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

Spring 2023

Outline

- String Matching
    - Introduction
    - Karp-Rabin Algorithm
    - Boyer-Moore Algorithm
    - Suffix Trees

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
- Example

$T =$ Little piglets cooked for mother pig

$P =$ pig

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

  - return smallest i such that

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

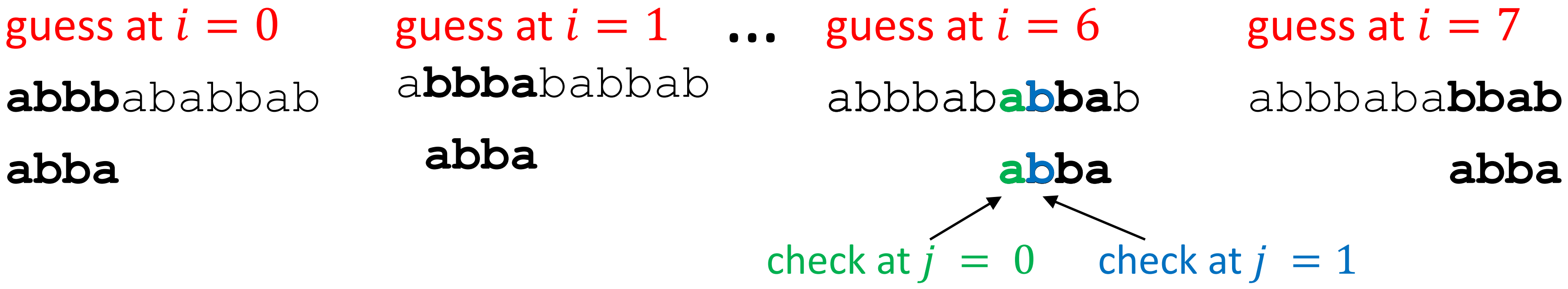
- If P does not occur in T, return FAIL
- Applications
  - information retrieval (text editors, search engines), bioinformatics, data mining

More Definitions [2]

antidisestablishmentarianism

- Substring $T[i\ldots j]$ $0 \leq i \leq j < n$ 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 $0 \leq i \leq n-1$
- Suffix of T is a substring $T[i\ldots n-1]$ of T for some $0 \leq i \leq n-1$
- With this definition, prefix and suffix are never empty strings
    - sometimes want to allow empty string prefix and suffix

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

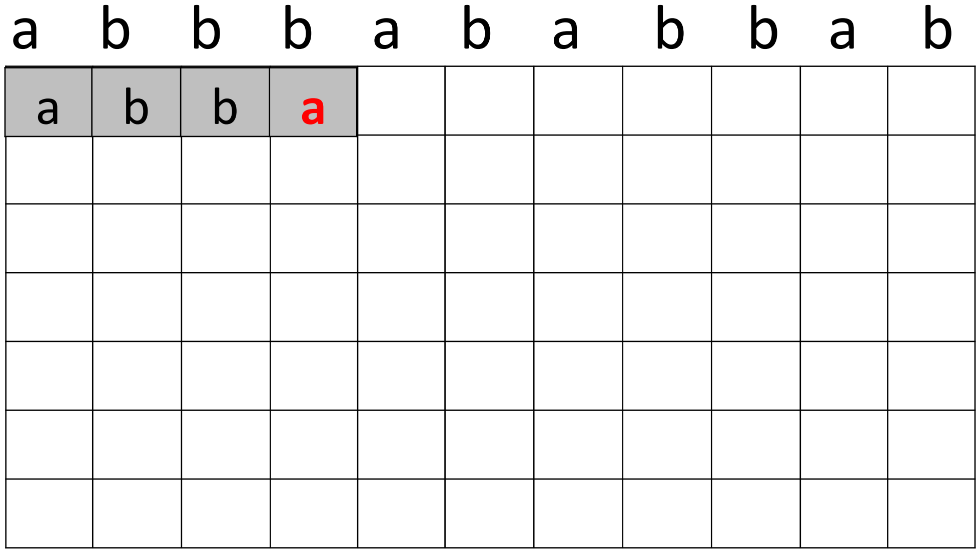
abbbaba**bbab**
abba

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

- 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

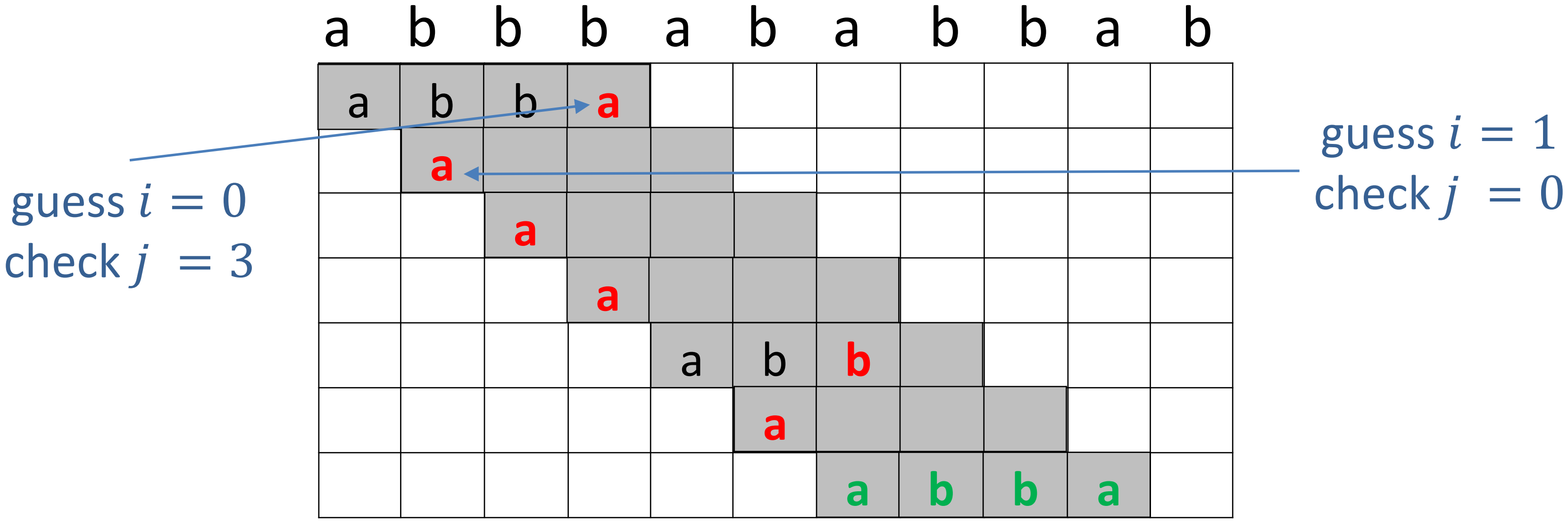
- Diagram single run of pattern matching algorithm by matrix of checks
    - each row represents a single guess



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

Brute-Force Algorithm: Example

Example: T = abbbababbab, P = abba



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

guess $i = 1$
check $j = 0$

- 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 $\Theta((n-m)m)$ runtime
    - this is $\Theta(nm)$ if $m \le n/2$
    - worst running time if $m = n/2$
        - $\Theta(n^2)$

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

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

```
strcmp($T[i \ldots i+m-1], P[0\ldots m-1]$)
for $j \leftarrow 0$ to $m-1$ do
    if $T[i+j]$ is before $P[j]$ in $\Sigma$ then return -1
    if $T[i+j]$ is after $P[j]$ in $\Sigma$ then return 1
return 0
```

How to improve?

- Extra preprocessing on pattern P
  - **Karp-Rabin**
  - **KMP**
  - **Boyer-Moore**
  - 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

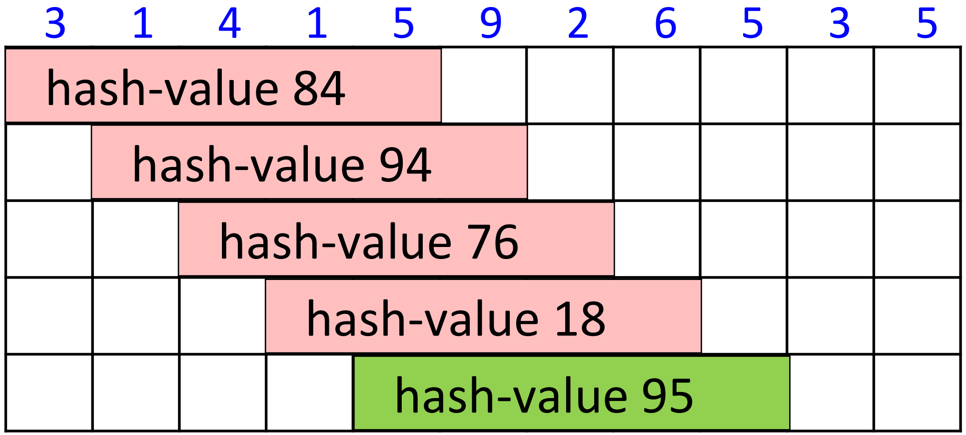
Outline

- String Matching
    - Introduction
    - Karp-Rabin Algorithm
    - Boyer-Moore Algorithm
    - Suffix Trees

Karp-Rabin Fingerprint Algorithm: Idea

- Hash functions are useful not just for hash tables!
- **Idea:** use hashing to eliminate guesses faster
    - compute hash function for each guess, compare with pattern hash
        - if values are unequal, then current guess cannot match the pattern
        - 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 $R = 10$):

$$h(59265) = (5 \cdot 10^4 + 9 \cdot 10^3 + 2 \cdot 10^2 + 6 \cdot 10^1 + 5) \; mod \; 97 = 59265 \; mod \; 97 = 95$$


3 1 4 1 5 9 2 6 5 3 5

hash-value 84

hash-value 94

hash-value 76

hash-value 18

hash-value 95

$h(31415) = 84$

$h(14159) = 94$

$h(41592) = 76$

$h(15926) = 18$

$h(59265) = 95$

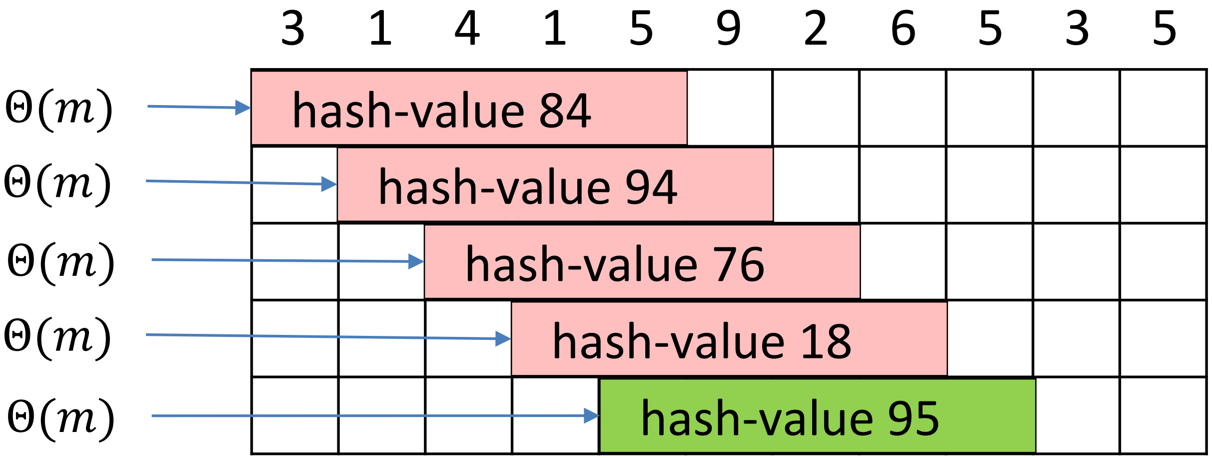
Karp-Rabin Fingerprint Algorithm – First Attempt

```
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
```

$\Theta(m)$

- 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



3 1 4 1 5 9 2 6 5 3 5

$\Theta(m)$ hash-value 84

$\Theta(m)$ hash-value 94

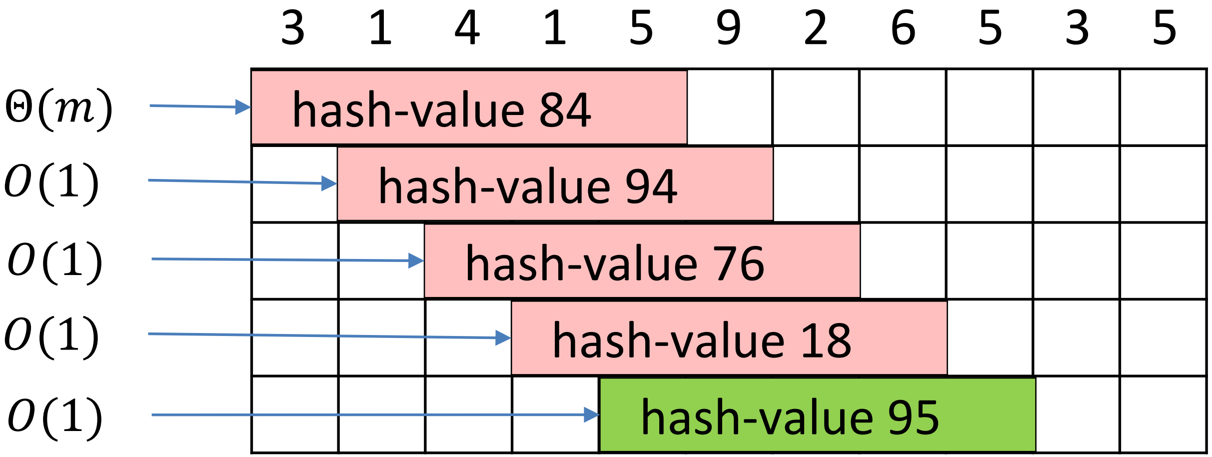
$\Theta(m)$ hash-value 76

$\Theta(m)$ hash-value 18

$\Theta(m)$ hash-value 95

- For each shift, $\Theta(m)$ time to compute hash value
    - since $h(T[i...i+m-1])$ depends on all m characters
    - worse than brute-force!
        - it is possible for brute force matching to use less than $\Theta(m)$ per shift, as 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
    - 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
    - if hash function is good, then whenever hash values are equal, pattern most likely matches the text

Karp-Rabin Fingerprint Algorithm – Fast Rehash

- For historical reasons, hashes are called **fingerprints**
- Insight: can update a fingerprint from previous fingerprint in constant time
  - $O(1)$ time to compute any hash, except first one
- **Example**

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

- Initialization of the algorithm
  1. compute first hash: $h(41592) = 41592\ mod\ 97 = 76$ [$\Theta(m)$ time]
  2. also compute $10000\ mod\ 97 = 9$
- Main loop: repeatedly compute next hash from the previous one
- Example : $\underline{1592}6\ mod\ 97$ from $4\underline{1592}\ mod\ 97$
  - 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

- Insight: can update a fingerprint from previous fingerprint in constant time
- **Example**

$T = 4\,1\,5\,9\,2\,6\,5\,3\,5, \qquad P = 5\;\,9\;\,2\;\,6\;\,5$

- Initialization of the algorithm
  1. compute first hash: $h(41592) = 41592 \; mod \; 97 = 76$ [$\Theta(m)$ time]
  2. also compute $10000 \; mod \; 97 = 9$
- Main loop: repeatedly compute next hash from the previous one
- Example: $\underline{1592}6 \; mod \; 97$ from $4\underline{1592} \; mod \; 97$

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

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

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

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

$$18 = 15926 \; mod \; 97$$

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 prime in $\{2, \ldots, mn^2\}$
- Expected running time is $O(m+n)$
- $\Theta(mn)$ worst-case, but this extremely is unlikely
- Improvement: reset M if no match at $h_T = h_P$

Outline

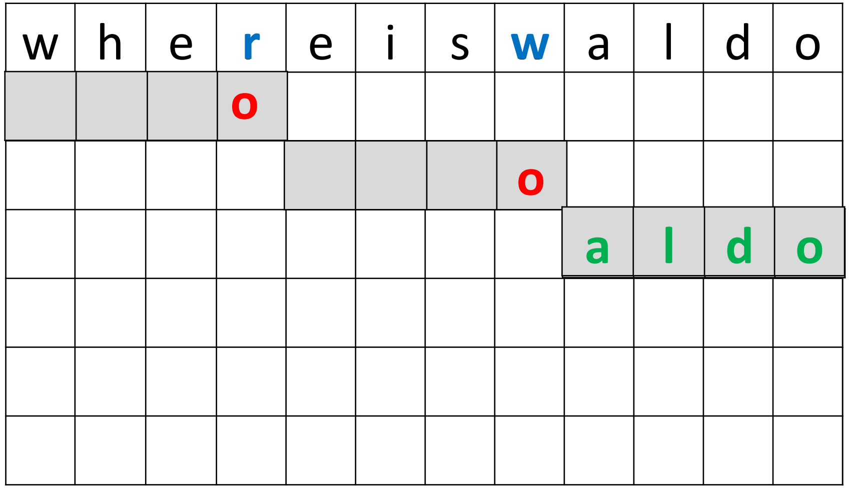
- String Matching
  - Introduction
  - Karp-Rabin Algorithm
  - Boyer-Moore Algorithm
  - Suffix Trees

Boyer-Moore Algorithm Motivation

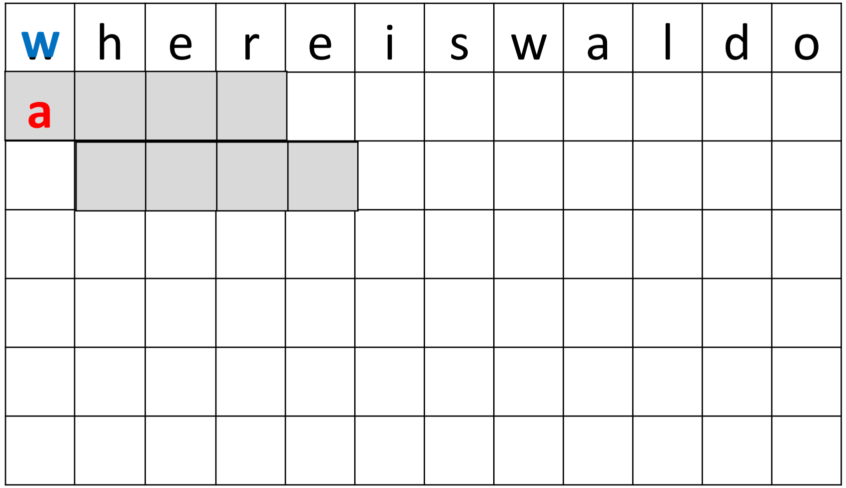
- Fastest pattern matching in practice on English Text
- Important components
  - Reverse-order searching
    - compare P with a guess moving *backwards*
- When a mismatch occurs choose the better option among the two below
  1. Bad character heuristic
     - eliminate shifts based on mismatched character 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



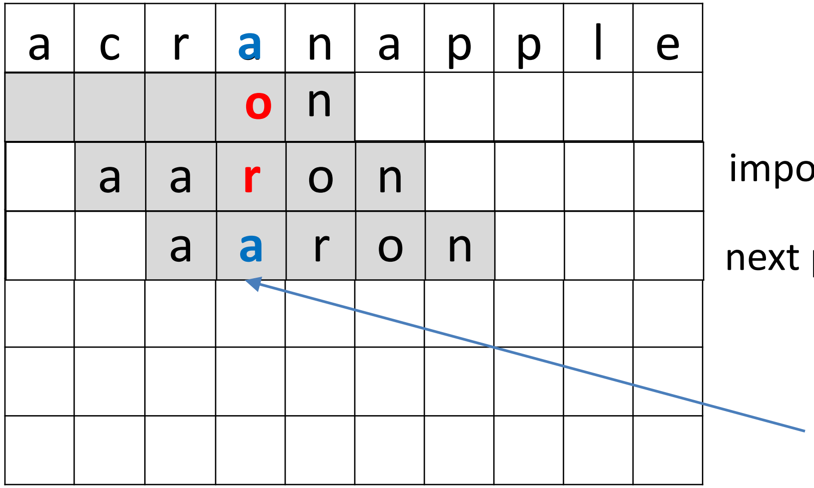
- **r** does not occur in P = aldo
- shift pattern past **r**
- **w** does not occur in P = aldo
- shift pattern past **w**
- bad character heuristic can rule out many shifts with reverse searching



- **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 rule out any shifts with forward searching when the first character of the pattern is mismatched

What if Mismatched Text Character Occurs in P?

T= acranapple, P = aaron



impossible shift

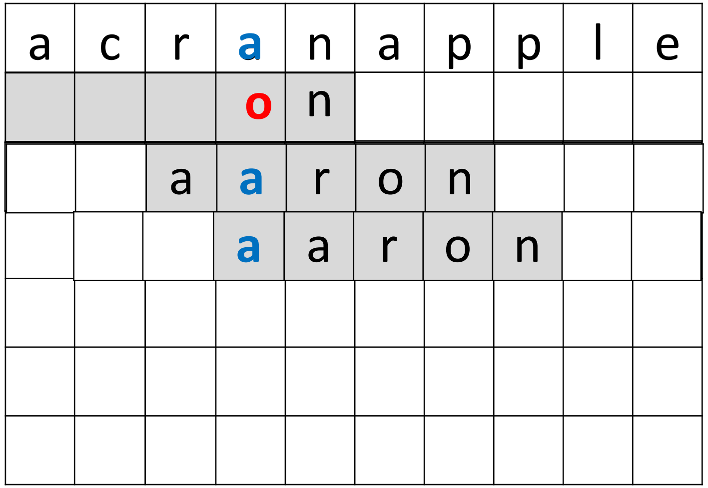
next possible shift

last occurrence of a in pattern

- Mismatched character in the text is **a**
- Find **last** occurrence of a in P
- Shift 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



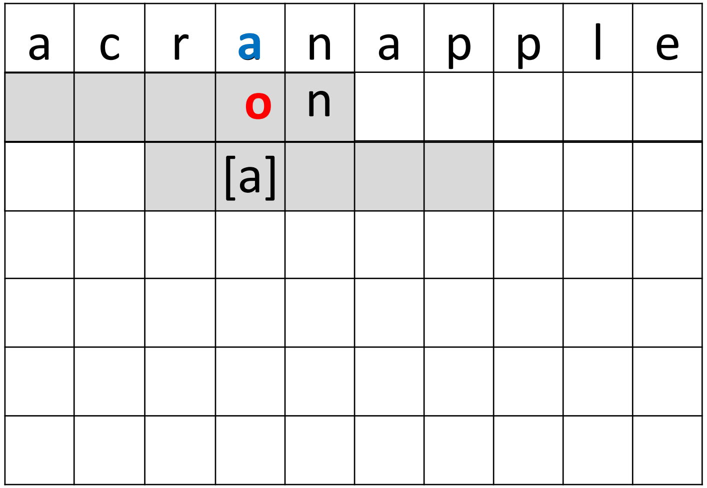
a	c	r	a	n	a	p	p	l	e	
			o	n						
		a	a	r	o	n				next possible shift
			a	a	r	o	n			also a valid shift

- If we shifted until the **first** a in P aligns with a in text
  - this would give a possible shift, but misses a previous possible shift, 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



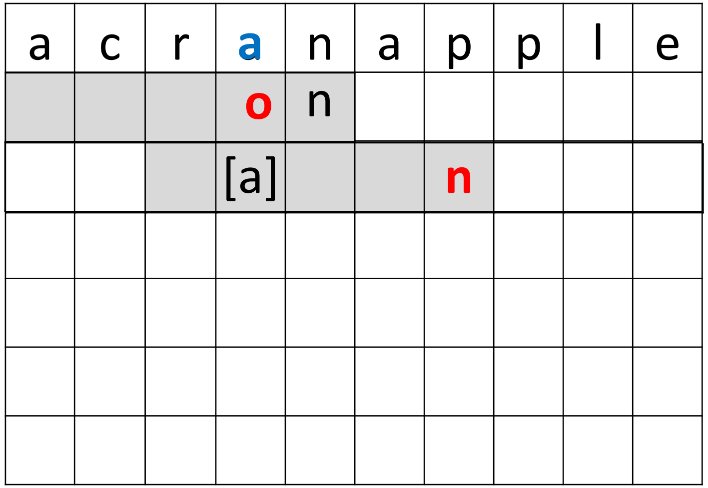
a	c	r	a	n	a	p	p	l	e
			o	n					
			[a]						

- Mismatched character in the text is **a**
- Shift 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
- Shift 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	f	…	x	y	z
-1	-1	-1	-1	-1	-1		-1	-1	-1

in actual implementation:

0	1	2	3	4	5	…	24	25	26
-1	-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

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

aaron

$i = 0$

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

aaron

$i = 2$

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

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

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

aaron

$i = 3$

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

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

aaron

$i = 4$

L is valid for $P =$ aaron

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

Boyer-More vs. Brute-force Indexing

$P = cad$

	$j=0$	$j=1$	$j=2$		
T	c	a	b	a	b
$i=0$	c	a	b		

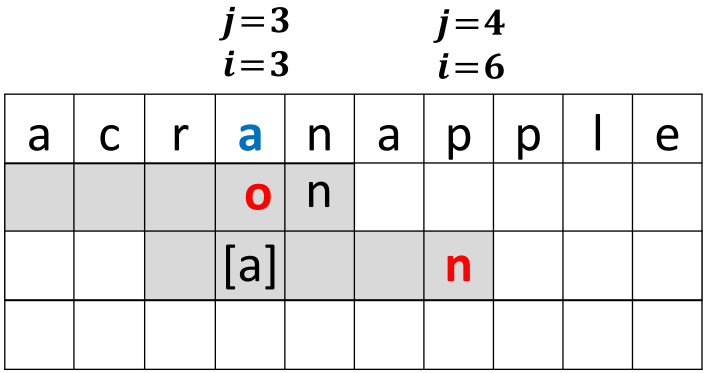
	$j=0$ $i=0$	$j=1$ $i=1$	$j=2$ $i=2$		
T	c	a	b	a	b
	c	a	b		

- Brute-force
  - maintain variables i and j
  - j is the position in the pattern
  - i is equal to the current shift
  - check is performed by determining if $T[i+j] = P[j]$
- Boyer-More
  - 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 shift is $i - j$

Bad Character Heuristic: Shifting Formula

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

T= acranapple, P = aaron

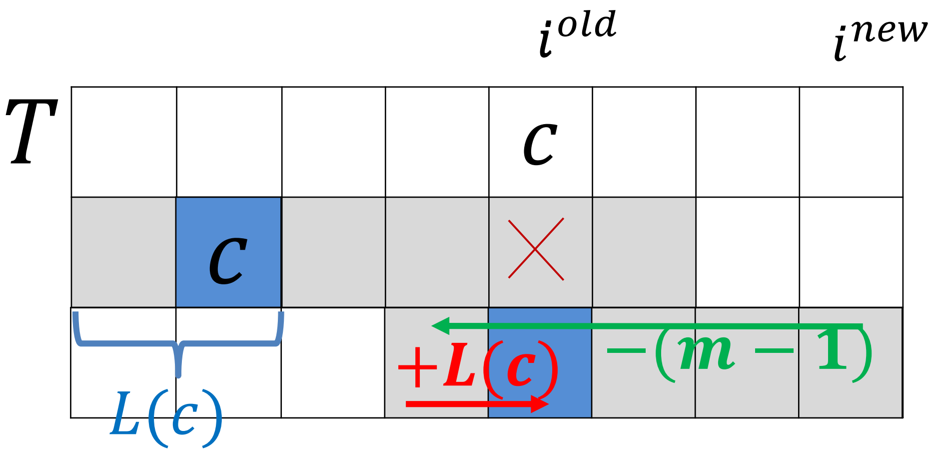


a	c	r	a	n	a	p	p	l	e
			o	n					
			[a]			n			

- 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: Shifting Formula Explained

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



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

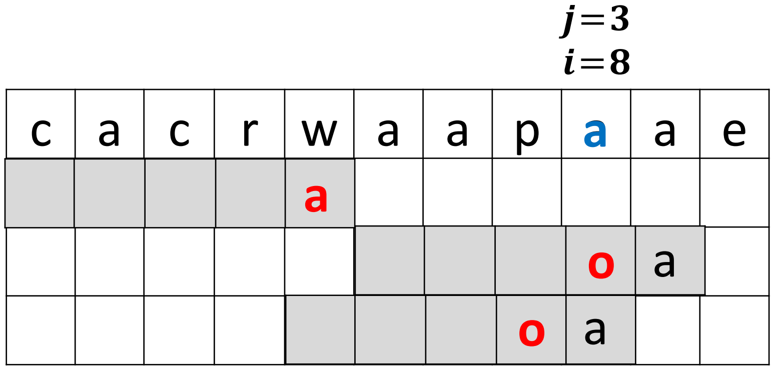
$$i^{new} = i^{old} + m - 1 - L(c)$$

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

Bad Character Heuristic: Important Use Condition

- Text character is c at the mismatch position i in the text
- $i \;\; = i + m - 1 - L(c)$
- Old shift: $i - j$
- New shift: $i + (m - 1) - L(c) - (m - 1) = i - L(c)$
- If $L(c) > j$, new shift < old shift, shifts P in the wrong direction, not useful
  - we already ruled that shift 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		

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

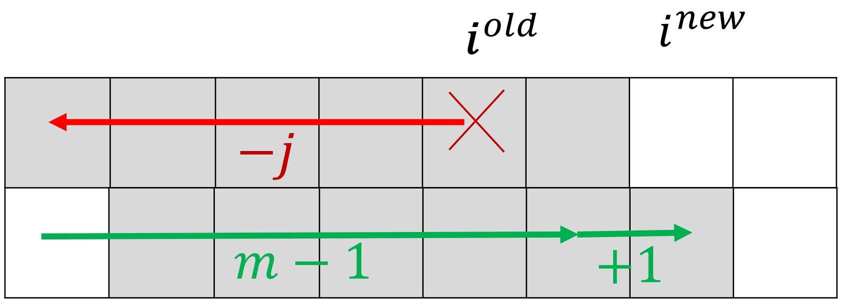
old shift: $i - j = 8 - 3 = 5$
$i \;\; = 8 + 5 \;\; - 1 - 4 = 8$
$j \;\; = 5 - 1 = 4$

new shift: $i - j = 8 - 4 = 4$

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

Bad Character Heuristic: Brute-Force Step

- If $L(c) > j$
  - pattern would shift 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

- If $L(c) < j$
  - $j = m - 1$
  - $i = i + m - 1 - L(c)$
- If $L(c) > j$
  - $j = m - 1$
  - $i = i - j + m$

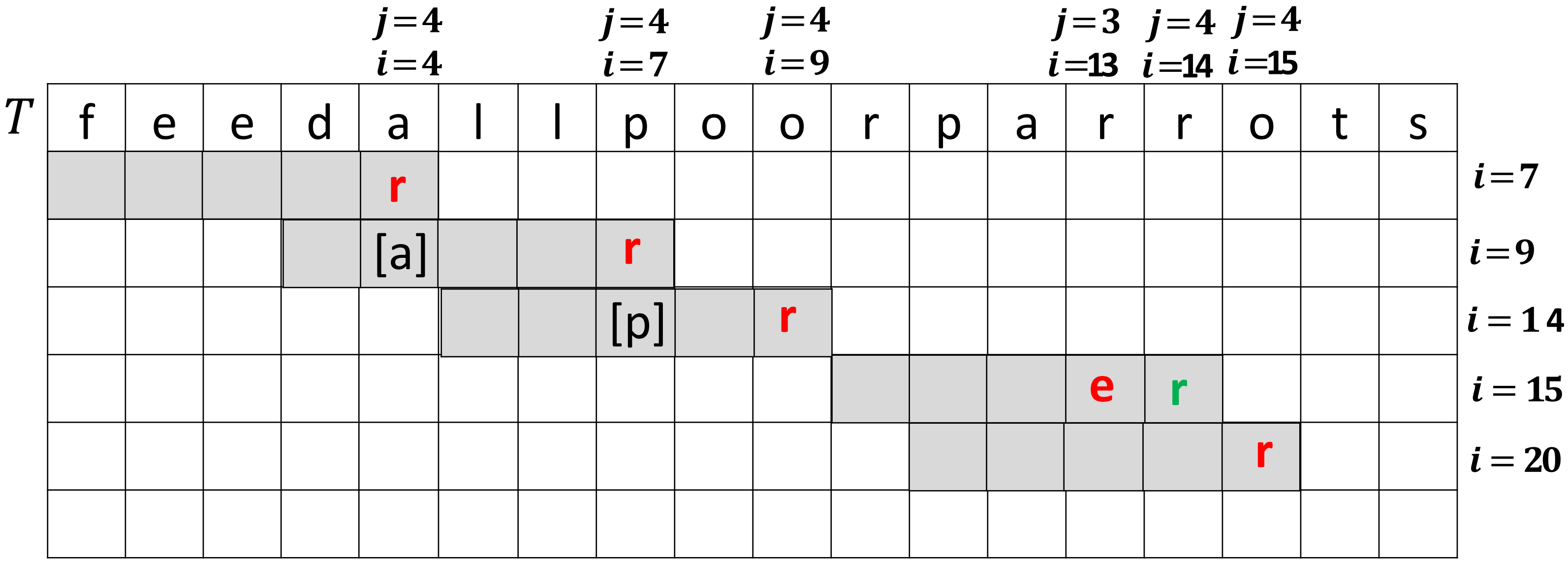
- Unified formula for i that works in all cases

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

Boyer-More Example

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

P = paper



not found!

- Unified formula for i that works in all cases

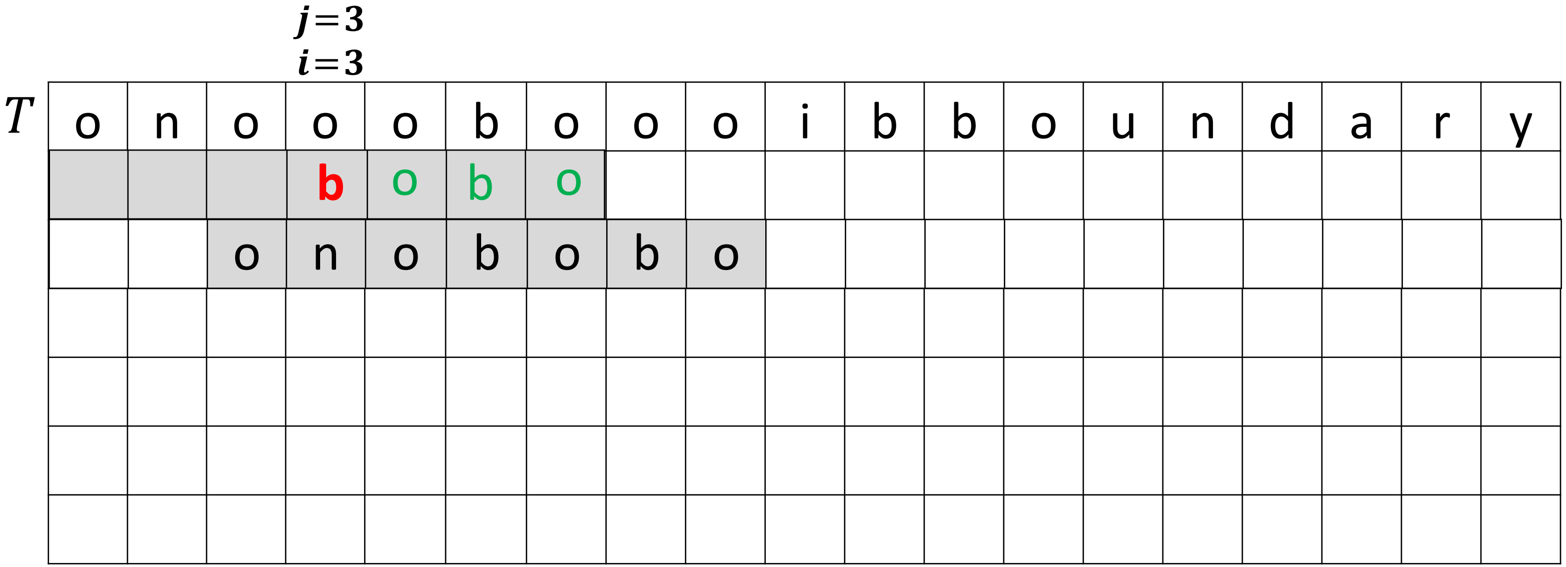
$$i = i + m - 1 - \min\{L(c), j - 1\}$$

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 shift $i + 1$” // $i$ moved one position to
                                               // the left of the first char in $T$
    else  return FAIL
```

Good Suffix Heuristic

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 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 Algorithm with Good Suffix

```
BoyerMoore($(T, P)$)
    $L \leftarrow$ last occurrence array computed from $P$
    $S \leftarrow$ good suffix 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
            $i \leftarrow i + m - 1 - \min\{L(T[i]), S[j]\}$
            $j \leftarrow m - 1$
    if $j = -1$ return “found at shift $i + 1$”
    else return FAIL
```

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
    - Boyer-Moore Algorithm
    - Suffix Trees

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

$$T = \text{establ}\underbrace{\overbrace{\text{ish}}^{\text{prefix}}\text{ment}}_{\text{suffix}}$$

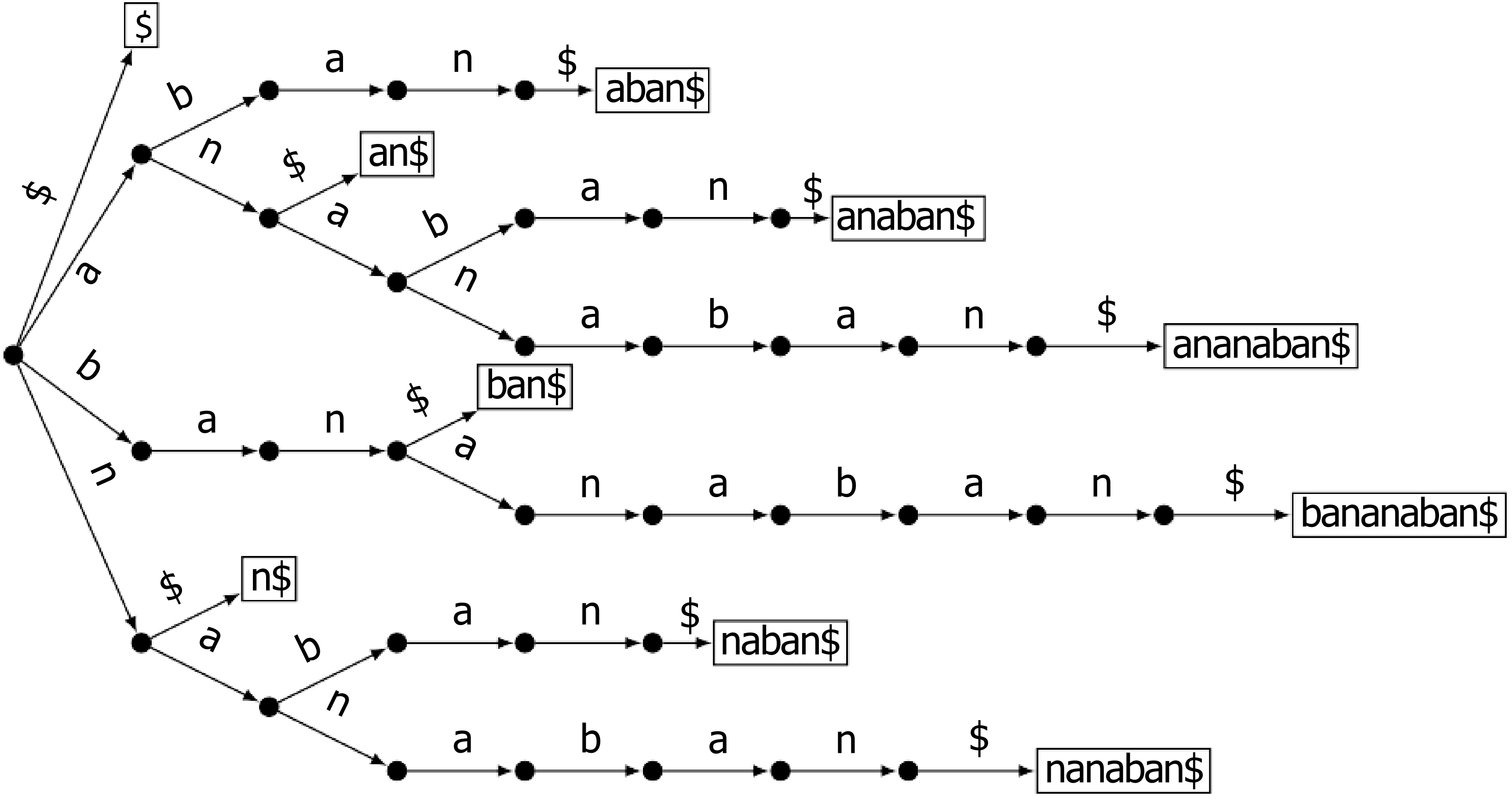
- Store all suffixes of T in a trie
- To save space
    - use compressed trie
    - store suffixes implicitly via indices into T
- This is called a **suffix tree**

Trie of suffixes: Example

- *T* = bananaban

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

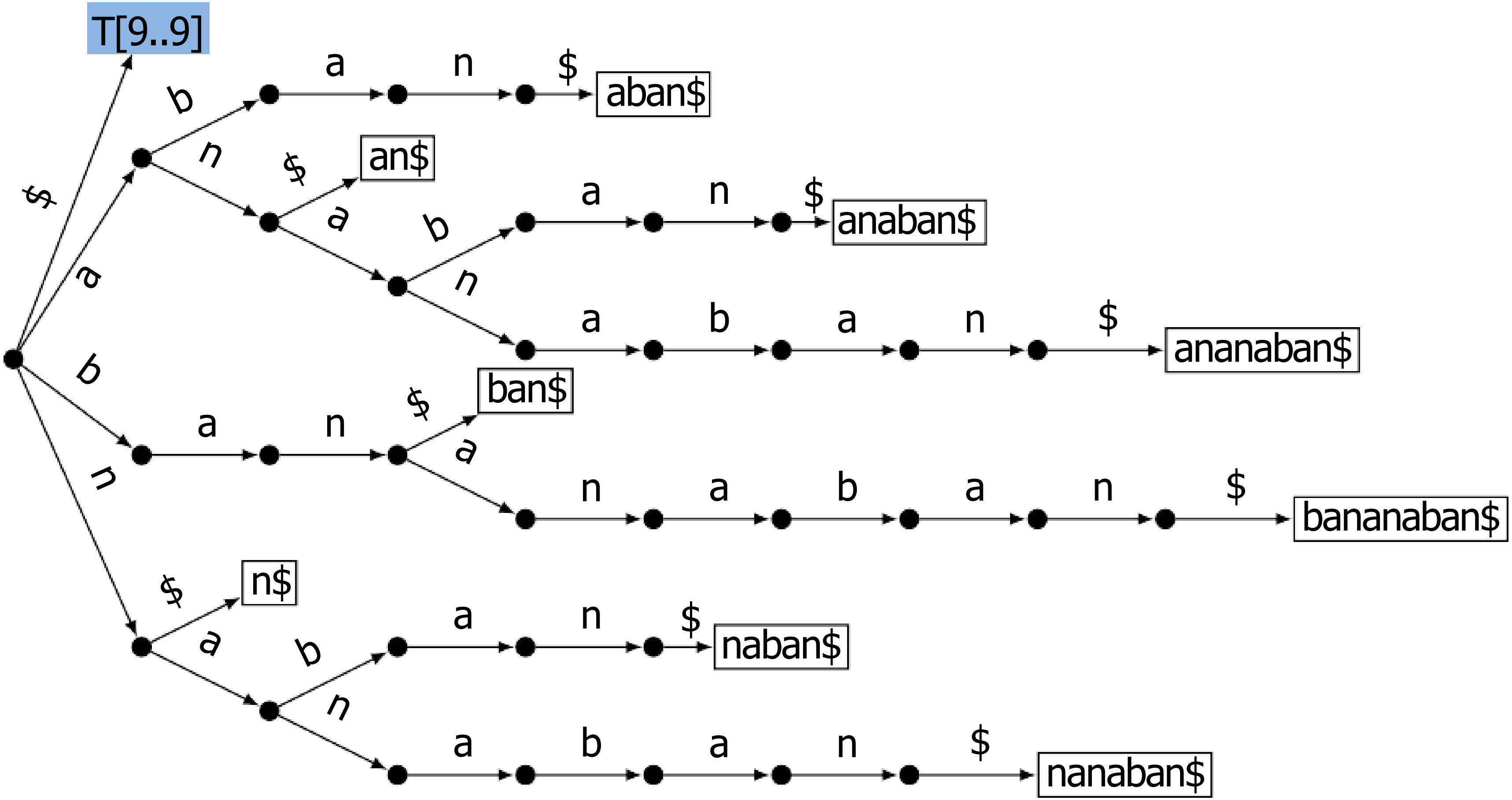
S = {bananaban$, ananaban$, nanaban$, anaban$,naban$,..., ban$, n$, $}



Trie of suffixes: Example

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

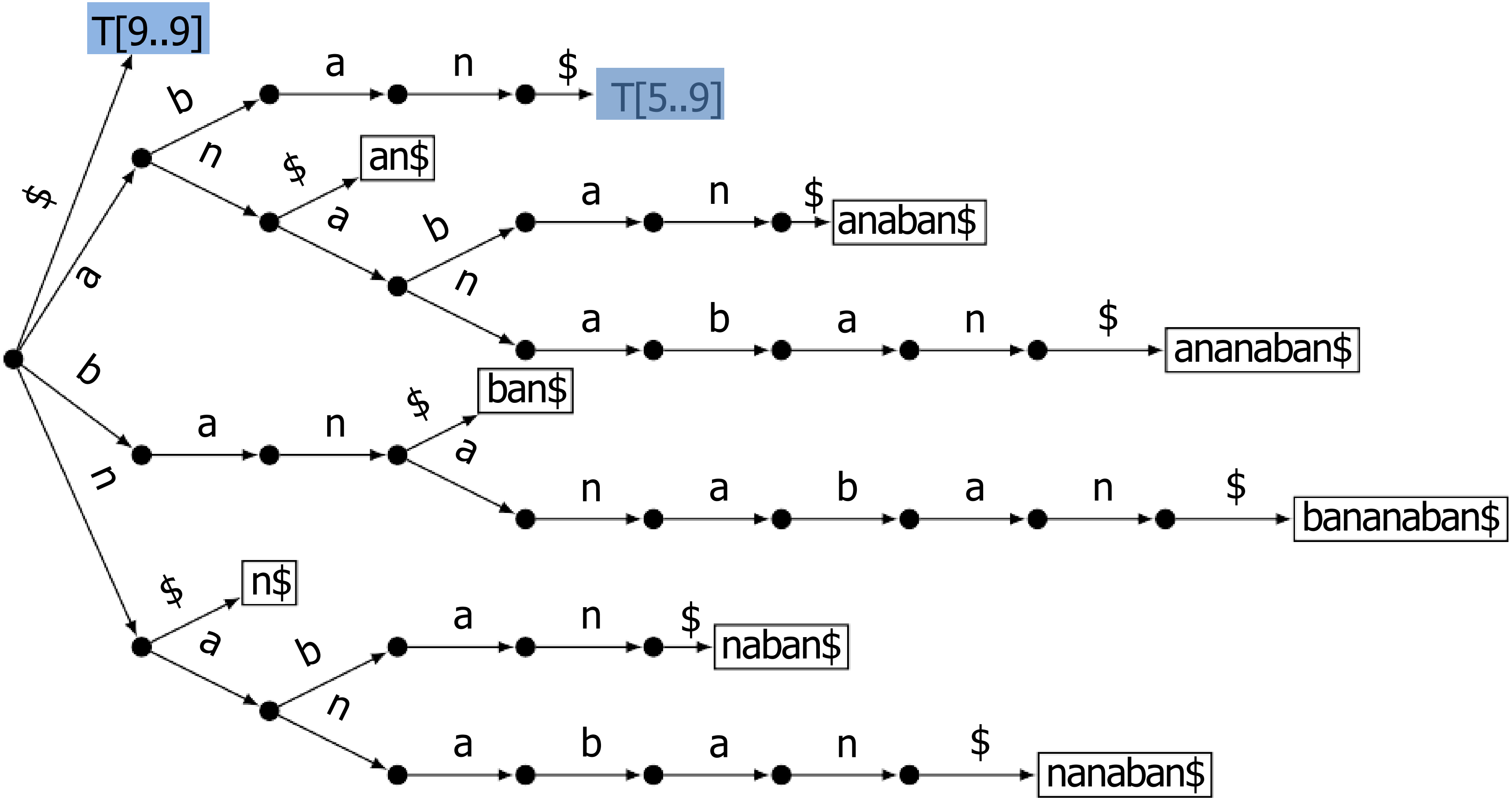
- Store suffixes via indices



Trie of suffixes: Example

- Store suffixes via indices

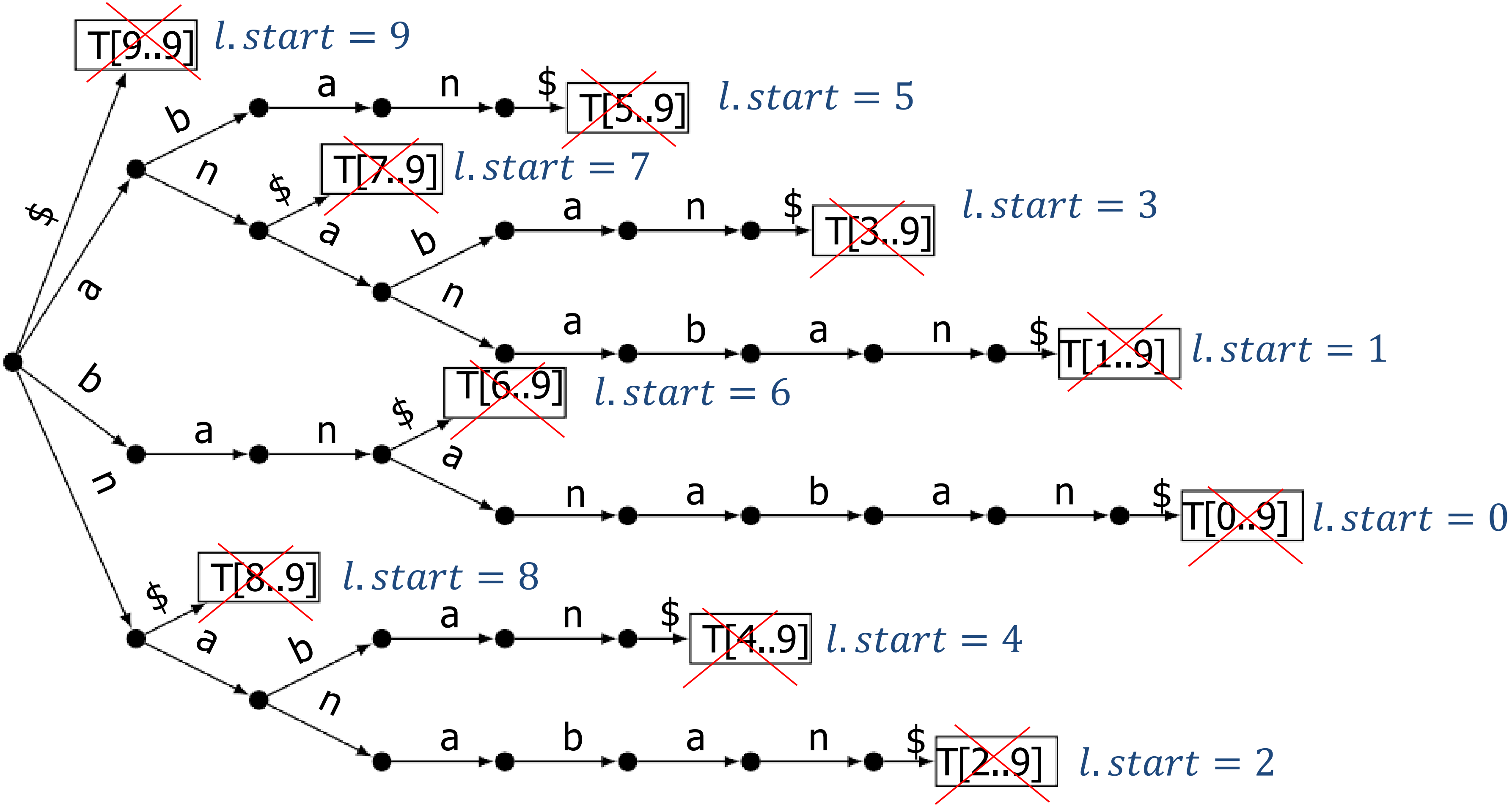
	0	1	2	3	4	5	6	7	8	9
T =	b	a	n	a	n	a	b	a	n	$



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	$

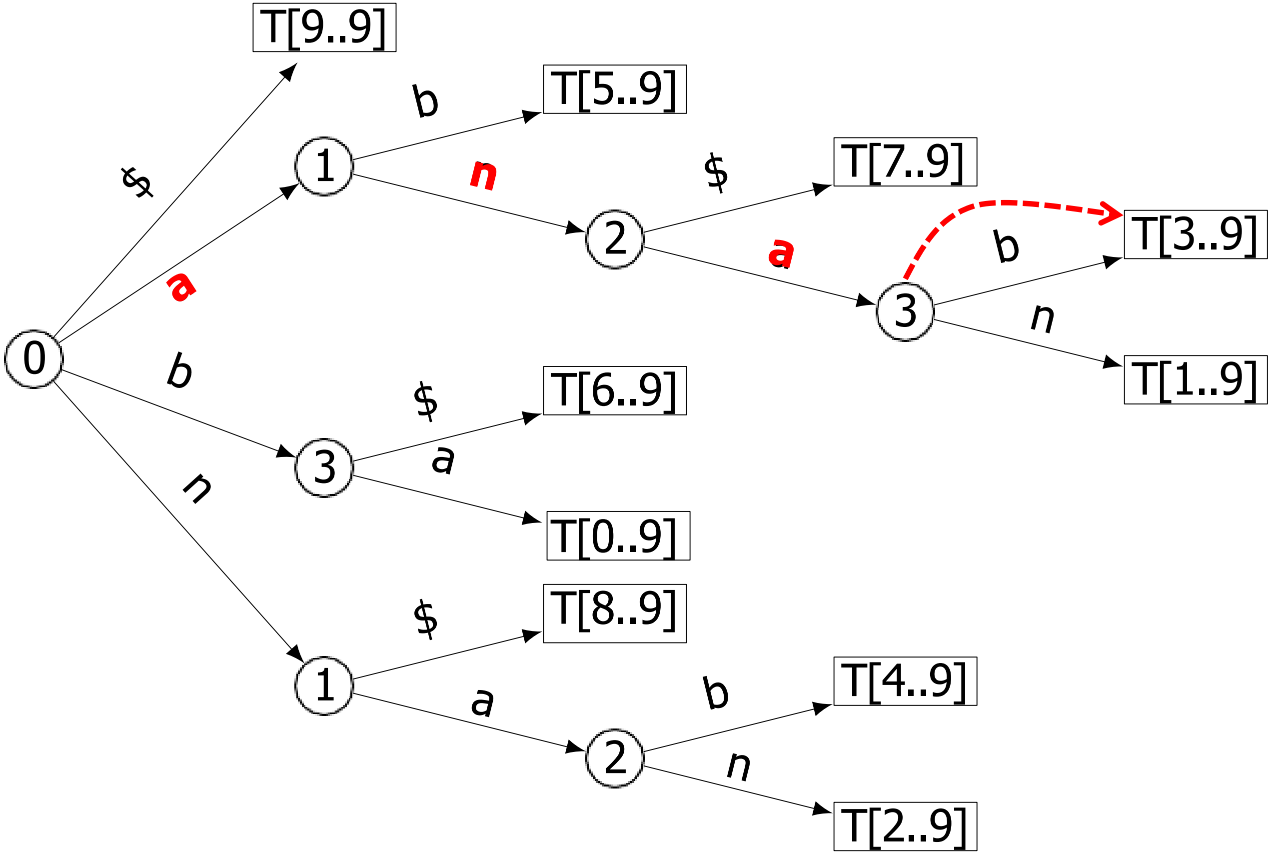


Suffix tree

T =

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

- Suffix tree: compressed trie of suffixes
- If P occurs in the text, it is a prefix of one (or more) strings stored in the trie
- Have to modify search in a trie to allow search for a prefix



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
        - $O(|\Sigma|m)$, assuming each node stores children in a linked list
        - $O(m)$, assuming each node stores children in an array
- Summary
    - theoretically good, but construction is slow or complicated and lots of space-overhead
    - rarely used in practice