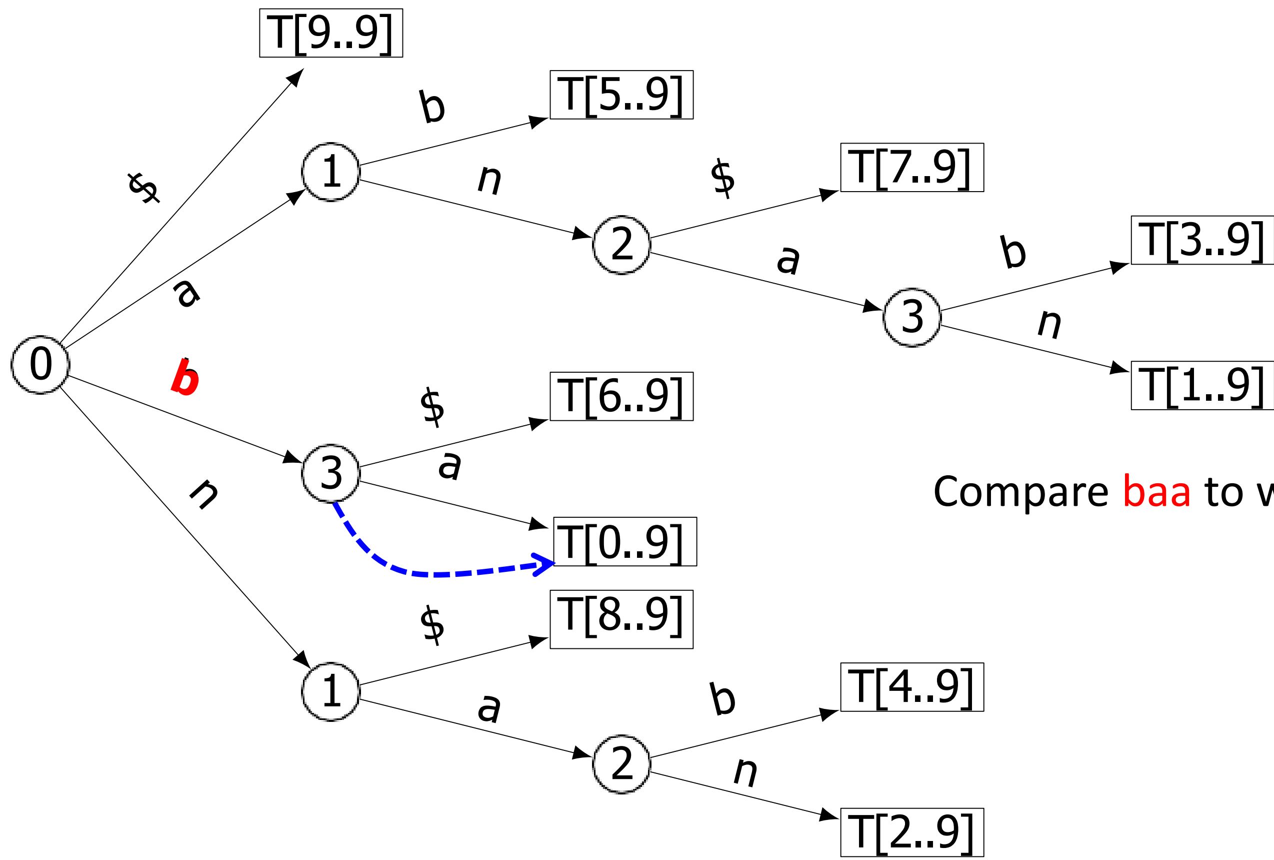
Suffix Tree Search

T =

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

not found!

- If P occurs in the text, it is a prefix of one (or more) strings stored in the trie
- To search for a pattern, use prefix search on tries
- Example: search for baa



Compare baa to what is stored in T[0..2]

Building Suffix Tree

- Building
  - text T has n characters and $n + 1$ suffixes
  - can build suffix tree by inserting each suffix of T into compressed trie
    - $\Theta(|\Sigma|n^2)$ time
  - there is a way to build a suffix tree of T in $\Theta(|\Sigma|n)$ time
    - beyond the course scope
- Pattern Matching
  - *prefix-search* for P in compressed trie
  - run-time is
    - $O(|\Sigma|m)$, assuming a node stores children in a linked list
    - $O(m)$, assuming a node stores children in an array
- Summary
  - theoretically good, but construction is slow or complicated and lots of space-overhead
  - rarely used in practice

Outline

- String Matching
  - Introduction
  - Karp-Rabin Algorithm
  - 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	9
T =	b	a	n	a	n	a	b	a	n	$

	0	1	2	3	4	5	6	7	8	9
Suffix Array =	9	5	7	3	1	6	0	8	4	2

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

sort lexicographically →

j	$A^s[j]$	
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 Construction

T =

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

- Easy to construct using MSD-Radix-Sort (pad with any character to get the same length)

	round 1	**round 2**	…	**round n**
bananaban$	$*********	$*********		$*********
ananaban$*	ananaban$	aban$****		aban$****
nanaban$**	anaban$***	ananaban$		an$*******
anaban$***	aban$*****	anaban$**		anaban$***
naban$****	an$*******	an$******		ananaban$*
aban$*****	bananaban$	bananaban$		ban$******
ban$******	ban$******	ban$******		bananaban$
an$*******	nanaban$**	nanaban$**		n$********
n$********	naban$****	naban$****		naban$****
$*********	n$********	n$********		nanaban$**

- Fast in practice, suffixes are unlikely to share many leading characters
- But worst case run-time is $\Theta(n^2)$
  - recursion depth is n, $\Theta(n)$ time at each recursion depth, example: $T = aa\ldots.a$
  - $\Theta(|\Sigma|n^2)$ if accounting for alphabet size

Suffix Array Construction

- Idea: we do not need n rounds
  - $\Theta(\log n)$ rounds enough → $\Theta(n \log n)$ run time
    - $\Theta((n + |\Sigma|) \log n)$ if accounting for alphabet size
- Construction-algorithm
  - MSD-radix sort plus some bookkeeping
    - needs only one extra array
    - easy to implement
  - details are covered in an algorithms course

Pattern Matching in Suffix Arrays

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

P = ban

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

ban > **ana**

	j	$A^s[j]$	
$l \rightarrow$	0	9	$
	1	5	aban$
	2	7	an$
	3	3	anaban$
$v \rightarrow$	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

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

ban < **n**

	j	$A^s[j]$	
	0	9	$
	1	5	aban$
	2	7	an$
	3	3	anaban$
	4	1	ananaban$
$l \rightarrow$	5	6	ban$
	6	0	bananaban$
$v \rightarrow$	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

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

found!

	j	$A^s[j]$	
	0	9	$
	1	5	aban$
	2	7	an$
	3	3	anaban$
	4	1	ananaban$
$v = l \rightarrow$	5	6	ban$
$r \rightarrow$	6	0	bananaban$
	7	8	n$
	8	4	naban$
	9	2	nanaban$

- $\Theta(\log n)$ comparisons
- Each comparison is *strcmp* $(P, T, A^s[v], m)$
- $\Theta(m)$ per comparison $\Longrightarrow$ run-time is $\Theta(m \log n)$

Pattern Matching in Suffix Arrays

SuffixArray-Search(T, P, A^s)

A^s: suffix array of T, P: pattern

$l \leftarrow 0, r \leftarrow$ last index of A^s

while $l \leq r$

$v \leftarrow \left\lfloor \frac{l+r}{2} \right\rfloor$

$i \leftarrow A^s[v]$

// assume *strcmp* handles out of bounds suitably

$s \leftarrow$ *strcmp*(T, P, i, m)

if $(s < 0)$ **do** $l \leftarrow v + 1$

else $(s > 0)$ **do** $r \leftarrow v - 1$

else return 'found at guess i'

return FAIL

- Does not always find the leftmost occurrence
- Can find the leftmost occurrence and reduce runtime to $O(m + \log n)$ with further pre-computations

Outline

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

String Matching Conclusion

	Brute Force	KR	BM	KMP	Suffix Trees	Suffix Array
preproc.	—	$O(m)$	$O(m + \|\sum\|)$	$O(m)$	$O(\|\sum\|n^2)$ $\rightarrow O(\|\sum\|n)$	$O(nlogn)$ $\rightarrow O(n)$
search time (preproc excluded)	$O(nm)$	$O(n + m)$ expected	$O(n + \|\sum\|)$ with good suffix often better	$O(n)$	$O(m(\|\sum\|)$	$O(mlogn)$ $\rightarrow O(m + logn)$
extra space	—	$O(1)$	$O(m + \|\sum\|)$	$O(m)$	$O(n)$	$O(n)$

- Algorithms stop once they found one occurrence
- Most of them can be adapted to find *all* occurrences within the same worst-case run-time