Advanced Data Structures

String Pattern Matching/Text Search
What is Pattern Matching?

• Definition:
  – given a text string $T$ and a pattern string $P$, find the pattern inside the text
    • $T$: “the rain in spain stays mainly on the plain”
    • $P$: “n th”
Text search

- Pattern matching directly
  - Brute force
  - KMP
  - BM
- Regular expressions (Not in this course)
- Indices for pattern matching
  - Inverted files
  - Signature files
  - Suffix trees and Suffix arrays
The Brute Force Algorithm

• Check each position in the text T to see if the pattern P starts in that position

T: andrew  T: andrew
P: rew    P: rew

P moves 1 char at a time through T
Analysis

• Brute force pattern matching runs in time $O(mn)$ in the worst case.

• But most searches of ordinary text take $O(m+n)$, which is very quick.
• The brute force algorithm is fast when the alphabet of the text is large
  – e.g. A..Z, a..z, 1..9, etc.

• It is slower when the alphabet is small
  – e.g. 0, 1 (as in binary files, image files, etc.)
• Example of a worst case:
  – T: "aaaaaaaaaaaaaaaaaaaaaaaaaaaaah"
  – P: "aaah"

• Example of a more average case:
  – T: "a string searching example is standard"
  – P: "store"
The KMP Algorithm

• The Knuth-Morris-Pratt (KMP) algorithm looks for the pattern in the text in a left-to-right order (like the brute force algorithm).

• But it shifts the pattern more intelligently than the brute force algorithm.
Summary

- If a mismatch occurs between the text and pattern $P$ at $P[j]$, what is the *most* we can shift the pattern to avoid wasteful comparisons?
- If a mismatch occurs between the text and pattern P at P[j], what is the *most* we can shift the pattern to avoid wasteful comparisons?

- *Answer*: the largest prefix of P[0 .. j-1] that is a suffix of P[1 .. j-1]
Example

T:   a b a c a  a b a c c  a b a c a  b a a b b

1 2 3 4 5 6
P:   a b a c a  a b

7
a b a c a b

8 9 10 11 12
a b a c a b

13
a b a c a b

14 15 16 17 18 19
a b a c a b

<table>
<thead>
<tr>
<th>k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>F(k)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
KMP Advantages

• KMP runs in optimal time: $O(m+n)$
  – very fast

• The algorithm never needs to move backwards in the input text, $T$
  – this makes the algorithm good for processing very large files that are read in from external devices or through a network stream
KMP Disadvantages

• KMP doesn’t work so well as the size of the alphabet increases
  – more chance of a mismatch (more possible mismatches)
  – mismatches tend to occur early in the pattern, but KMP is faster when the mismatches occur later
Boyer and Moore Algorithm


BOYER, R.S. and MOORE, J.S.
Boyer and Moore Algorithm

• The algorithm compares the pattern $P$ with the substring of sequence $T$ within a sliding window in the right-to-left order.

• The **bad character rule** and **good suffix rule** are used to determine the movement of sliding window.
Bad Character Rule

Suppose that $P_1$ is aligned to $T_s$ now, and we perform a pairwise comparing between text $T$ and pattern $P$ from right to left. Assume that the first mismatch occurs when comparing $T_{s+j-1}$ with $P_j$.

Since $T_{s+j-1} \neq P_j$, we move the pattern $P$ to the right such that the largest position $c$ in the left of $P_j$ is equal to $T_{s+j-1}$. We can shift the pattern at least $(j-c)$ positions right.
Character Matching Rule

• Bad character rule uses Rule 2-1 (Character Matching Rule).

• For any character \( x \) in \( T \), find the nearest \( x \) in \( P \) which is to the left of \( x \) in \( T \).
Implication

• Case 1. If there is a $x$ in $P$ to the left of $T$, move $P$ so that the two $x$’s match.
• Case 2: If no such a $x$ exists in $P$, consider the partial window defined by $x$ in $T$ and the string to the left of it.
• Ex: Suppose that $P_1$ is aligned to $T_6$ now. We compare pair-wise between $T$ and $P$ from right to left. Since $T_{16,17} = P_{11,12} = \text{“CA”}$ and $T_{15} = \text{“G”} \neq P_{10} = \text{“T”}$. Therefore, we find the rightmost position $c=7$ in the left of $P_{10}$ in $P$ such that $P_c$ is equal to “G” and we can move the window at least $(10-7=3)$ positions.

![Diagram showing the alignment process between T and P, highlighting the mismatch at position 15 and the new alignment position 7.]

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array}
\]
Good Suffix Rule 1

- If a mismatch occurs in $T_{s+j-1}$, we match $T_{s+j-1}$ with $P_{j'-m+j}$, where $j' \ (m-j+1 \leq j' < m)$ is the **largest position** such that
  
  1. $P_{j+1,m}$ is a suffix of $P_{1,j'}$
  2. $P_{j'-(m-j)} \neq P_{j}$

- We can move the window at least $(m-j')$ position(s).
Rule: The Substring Matching Rule

- For any substring $u$ in $T$, find a nearest $u$ in $P$ which is to the left of it. If such a $u$ in $P$ exists, move $P$; otherwise, we may define a new partial window.
Ex: Suppose that \( P_1 \) is aligned to \( T_6 \) now. We compare pairwise between \( P \) and \( T \) from right to left. Since \( T_{16,17} = \text{"CA"} = P_{11,12} \) and \( T_{15} = \text{"A"} \neq P_{10} = \text{"T"} \). We find the substring "CA" in the left of \( P_{10} \) in \( P \) such that "CA" is the suffix of \( P_{1,6} \) and the left character to this substring "CA" in \( P \) is not equal to \( P_{10} = \text{"T"} \). Therefore, we can move the window at least \( m-j' \) (12-6=6) positions right.

\[
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\end{array}
\]
Good Suffix Rule 2

Good Suffix Rule 2 is used only when Good Suffix Rule 1 cannot be used. That is, $t$ does not appear in $P(1, j)$. Thus, $t$ is unique in $P$.

- If a mismatch occurs in $T_{s+j-1}$, we match $T_{s+m-j'}$ with $P_1$, where $j'$ (1 $\leq j' \leq m-j$) is the largest position such that $P_{1,j'}$ is a suffix of $P_{j+1,m}$.

P.S. : $t'$ is suffix of substring $t$. 

$P_1,j'$ is a suffix of $P_{j+1,m}$. 
Rule: Unique Substring Rule

- The substring $u$ appears in $P$ exactly once.
- If the substring $u$ matches with $T_{i,j}$, no matter whether a mismatch occurs in some position of $P$ or not, we can slide the window by $l$.

The string $s$ is the longest prefix of $P$ which equals to a suffix of $u$. 
The Suffix to Prefix Rule

• For a window to have any chance to match a pattern, in some way, there must be a suffix of the window which is equal to a prefix of the pattern.

```
T                          P
```

T: window

P: pattern

The suffix to prefix rule illustrates the condition necessary for a window to potentially match a pattern.
• Note that the above rule also uses Rule 1.
• It should also be noted that the unique substring is the shorter and the more right-sided the better.
• A short $u$ guarantees a short (or even empty) $s$ which is desirable.
• Ex: Suppose that $P_1$ is aligned to $T_6$ now. We compare pair-wise between $P$ and $T$ from right to left. Since $T_{12} \neq P_7$ and there is no substring $P_{8,12}$ in left of $P_8$ to exactly match $T_{13,17}$. We find a longest suffix “AATC” of substring $T_{13,17}$, the longest suffix is also prefix of $P$. We shift the window such that the last character of prefix substring to match the last character of the suffix substring. Therefore, we can shift at least $12-4=8$ positions.
• Let $Bc(a)$ be the rightmost position of $a$ in $P$. The function will be used for applying *bad character rule*.

<table>
<thead>
<tr>
<th>$j$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>12</td>
<td>11</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

• We can move our pattern right $j-B(T_{s+j-1})$ position by above $Bc$ function.

Move $10-B(G) = 10$ positions
Let $G_s(j)$ be the largest number of shifts by *good suffix rule* when a mismatch occurs for comparing $P_j$ with some character in $T$. 
• \(gs_1(j)\) be the largest \(k\) such that \(P_{j+1,m}\) is a suffix of \(P_{1,k}\) and \(P_{k-m+j} \neq P_j\), where \(m-j+1 \leq k < m\); 0 if there is no such \(k\).

\((gs_1\) is for Good Suffix Rule 1)\\

• \(gs_2(j)\) be the largest \(k\) such that \(P_{1,k}\) is a suffix of \(P_{j+1,m}\), where \(1 \leq k \leq m-j\); 0 if there is no such \(k\).

\((gs_2\) is for Good Suffix Rule 2.)\\

• \(Gs(j) = m - \max\{gs_1, gs_2\}\), if \(j = m\), \(Gs(j)=1\).\\

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P)</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>(gs_1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(gs_2)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(Gs)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>11</td>
<td>6</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

\(gs_1(7)=9\)\\
\(\therefore P_{8,12}\) is a suffix of \(P_{1,9}\) and \(P_4 \neq P_7\)\\
\(gs_2(7)=4\)\\
\(\therefore P_{1,4}\) is a suffix of \(P_{8,12}\).
How do we obtain $gs_1$ and $gs_2$?

In the following, we shall show that by constructing the **Suffix Function**, we can kill two birds with one arrow.
Suffix function $f'$

- For $1 \leq j \leq m-1$, let the suffix function $f'(j)$ for $P_j$ be the smallest $k$ such that $P_{k,m} = P_{j+1,m-k+j+1}$; $(j+2 \leq k \leq m)$
  - If there is no such $k$, we set $f' = m+1$.
  - If $j=m$, we set $f'(m)=m+2$.

- Ex:

\[
P = \begin{array}{cccccccccccc}
\text{t} & \text{t} & \text{t} & \text{t} & \text{t} & \text{t} & \text{t} & \text{t} & \text{t} & \text{t} & \text{t} & \text{t} \\
\end{array}
\]

- $f'(4)=8$, it means that $P_{f'(4),m} = P_{8,12} = P_{5,9} = P_{4+1,4+1+m-f'(4)}$.
- Since there is no $k$ for $13 = j+2 \leq k \leq 12$, we set $f'(11)=13$. 

<table>
<thead>
<tr>
<th>$j$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>$f'$</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>
Suppose that the Suffix is obtained. How can we use it to obtain $gs_1$ and $gs_2$?

$gs_1$ can be obtained by scanning the Suffix function from right to left.
Example

<table>
<thead>
<tr>
<th>j</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>f'</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

T  | G | A | T | C | G | A | T | C | A | A | T | C | A | C | A | T | G | A | T | C | A

P  | A | T | C | A | C | A | T | C | A | C | A | T | C | A | A | T | G | A | T | C | A

Example
Example

As for Good Suffix Rule 2, it is relatively easier.

<table>
<thead>
<tr>
<th>$j$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>$f'$</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>
Question: How can we construct the Suffix function?

To explain this, let us go back to the prefix function used in the KMP Algorithm.
The following figure illustrates the prefix function in the KMP Algorithm.

P

The following figure illustrates the suffix function of the BM Algorithm.

P
We now can see that actually the suffix function is the same as the prefix. The only difference is now we consider a suffix. Thus, the recursive formula for the prefix function in KMP Algorithm can be slightly modified for the suffix function in BM Algorithm.
The formula of suffix function $f''$ as follows:

Let $f''^x (y) = f'( f''^{x-1} (y))$ for $x > 1$ and $f''^1 (y) = f' (y)$

$$f''(j) = \begin{cases} 
m + 2, & \text{if } j = m \\
f''^k (j + 1) - 1, & \text{if } 1 \leq j \leq m - 1 \text{ and there exists the smallest } k \geq 1 \text{ such that } P_{j+1} = P_{f''^k (j+1) - 1}; \\
m + 1, & \text{otherwise} \end{cases}$$
No $k$ satisfies

$$P_{j+1} = P_{f(k_{(j+1)}-1)}$$

$$f' = m+1 = 12+1 = 13$$

$$k = 1 \rightarrow P_{12} \neq P_{13}$$
No $k$ satisfies $P_{j+1} = P_{f'}^k(j+1) - 1$, $f' = m+1 = 12+1 = 13$

$k = 1 \Rightarrow P_{11} \neq P_{12}$

No $k$ satisfies $P_{j+1} = P_{f'}^k(j+1) - 1$, $f' = m+1 = 12+1 = 13$

$k = 1 \Rightarrow P_{10} \neq P_{12}$
\[
\begin{align*}
\therefore P_{j+1} &= P_{f'}^{(j+1)-1} \\
\Rightarrow P_9 &= P_{12}, \\
f' &= f' (j+1) - 1 = 12 - 1 = 12
\end{align*}
\]

\[
\begin{align*}
\therefore P_{j+1} &= P_{f'}^{(j+1)-1} \\
\Rightarrow P_8 &= P_{11}, \\
f' &= f' (j+1) - 1 = 12 - 1 = 11
\end{align*}
\]
\[
\begin{array}{cccccccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
  f' & & & & & & & & & & & & \\
  \end{array}
\]

\[
\begin{array}{cccccccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
  f' & & & & & & & & & & & & \\
  \end{array}
\]

\[
\because P_{j+1} = P_{f'}^{1(j+1)-1} \Rightarrow P_5 = P_8, \\
f' = f'^{(j+1) - 1} = 9 - 1 = 8
\]

\[
\begin{array}{cccccccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
  f' & & & & & & & & & & & & \\
  \end{array}
\]

\[
\begin{array}{cccccccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
  f' & & & & & & & & & & & & \\
  \end{array}
\]

\[
\because P_{j+1} = P_{f'}^{3(j+1)-1} \Rightarrow P_4 = P_{f'}^{3(4)-1} = P_{12}, \\
f' = f'^{3(j+1) - 1} = 13 - 1 = 12
\]
\[
\begin{array}{cccccccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
  P & \text{A T C A C A T C A T C A A} \\
  f' & 11 & 12 & 8 & 9 & 10 & 11 & 12 & 13 & 13 & 13 & 14
\end{array}
\]

\[
\because P_{j+1} = P_{f'}^{(j+1)-1} \Rightarrow P_3 = P_{f'}^{(3)-1} = P_{11},
\]
\[
f' = f' (j+1) - 1 = 12 - 1 = 11
\]

\[
\begin{array}{cccccccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
  P & \text{A T C A C A T C A T C A A} \\
  f' & 10 & 11 & 12 & 8 & 9 & 10 & 11 & 12 & 13 & 13 & 13 & 14
\end{array}
\]

\[
\because P_{j+1} = P_{f'}^{(j+1)-1} \Rightarrow P_2 = P_{f'}^{(2)-1} = P_{10},
\]
\[
f' = f' (j+1) - 1 = 11 - 1 = 10
\]
• Let $G'(j)$, $1 \leq j \leq m$, to be the largest number of shifts by good suffix rules.

• First, we set $G'(j)$ to zeros as their initializations.

<table>
<thead>
<tr>
<th>$j$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>$f'$</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>$G'$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
• **Step 1:** We scan from right to left and $g_{s_1}(j)$ is determined during the scanning, then $g_{s_1}(j) \geq g_{s_2}(j)$

---

**Observe:**

- If $P_j = P_4 \neq P_7 = P_{f'(j)-1}$, we know $g_{s_1}(f'(j)-1) = m + j - f'(j) + 1 = 9$.
- If $t = f'(j)-1 \leq m$ and $P_j \neq P_t$, $G'(t) = m - g_{s_1}(f'(j)-1) = f'(j) - 1 - j$.

$$f'^{(k)}(x) = f'^{(k-1)}(f'(x) - 1), \ k \geq 2$$

- When $j=12$, $t=13$. $t > m$.
- When $j=11$, $t=12$. Since $P_{11} = 'C' \neq 'A' = P_{12}$, $G'(t) = m - \max\{g_{s_1}(t), g_{s_2}(t)\} = m - g_{s_1}(t)$

$$= f'(j) - 1 - j$$

$$\Rightarrow G'(12) = 13 - 1 - 11 = 1.$$
If \( t = f'(j) - 1 \leq m \) and \( P_j \neq P_t \), \( G'(t) = f'(j) - 1 - j \).

\[ f^{(k)}(x) = f^{(k-1)}(f'(x) - 1), \ k \geq 2 \]

- When \( j = 10 \), \( t = 12 \). Since \( P_{10} = 'T' \neq 'A' = P_{12} \), \( G'(12) \neq 0 \).
- When \( j = 9 \), \( t = 12 \). \( P_9 = 'A' = P_{12} \).
- When \( j = 8 \), \( t = 11 \). \( P_8 = 'C' = P_{11} \).
- When \( j = 7 \), \( t = 10 \). \( P_7 = 'T' = P_{10} \).
- When \( j = 6 \), \( t = 9 \). \( P_6 = 'A' = P_9 \).
- When \( j = 5 \), \( t = 8 \). \( P_5 = 'C' = P_8 \).
- When \( j = 4 \), \( t = 7 \). Since \( P_4 = 'A' \neq P_7 = 'T' \), \( G'(7) = 8 - 1 - 4 = 3 \).

Besides, \( t = f''(4) - 1 = f'(f'(4) - 1) - 1 = 10 \). Since \( P_4 = 'A' \neq P_{10} = 'T' \), \( G'(10) = f''(7) - 1 - j = 11 - 1 - 4 = 6 \).

<table>
<thead>
<tr>
<th>( j )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P )</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>( f' )</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>( G' )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
If \( t = f' (j) - 1 \leq m \) and \( P_j \neq P_t \), \( G' (t) = f'' (j) - 1 - j \).

\[ f''^{(k)}(x) = f''^{(k-1)}(f' (x) - 1), \quad k \geq 2 \]

- When \( j=3 \), \( t=11 \). \( P_3 = 'C' = P_{11} \).
- When \( j=2 \), \( t=10 \). \( P_2 = 'T' = P_{10} \).
- When \( j=1 \), \( t=9 \). \( P_1 = 'A' = P_9 \).

<table>
<thead>
<tr>
<th>( j )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P )</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>( f' )</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>( G' )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

• By the above discussion, we can obtain the values using the Good Suffix Rule 1 by scanning the pattern from right to left.
- **Step 2**: Continuously, we will try to obtain the values using *Good Suffix Rule 2* and those values are still zeros now and scan from left to right.

\[
\begin{array}{cccccccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
  f' & 10 & 11 & 12 & 8 & 9 & 10 & 11 & 12 & 13 & 13 & 13 & 14 \\
  G' & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 6 & 0 & 1 \\
\end{array}
\]
Let $k'$ be the smallest $k$ in $\{1, \ldots, m\}$ such that $P_{f'}^{(k)}(1)-1 = P_1$ and $f^{(k)}(1)-1 \leq m$.

Observe:

$P_{1,4} = P_{9,12}$, $g_s(j) = m - (f'(1)-1) + 1 = 4$, where $1 \leq j \leq f'^{(k)}(1)-2$.

• If $G'\ j\) is not determined in the first scan and $1 \leq j \leq f'^{(k')}(1)-2$, thus, in the second scan, we set $G'(j) = m - \max\{g_s(j), g_s(j)\} = m - g_s(j) = f'^{(k')}(1) - 2$. If no such $k$ exists, set each undetermined value of $G$ to $m$ in the second scan.

• $k = 1 = k'$, since $P_{f'}^{(1)-1} = P_9 = "A" = P_1$, we set $G'(j) = f''(1)-2$ for $j = 1, 2, 3, 4, 5, 6, 8$.

<table>
<thead>
<tr>
<th>$j$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>$f'$</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>$G'$</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Let \( z \) be \( f''(k') (1) - 2 \). Let \( k'' \) be the largest value \( k \) such that \( f''(k)(z) - 1 \leq m \).

Then we set \( G'(j) = m - gs_2(j) = m - (m - f''(i)(z) - 1) = f''(i)(z) - 1 \), where \( 1 \leq i \leq k'' \) and \( f''(i-1)(z) < j \leq f''(i)(z) - 1 \) and \( f''(0)(z) = z \).

For example, \( z = 8 \):

- \( k = 1, f''(1)(8) - 1 = 11 \leq m = 12 \)
- \( k = 2, f''(2)(8) - 1 = 12 \leq m = 12 \) \( \Rightarrow k'' = 2 \)
- \( i = 1, f''(0)(8) - 1 = 7 < j \leq f''(1)(8) - 1 = 11 \).
- \( i = 2, f''(1)(8) - 1 = 11 < j \leq f''(2)(8) - 1 = 12 \).
- We set \( G(9) \) and \( G(11) = f''(1)(8) - 1 = 12 - 1 = 11 \).
We essentially have to decide the maximum number of steps. We can move the window right when a mismatch occurs. This is decided by the following function:

$$\max \{ G' (j), j-B(T_{s+j-1}) \}$$
We compare \( T \) and \( P \) from right to left. Since \( T_{12} = "T" \neq P_{12} = "A" \), the largest movement = \( \max \{ G'(j), j-B(T_{s+j-1}) \} = \max \{ G'(12), 12-B(T_{12}) \} = \max \{ 1, 12-10 \} = 2 \).
After moving, we compare $T$ and $P$ from right to left. Since $T_{14} = "T" \neq P_{12} = "A"$, the largest movement = \[
\max \{G'(j), j-B(Ts+j-1)\} = \max \{G'(12), 12-B(T_{14})\} = \max \{1, 12-10\} = 2.
\]
Time Complexity

• The preprocessing phase in $O(m+\Sigma)$ time and space complexity and searching phase in $O(mn)$ time complexity.

• The worst case time complexity for the Boyer-Moore method would be $O((n-m+1)m)$.

• It was proved that this algorithm has $O(m)$ comparisons when $P$ is not in $T$. However, this algorithm has $O(mn)$ comparisons when $P$ is in $T$. 
Reference

• Analyse exacte et en moyenne d'algorithmes de recherche d'un motif dans un texte, HANCART, C., University Paris 7, France, 1993.
Suffix trees and suffix arrays
String/Pattern Matching

- You are given a source string \( S \).
- Answer queries of the form: is the string \( p_i \) a substring of \( S \)?
- Knuth-Morris-Pratt (KMP) string matching.
  - \( O(|S| + |p_i|) \) time per query.
  - \( O(n|S| + S_i | p_i |) \) time for \( n \) queries.
- Suffix tree solution.
  - \( O(|S| + S_i | p_i |) \) time for \( n \) queries.
String/Pattern Matching

• KMP/BM preprocesses the query string $p_i$, whereas the suffix tree method preprocesses the source string $S$. 
Trie

• A tree representing a set of strings.

{ aeef, ad, bbfe, bbfg, c }
• **Assume no string is a prefix of another**

Each edge is labeled by a letter, no two edges outgoing from the same node are labeled the same.

Each string corresponds to a leaf.
Compressed Trie

- Compress unary nodes, label edges by strings
Suffix tree

Given a string $s$ a suffix tree of $s$ is a compressed trie of all suffixes of $s$

To make these suffixes prefix-free we add a special character, say $\$$, at the end of $s$
The suffix tree Tree(T) of T

- data structure **suffix tree**, Tree(T), is **compacted trie** that represents all the suffixes of string T
- linear size: $|\text{Tree}(T)| = O(|T|)$
- can be constructed in linear time $O(|T|)$
- has *myriad virtues* (A. Apostolico)
- is well-known: Google hits
Let $s=abab$, a suffix tree of $s$ is a compressed trie of all suffixes of $s=abab$.

\{ $, b$, ab$, bab$, abab$ \}$
Trivial algorithm to build a Suffix tree

Put the largest suffix in

Put the suffix bab$ in
Put the suffix ab$ in
Put the suffix b$ in
Put the suffix $ in
We will also label each leaf with the starting point of the corres. suffix.
On-line construction of Trie(T)

• $T = t_1 t_2 \ldots t_n$

• $P_i = t_1 t_2 \ldots t_i$  \textit{i:th prefix of T}

• \underline{on-line idea}: update $Trie(P_i)$ to $Trie(P_{i+1})$

• => very simple construction
Trie(abaab)

Trie(a)    Trie(ab)    Trie(aba)

aba

ba

a

ε

ε

chain of links connects the end points of current suffixes
Trie(abaab)

Trie(abaa)
Trie(abaab)

Trie(abaa)

Add next symbol = b
Trie(abaab)

Add next symbol = b

From here on b-arc already exists
Trie(abaab)
What happens in $\text{Trie}(P_i) \Rightarrow \text{Trie}(P_{i+1})$?

Before

After

From here on the $a_i$-arc exists already => stop updating here

New suffix links

New nodes
What happens in $Trie(P_i) => Trie(P_{i+1})$?

- time: $O(\text{size of Trie(T)})$
- suffix links:
  \[ \text{slink(node}(a\alpha)) = \text{node}(\alpha) \]
What can we do with it?

Exact string matching:

Given a Text $T$, $|T| = n$, preprocess it such that when a pattern $P$, $|P| = m$, arrives, you can quickly decide when it occurs in $T$.

We may also want to find all occurrences of $P$ in $T$. 
Exact string matching

In preprocessing we just build a suffix tree in $O(n)$ time

Given a pattern $P = ab$ we traverse the tree according to the pattern.
Exact string matching

In preprocessing we just build a suffix tree in $O(n)$ time

![Suffix tree diagram]

Given a pattern $P = \text{ab}$ we traverse the tree according to the pattern.
If we did not get stuck traversing the pattern then the pattern occurs in the text.

Each leaf in the subtree below the node we reach corresponds to an occurrence.

By traversing this subtree we get all k occurrences in $O(n+k)$ time
Generalized suffix tree

Given a set of strings $S$ a generalized suffix tree of $S$ is a compressed trie of all suffixes of $s \in S$

To make these suffixes prefix-free we add a special char, say $\$\$, at the end of $s$

To associate each suffix with a unique string in $S$ add a different special char to each $s$
Generalized suffix tree (Example)

Let $s_1=abab$ and $s_2=aab$ here is a generalized suffix tree for $s_1$ and $s_2$

{ 
  $\ \ #
  b$ $\ b#$
  ab$ $\ ab#$
  bab$ $\ aab#$
  abab$ 
}
So what can we do with it?

Matching a pattern against a database of strings
Longest common substring (of two strings)

Every node with a leaf descendant from string $S_1$ and a leaf descendant from string $S_2$ represents a maximal common substring and vice versa.

Find such node with largest “string depth”
Longest common substring (of two strings)

Every node with a leaf descendant from string $S_1$ and a leaf descendant from string $S_2$ represents a maximal common substring and vice versa.

Find such node with largest “string depth”
Lowest common ancestor

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it.
Lowest common ancestor

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it.
Lowest common ancestor

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it
Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes.
Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes.
Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes.
Finding maximal palindromes

• A palindrome: caabaac, cbaabc
• Want to find all maximal palindromes in a string $s$

Let $s = cbaaba$

The maximal palindrome with center between $i-1$ and $i$ is the LCP of the suffix at position $i$ of $S$ and the suffix at position $m-i+1$ of $Sr$
Maximal palindromes algorithm

Prepare a generalized suffix tree for
\( s = \text{cbaaba}\$ \) and \( s_r = \text{abaabc}\# \)

For every \( i \) find the LCA of suffix \( i \) of \( s \) and suffix \( m-i+1 \) of \( s_r \)
Let $s = \text{cbaaba}$ then $s_r = \text{abaabc}$
Let $s = \text{cbaaba}$ then $s_r = \text{abaabc}$
Let $s = cbaaba\$ then $s_r = abaabc#$
Analysis

$O(n)$ time to identify all palindromes
Drawbacks

• Suffix trees consume a lot of space

• It is $O(n)$ but the constant is quite big

• Notice that if we indeed want to traverse an edge in $O(1)$ time then we need an array of ptrs. of size $|\Sigma|$ in each node
Suffix array

- We loose some of the functionality but we save space.

Let \( s = \text{abab} \)

Sort the suffixes lexicographically: \( \text{ab, abab, b, bab} \)

The suffix array gives the indices of the suffixes in sorted order

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>
How do we build it?

• Build a suffix tree
• Traverse the tree in DFS, lexicographically picking edges outgoing from each node and fill the suffix array.

• $O(n)$ time
How do we search for a pattern?

- If P occurs in T then all its occurrences are consecutive in the suffix array.
- Do a binary search on the suffix array
- Takes $O(m \log n)$ time
Example

Let $S = \text{mississippi}$

Let $P = \text{issa}$
Supra index

- Structure
  - **Suffix arrays** are space efficient implementation of **suffix trees**.
  - Simply an array containing all the pointers to the text suffixes listed in lexicographical order.
  - **Supra-indices:**
    - If the suffix array is **large**, this binary search can perform **poorly** because of the number of random disk accesses.
    - Suffix arrays are designed to allow **binary searches** done by comparing the contents of each pointer.
    - To remedy this situation, the use of **supra-indices** over the suffix array has been proposed.
Supra index

• Example

This is a text. A text has many words. Words are made from letters

<table>
<thead>
<tr>
<th>60</th>
<th>50</th>
<th>28</th>
<th>19</th>
<th>11</th>
<th>40</th>
<th>33</th>
</tr>
</thead>
</table>

SuffixArray

<table>
<thead>
<tr>
<th>lett</th>
<th>text</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>50</td>
<td>28</td>
</tr>
<tr>
<td>19</td>
<td>11</td>
<td>40</td>
</tr>
<tr>
<td>33</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Supra-Index

SuffixArray
Supra index

• Example

This is a text. A text has many words. Words are made from letters.
Tree(hattivatti)
Tree(hattivatti)

substring labels of edges represented as pairs of pointers
Tree(hattivatti)
Tree(T) is *full* text index

P occurs in T ⇔ P is a prefix of some suffix of T ⇔
Path for P exists in Tree(T)

All occurrences of P in time $O(|P| + \#occ)$
Find **att** from Tree(hattivatti)
Linear time construction of Tree(T)

Weiner (1973), ‘algorithm of the year’

McCreight (1976)

‘on-line’ algorithm
(Ukkonen 1992)