CS 240 – Data Structures and Data Management

Module 9: String Matching

T. Biedl E. Schost O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2021

Outline

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

Pattern Matching Definitions [1]

- 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
- Return the first occurrence of P in T, that is return smallest i such that

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

- 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

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

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

More Definitions [2]

antidisestablishmentarianism

- Substring $T[i\ldots j]$ $0 \le i \le j < n$ is a string consisting of characters $T[i], T[i+1], \ldots, T[j]$
  - length is $j - i + 1$
- Prefix of T is a substring $T[0\ldots i]$ of T for some $0 \le i < n$
- Suffix of T is a substring $T[i\ldots n-1]$ of T for some $0 \le i \le n-1$

General Idea of Algorithms

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

abbbababbab

abba

a**bbba**babbab

abba

abbbab**abba**b

abba

check at $j = 0$

check at $j = 1$

abbbaba**bbab**

abba

- Pattern matching algorithms consist of guesses and checks
  - a **guess** or **shift** 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

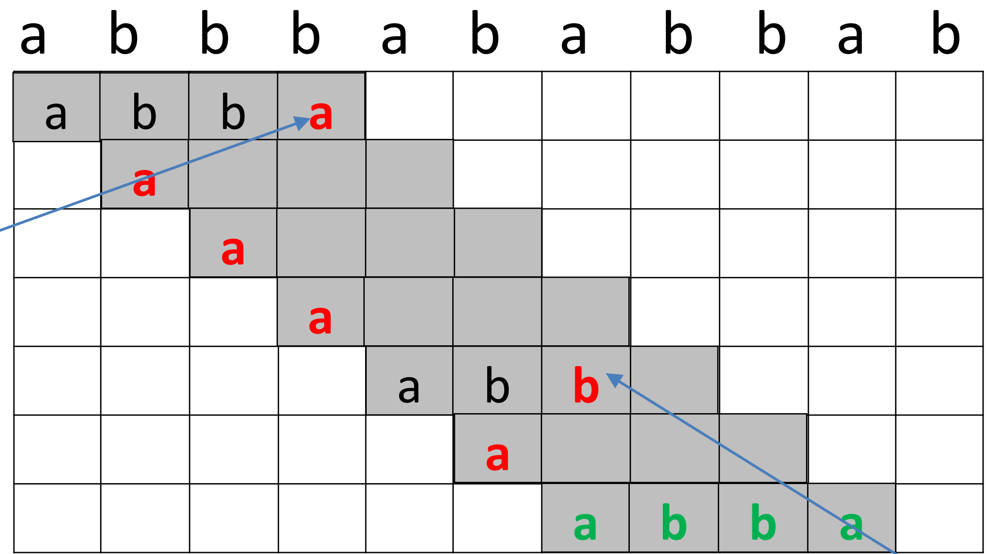
Diagrams for Matching

- 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 Example

Example: T = abbbababbab, P = abba



guess $i = 0$,
check $j \ = 3$

guess $i = 4$,
check $j \ = 2$

- Worst possible input
  - $P \ = \underbrace{a \dots a}_{m-1 \text{ times}} b, T \ = \underbrace{aaaaaaaa \dots aaaaaaa}_{n \text{ times}}$
- Have to perform $(n - m + 1)m$ checks, which is $\Theta(nm)$ running time
  - very inefficient if m is large, i.e. $m \ = \ n/2$

Brute-force Algorithm

- Idea: Check 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*$(T\ [i\ \ldots\ i+m-1],\ P) = 0$
            return "found at guess i"
    return FAIL

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

strcmp$(T\ [i\ \ldots\ i+m-1], P[0...m-1])$
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

How to improve?

- More sophisticated algorithms
  - Extra preprocessing on pattern P
    - **Karp-Rabin**
    - **Boyer-Moore**
    - **KMP**
    - Eliminate guesses based on completed matches and mismatches
  - Do extra preprocessing on the text T
    - **Suffix-trees**
    - **Suffix-arrays**
    - Create a data structure to find matches easily

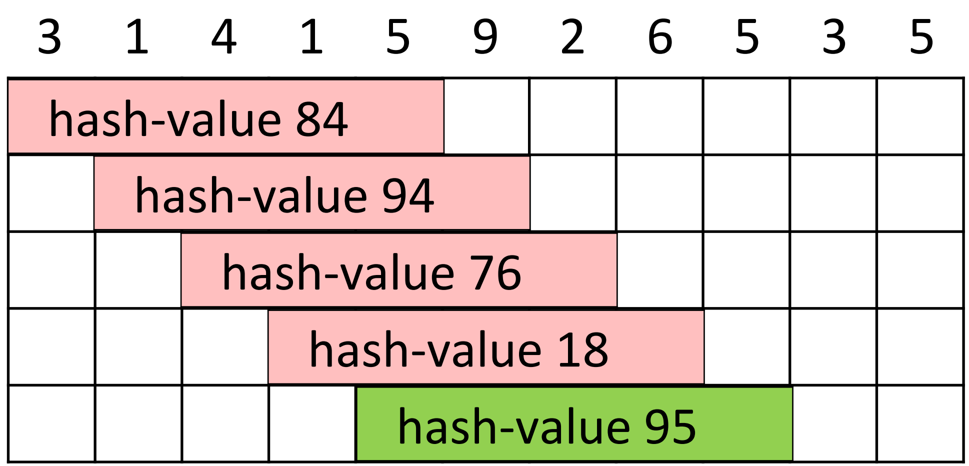
Outline

- 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 hashing to eliminate guesses faster
  - compute hash function for each guess, compare with pattern hash
    - if values are unequal, then the guess cannot be an occurrence
    - if values are equal, verify that pattern actually matches text
      - equal hash value does not guarantee equal keys
      - although if hash function is good, most likely keys are equal
      - $O(m)$ time to verify, but happens rarely, and most likely only for true match
  - example P = 5 9 2 6 5, T = 3 1 4 1 5 9 2 6 5 3 5
    - standard hash function: flattening + modular (radix $R = 10$):

$$h(P) = 59265 \; mod \; 97 = 95$$



$h(31415) = 84$

$h(14159) = 94$

$h(41592) = 76$

$h(15926) = 18$

$h(59265) = 95$

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 $n - m$

$h_T \leftarrow h(T\,[i...i+m-1])$

if $h_T = h_P$

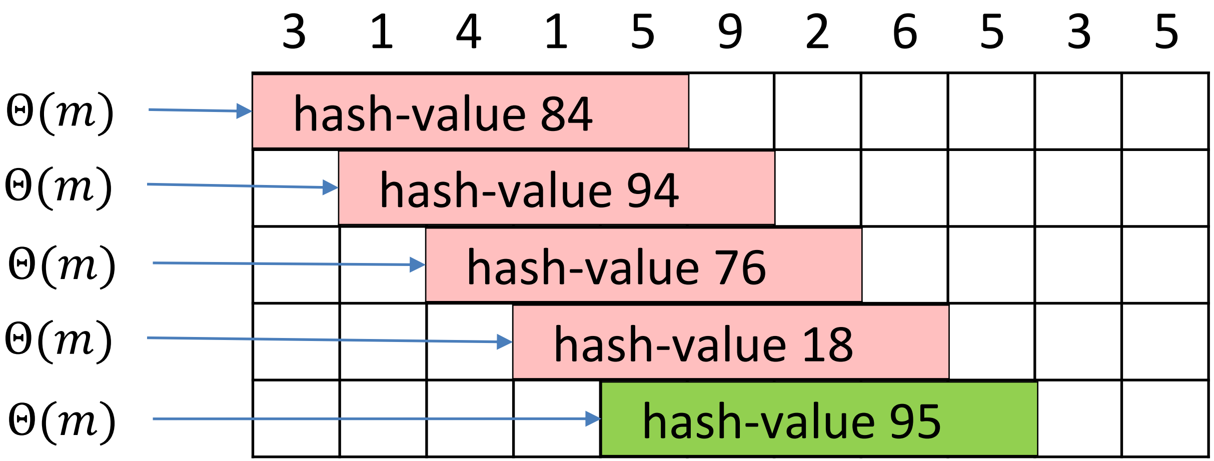
if *strcmp*$(T\,[i\,...\,i+m-1], P) = 0$

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 shift, $\Theta(m)$ time to compute hash value
  - worse than brute-force,
  - brute force can use less than $\Theta(m)$ per shift, it stops at the first mismatched character
- $n - m + 1$ shifts in text to check
- total time is $\Theta(mn)$ if pattern not in text

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 $n - m$

$h_T \leftarrow h(T\,[i...i+m-1])$

if $h_T = h_P$

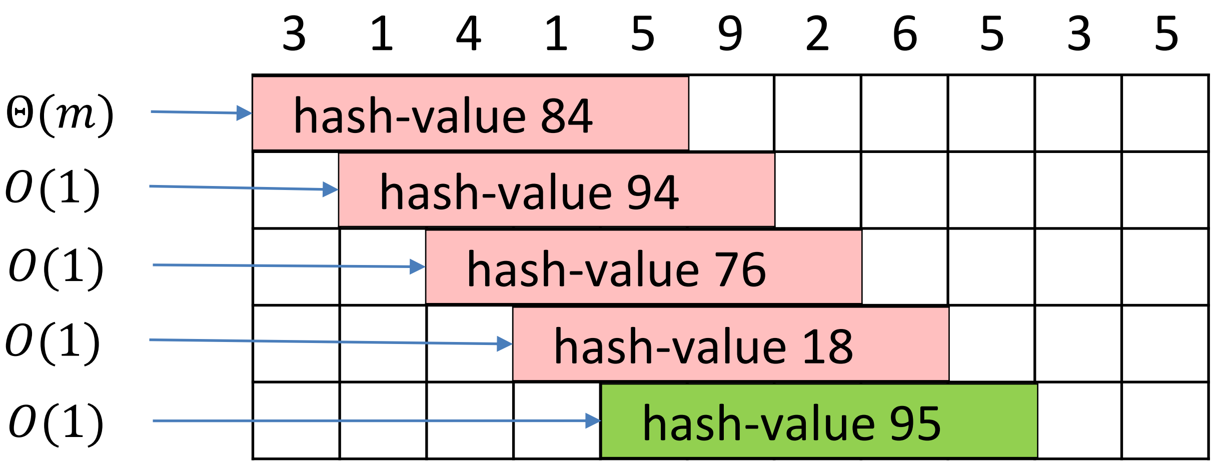
if *strcmp*$(T\,[i\,...\,i+m-1], P) = 0$

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
- $h(T[i...i+m-1])$ depends on m characters
  - naive computation takes $\Theta(m)$ time per guess
- Running time is $\Theta(mn)$ if P not in T
- How can we improve this?

Karp-Rabin Fingerprint Algorithm: Idea



- Idea: compute next hash from previous one in $O(1)$ time
- $n - m + 1$ shifts in text to check
- $\Theta(m)$ to compute the first hash value
- $O(1)$ to compute all other hash values
- $\Theta(n + m)$ expected 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
    - assuming a good hash function
        - if hash values are equal, pattern most likely matches

Karp-Rabin Fingerprint Algorithm – Fast Rehash

- Hashes are called **fingerprints**
- Insight: can update a fingerprint from previous fingerprint in constant time
  - $O(1)$ time per hash, except first one
- **Example**

T = 4 1 5 9 2 6 5 3 5, P = 5 9 2 6 5

- At the start of the algorithm, compute
  - $h(41592) = 41592 \ mod \ 97 = 76$
    - the first hash (fingerprint), $\Theta(m)$ time
  - $10000 \ mod \ 97 = 9$, precomputed one time, $\Theta(m)$ time
- How to compute $15926 \ mod \ 97$ from $41592 \ mod \ 97$?
  - to get from 41592 to 15926, need to 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

- Hashes are called **fingerprints**
- Insight: can update a fingerprint from previous fingerprint in constant time
  - $O(1)$ time per hash, except first one
- **Example**

T = 4 1 5 9 2 6 5 3 5, P = 5 9 2 6 5

- At the start of the algorithm, compute
  - $h(41592) = 41592 \ mod \ 97 = 76$
    - the first hash (fingerprint), $\Theta(m)$ time
  - $10000 \ mod \ 97 = 9$, precomputed one time, $\Theta(m)$ time
- How to compute $15926 \ mod \ 97$ from $41592 \ mod \ 97$?

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

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

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

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

Karp-Rabin Fingerprint Algorithm – Conclusion

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

$M \leftarrow$ suitable prime number
$h_P \leftarrow h(P[0 \ldots m-1)])$
$h_T \leftarrow h(T[0..m-1)])$
$s \leftarrow 10^{m-1} \, mod \, M$
for $i \leftarrow 0$ to $n-m$
  if $h_T = h_P$
    if *strcmp*$(T[i \ldots i+m-1], \; P) = 0$
      return "found at guess i"
  if $i < n-m$ // compute hash-value for next guess
    $h_T \leftarrow \big((h_T - T[i] \cdot s) \cdot 10 + T[i+m]\big) mod \, M$
return FAIL

- Choose "table size" M at random to be a large prime
- Expected running time is $O(m+n)$
- $\Theta(mn)$ worst-case, but this is (unbelievably) unlikely

Outline

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

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
    - check if $T[i] = P[j]$
      - note brute force checks if $T[i+j] = P[j]$, different usage of i
- Begin matching with $i = 0,\ j = 0$
- If $T[i] \neq P[j]$ and $j = 0$, shift pattern by 1, the same action as in brute-force
  - $i = i + 1$
  - j is unchanged

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
  - $i = i + 1$
  - $j = j + 1$
- 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

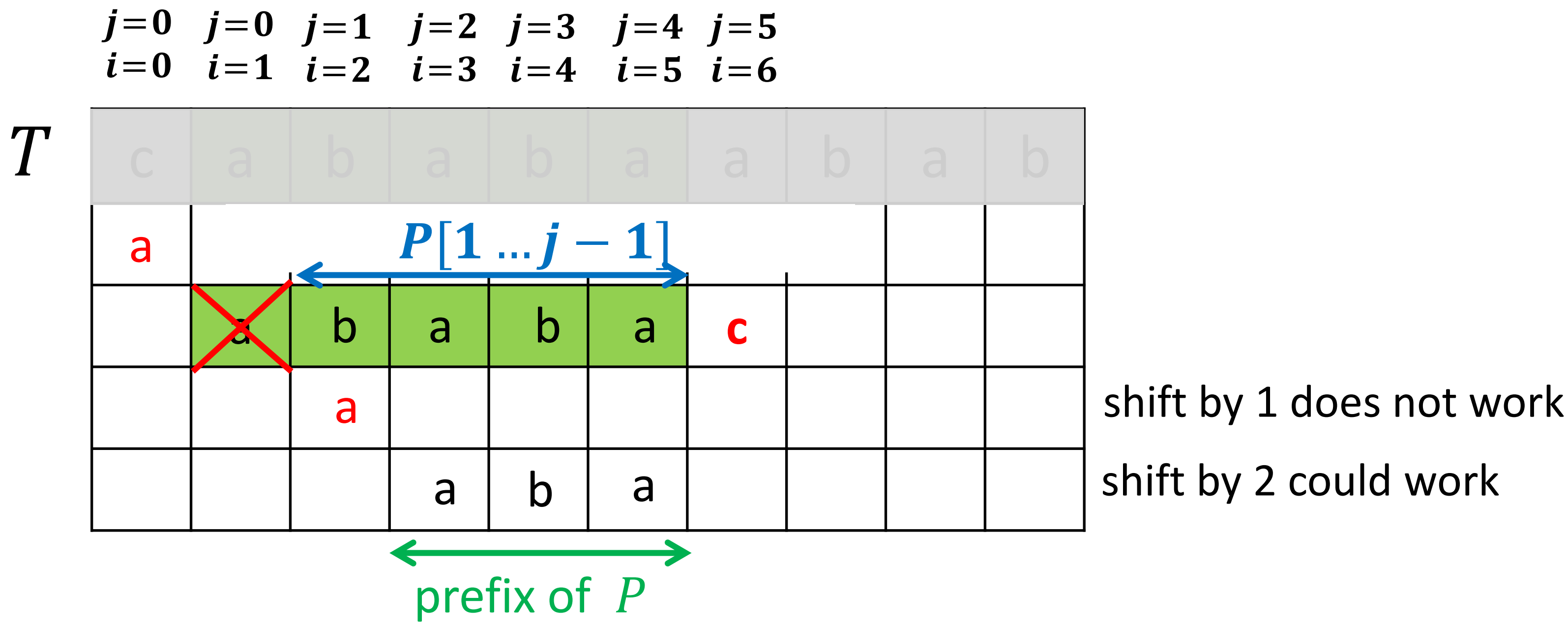
$P = \ ababaca$

	$j=0$ $i=0$	$j=0$ $i=1$	$j=1$ $i=2$	$j=2$ $i=3$	$j=3$ $i=4$	$j=4$ $i=5$	$j=5$ $i=6$				
T	c	a	b	a	b	a	a	b	a	b	
	a										
		a	b	a	b	a	c				
			a								shift by 1 does not work
				a	b	a					shift by 2 could work

- 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
  - can precompute the shift before matching even begins
- If failure at $j = 5$, shift pattern by 2 **and** start matching with $j = 3$
  - equivalently: i stays the same, new $j = 3$
  - skipped one shift, and also 3 character checks at the next shift

Knuth-Morris-Pratt Motivation

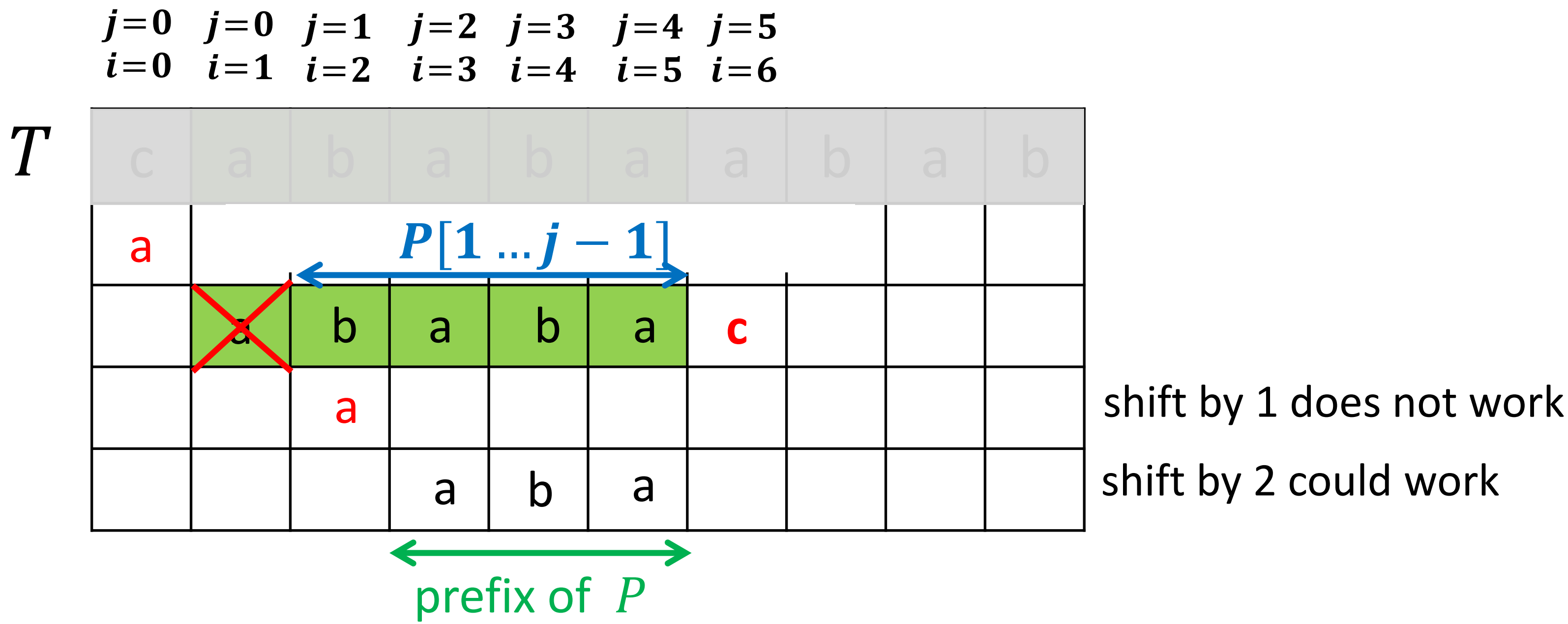
$P = ababaca$



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

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
- Brief rule for determining new j
  - find longest suffix of $P[1 \ldots j-1]$ which is also prefix of P
  - call a suffix valid if it is a prefix of P
  - new $j =$ the 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 $P[1 \ldots j-1]$
- Computed previously for $j = 5$, but need to compute for all j
- Store this information in array $F[0...m-1]$, called failure-function
  - $F[j]$ is length of the longest valid suffix of $P[1...j]$
  - if failure at pattern index $j > 0$, new $j = F[j-1]$
- $P = ababaca$

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

- $j = 0$
  - $P[1 \ldots 0] =$ “”, $P = ababaca$, longest valid suffix is “”
  - note that $F[0] = 0$ 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
- $j = 3$
  - $P[1 \ldots 3] = bab$, $P = ababaca$, longest valid suffix is ab

KMP Failure Array Computation: Slow

- Store this information in array $F[0...m-1]$, called failure-function
  - $F[j]$ is length of the longest valid suffix of $P[1...j]$
  - if failure at pattern index $j > 0$, new $j = F[j-1]$

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

- $j = 4$
  - $P[1 \dots 4] = baba$, $P = ababaca$, 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 of failure array F is $O(m^3)$ time

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

String matching with KMP: Example

- $T = cabababcabababaca, P = ababaca$

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

- $T = cabababcababaca,\ P = ababaca$

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

	$j=0$	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=3$ $\cancel{j=5}$	$j=0$ $\cancel{j=2}$ $\cancel{j=4}$	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$	$i=8$	$i=9$	$i=10$	$i=11$	$i=12$	$i=13$	$i=14$	
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(T, P)
    F ← failureArray(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"
              // location i in T is the end of matched P in text
            else // rule 1
                i ← i + 1
                j ← j + 1
        else // P[j] ≠ T[i]
            if j > 0
                j ← F[j − 1]  // rule 2
            else // rule 3
                i ← i + 1
    return FAIL
```

KMP: Time Complexity, informally

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

i increases

j decreases

								$j=0$							
							$j=3$	$\cancel{j=2}$							
	$j=0$	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$\cancel{j=5}$	$\cancel{j=4}$	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$	$i=8$	$i=9$	$i=10$	$i=11$	$i=12$	$i=13$	$i=14$
T:	c	a	b	a	b	a	b	c	a	b	a	b	a	c	a
P:															

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

- For now, ignore the cost of computing failure array
- Total time = 'horizontal iterations' + 'vertical iterations'
- i can increase at most n times
- number of decreases of $j \leq$ number of increases of j $\leq n$
- $O(n)$ total iterations, more formal analysis later

KMP: Running Time, informally

F

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

i increases

j decreases

							$j=0$								
						$j=3$	~~$j=2$~~								
	$j=0$	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$ ~~$j=5$~~	~~$j=4$~~	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	
	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$	$i=8$	$i=9$	$i=10$	$i=11$	$i=12$	$i=13$	$i=14$
T:	c	a	b	a	b	a	b	c	a	b	a	b	a	c	a
P:															

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

- For now, ignore the cost of computing failure array
- Total time = 'horizontal iterations' + 'vertical iterations'
- i can increase at most n times
- number of decreases of $j \leq$ number of increases of j $\leq n$
- $O(n)$ total iterations, more formal analysis later

Fast Computation of F

- $P = ababaca$

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

- After processing T, the final value of j is longest suffix of T equal to prefix of P
  - or, using our terminology, the final value of j is the longest valid suffix of T
- Useful for failure array computation
  - but first, let us rename variable j as l (only for failure array computation)
  - otherwise things get confusing
    - already have j when talking about failure array

Fast Computation of F

- $P = ababaca$

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

- After processing T, the final value of l is longest suffix of T equal to prefix of P
  - or, using our terminology, the final value of l is the longest valid suffix of T
- $F[j]$ = length of the longest valid suffix of $P[1...j]$
  - need to compute $F[j]$ for $0 < j < m$
    - $F[0] = 0$, no need to compute
- Big idea

$T = P[1 \dots 1] \rightarrow$ KMP $\xrightarrow{\text{final } l}$ $F[1] = l$

$T = P[1 \dots 2] \rightarrow$ KMP $\xrightarrow{\text{final } l}$ $F[2] = l$

⋮

$T = P[1 \dots m-1] \rightarrow$ KMP $\xrightarrow{\text{final } l}$ $F[m-1] = l$

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

Fast Computation of F: Big Idea Saved

- $j = 1$

  $$T = P[1 \dots 1] \longrightarrow \boxed{\text{KMP}} \xrightarrow{\text{final } l} F[1] = l$$

  - start with $l = 0$
  - text has one letter, can reach at most $l = 1$
  - need at most $F[0]$, and already have it

- $j = 2$

  $$T = P[1 \dots 2] \longrightarrow \boxed{\text{KMP}} \xrightarrow{\text{final } l} F[2] = l$$

  - start with $l = 0$
  - text has two letters, can reach at most $l = 2$
  - need at most $F[0], F[1]$, and already have it

$\vdots$

- $j = m - 1$

  $$T = P[1 \dots m-1] \longrightarrow \boxed{\text{KMP}} \xrightarrow{\text{final } l} F[m-1] = l$$

  - start with $l = 0$
  - text has $m - 1$ letters, can reach at most $l = m - 1$
  - need at most $F[0], F[1], \dots, F[m-2]$, and already have it

Fast Computation of F: Big Idea Made Bigger

$T = P[1 \dots 1] \longrightarrow \boxed{\text{KMP}} \xrightarrow{\text{final } l} F[1] = l$

$T = P[1 \dots 2] \longrightarrow \boxed{\text{KMP}} \xrightarrow{\text{final } l} F[2] = l$

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

$T = P[1 \dots 3] \longrightarrow \boxed{\text{KMP}} \xrightarrow{\text{final } l} F[3] = l$

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

$\vdots$

$T = P[1 \dots m-1] \longrightarrow \boxed{\text{KMP}} \xrightarrow{\text{final } l} F[m-1] = l$

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

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

Fast Computation of F

F	0	1	2	3	4	5	6
	0						

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

Fast Computation of F

F	0	1	2	3	4	5	6
	0	0					

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

	$l=0$ $i=0$	$l=0$ $i=1$									
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:	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 $T = P[1 \dots j]$, $F[j]$ = final l
- $P = ababaca$
- $j = 3, T = P[1 \dots j] = bab$

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

	$l=0$ $i=0$	$l=0$ $i=1$	$l=1$ $i=2$	$l=2$ $i=3$	$l=3$ $i=4$						
T:	b	a	b	a							
P:	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 $T = P[1 \ldots j]$, $F[j] =$ final l
- $P = ababaca$
- $j = 5, T = P[1 \ldots j] = babac$

	$l=0$ $i=0$	$l=0$ $i=1$	$l=1$ $i=2$	$l=2$ $i=3$	$l=0$ $\cancel{l=1}$ $\cancel{l=3}$ $i=4$	$l=0$ $i=5$						
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 $T = P[1 \dots j]$, $F[j]$ = final l
- $P = ababaca$
- $j = 6, T = P[1 \dots j] = babaca$

	$l=0$, $i=0$	$l=0$, $i=1$	$l=1$, $i=2$	$l=2$, $i=3$	$l=0$, $\cancel{l=1}$, $\cancel{l=3}$, $i=4$	$l=0$, $i=5$	$l=1$, $i=6$					
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

KMP: Computing Failure Array

- Pseudocode is almost identical to KMP(T, P)
  - main difference: $F[j]$ gets both used and updated
- More formal analysis
  - consider how $2j - l$ changes in each iteration of while loop
  - one of the three case below applies
    1) j and l both increase by 1
       - $2j - l$ increases by 1
    2) l decreases $(F[l-1] < l)$
       - $2j - l$ increases by 1 or more
    1) j increases by 1
       - $2j - l$ increases by 2
  - initially $2j - l = 2 \geq 0$
  - at the end $2j - l \leq 2m$
    - $j = m, l \geq 0$
  - no more than $2m$ iterations of while loop
  - time is $\Theta(m)$

failureArray(P)
P: String of length m (pattern)

```
F[0] ← 0
j ← 1 // parsing P[1 ... j]
l ← 0
while j < m do
    if P[j] = P[l]
        l ← l + 1
        F[j] ← l
        j ← j + 1
    else if l > 0
        l ← F[l − 1]
    else
        F[j] ← 0
        j ← j + 1
```

KMP: main function runtime

$KMP(T, P)$
    $F \leftarrow failureArray\ (P)$
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $i < n$ **do**
        if $P[j] = T[i]$
            if $j = m - 1$
                return "found at guess $i - m + 1$"
            else
                $i \leftarrow i + 1$
                $j \leftarrow j + 1$
        else // $P[j] \neq T[i]$
            if $j > 0$
                $j \leftarrow F[j - 1]$
            else
                $i \leftarrow i + 1$
    return $FAIL$

- KMP main function
    - failureArray can be computed $in\ \Theta(m)$ time
    - Same analysis gives at most $2n$ iterations of while loop since $2i - j \leq 2n$
    - Running time KMP altogether: $\Theta(n + m)$

Outline

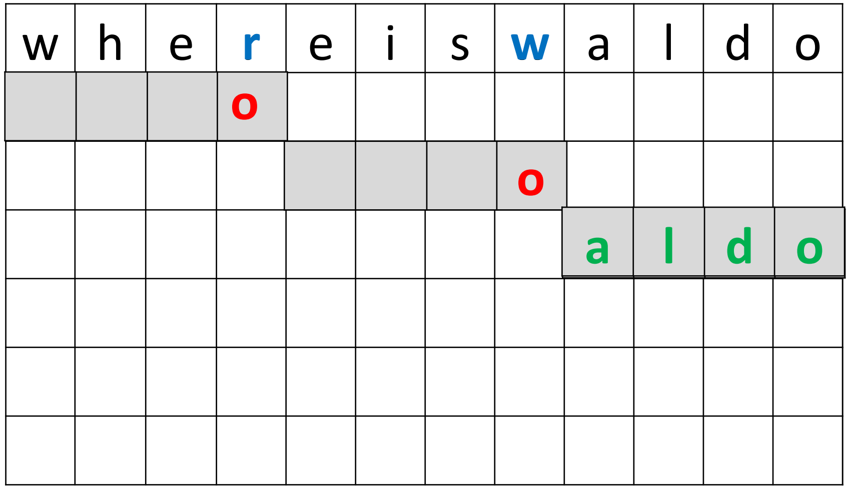
- String Matching
  - Introduction
  - Karp-Rabin Algorithm
  - Knuth-Morris-Pratt algorithm
  - Boyer-Moore Algorithm
  - Suffix Trees
  - Suffix Arrays
  - Conclusion

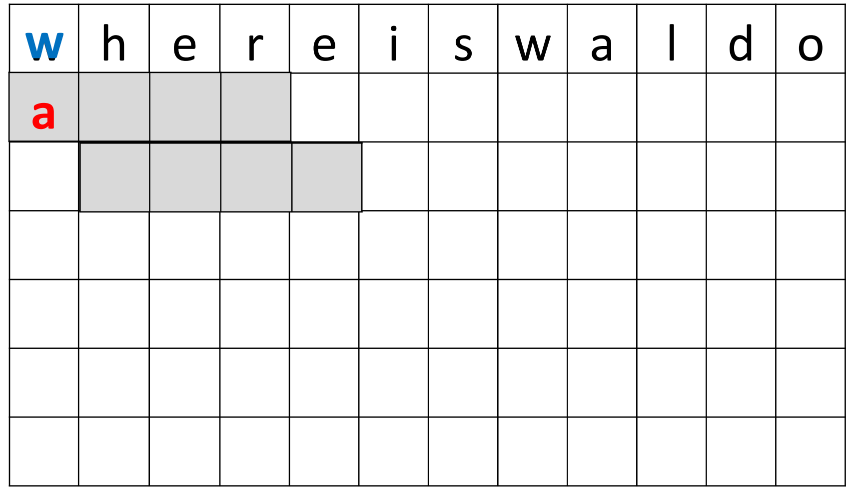
Boyer-Moore Algorithm Motivation

- Fastest pattern matching 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
- shift pattern past **r**
- **w** does not occur in P = aldo
- shift pattern past **w**
- this bad character heuristic works well with reverse searching

- **w** does not occur in P = aldo
- move pattern past **w**
- the first shift moves pattern past **w**
- no shifts are ruled out
- bad character heuristic does not work well with forward searching

Bad Character Heuristic: Full Version

- Extends to the case when mismatched text character occurs in P

T= acranapple, P = aaron

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

- Mismatched character in the text is **a**
- Find last occurrence of **a** in P
- Shift the pattern to the left until last **a** in P aligns with **a** in text

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**
- Find last occurrence of **a** in P
- Shift the pattern to the left until last **a** in P aligns with **a** in text
- This is the next possible shift of pattern to explore, skipped shifts are impossible because they do not match **a**
  - start matching at the end

Bad Character Heuristic: The Shifting Formula

T= acranapple, P = aaron

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

- Let $L(c)$ be the last occurrence of character c in P
  - $L(\mathbf{a}) = 1$ in our example
  - define $L(c) = -1$ if character c does not occur in P
- When mismatch occurs at text position i, pattern position j, update
  - $j = m - 1$
    - start matching at the end of the pattern
  - $i = i + m - 1 - L(c)$
    - bad character heuristic can be used only if $L(c) < j$

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 i such that $P[i] = c$

- Example: P = aaron
  - initialization

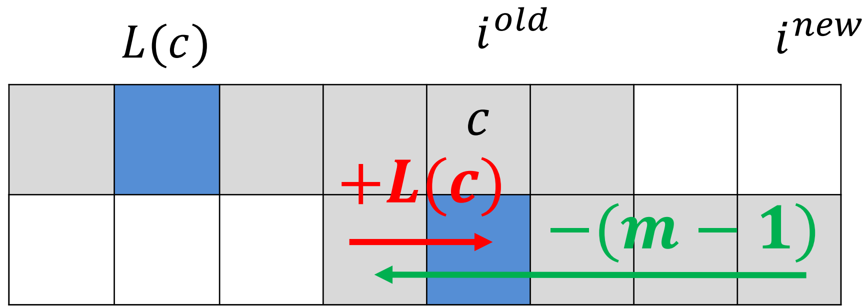
$char$	a	n	o	r	all others
$L(c)$	-1	-1	-1	-1	-1

  - computation

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

  - $O(m + |\Sigma|)$ time

Bad Character Heuristic: Shifting Formula Explained

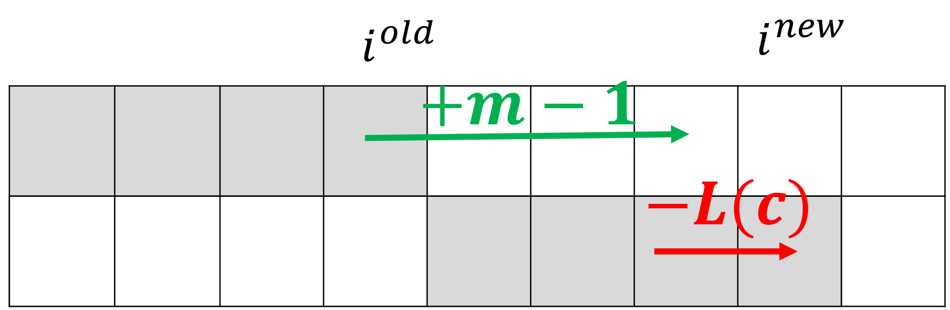


$$i^{new} - (m-1) + L(c) = i^{old}$$

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

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

- recall $L(c) = -1$ for any character c that does not occur in P
- formula also works when mismatched character c does not occur in P



Bad Character Heuristic, Last detail

- Can use bad character heuristic **only** if $L(c) < j$
- Example when $L(c) > j$

T= acraaapple, P = aaroa

$\boldsymbol{j=3}$
$\boldsymbol{i=3}$

	a	c	r	a	a	a	p	p	l	e
				o	a					

  - $i = i + m - 1 - L(c)$
    - $L(\mathbf{a}) = 4 > j = 3$
    - $i = 3 + 4 - 4 = 3$
  - shifts the pattern in the wrong direction!

- If $L(c) > j$, do brute-force step
  - $i = i - j + m$
  - $j = m - 1$
- Unified formula 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**

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** $i + 1$

else return FAIL

Good Suffix Heuristic

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

P = onobobo

				$j=3$, $i=3$					$j=6$, $i=8$										
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 shift 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, but in practice much faster
- On typical English text, Boyer-Moore looks only at ≈25% of text
- 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: peprocess 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

$$\text{establ}\underbrace{\text{ishment}}_{\text{suffix}}$$

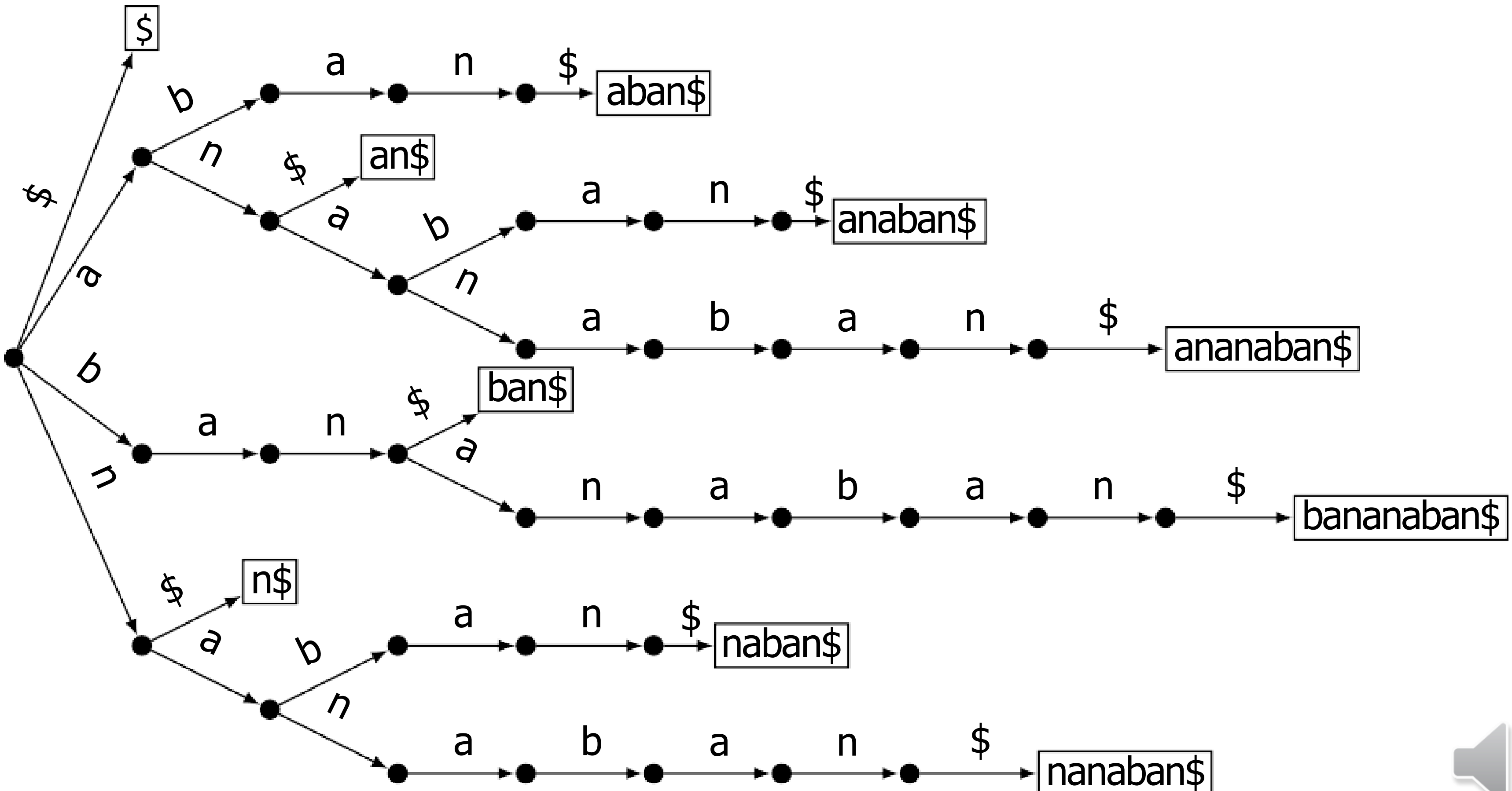
- Store all suffixes of T in a trie
  - generalize search to prefixes of stored strings
- To save space
  - use compressed trie
  - store suffixes implicitly via indices into T
- This is called a **suffix tree**

Trie of suffixes: Example

- T = bananaban

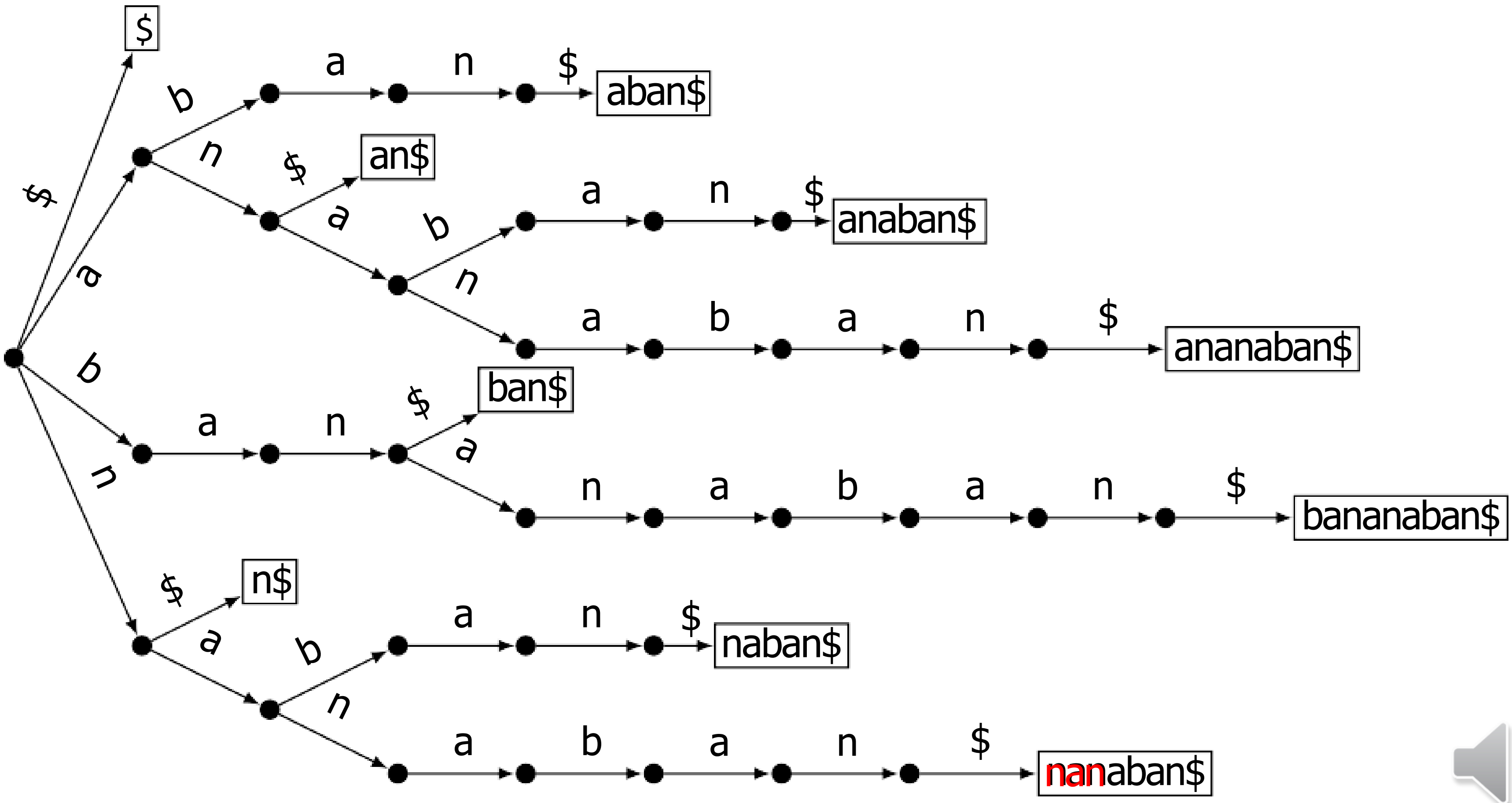
Suffixes = {bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n, Λ}

S = {bananaban\$, ananaban\$, nanaban\$, anaban\$,naban\$,..., ban\$, n\$, \$}



Trie of suffixes: Example

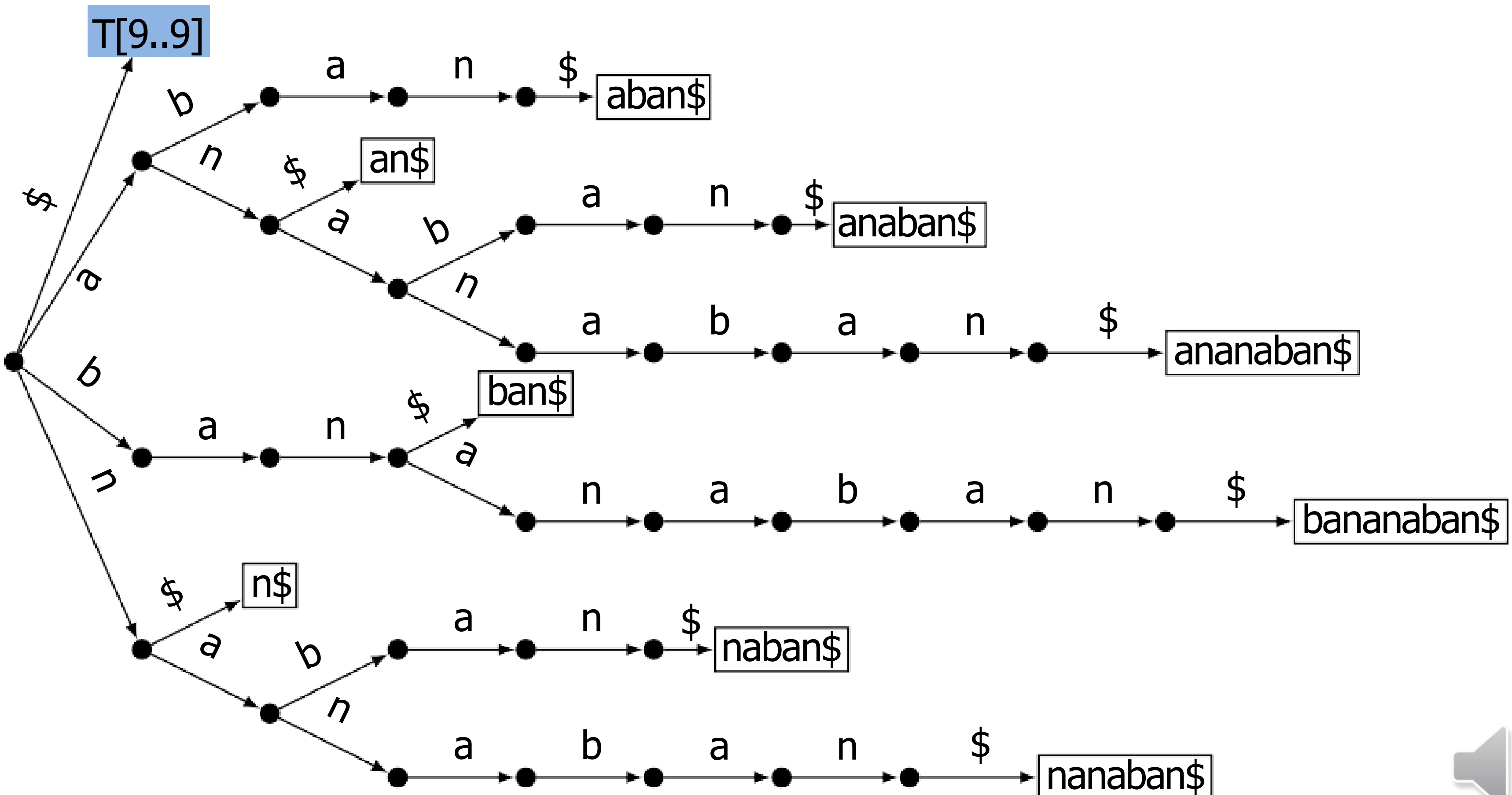
- T = bananaban
- If P occurs in the text, it is a prefix of one (or more) strings stored in the trie
- Will have to modify search in a trie to allow search for a prefix



Trie of suffixes: Example

- Store suffixes via indices

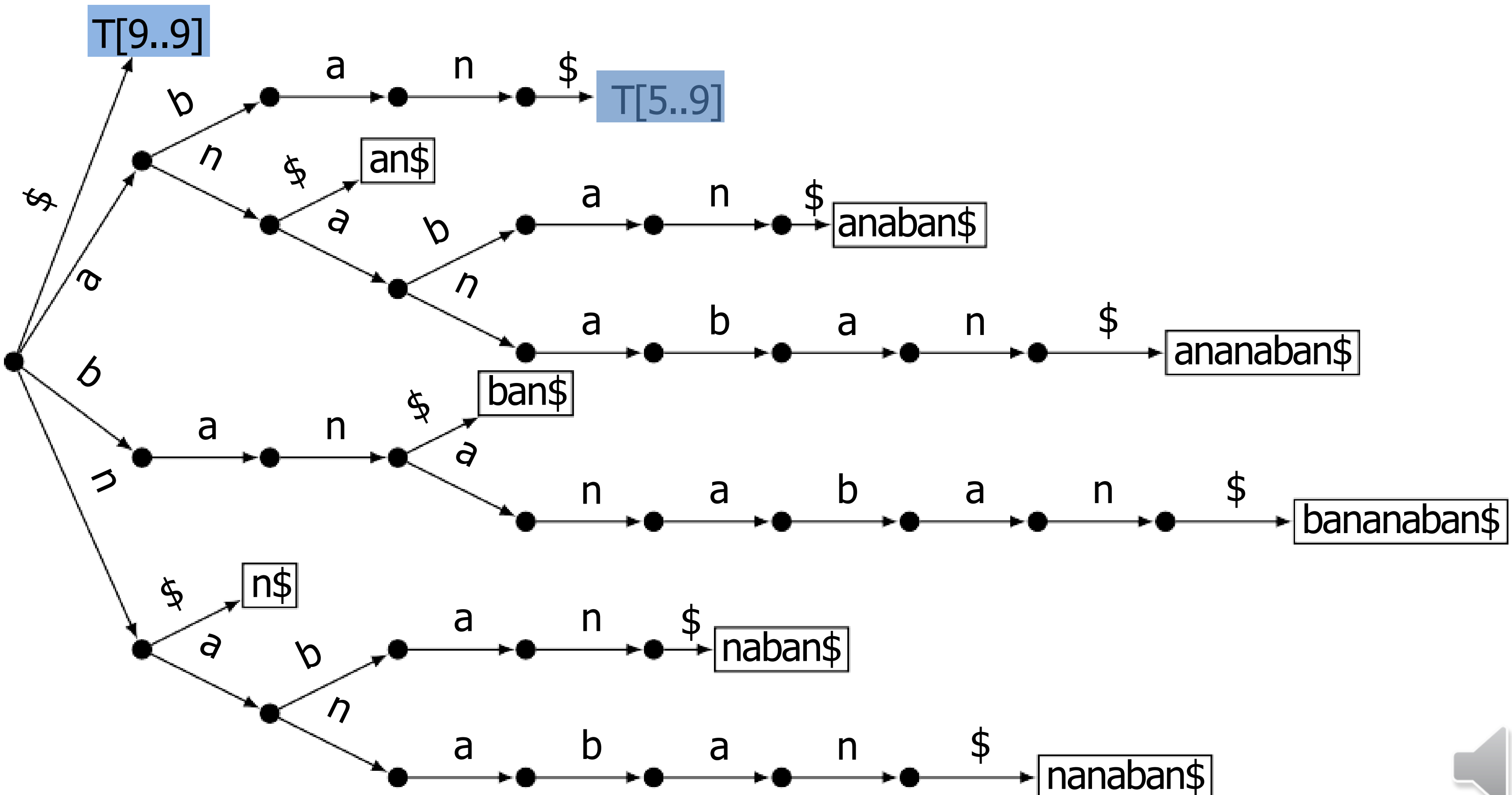
	0	1	2	3	4	5	6	7	8	9
T =	b	a	n	a	n	a	b	a	n	$



Trie of suffixes: Example

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

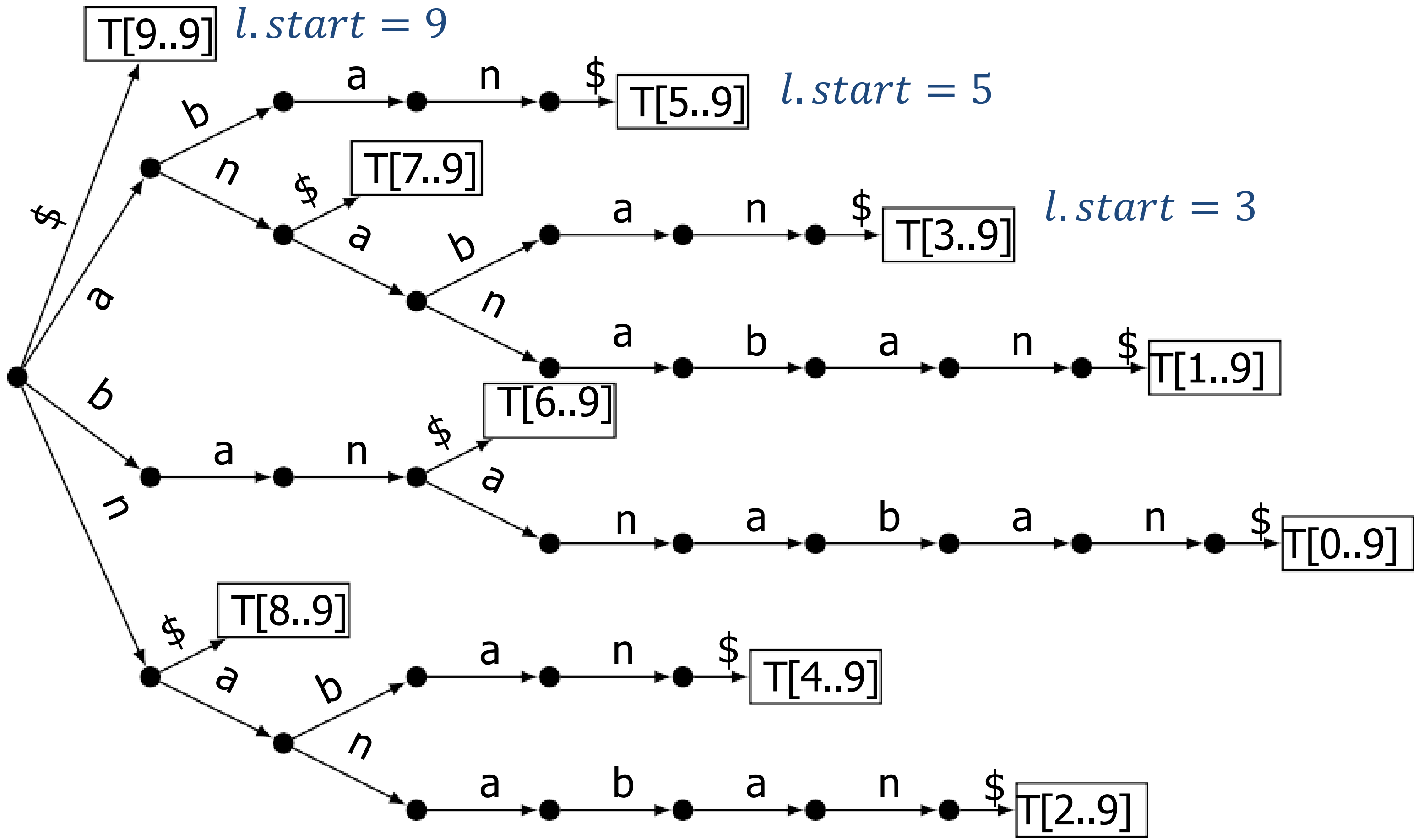
- Store suffixes via indices



Tries of suffixes

- each leaf l stores the start of its suffix in variable $l.start$

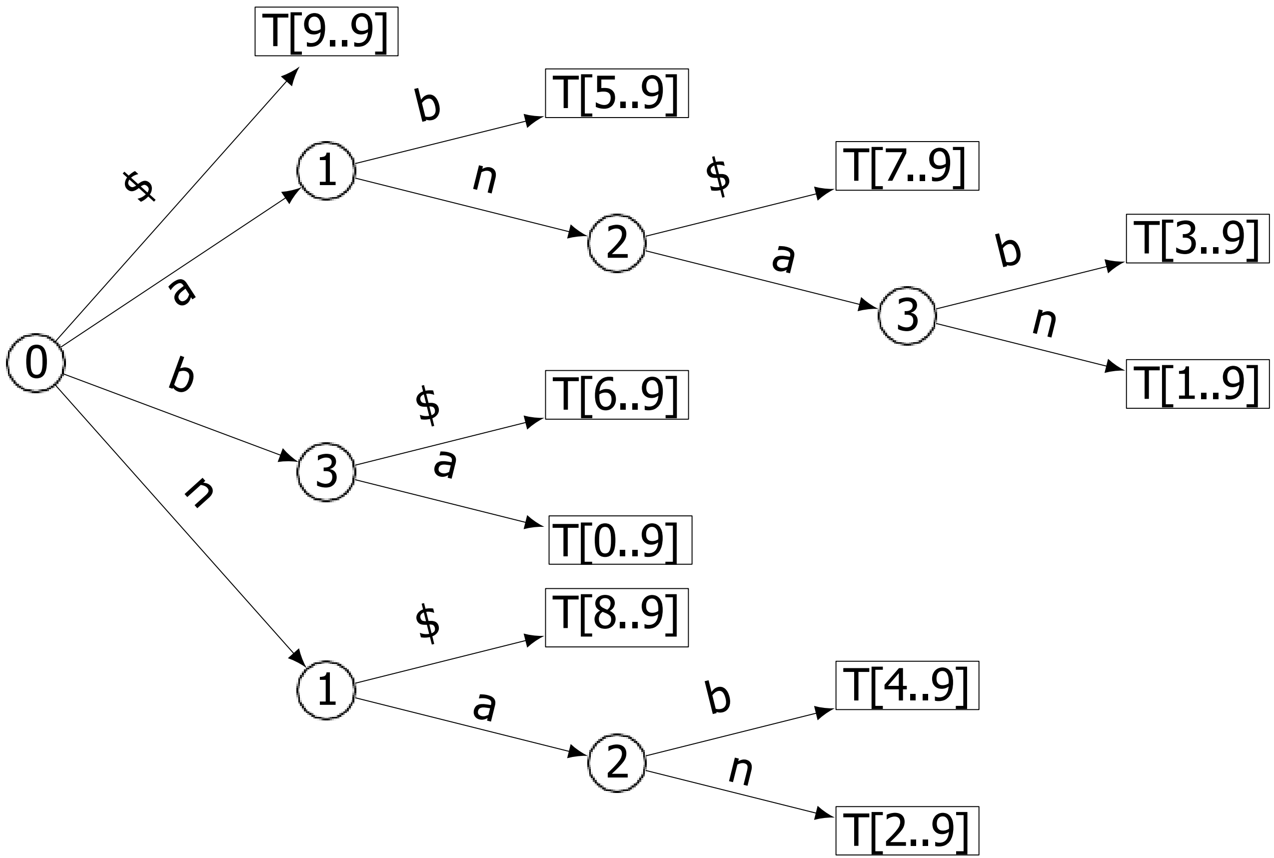
	0	1	2	3	4	5	6	7	8	9
T =	b	a	n	a	n	a	b	a	n	$



Suffix tree

- Suffix tree: compressed trie of suffixes

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



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
    - takes $\Theta(|\Sigma|n^2)$ time
  - there is a way to build a suffix tree of T in $\Theta(|\Sigma|n)$ time
    - beyond the course scope
- Pattern Matching
  - essentially search for P in compressed trie
    - some changes needed, since P may only be prefix of stored word
  - run-time is $O(|\Sigma|m)$
- 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 in T

Suffix Array Example

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

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$

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

Suffix Array Example

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

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$

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

Suffix Array Construction

	0	1	2	3	4	5	6	7	8	9
T =	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 $\Theta(n^2)$
  - n rounds of recursion, each round takes $\Theta(n)$ time (bucket sort)

Suffix Array Construction

- Idea: we do not need n rounds
  - $\Theta(\log n)$ rounds enough $\rightarrow \Theta(n \log n)$ run time
- 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

	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

	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

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

- $\Theta(\log n)$ comparisons
- Each comparison is *strcmp*$(P,\ T[A^s[v] \ldots A^s[v+m-1]])$
- $\Theta(m)$ per comparison $\Longrightarrow$ run-time is $\Theta(m \log n)$

Pattern Matching in Suffix Arrays

SuffixArray-Search$(A^s[j],\ P[0 \ldots m-1],\ T)$

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

$l \leftarrow 0, r \leftarrow n-1$

while $l < r$

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

$i \leftarrow A^s[v]$

// assume *strcmp* handles out of bounds suitably

$s \leftarrow$ *strcmp*$(T[i \ldots i+m-1], P)$

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

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

else return 'found at guess $T[i \ldots i+m-1]$'

if *strcmp*$(P, T[A^s[l], A^s[l]+m-1]])$

return 'found at guess $T[l \ldots l+m-1]$'

return FAIL

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.	—	$O(m)$	$O(m + \|\Sigma\|)$	$O(m)$	$O(\|\Sigma\| n^2)$ $\rightarrow O(\|\Sigma\| n)$	$O(nlogn)$ $\rightarrow O(n)$
search time (preproc excluded)	$O(nm)$	$O(n+m)$ expected	$O(n)$ often better	$O(n)$	$O(m)$	$O(mlogn)$
extra space	—	$O(1)$	$O(m + \|\Sigma\|)$	$O(m)$	$O(n)$	$O(n)$

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