Lecture 12: String Matching III

COMP526: Efficient Algorithms

Updated: November 12, 2024

Will Rosenbaum
University of Liverpool

Announcements

1. Programming Assignment 1 **DUE WEDNESDAY**
   - Use updated testing code (from last Wednesday) ←
   - Submission through Canvas
   - *Only* submit ~~`pr_tester.py`~~ pr_sorter.py
   - Late Policy: 5% off per day down to 50%
2. Quiz due Friday
   - Covers string matching
     - [*including today's lecture*] **
   - 2 questions (multiple choice)
3. Attendance Code:

174786

Meeting Goals

Discuss String Matching procedures:

- Knuth-Morris-Pratt ←
- Boyer-Moore ←

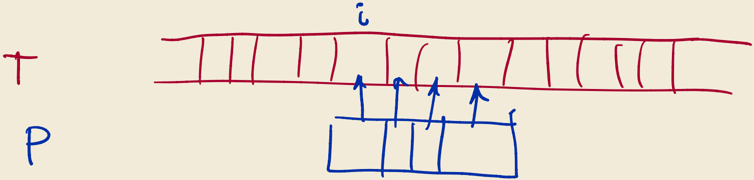
The String Matching Problem

Input:

- A **text** $T \in \Sigma^*$ of length n
- A **pattern** $P \in \Sigma^*$ of length m

Output:

- The index of the **first occurrence** of P in T



The String Matching Problem

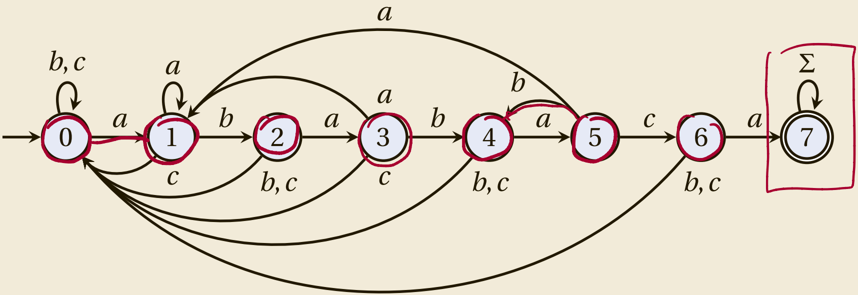
Input:

- A **text** $T \in \Sigma^*$ of length n
- A **pattern** $P \in \Sigma^*$ of length m

Output:

- The index of the **first occurrence** of P in T

Last Time. Search with DFA



1 2 3 4 5 4 5 6

Example: $T =$ `abababac`

The String Matching Problem

Fix Typos
$n \to m$

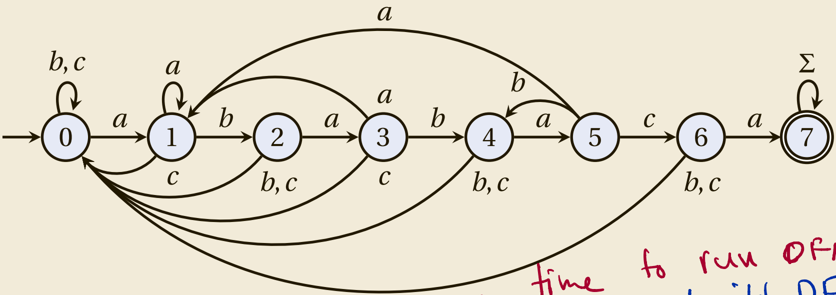
Input:

- A **text** $T \in \Sigma^*$ of length n
- A **pattern** $P \in \Sigma^*$ of length m

Output:

- The index of the **first occurrence** of P in T

Last Time. Search with DFA



time to run DFA

build DFA

Result: Search in time $\Theta(\underline{n} + \boxed{|\Sigma|\, n})$ with space overhead $\boxed{|\Sigma|\, n}$.

Knuth-Morris-Pratt

Failure Link Automaton

DFA efficiency.

- Space/time to build DFA: $\Theta(m|\Sigma|)$
- Time to execute DFA: $\Theta(n)$

$\Longrightarrow$ Overall time is $\Theta(n + m|\Sigma|)$

  - additional space overhead is $\Theta(m|\Sigma|)$

Question. Can we perform string matching in time $O(n)$ with *less space overhead?*

Failure Link Automaton

DFA efficiency.

- Space/time to build DFA: $\Theta(m|\Sigma|)$
- Time to execute DFA: $\Theta(n)$

$\Longrightarrow$ Overall time is $\Theta(n+m|\Sigma|)$

  - additional space overhead is $\Theta(m|\Sigma|)$

Question. Can we perform string matching in time $O(n)$ with *less space overhead?*

Idea. When comparison fails, don't have a separate transition for each failing character

- Just record failure and "shift" pattern as far forward as possible

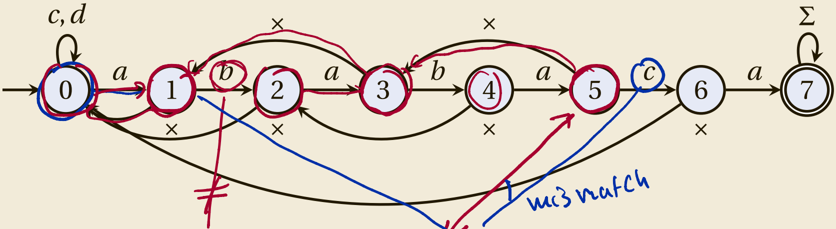
Failure Link Automaton

x = any mismatch

[On mismatch try matching same char in T again]

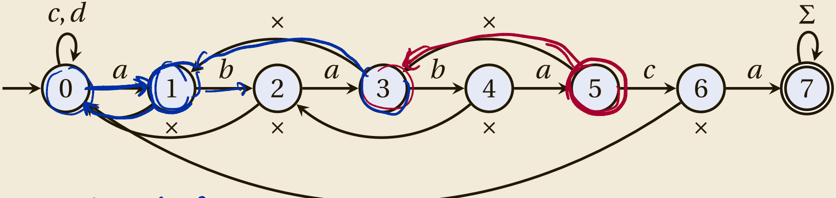
Example

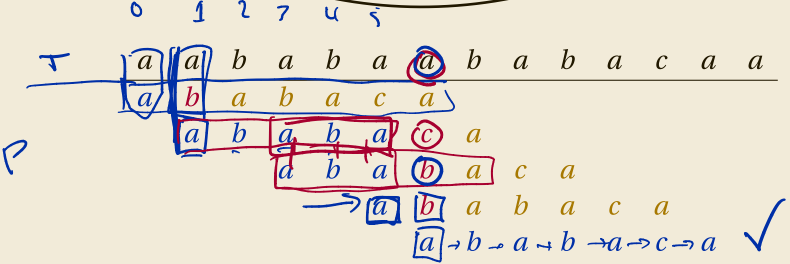
- T = aababaababacaa
- P = ababaca



text	*a*	*a*	*b*	*a*	*b*	*a*	*a*	*b*	*a*	*b*	*a*	*c*	*a*	*a*
states	1	0 1	2	3	4	5	3 1 0 1	2	3	4	5	6	7 × × ×	7

States and Shifts





	0	1	2	7	4	5								
T	a	a	b	a	b	a	a	b	a	b	a	c	a	a
P	a	b	a	b	a	c	a							
		a	b	a	b	a	c	a						
				a	b	a	b	a	c	a				
						a	b	a	b	a	c	a		
							a →	b →	a →	b →	a →	c →	a	✓

Correspondence: matches increment T index i, mismatches shift P

- shift amount aligns largest possible number of matches

FLA Execution

A **Failure Link Automaton** (**FLA**) consists of:

- A finite set Q of **states**
- A finite **alphabet** Σ
- A **transition function** $\varphi : \underline{Q} \times (\underline{\Sigma \cup \boxed{\{\times\}}}) \to Q$
- An **initial state** $\underline{q_0} \in Q$
- A set $\underline{F \subseteq Q}$ of **accepting states**

#s 0 … m

\# of matching chars aligned @ current index of T

FLA Execution

A **Failure Link Automaton** (**FLA**) consists of:

- A finite set Q of **states**
- A finite **alphabet** Σ
- A **transition function** $\varphi : Q \times (\Sigma \cup \{\times\}) \to Q$
- An **initial state** $q_0 \in Q$
- A set $F \subseteq Q$ of **accepting states**

Execution. To apply and FLA to T

- Start at the state q_0
- Read characters from T sequentially
  - if in state q and read character c:
    - if $\varphi(q, c)$ is defined, move to state $\varphi(q, c)$
    - otherwise move to state $\varphi(q, \times)$ and **re-read** c
- Return TRUE if end in "accepting" state

text char
=
pattern char

FLA Execution

PollEverywhere Question

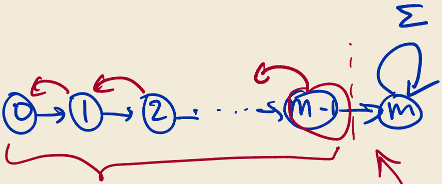
Given an FLA for a pattern P of length m, how many times could we follow failure links for a single character c read from T in the worst case?

`pollev.com/comp526`

Execution. To apply and FLA to T

- Start at the state q_0
- Read characters from T sequentially
  - if in state q and read character c:
    - if $\varphi(q, c)$ is defined, move to state $\varphi(q, c)$
    - otherwise move to state $\varphi(q, \times)$ and **re-read** c
- Return TRUE if end in "accepting" state

FLA Execution



m states

→ m-1 possible failures

Naive analysis
each char from T
gives $\Theta(m)$ running time

$\Rightarrow \boxed{\Theta(n \cdot m)}$ running time?

Execution. To apply and FLA to T

- Start at the state q_0
- Read characters from T sequentially
  - if in state q and read character c:
    - if $\varphi(q, c)$ is defined, move to state $\varphi(q, c)$
    - otherwise move to state $\varphi(q, \times)$ and **re-read** c
- Return TRUE if end in "accepting" state

P = AAA---AB

FLA Running Time

More careful analysis

- If we match up to $P[j]$, then we can only follow up to j back links
- In order to witness j failures, must have witnessed j successes!

FLA Running Time

More careful analysis

- If we match up to $P[j]$, then we can only follow up to j back links
- In order to witness j failures, must have witnessed j successes!

Amortized cost of each character read from $\underline{T}$

- If read character c is a **match**:
  - pay 1 for comparison ✓
  - put 1 unit cost in the **bank** ✓
- If read character c is a **mismatch**
  - *withdraw* 1 from the bank

pay $\boxed{2}$ from pocket per read (match)

withdraw 1 per mismatch

- By analysis above account balance is always non-negative

$\Longrightarrow$ amortized cost of each comparison is $\boxed{2}$

$\Longrightarrow$ hence overall running time of execution is $O(n)$

Worst case $\longrightarrow$ $\underline{2n}$ comparisons when running on T[0,..n).

FLA Construction

Observation. Each state q has

- 1 forward link to state $q+1$
- 1 fail link

Given P, we don't need to store forward link label:

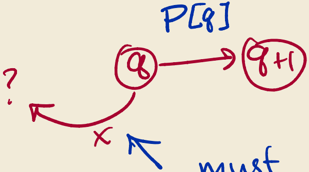
- forward link label from q to $q+1$ is $P[q]$

Only need to store fail link state!

- this can be stored as a single array of size m

$\Longrightarrow$ only $O(m)$ space overhead

no need to store if we know P



must store this

Store in an array fail[q] store "back link" on mismatch from q.

FLA Construction

Definition. The **failure link array** *fail* of P the array of m numbers that stores the (index of) the next state for each failure

- How do we construct it?

FLA Construction

Definition. The **failure link array** *fail* of P the array of m numbers that stores the (index of) the next state for each failure

- How do we construct it?
- Again x is length of largest prefix that matches a suffix of $P[1, q)$

Example. $P[0..6) =$ `ababaca`

q	0	1	2	3	4	5	6
$fail[q]$	0	0	0	1	2	3	0

```
procedure FailureLink($P[0, m)$)
    $fail[0] \leftarrow 0$
    $x \leftarrow 0$   ← prefix match
    for $j = 1, 2, \ldots, m-1$ do
        $fail[j] \leftarrow x$
        while $P[x] \neq P[j]$ do
            if $x = 0$ then
                $x \leftarrow -1$
                break
            else
                $x \leftarrow fail[x]$   update to 1
            end if
        end while
        $x \leftarrow x + 1$
    end for
end procedure
```

$x = \cancel{0}\,1,\quad x = 2,\ j = 4$

$x = 3,\ j = 5$

FLA Construction

Question. What is the running time of FAILURELINK on input of size m?

```
procedure FAILURELINK($P[0, m)$)
    $fail[0] \leftarrow 0$
    $x \leftarrow 0$
    for $j = 1, 2, \ldots, m-1$ do
        $fail[j] \leftarrow x$
        while $P[x] \neq P[j]$ do
            if $x = 0$ then
                $x \leftarrow -1$
                break
            else
                $x \leftarrow fail[x]$
            end if
        end while
        $x \leftarrow x + 1$
    end for
end procedure
```

x decremented

x gets incremented

FLA Construction

Question. What is the running time of FailureLink on input of size m?

Observations.

- x incremented once per j
- $fail[x] < x$
- Each "while" iteration decrements x

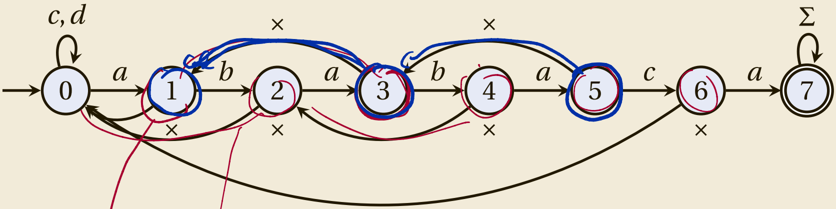
So at most $2m$ updates to x

- cf. amortized analysis
- x = bank balance

```
procedure FailureLink($P[0, m)$)
    $fail[0] \leftarrow 0$
    $x \leftarrow 0$
    for $j = 1, 2, \ldots, m-1$ do
        $fail[j] \leftarrow x$
        while $P[x] \neq P[j]$ do
            if $x = 0$ then
                $x \leftarrow -1$
                break
            else
                $x \leftarrow fail[x]$
            end if
        end while
        $x \leftarrow x + 1$
    end for
end procedure
```

$\Rightarrow$ $\Theta(m)$ running time

Failue Links: 3 Views



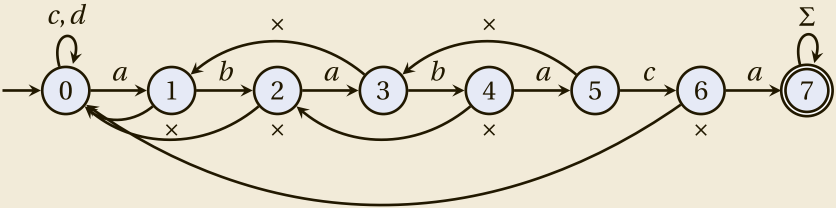
a b a b a c a

a b a b a c a

a b a b a c a

q	0	1	2	3	4	5	6
$fail[q]$	0	0	0	1	2	3	0

Failue Links: 3 Views



a	b	a	b	a	c	a				
		a	b	a	b	a	c	a		
				a	b	a	b	a	c	a

q	0	1	2	3	4	5	6
$fail[q]$	0	0	0	1	2	3	0

$fail[q]$ is

- the max of alignments formed by shifting P if first mismatch at $P[q]$
- longest prefix of $P[0, q)$ that is a suffix of $P[1, q)$

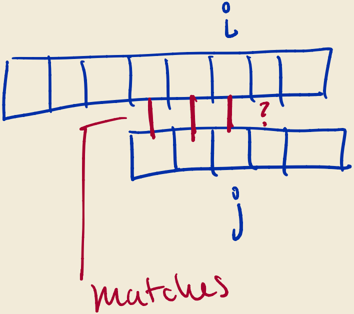
KMP Algorithm

Question. How do we apply the failure link array to find a match?

KMP Algorithm

Question. How do we apply the failure link array to find a match?

- Scan along $T[0, n)$
    - index i
- Maintain position in $P[0, m)$
    - index j
    - current prefix match
- When $T[i] = P[j]$, increment i and j
- Otherwise, $\boxed{j \leftarrow \mathit{fail}[j]}$
    - unless $j = 0$, then $i \leftarrow i + 1$



KMP Algorithm

Question. How do we apply the failure link array to find a match?

- Scan along $T[0, n)$
  - index i
- Maintain position in $P[0, m)$
  - index j
  - current prefix match
- When $T[i] = P[j]$, increment i and j
- Otherwise, $j \leftarrow fail[j]$
  - unless $j = 0$, then $i \leftarrow i + 1$

```
procedure KMP($T[0..n), P[0..m)$)
    $fail \leftarrow$ FAILURELINK($P$)
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $i < n$ do
        if $T[i] = P[q]$ then
            $i \leftarrow i + 1, j \leftarrow j + 1$
            if $j = m$ then return $i - j$
        else
            if $j \geq 1$ then
                $j \leftarrow fail[j]$
            else
                $i \leftarrow i + 1$
            end if
        end if
    end while
end procedure
```

KMP Algorithm

Analysis:

- Running time $O(n+m)$
  - $O(m)$ to build *fail*
  - $O(n)$ to apply KMP
  - analysis uses **amortized analysis**
- Additional space $\boxed{O(m)}$
  - just need to store *fail* and indices

size

```
procedure KMP($T[0..n), P[0..m)$)
    $\boxed{fail}$ ← FAILURELINK($P$)   } O(m)
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $i < n$ do
        if $T[i] = P[q]$ then                 O(n)
            $i \leftarrow i+1$, $j \leftarrow j+1$
            if $j = m$ then return $i - j$
        else
            if $j \geq 1$ then
                $j \leftarrow fail[j]$
            else
                $i \leftarrow i+1$
            end if
        end if
    end while
end procedure
```

Fix typos

KMP Algorithm

Analysis:

- Running time $O(n+m)$
    - $O(m)$ to build *fail*
    - $O(n)$ to apply KMP
    - analysis uses **amortized analysis**
- Additional space $O(m)$
    - just need to store *fail* and indices

Clean Takeaway:
$\mathit{fail}[j)$ is the length of the longest prefix of $P[0..j)$ that is a suffix of $P[1..j]$

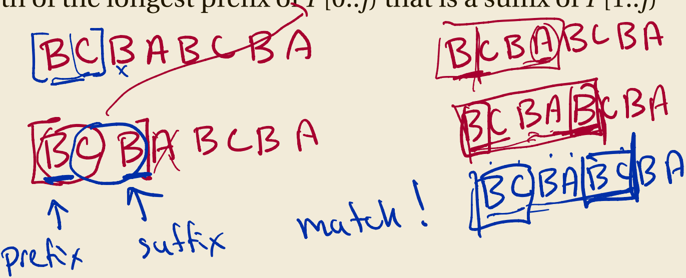
```
procedure KMP($T[0..n), P[0..m)$)
    $fail \leftarrow$ FAILURELINK($P$)
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $i < n$ do
        if $T[i] = P[q]$ then
            $i \leftarrow i+1, j \leftarrow j+1$
            if $j = m$ then return $i - j$
        else
            if $j \geq 1$ then
                $j \leftarrow fail[j]$
            else
                $i \leftarrow i+1$
            end if
        end if
    end while
end procedure
```

KMP Example

Example. Find the failure link array for $P[0,8) = \texttt{BCBABCBA}$.

i	0	1	2	3	4	5	6	7
$fail[i]$	0	0	0	1	0	1	2	3

$fail[j]$ is the length of the longest prefix of $P[0..j)$ that is a suffix of $P[1..j)$



KMP Example

Example. Find the failure link array for $P[0,8) =$ `BCBABCBA`.

i	0	1	2	3	4	5	6	7
$fail[i]$	0	0	0	1	0	1	2	3

KMP Example

Example. Find the failure link array for $P[0,8) = \texttt{BCBABCBA}$.

i	0	1	2	3	4	5	6	7
$fail[i]$	0	0	0	1	0	1	2	3

Interpretation. If $T[i..i+j)$ matches $P[0..j)$, but $T[i+j] \neq P[j]$, then $fail[j]$ is the maximum number matches between $T[i+1,i+j]$ and P.

1	2	3	4	5	6	7					
B	C	B	A	B	C	B	A				
		B	C	B	A	B	C	B	A		
				B	C	B	A	B	C	B	A

KMP Example

Example. Find the failure link array for $P[0,8) = \texttt{BCBABCBA}$.

i	0	1	2	3	4	5	6	7
$fail[i]$	0	0	0	1	0	1	2	3

Interpretation. If $T[i..i+j)$ matches $P[0..j)$, but $T[i+j] \neq P[j]$, then $fail[j]$ is the maximum number matches between $T[i+1,i+j]$ and P.

1	2	3	4	5	6	7					
B	C	B	A	B	C	B	A				
		B	C	B	A	B	C	B	A		
				B	C	B	A	B	C	B	A

Visualization. See website.

Spot the bug!

DFA vs FLA

Question. Which is better? DFA matching or KMP algorithm?

- KMP has overall running time $O(n+m)$ ✓
    - amortized 2 comparisons per T access
- DFA has overall running time $O(n+m|\Sigma|)$
    - 1 comparison per T access −
    - $|\Sigma|$ dependence −

Boyer-Moore

Beyond Worst-Case Pattern Matching?

A Puzzle. Suppose we have

- $P[0,4)$ = `AAAA`
- $T[0,14)$ = `BBBBBBBBBBBBBB`

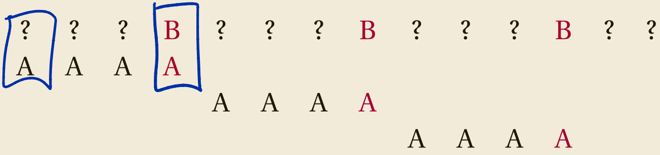
If we know P, what is the fewest number of accesses we can make to T to **certify** that T does not contain P?

Beyond Worst-Case Pattern Matching?

A Puzzle. Suppose we have

- $P[0,4) = \texttt{AAAA}$
- $T[0,14) = \texttt{BBBBBBBBBBBBBB}$

If we know P, what is the fewest number of accesses we can make to T to **certify** that T does not contain P?

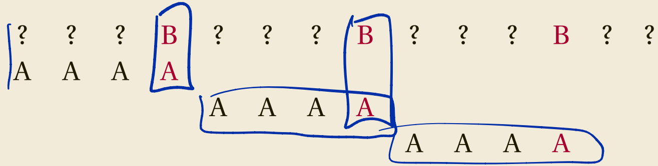


Beyond Worst-Case Pattern Matching?

A Puzzle. Suppose we have

- $P[0,4) =$ `AAAA`
- $T[0,14) =$ BBBBBBBBBBBBBB

If we know P, what is the fewest number of accesses we can make to T to **certify** that T does not contain P?



Observation.

- By starting comparisons from the *end* of P, we could eliminate more possible alignments.

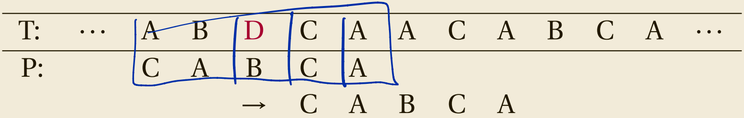
Two Heuristics

Strategy. To test match of $P[0..m)$ with $T[j..j+m)$, perform comparisons from *right to left*

Two Heuristics

Strategy. To test match of $P[0..m)$ with $T[j..j+m)$, perform comparisons from *right to left*

Heuristic 1. If we encounter $T[i]$ that does not occur in P, shift P entirely past index i.



T:	⋯	A	B	D	C	A	A	C	A	B	C	A	⋯
P:		C	A	B	C	A							
				→	C	A	B	C	A				

Two Heuristics

Strategy. To test match of $P[0..m)$ with $T[j..j+m)$, perform comparisons from *right to left*

Heuristic 1. If we encounter $T[i]$ that does not occur in P, shift P entirely past index i.

T:	⋯	A	B	D	C	A	A	C	A	B	C	A	⋯
P:		C	A	B	C	A							
				→	C	A	B	C	A				

Heuristic 2. If we match on a suffix of P but mismatch at index i, shift P to next alignment of suffix.

T:	⋯	A	B	D	C	A	A	C	A	B	C	A	⋯
P:					C	A	B	C	A				
							→	C	A	B	C	A	

Boyer-Moore Algorithm

Combining these heuristics gives the **Boyer-Moore algorithm**

- Compare alignments from right to left
- If we encounter $T[i]$ that does not occur in P, shift P entirely past index i.
- If we match on a suffix of P but mismatch at index i, shift P to next alignment of suffix

T:	⋯	A	B	D	C	A	A	C	A	B	C	A	⋯
P:		C	A	B	C	A							
				→	C	A	B	C	A				

T:	⋯	A	B	D	C	A	A	C	A	B	C	A	⋯
P:					C	A	B	C	A				
							→	C	A	B	C	A	

Boyer-Moore Algorithm

Combining these heuristics gives the **Boyer-Moore algorithm**

- Compare alignments from right to left
- If we encounter $T[i]$ that does not occur in P, shift P entirely past index i.
- If we match on a suffix of P but mismatch at index i, shift P to next alignment of suffix

Features of this approach:

- Worst-case running time on $P[0..m)$ and $T[0..n)$ is $\boxed{\Theta(nm)}$
  - achieved if all instances of P must be reported
  - can be improved to $\underline{\Theta(n+m+|\Sigma|)}$ with some care if T does not contain P

Boyer-Moore Algorithm

Combining these heuristics gives the **Boyer-Moore algorithm**

- Compare alignments from right to left
- If we encounter $T[i]$ that does not occur in P, shift P entirely past index i.
- If we match on a suffix of P but mismatch at index i, shift P to next alignment of suffix

Features of this approach:

- Worst-case running time on $P[0..m)$ and $T[0..n)$ is $\Theta(nm)$
    - achieved if all instances of P must be reported
    - can be improved to $\Theta(n + m + |\Sigma|)$ with some care if T does not contain P
- Typical running time can be much better!
    - For some *random* string models, expected running time is $\boxed{O(n/m)}$
    - For English text, typically uses $\sim \underline{0.25n}$ comparisons if no match

Boyer-Moore Algorithm

Combining these heuristics gives the **Boyer-Moore algorithm**

- Compare alignments from right to left
- If we encounter $T[i]$ that does not occur in P, shift P entirely past index i.
- If we match on a suffix of P but mismatch at index i, shift P to next alignment of suffix

Features of this approach:

- Worst-case running time on $P[0..m)$ and $T[0..n)$ is $\Theta(nm)$
  - achieved if all instances of P must be reported
  - can be improved to $\Theta(n + m + |\Sigma|)$ with some care if T does not contain P
- Typical running time can be much better!
  - For some *random* string models, expected running time is $O(n/m)$
  - For English text, typically uses $\sim 0.25n$ comparisons if no match
- Space overhead is $\Theta(m + |\Sigma|)$ ←

Summary of String Matching

- **Brute Force**:
  - simplest description
  - $\Theta(nm)$ running time
  - $O(1)$ space overhead

Summary of String Matching

- **Brute Force**:
    - simplest description
    - $\Theta(nm)$ running time
    - $O(1)$ space overhead
- **DFA**
    - few comparisons (worst case)
    - $\Theta(n+m|\Sigma|)$ running time
    - $\Theta(m|\Sigma|)$ space overhead (DFA table)

Summary of String Matching

- **Brute Force**:
  - simplest description
  - $\Theta(nm)$ running time
  - $O(1)$ space overhead
- **DFA**
  - few comparisons (worst case)
  - $\Theta(n + m|\Sigma|)$ running time
  - $\Theta(m|\Sigma|)$ space overhead (DFA table)
- **Knuth-Morris-Pratt**
  - simple description
  - $\Theta(n + m)$ running time (inc. all occurrences)
  - $\Theta(m)$ space overhead (fail array)

Summary of String Matching

- **Brute Force**:
  - simplest description
  - $\Theta(nm)$ running time
  - $O(1)$ space overhead
- **DFA**
  - few comparisons (worst case)
  - $\Theta(n + m|\Sigma|)$ running time
  - $\Theta(m|\Sigma|)$ space overhead (DFA table)
- **Knuth-Morris-Pratt**
  - simple description
  - $\Theta(n + m)$ running time (inc. all occurrences)
  - $\Theta(m)$ space overhead (fail array)
- **Boyer-Moore**
  - efficient in practice (English text)
  - $\Theta(nm)$ worst case to find all occurrences, can be as small as $O(\underline{n/m})$
  - $\underline{\Theta(m)}$ overhead $+ |\Sigma|$

Summary of String Matching

- **Brute Force**:
  - simplest description
  - $\Theta(nm)$ running time
  - $\boxed{O(1)}$ space overhead
- **DFA**
  - few comparisons (worst case)
  - $\Theta(n+m|\Sigma|)$ running time
  - $\Theta(m|\Sigma|)$ space overhead (DFA table)
- **Knuth-Morris-Pratt**
  - simple description
  - $\Theta(n+m)$ running time (inc. all occurrences)
  - $\Theta(m)$ space overhead (fail array)
- **Boyer-Moore**
  - efficient in practice (English text)
  - $\Theta(nm)$ worst case to find all occurrences, can be as small as $O(n/m)$
  - $\Theta(m)$ overhead
- **Rabin-Karp**
  - based on **hashing**
  - generalizes beyond one-dimensional strings
  - expected running time $\boxed{O(n+m)}$
  - $\boxed{O(1)}$ space overhead

Next Time

Data Compression!

- How much **space** do we need to store our data?

Scratch Notes