CS 240 – Data Structures and Data Management

Module 9: String Matching

T. Biedl   É. Schost   O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2021

version 2021–03–22 10:18

Outline

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

Outline

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

Pattern Matching Definition [1]

- Search for a string (pattern) in a large body of text
- $T[0..n-1]$ – The **text** (or **haystack**) being searched within
- $P[0..m-1]$ – The **pattern** (or **needle**) being searched for
- Strings over **alphabet** Σ
- Return smallest i such that

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

- This is the first **occurrence** of P in T
- If P does not **occur** in T, return `FAIL`
- Applications:
  - Information Retrieval (text editors, search engines)
  - Bioinformatics
  - Data Mining

Pattern Matching Definition [2]

Example:

- $T =$ "`Where is he?`"
- $P_1 =$ "`he`"
- $P_2 =$ "`who`"

Definitions:

- **Substring** $T[i..j]$ $0 \le i \le j < n$: a string of length $j - i + 1$ which consists of characters $T[i], \ldots T[j]$ in order
- A **prefix** of T:
  a substring $T[0..i]$ of T for some $0 \le i < n$
- A **suffix** of T:
  a substring $T[i..n-1]$ of T for some $0 \le i \le n-1$

General Idea of Algorithms

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 $P[j]$. We must perform m checks of a single **correct** guess, but may make (many) fewer checks of an **incorrect** guess.

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

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 ← 0 to n − m do
        if strcmp(T[i..i+m−1], P) = 0
            return "found at guess i"
    return FAIL
```

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

```
strcmp(T[i..i+m−1], P[0..m − 1])
    for j ← 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
```

Brute-Force Example

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

a	b	b	b	a	b	a	b	b	a	b
a	b	b	**a**							
	a									
		a								
			a							
				a	b	**b**				
					a					
						a	b	b	a	

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

How to improve?

More sophisticated algorithms

- Do extra **preprocessing** on the pattern P
  - **Karp-Rabin**
  - **Boyer-Moore**
  - Deterministic finite automata (**DFA**), **KMP**
  - We **eliminate guesses** based on completed matches and mismatches.
- Do extra **preprocessing** on the text T
  - **Suffix-trees**
  - We **create a data structure** to find matches easily.

Outline

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

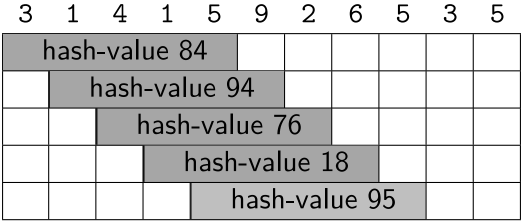
Karp-Rabin Fingerprint Algorithm – Idea

Idea: use hashing to eliminate guesses

- Compute hash function for each guess, compare with pattern hash
- If values are unequal, then the guess cannot be an occurrence
- Example: $P = 5\ 9\ 2\ 6\ 5$, $T = 3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3\ 5$
  - Use standard hash-function: flattening + modular (radix $R = 10$):

$$h(x_0 \dots x_4) = \big(x_0 x_1 x_2 x_3 x_4\big)_{10} \bmod 97$$

  - $h(P) = 59265 \bmod 97 = 95$.



Karp-Rabin Fingerprint Algorithm – First Attempt

```
Karp-Rabin-Simple::patternMatching(T, P)
1.   h_P ← h(P[0..m−1)])
2.   for i ← 0 to n − m
3.       h_T ← h(T[i..i+m−1])
4.       if h_T = h_P
5.           if strcmp(T[i..i+m−1], P) = 0
6.               return "found at guess i"
7.   return FAIL
```

- Never misses a match: $h(T[i..i{+}m{-}1]) \neq h(P) \Rightarrow$ guess i is not P
- $h(T[i..i{+}m{-}1])$ depends on m characters, so 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 – Fast Update

The initial hashes are called **fingerprints**.

Crucial insight: We can update these fingerprints in constant time.

- Use previous hash to compute next hash
- $O(1)$ time per hash, except first one

Example:

- Pre-compute: $10000 \bmod 97 = 9$
- Previous hash: $\mathbf{4}1592 \bmod 97 = 76$
- Next hash: $1592\mathbf{6} \bmod 97 = ??$

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

$$15926 \bmod 97 = \Big((\underbrace{41592 \bmod 97}_{76 \text{ (previous hash)}} - 4 \cdot \underbrace{10000 \bmod 97}_{9 \text{ (pre-computed)}} \Big) \cdot 10 + 6) \bmod 97$$

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

Karp-Rabin Fingerprint Algorithm – Conclusion

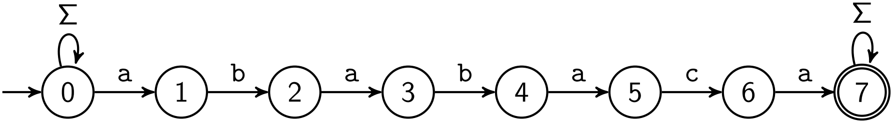
```
Karp-Rabin-RollingHash::patternMatching(T, P)
    M ← suitable prime number
    h_P ← h(P[0..m−1)])
    h_T ← h(T[0..m−1)])
    s ← 10^(m−1) mod M
    for i ← 0 to n − m
        if h_T = h_P
            if strcmp(T[i..i+m−1], P) = 0
                return "found at guess i"
        if i < n − m // compute hash-value for next guess
            h_T ← ((h_T − T[i] · s) · 10 + T[i+m]) mod M
    return "FAIL"
```

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

Outline

String Matching with Finite Automata

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

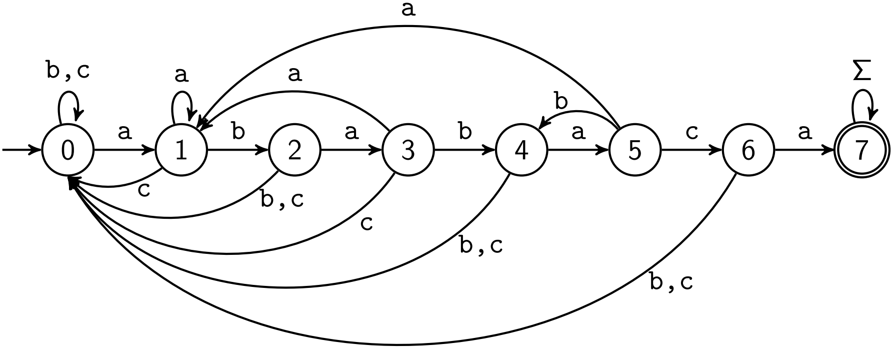


(You should be familiar with:
- finite automaton, DFA, NFA, converting NFA to DFA
- transition function δ, states Q, accepting states F)

- The above finite automation is an **NFA**
- State q expresses "we have seen $P[0..q-1]$"
  - NFA accepts T if and only if T contains ababaca
  - But evaluating NFAs is very slow.

String matching with DFA

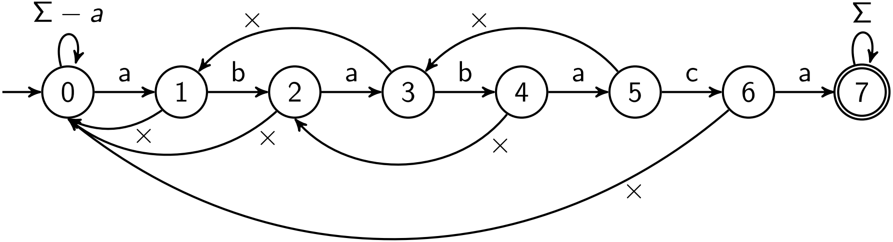
Can show: There exists an equivalent small DFA.



- Easy to test whether P is in T.
- But how do we find the arcs?
- We will not give the details of this since there is an even better automaton.

Outline

Knuth-Morris-Pratt Motivation



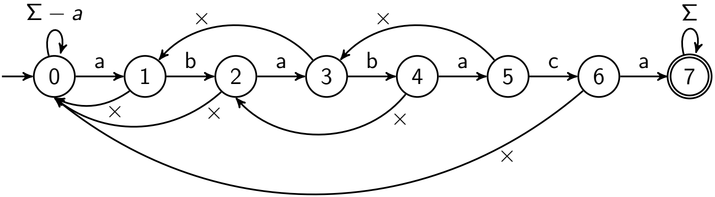
- Use a new type of transition × (*"failure"*):
  - Use this transition only if no other fits.
  - Does **not** consume a character.
  - With these rules, computations of the automaton are deterministic. (But it is formally not a valid DFA.)
- Can store **failure-function** in an array $F[0..m-1]$
  - The failure arc from state j leads to $F[j-1]$
- Given the failure-array, we can easily test whether P is in T: Automaton accepts T if and only if T contains `ababaca`

Knuth-Morris-Pratt Algorithm

```
KMP::patternMatching(T, P)
    F ← failureArray(P)
    i ← 0 // current character of T to parse
    j ← 0 // current state: we have seen P[0..j−1]
    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 // i.e. P[j] ≠ T[i]
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return FAIL
```

String matching with KMP – Example

Example: T = `abababbaca`...

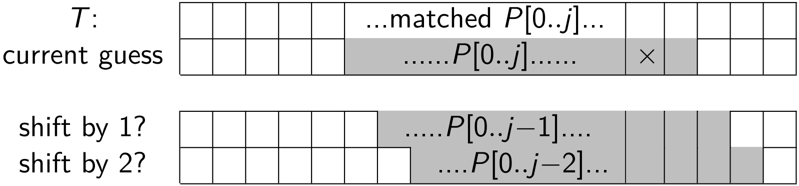


T:	a	b	a	b	a	b	b	c	a	b	a	b	a	c	a
	a	b	a	b	a	×									
			(a)	(b)	(a)	b	×								
					(a)	(b)	×								
							×								
								×							
									a	b	a	b	a	c	a
q:	1	2	3	4	5	3, 4	2, 0	0	1	2	3	4	5	6	7

(after reading this character)

String matching with KMP – Failure-function

Assume we reach state $j+1$ and now have mismatch.



- Can eliminate "shift by 1" if $P[1..j] \neq P[0..j-1]$.
- Can eliminate "shift by 2" if $P[1..j]$ does not end with $P[0..j-2]$.
- Generally eliminate guess if that prefix of P is not a suffix of $P[1..j]$.
- So want longest prefix $P[0..\ell-1]$ that is a suffix of $P[1..j]$.
- The ℓ characters of this prefix are matched, so go to state ℓ.

$$\begin{aligned} F[j] &= \text{head of failure-arc from state } j+1 \\ &= \text{length of the longest prefix of } P \text{ that is a suffix of } P[1..j]. \end{aligned}$$

KMP Failure Array – Example

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

Consider $P = \texttt{ababaca}$

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

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

Computing the Failure Array

```
KMP::failureArray(P)
P: String of length m (pattern)
    F[0] ← 0
    j ← 1            // index within parsed text
    ℓ ← 0            // reached state
    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
```

Correctness-idea: $F[j]$ is defined via pattern matching of P in $P[1..j]$. So KMP uses itself! Already-built parts of $F[\cdot]$ are used to expand it.

KMP – Runtime

failureArray

- Consider how $2j - \ell$ changes in each iteration of the while loop
  - j and ℓ both increase by 1 $\Rightarrow 2j - \ell$ increases –OR–
  - ℓ decreases ($F[\ell - 1] < \ell$) $\Rightarrow 2j - \ell$ increases –OR–
  - j increases $\Rightarrow 2j - \ell$ increases
- Initially $2j - \ell \geq 0$, at the end $2j - \ell \leq 2m$
- So no more than $2m$ iterations of the while loop.
- Running time: $\Theta(m)$

KMP main function

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

Outline

Boyer-Moore Algorithm

Fastest pattern matching on English text.

Important components:

- **Reverse-order searching**: Compare P with a guess moving **backwards**

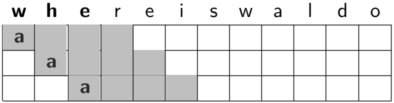
When a mismatch occurs, choose the better of the following two options:

- **Bad character jumps**: Eliminate guesses based on mismatched characters of T.
- **Good suffix jumps**: Eliminate guesses based on matched suffix of P.

Forward-searching vs. reverse-searching

P: aldo
T: whereiswaldo

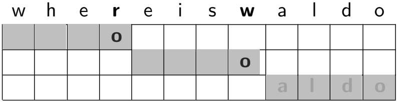
Forward-searching:



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

With forward-searching, no guesses are ruled out.

Reverse-searching:



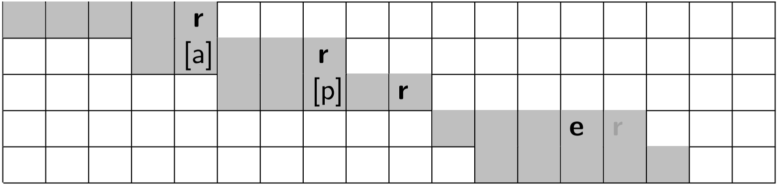
- r does not occur in P.
  $\Rightarrow$ shift pattern past r.
- w does not occur in P.
  $\Rightarrow$ shift pattern past w.

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

Bad character heuristic details

P : p a p e r

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



- Mismatched character in the text is `a`
- Shift the guess until `a` in P aligns with `a` in T
  - All skipped guessed are impossible since they do not match `a`
- Shift the guess until *last* `p` in P aligns with `p` in T
  - Use "last" since we cannot rule out this guess.
- As before, shift completely past `o` since `o` is not in P.
- Finding `r` does not help ⇒ shift by one unit.
  - Here the other strategy will do better.

Last-Occurrence Array

- Build the **last-occurrence array** L mapping Σ to integers
- $L[c]$ is the largest index i such that $P[i] = c$
- We will see soon: If c is not in P, then we should set $L[c] = -1$

Pattern: `paper`

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

- We can build this in time $O(m + |\Sigma|)$ with simple for-loop

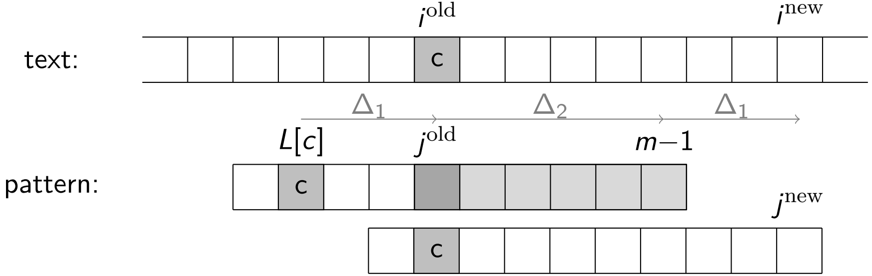
BoyerMoore::lastOccurrenceArray($P[0..m-1]$)
1. initialize array L indexed by Σ with all -1
2. **for** $j \leftarrow 0$ **to** $m-1$ **do** $L[P[j]] \leftarrow j$
3. **return** L

- But how should we do the update?

Bad character heuristic formula

We will always compare $T[i]$ and $P[j]$. How to update at a mismatch?

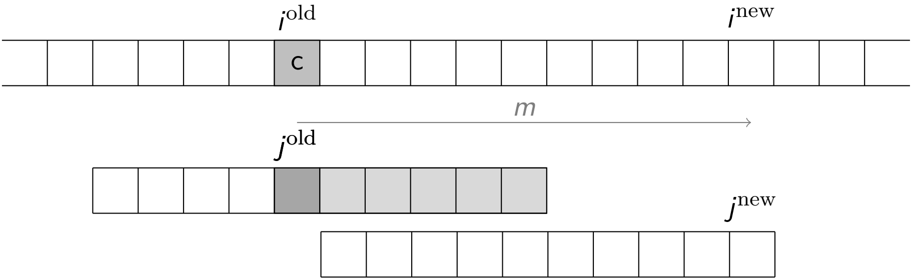
"Good" case: $L[c] < j$, so c is left of $P[j]$.



- $j^{\text{new}} = m-1$ (we re-start the search from the right end)
- $i^{\text{new}} =$ corresponding index in T. What is it?
  - $\Delta_1 =$ amount that we should shift $= j^{\text{old}} - L[c]$
  - $\Delta_2 =$ how much we had compared $= (m-1) - j^{\text{old}}$
  - $i^{\text{new}} = i^{\text{old}} + \Delta_2 + \Delta_1 = i^{\text{old}} + (m-1) - L[c]$

Bad character heuristic formula

Bad case 1: c does not occur in P.

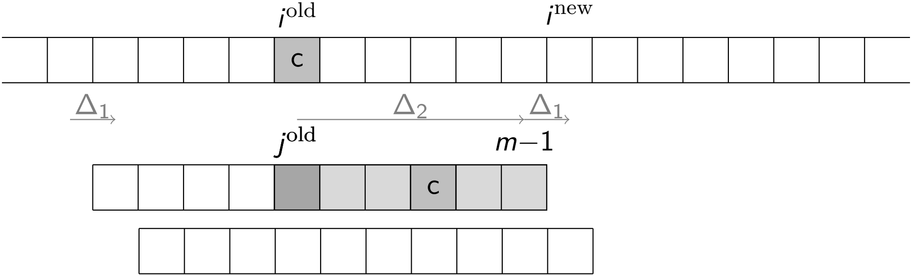


- We want to shift past $T[i^{\text{old}}]$, so need $i^{\text{new}} = i^{\text{old}} + m$
- What value of $L[c]$ would achieve this automatically?
  - formula was $i^{\text{new}} = i^{\text{old}} + (m-1) - L[c]$

  $\Rightarrow$ set $L[c] := -1$

Bad character heuristic formula

Bad case 2: $L[c] > j$, so c is right of $P[j]$.



- Bad character heuristic not helpful in this case.
- We want to shift by $\Delta_1 := 1$ units

$$i^{\text{new}} = i^{\text{old}} + \Delta_2 + \Delta_1 = i^{\text{old}} + 1 + (m{-}1) - j^{\text{old}}$$

Unified formula for all cases: $\boxed{i^{\text{new}} = i^{\text{old}} + (m{-}1) - \min\left\{L[c], j^{\text{old}}{-}1\right\}}$

Boyer-Moore Algorithm

```
Boyer-Moore::patternMatching(T,P)
1.    L ← lastOccurrenceArray(P)
2.    S ← good suffix array computed from P
3.    i ← m − 1,    j ← m − 1
4.    while i < n and j ≥ 0 do
5.        if T[i] = P[j]
6.            i ← i − 1
7.            j ← j − 1
8.        else
9.            i ← i + m−1 − min{L[T[i]], j−1, S[j]}
10.           j ← m − 1
11.   if j = −1 return "found at T[i+1..i+m]"
12.   else return FAIL
```

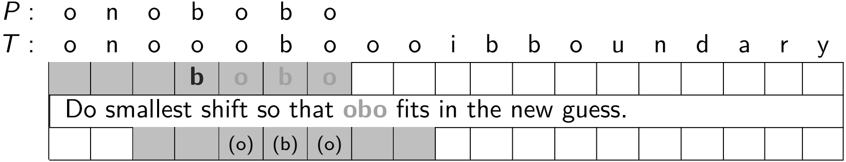
S will be explained below.

Can show: '$j-1$' is not needed in line 9 since $\min\{L[T[i]], S[j]\} \leq j-1$

Good Suffix Heuristic

$S[j]$ expresses

"since $P[j+1..m-1]$ was matched, how much should we shift?"



- Doing examples is easy, but the formula is complicated (no details)
- $S[\cdot]$ computable (similar to KMP failure function) in $\Theta(m)$ time.

Summary:

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

Outline

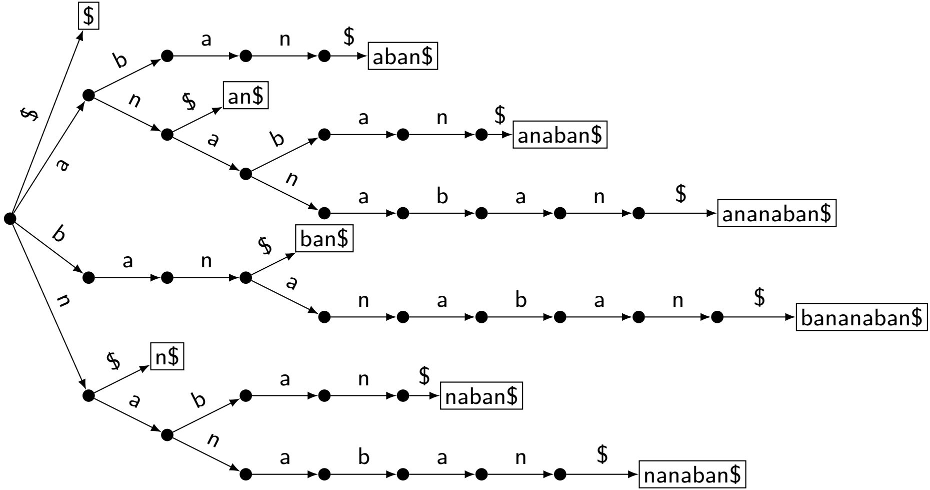
Tries of Suffixes and Suffix Trees

- What if we want to search for **many patterns** P within the same **fixed text** T?
- **Idea:** Preprocess the text T rather than the pattern P
- **Observation:** P is a substring of T if and only if P is a prefix of some suffix of T.
- So want to store all suffixes of T in a trie.
- To save space:
  - Use a compressed trie.
  - Store suffixes implicitly via indices into T.
- This is called a **suffix tree**.

Trie of suffixes: Example

$T =$ bananaban has suffixes

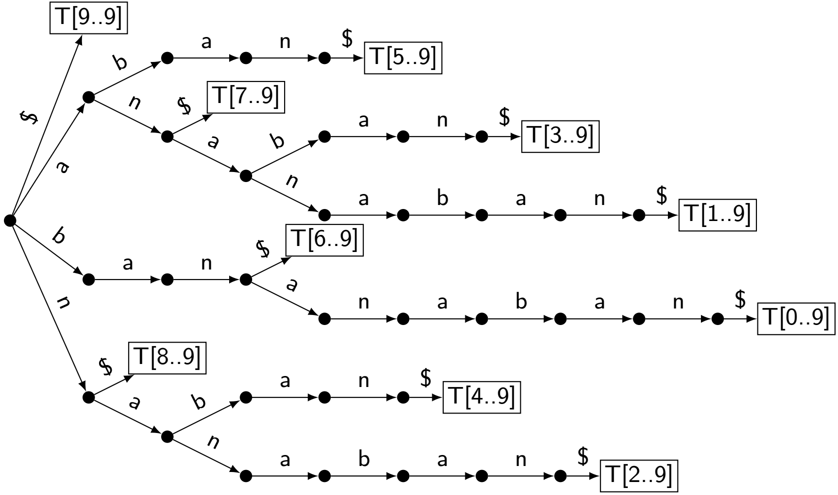
{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n, Λ}



Tries of suffixes

Store suffixes via indices: $T =$

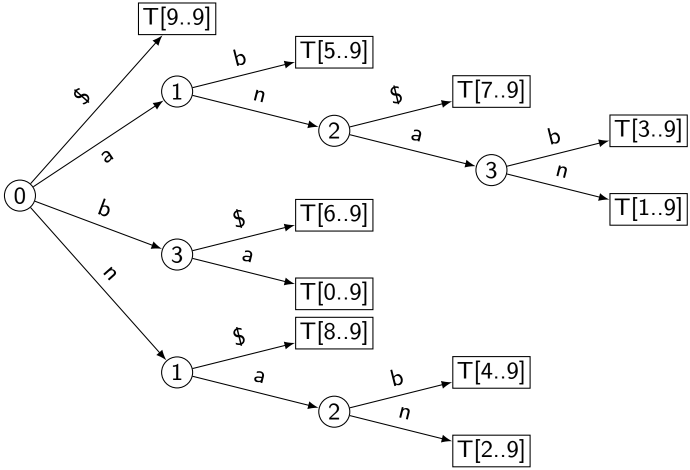
0	1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	b	a	n	$



Suffix tree

Suffix tree: Compressed trie of suffixes

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



More on Suffix Trees

Building:

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

Pattern Matching:

- Essentially *search* for P in compressed trie. Some changes are needed, since P may only be prefix of stored word.
- Run-time: $O(|\Sigma|m)$.

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

Outline

Suffix Arrays

- Relatively recent development (popularized in the 1990s)
- Sacrifice some performence 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

Text T:

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

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

$\longrightarrow$ 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:

0	1	2	3	4	5	6	7	8	9
9	5	7	3	1	6	0	8	4	2

Suffix Array Construction

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

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

)

  - $O(\log n)$ rounds enough $\Rightarrow$ $O(n \log n)$ **run-time**
- Construction-algorithm: MSD-radix-sort plus some bookkeeping
  - needs only one extra array
  - easy to implement
- You do not need to know details.

Pattern matching in suffix arrays

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

$P =$ban:

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

- $O(\log n)$ comparisons.
- Each comparison is *strcmp*$(P, T[A^s[\nu]..A^s[\nu + m - 1]])$
- $O(m)$ time per comparison $\Rightarrow$ **run-time** $O(m \log n)$

Pattern matching in suffix arrays

SuffixArray-search$(A^s[0...n-1], P[0..m-1])$
A^s: suffix array of T, P: pattern

- $\ell \leftarrow 0$, $r \leftarrow n-1$
- **while** $(\ell < r)$
  - $\nu \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
  - $i \leftarrow A^s[\nu]$ // Suffix is $T[i..n-1]$
  - $s \leftarrow$ *strcmp*$(T[i..i+m-1], P)$
    - // Assuming *strcmp* handles "out of bounds" suitably
  - **if** $(s < 0)$ **do** $\ell \leftarrow \nu + 1$
  - **else if** $(s > 0)$ **do** $r \leftarrow \nu - 1$
  - **else return** "found at guess $T[i..i+m-1]$"
- **if** *strcmp*$(T, P, A^s[\ell], A^s[\ell]+m-1) = 0$
  - **return** "found at guess $T[\ell..\ell+m-1]$"
- **return** `FAIL`

Outline

String Matching Conclusion

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

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