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
- $T[0...n-1]$ text (or haystack) being searched
- $P[0...m-1]$ pattern (or needle) being searched for
- Strings over alphabet $\Sigma$
- Return the first occurrence of $P$ in $T$, that is return smallest $i$ such that
  \[ P[j] = T[i+j] \text{ for } 0 \leq j \leq m-1 \]

- Example
  \[ T = \text{Little piglets cooked for mother pig} \]
  \[ P = \text{pig} \]
  \[ n = 36, \: m = 3, \: i = 7 \]

- If $P$ does not occur in $T$, return FAIL

- Applications
  - information retrieval (text editors, search engines)
  - bioinformatics, data mining
More Definitions [2]

antidisestablishmentarianism

- **Substring** $T[i...j]$  $0 \leq i \leq j < n$ is a string consisting of characters $T[i], T[i+1], ..., T[j]$
  - length is $j - i + 1$

- **Prefix** of $T$ is a substring $T[0...i]$ of $T$ for some $0 \leq i < n$

- **Suffix** of $T$ is a substring $T[i...n-1]$ of $T$ for some $0 \leq i \leq n - 1$
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
Brute-Force Example

Example: \( T = \text{abbbababbab}, \ P = \text{abba} \)

- Worst possible input
  - \( P = a \ldots ab, \ T = aaaaaaaa \ldots aaaaaa \)

- 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

```c
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

```c
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 $\Sigma$ then return -1
    if T[i + j] is after P[j] in $\Sigma$ 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 $R = 10$):
    $$h(P) = 59265 \mod 97 = 95$$

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>1</th>
<th>4</th>
<th>1</th>
<th>5</th>
<th>9</th>
<th>2</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash-value 84</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash-value 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash-value 76</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash-value 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash-value 95</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$h(31415) = 84$
$h(14159) = 94$
$h(41592) = 76$
$h(15926) = 18$
$h(59265) = 95$
# Karp-Rabin Fingerprint Algorithm – First Attempt

<table>
<thead>
<tr>
<th>Karp-Rabin-Simple::patternMatching($T, P$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_P \leftarrow h(P[0..m-1])$</td>
</tr>
<tr>
<td>for $i \leftarrow 0$ to $n - m$</td>
</tr>
<tr>
<td>$h_T \leftarrow h(T[i...i+m-1])$</td>
</tr>
<tr>
<td>if $h_T = h_P$</td>
</tr>
<tr>
<td>if strcmp($T[i...i+m-1], P$) = 0</td>
</tr>
<tr>
<td>return “found at guess $i$”</td>
</tr>
<tr>
<td>return FAIL</td>
</tr>
</tbody>
</table>

- Algorithm correctness: match is not missed
  - $h(T[i...i+m-1]) \neq h(P) \Rightarrow \text{guess } i \text{ is not } P$
- What about running time?
Karp-Rabin Fingerprint Algorithm: First Attempt

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>4</th>
<th>1</th>
<th>5</th>
<th>9</th>
<th>2</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(m)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Theta(m)$</td>
<td>hash-value 84</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Theta(m)$</td>
<td>hash-value 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Theta(m)$</td>
<td></td>
<td>hash-value 76</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Theta(m)$</td>
<td></td>
<td></td>
<td>hash-value 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Theta(m)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>hash-value 95</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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**(\(T, P\))

\[
\begin{align*}
    h_P & \leftarrow h(P[0..m-1]) \\
    \text{for } i & \leftarrow 0 \text{ to } n - m \\
    h_T & \leftarrow h(T[i...i+m-1]) \\
    \text{if } h_T = h_P \\
    & \text{if } 	ext{strcmp}(T[i...i+m-1], P) = 0 \\
    & \quad \text{return } \text{“found at guess } i\text{“} \\
\end{align*}
\]

return FAIL

- Algorithm correctness: match is not missed
  - \(h(T[i...i+m-1]) \neq h(P) \Rightarrow \text{guess } i \text{ is not } P\)
  - \(h(T[i...i+m-1])\) 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**

  $T = 415926535, \quad P = 5\ 9\ 2\ 6\ 5$

- 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

  \[
  41592 \quad \rightarrow \quad 1592 \quad \rightarrow \quad 15920 \quad \rightarrow \quad 15926
  \]

- Algebraically,

  \[
  (41592 - (4 \cdot 10000)) \cdot 10 + 6 = 15926
  \]
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**

  $T = 415926535$, $P = 59265$

  - 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 $41592 \mod 97$?

      $$
      \left(41592 - (4 \cdot 10000)\right) \cdot 10 + 6 = 15926
      \left(\left(41592 - (4 \cdot 10000)\right) \cdot 10 + 6\right) \mod 97 = 15926 \mod 97
      \left(\left(41592 \mod 97 - (4 \cdot 10000 \mod 97)\right) \cdot 10 + 6\right) \mod 97 = 15926 \mod 97
      \left(\left(76 - (4 \cdot 9)\right) \cdot 10 + 6\right) \mod 97 = 15926 \mod 97
      $$

      constant number of operations, independent of $m$
### Karp-Rabin Fingerprint Algorithm – Conclusion

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

- \(M \leftarrow \text{suitable prime number}\)
- \(h_P \leftarrow h(P[0\ldots m - 1])\)
- \(h_T \leftarrow h(T[0.. m - 1])\)
- \(s \leftarrow 10^{m-1} \mod M\)

```cpp
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
            \(h_T \leftarrow ((h_T - T[i] \cdot s) \cdot 10 + T[i + m]) \mod M\)
    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

\[ P = ababaca \]

- 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

\[ P = ababaca \]

\[
\begin{array}{cccccccc}
  j=0 & j=0 & j=1 & j=2 & j=3 & j=4 & j=5 \\
  i=0 & i=1 & i=2 & i=3 & i=4 & i=5 & i=6 \\
\end{array}
\]

\[ T \]
\[
\begin{array}{cccccccc}
  c & a & b & a & b & a & a & b & a & b \\
  a & & & & & & & & & \\
  a & b & a & b & a & c & & & & \\
\end{array}
\]

- 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

$P = \text{ababaca}$

When failure is at pattern position $j > 0$, do something smarter than brute force

Prior to $j = 5$, pattern and text are equal

- find how to shift pattern looking only at pattern
- 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

\( P = \text{ababaca} \)

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( i=0 )</td>
<td>( i=1 )</td>
<td>( i=2 )</td>
<td>( i=3 )</td>
<td>( i=4 )</td>
<td>( i=5 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( j=0 )</td>
<td>( j=1 )</td>
<td>( j=2 )</td>
<td>( j=3 )</td>
<td>( j=4 )</td>
<td>( j=5 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( T \)

\[
\begin{array}{cccccccc}
   & c & a & b & a & b & a & a & b & a \\
\hline
   a & & & & & & & & & \\
   & & & & & & & & & \\
   a & & b & a & b & a & c & & & \\
   a & & & & & & & & & \\
   a & b & a & & & & & & & \\
\end{array}
\]

- 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 \( P[1...j-1] \) 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 \( P[1...j-1] \)
Knuth-Morris-Pratt Motivation

$P = ababaca$

<table>
<thead>
<tr>
<th>$j=0$</th>
<th>$j=1$</th>
<th>$j=2$</th>
<th>$j=3$</th>
<th>$j=4$</th>
<th>$j=5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
<td>$i=1$</td>
<td>$i=2$</td>
<td>$i=3$</td>
<td>$i=4$</td>
<td>$i=5$</td>
</tr>
</tbody>
</table>

\[ \begin{array}{cccccccc}
T & c & a & b & a & b & a & a & b \\
\hline
a & & & & & & & & \\
\hline
a & b & a & b & a & c & & & \\
\hline
| & a & b & a & & & & & \\
\hline
\end{array} \]

- 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 $j = \text{the length of the longest valid suffix of } P[1 \ldots j - 1]$
KMP Failure Array Computation: Slow

- **Rule**: if failure at pattern index $j > 0$, continue matching with the same $i$ and new $j = \text{the length of the longest valid suffix of } P[1 \ldots j-1]$
- Computed previously for $j = 5$, but need to compute for all $j$
- Store this information in array $F[0 \ldots m-1]$, called **failure-function**
  - $F[j]$ is length of the longest valid suffix of $P[1 \ldots j]$
  - if failure at pattern index $j > 0$, new $j = F[j-1]$
- $P = ababaca$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $j = 0$
  - $P[1 \ldots 0] = "", P = ababaca$, 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 $j = F[j-1]$

<table>
<thead>
<tr>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6</td>
</tr>
<tr>
<td>0 0 1 2 3 0 1</td>
</tr>
</tbody>
</table>

- $j = 4$
  - $P[1...4] = baba$, $P = ababaca$, longest valid suffix is *aba*

- $j = 5$
  - $P[1...5] = babac$, $P = ababaca$, longest valid suffix is “”

- $j = 6$
  - $P[1...6] = babaca$, $P = ababaca$, 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 $P[1...j]$
  
  for $k = 0$ to $i$ // compare next suffix to prefix of $P$
String matching with KMP: Example

- \( T = \text{cabababcababaca}, P = \text{ababaca} \)

\[ \begin{array}{cccccccccccccccc}
T: & c & a & b & a & b & a & b & c & a & b & a & b & a & c & a \\
i=0 & j=0 & F & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 0 & 0 & 1 & 2 & 3 & 0 & 1 \\
\end{array} \]

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

- \( T = cabababcababaca, P = ababaca \)

<table>
<thead>
<tr>
<th>( T )</th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P )</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( j = 3 \quad j = 2 \)

\( j = 0 \quad j = 1 \quad j = 2 \quad j = 3 \quad j = 4 \quad j = 5 \quad j = 6 \quad j = 10 \quad j = 11 \quad j = 12 \quad j = 13 \quad j = 14 \)

- \( T[i] = P[j] \):
  - \( i = i + 1 \)
  - \( j = j + 1 \)
- \( T[i] \neq P[j] \) and \( j > 0 \):
  - \( i \) unchanged
  - \( j = F[j - 1] \)
- \( T[i] \neq P[j] \) and \( j = 0 \):
  - \( i = i + 1 \)
  - \( j \) is unchanged

\[
F = \begin{pmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
0 & 0 & 1 & 2 & 3 & 0 & 1 \\
\end{pmatrix}
\]
Knuth-Morris-Pratt Algorithm

\[ \text{KMP}(T, P) \]

\[
F \leftarrow \text{failureArray}(P) \\
i \leftarrow 0 \quad // \text{current character of } T \\
j \leftarrow 0 \quad // \text{current character of } P \\
\text{while } i < n \text{ do} \\
\quad \text{if } P[j] = T[i] \\
\quad \quad \text{if } j = m - 1 \\
\quad \quad \quad \text{return} \text{ “found at guess } i - m + 1\text{”} \\
\quad \quad \quad \text{// location } i \text{ in } T \text{ is the end of matched } P \text{ in text} \\
\quad \quad \text{else // rule 1} \\
\quad \quad i \leftarrow i + 1 \\
\quad \quad j \leftarrow j + 1 \\
\quad \text{else // } P[j] \neq T[i] \\
\quad \quad \text{if } j > 0 \\
\quad \quad \quad j \leftarrow F[j - 1] \quad // \text{rule 2} \\
\quad \quad \text{else // rule 3} \\
\quad \quad \quad i \leftarrow i + 1 \\
\text{return } FAIL\]
**KMP: Time Complexity, informally**

- 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

\[
\begin{array}{cccccccccc}
T: & c & a & b & a & b & a & b & c & a & b & a & b & a & c & a \\
\hline
\end{array}
\]

- 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
KMP: Running Time, informally

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 $F$

- $P = ababaca$

<table>
<thead>
<tr>
<th>$j=0$</th>
<th>$j=1$</th>
<th>$j=2$</th>
<th>$j=3$</th>
<th>$j=4$</th>
<th>$j=5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 $F$

- $P = ababaca$

- 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 $T$

- $F[j] = \text{length of the longest valid suffix of } P[1\ldots j]$
  - need to compute $F[j]$ for $0 < j < m$
    - $F[0] = 0$, no need to compute

- Big idea:
  - $T = P[1\ldots 1] \xrightarrow{\text{KMP}} F[1] = l$
  - $T = P[1\ldots 2] \xrightarrow{\text{KMP}} F[2] = l$
    - $\vdots$
  - $T = P[1\ldots m-1] \xrightarrow{\text{KMP}} F[m-1] = l$

    "chicken and egg" problem with big idea: need $F$ to put text through KMP
Fast Computation of $F$: Big Idea Saved

- $j = 1$
  \[ T = P[1 \ldots 1] \rightarrow \text{KMP} \rightarrow \text{final } l \quad F[1] = l \]
  - start with $l = 0$
  - text has one letter, can reach at most $l = 1$
  - need at most $F[0]$, and already have it

- $j = 2$
  \[ T = P[1 \ldots 2] \rightarrow \text{KMP} \rightarrow \text{final } l \quad F[2] = l \]
  - 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

- $j = m - 1$
  \[ T = P[1 \ldots m - 1] \rightarrow \text{KMP} \rightarrow \text{final } l \quad F[m - 1] = l \]
  - start with $l = 0$
  - text has $m - 1$ letters, can reach at most $l = m - 1$
  - need at most $F[0], F[1], \ldots, F[m - 2]$, and already have it
Fast Computation of $F$: Big Idea Made Bigger

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

- Process $T = P[1 \ldots j]$, $F[j] = \text{final } l$
- $P = ababaca$
- Initialize $F[0] = 0$
Fast Computation of $F$

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

<table>
<thead>
<tr>
<th>$l$</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T$:</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$P$:</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 $F$

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

<table>
<thead>
<tr>
<th>$l$</th>
<th>$i$</th>
<th>$a_l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>if $T[i] = P[l]$</th>
<th>if $T[i] \neq P[l]$ and $l &gt; 0$</th>
<th>if $T[i] \neq P[l]$ and $l = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = i + 1$</td>
<td>$i$ unchanged</td>
<td>$i = i + 1$</td>
</tr>
<tr>
<td>$l = l + 1$</td>
<td>$l = F[l - 1]$</td>
<td>$l$ is unchanged</td>
</tr>
</tbody>
</table>
Fast Computation of $F$

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

<table>
<thead>
<tr>
<th>$l=0$</th>
<th>$l=0$</th>
<th>$l=1$</th>
<th>$l=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
<td>$i=1$</td>
<td>$i=2$</td>
<td>$i=3$</td>
</tr>
</tbody>
</table>

$T$: b a b

$P$: a

$P$: a b

<table>
<thead>
<tr>
<th>$i$ if $T[i] = P[l]$</th>
<th>$i$ if $T[i] \neq P[l]$ and $l &gt; 0$</th>
<th>$i$ if $T[i] \neq P[l]$ and $l = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = i + 1$</td>
<td>$i$ unchanged</td>
<td>$i = i + 1$</td>
</tr>
<tr>
<td>$l = l + 1$</td>
<td>$l = F[l - 1]$</td>
<td>$l$ is unchanged</td>
</tr>
</tbody>
</table>
Fast Computation of $F$

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

<table>
<thead>
<tr>
<th></th>
<th>$l=0$</th>
<th>$l=1$</th>
<th>$l=2$</th>
<th>$l=3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
<td>$T$: b a b a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=1$</td>
<td>$P$: a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=2$</td>
<td>a b a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$:</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Fast Computation of $F$**

- Process $T = P[1 \ldots j]$, $F[j] = \text{final } l$
- $P = a babaca$
- $j = 5$, $T = P[1 \ldots j] = babac$

<table>
<thead>
<tr>
<th>$T$</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$l=0$</th>
<th>$l=1$</th>
<th>$l=2$</th>
<th>$l=3$</th>
<th>$l=0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
<td>$i=1$</td>
<td>$i=2$</td>
<td>$i=3$</td>
<td>$i=4$</td>
</tr>
</tbody>
</table>

- $F$:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

New $l = 1$

New $l = 0$

- 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 $F$

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

<table>
<thead>
<tr>
<th>$l=0$</th>
<th>$l=1$</th>
<th>$l=2$</th>
<th>$l=3$</th>
<th>$l=4$</th>
<th>$l=5$</th>
<th>$l=6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
<td>$i=1$</td>
<td>$i=2$</td>
<td>$i=3$</td>
<td>$i=4$</td>
<td>$i=5$</td>
<td>$i=6$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T$:</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$:</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
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)
  \(F[0] \leftarrow 0\)
  \(j \leftarrow 1 // parsing P[1 ... j]\)
  \(l \leftarrow 0\)
  \(\textbf{while } j < m \textbf{ do}\)
  \(\text{if } P[j] = P[l]\)
  \(\quad l \leftarrow l + 1\)
  \(\quad F[j] \leftarrow l\)
  \(\quad j \leftarrow j + 1\)
  \(\text{else if } l > 0\)
  \(\quad l \leftarrow F[l - 1]\)
  \(\text{else}\)
  \(\quad F[j] \leftarrow 0\)
  \(\quad j \leftarrow j + 1\)
```
KMP: main function runtime

- **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: $\Theta(n + m)$

```
KMP(T, P)
    F ← failureArray (P)
    i ← 0
    j ← 0
    while i < n do
        if P[j] = T[i]
            if j = m − 1
                return “found at guess i − m + 1”
            else
                i ← i + 1
                j ← j + 1
        else // P[j] ≠ T[i]
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return FAIL
```
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

$T = \text{whereiswaldo}$, $P = \text{aldo}$

- $r$ does not occur in $P = \text{aldo}$
- shift pattern past $r$
- $w$ does not occur in $P = \text{aldo}$
- shift pattern past $w$
- this bad character heuristic works well with reverse searching

- $w$ does not occur in $P = \text{aldo}$
- 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 $P$

\[ T = \text{acranapple}, \quad P = \text{aaron} \]

- 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 $P$

$$T = \text{acranapple}, \ P = \text{aaron}$$

- 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

\(j=3\)
\(i=3\)

\(j=4\)
\(i=6\)

<table>
<thead>
<tr>
<th>a</th>
<th>c</th>
<th>r</th>
<th>a</th>
<th>n</th>
<th>a</th>
<th>p</th>
<th>p</th>
<th>l</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>o</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Let \(L(c)\) be the last occurrence of character \(c\) in \(P\)
  - \(L(a) = 1\) 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

-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 \( P \)

\[
i^{\text{new}} - (m - 1) + L(c) = i^{\text{old}}
\]

\[
i^{\text{new}} = i^{\text{old}} + m - 1 - L(c)
\]

\[
i = i + m - 1 - L(c)
\]
Bad Character Heuristic, Last detail

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

$$T = \text{acraaapple}, \quad P = \text{aaroa}$$

$$j = 3 \quad i = 3$$

<table>
<thead>
<tr>
<th>a</th>
<th>c</th>
<th>r</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>p</th>
<th>p</th>
<th>l</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>o</td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $i = i + m - 1 - L(c)$
  - $L(a) = 4 > j = 3$
  - $i = 3 + 4 - 4 = 3$
- 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\((T, P)\)

\[ L \leftarrow \text{last occurrence array computed from } P \]

\[ j \leftarrow m - 1 \]

\[ i \leftarrow m - 1 \]

\textbf{while } i < n \text{ and } j \geq 0 \textbf{ do}

\textbf{if } T[i] = P[j] \textbf{ then}

\[ i \leftarrow i - 1 \]

\[ j \leftarrow j - 1 \]

\textbf{else}

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

\[ j \leftarrow m - 1 \]

\textbf{if } j = -1 \textbf{ return } i + 1

\textbf{else } \textbf{return } \text{FAIL}
**Good Suffix Heuristic**

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

\[ P = \text{onobobo} \]

\[
\begin{array}{cccccccccccccccc}
\hline
T & o & n & o & o & o & b & o & o & o & i & b & b & o & u & n & d & a & r & y \\
\hline
& & & & & & b & o & b & o & & & & & & & & & & & \\
& & o & n & o & b & o & b & o & & & & & & & & & & & & \\
\hline
\end{array}
\]

- 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 $\approx 25\%$ 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: 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 $T$

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**

- $T = \text{bananaban}$
- 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

\[ T = \begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{b} & \text{a} & \text{n} & \text{a} & \text{n} & \text{a} & \text{b} & \text{a} & \text{n} & \$
\end{array} \]
Trie of suffixes: Example

- Store suffixes via indices

```
T = b a n a n a b a n $
```

```
T[5..9]
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T[5..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T[5..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```

```
T = a b a n
```

```
T[9..9]
```
Tries of suffixes

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

$$T = \begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
& b & a & n & a & n & a & b & a & n & $\
\end{array}$$
Suffix tree

- **Suffix tree**: compressed trie of suffixes

\[
T = \begin{array}{c}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\begin{array}{c}
b & a & n & a & n & a & b & a & n & \$
\end{array}
\]

\[T = \]

```
0
  \_1
    \_b
      \_n
        \_\$
          \_a
            \_b
              \_n
                \_T[9..9]
                  \_T[5..9]
                    \_T[7..9]
                      \_T[3..9]
                        \_T[1..9]
                          \_T[6..9]
                            \_T[0..9]
                              \_T[2..9]
                                \_T[4..9]
                                  \_T[8..9]
```


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

### Given String:

$$T = \text{bananaban}$$

### Suffix Array Calculation:  

1. **suffixes** of **$T$**:
   - **0**: bananaban$ 
   - **1**: ananaban$ 
   - **2**: nanaban$ 
   - **3**: anaban$ 
   - **4**: naban$ 
   - **5**: aban$ 
   - **6**: ban$ 
   - **7**: an$ 
   - **8**: n$ 
   - **9**: $ 

2. **Sort lexicographically**:

<table>
<thead>
<tr>
<th>$i$</th>
<th>suffix $T[i...n]$</th>
<th>$j$</th>
<th>$A^s[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bananaban$</td>
<td>0</td>
<td>9 $</td>
</tr>
<tr>
<td>1</td>
<td>ananaban$</td>
<td>1</td>
<td>5 aban$</td>
</tr>
<tr>
<td>2</td>
<td>nanaban$</td>
<td>2</td>
<td>7 an$</td>
</tr>
<tr>
<td>3</td>
<td>anaban$</td>
<td>3</td>
<td>3 anaban$</td>
</tr>
<tr>
<td>4</td>
<td>naban$</td>
<td>4</td>
<td>1 ananaban$</td>
</tr>
<tr>
<td>5</td>
<td>aban$</td>
<td>5</td>
<td>6 ban$</td>
</tr>
<tr>
<td>6</td>
<td>ban$</td>
<td>6</td>
<td>0 bananaban$</td>
</tr>
<tr>
<td>7</td>
<td>an$</td>
<td>7</td>
<td>8 n$</td>
</tr>
<tr>
<td>8</td>
<td>n$</td>
<td>8</td>
<td>4 naban$</td>
</tr>
<tr>
<td>9</td>
<td>$</td>
<td>9</td>
<td>2 nanaban$</td>
</tr>
</tbody>
</table>

3. **Suffix Array**:

$$A^s = [9, 5, 7, 3, 1, 6, 0, 8, 4, 2]$$
Suffix Array Example

\[
T = \begin{array}{c}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} & \\
\text{bananaban$} & \text{ananaban$} & \text{nanaban$} & \text{anaban$} & \text{naban$} & \text{aban$} & \text{ban$} & \text{an$} & \text{n$} &
\end{array}
\]

Suffix Array Example:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|}
\text{i} & \text{suffix } T[i \ldots n] & \text{j} & A^s[j] \\
\hline
0 & \text{bananaban$} & 0 & 9 & $ \\
1 & \text{ananaban$} & 1 & 5 & \text{aban$} \\
2 & \text{nanaban$} & 2 & 7 & \text{an$} \\
3 & \text{anaban$} & 3 & 3 & \text{anaban$} \\
4 & \text{naban$} & 4 & 1 & \text{ananaban$} \\
5 & \text{aban$} & 5 & 6 & \text{ban$} \\
6 & \text{ban$} & 6 & 0 & \text{bananaban$} \\
7 & \text{an$} & 7 & 8 & \text{n$} \\
8 & \text{n$} & 8 & 4 & \text{naban$} \\
9 & $ & 9 & 2 & \text{nanaban$} \\
\end{array}
\]

Sort lexicographically:

Suffix Array = [9, 5, 7, 3, 1, 6, 0, 8, 4, 2]
Suffix Array Construction

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

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

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

round n
$********
aban$*****
ananaban$
anaban$**
an$*******
bananaban$
ban$******
nanaban$**
naban$*****
n$********
```

- 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

<table>
<thead>
<tr>
<th>l</th>
<th>j</th>
<th>$A^s[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>$$</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>aban$</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>an$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>anaban$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>ananaban$</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>ban$</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>bananaban$</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>n$</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>naban$</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>nanaban$</td>
</tr>
</tbody>
</table>
Pattern Matching in Suffix Arrays

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

\[ P = \text{ban} \]

<table>
<thead>
<tr>
<th>( j )</th>
<th>( A^s[j] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>
Pattern Matching in Suffix Arrays

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

\[
P = \text{ban}
\]

<table>
<thead>
<tr>
<th>(j)</th>
<th>(A^s[j])</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

- \(\Theta(\log n)\) comparisons
- Each comparison is `strcmp(P, T[A^s[v] \ldots A^s[v + m - 1]]))`
- \(\Theta(m)\) per comparison \(\Rightarrow\) run-time is \(\Theta(m \log n)\)
Pattern Matching in Suffix Arrays

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

\textbf{A}^s: suffix array of \textbf{T}, \textbf{P}: pattern

\begin{align*}
l & \leftarrow 0, r \leftarrow n - 1 \\
\textbf{while } l < r \textbf{ do} \\
& v \leftarrow \left\lceil \frac{l+r}{2} \right\rceil \\
& i \leftarrow A^s[v] \\
& \text{ // assume \texttt{strcmp} handles out of bounds suitably} \\
& s \leftarrow \texttt{strcmp}(T[i \ldots i + m - 1], P) \\
& \textbf{if } (s < 0) \textbf{ do } l \leftarrow v + 1 \\
& \textbf{else } (s > 0) \textbf{ do } r \leftarrow v - 1 \\
& \textbf{else return}\ ‘found at guess } T[i \ldots i + m - 1]' \\
& \textbf{if } \texttt{strcmp}(P, T[A^s[l], A^s[l] + m - 1]]) \\
& \texttt{return} ‘found at guess } T[l \ldots l + m - 1]' \\
\texttt{returnFAIL}
\end{align*}
Outline

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

<table>
<thead>
<tr>
<th></th>
<th>Brute Force</th>
<th>KR</th>
<th>BM</th>
<th>KMP</th>
<th>Suffix Trees</th>
<th>Suffix Array</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>preproc.</strong></td>
<td></td>
<td>—</td>
<td>$O(m)$</td>
<td>$O(m)$</td>
<td>$O(</td>
<td>\Sigma</td>
</tr>
<tr>
<td><strong>search time</strong></td>
<td>$O(nm)$</td>
<td>$O(n + m)$</td>
<td>$O(n)$ often</td>
<td>$O(n)$</td>
<td>$O(m)$</td>
<td>$O(m\log n)$</td>
</tr>
<tr>
<td><strong>excluded</strong></td>
<td></td>
<td>—</td>
<td>$O(1)$</td>
<td>$O(m)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

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