The Shift-And Method

- Define $M$ to be a binary $n$ by $m$ matrix such that:

  $$M(i,j) = 1 \text{ iff the first } i \text{ characters of } P \text{ exactly match the } i \text{ characters of } T \text{ ending at character } j.$$  

  $$M(i,j) = 1 \text{ iff } P[1 .. i] \equiv T[j-i+1 .. j]$$
The *Shift-And* Method

- Let $T = \text{california}$
- Let $P = \text{for}$

$$M = \begin{array}{cccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & m = 10 \\
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
3 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\end{array}$$

- $M(i,j) = 1$ iff the first $i$ characters of $P$ exactly match the $i$ characters of $T$ ending at character $j$. 
How to construct M

- We will construct M column by column.
- Two definitions:
  - Bit-Shift(j-1) is the vector derived by shifting the vector for column j-1 down by one and setting the first bit to 1.
- Example:

\[
\begin{pmatrix}
0 \\
1 \\
1 \\
0 \\
1 \\
\end{pmatrix}
= 
\begin{pmatrix}
1 \\
0 \\
1 \\
1 \\
0 \\
\end{pmatrix}
\]
How to construct M

- We define the n-length binary vector U(x) for each character x in the alphabet. U(x) is set to 1 for the positions in P where character x appears.

- Example:

\[
P = \text{abaac} \quad \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = U(a) \quad \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = U(b) \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = U(c)
\]
How to construct $M$

- Initialize column 0 of $M$ to all zeros
- For $j > 1$ column $j$ is obtained by

$$M(j) = \text{BitShift}(j - 1) \land U(T(j))$$
An example $j = 1$

$T = \begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
{\text{x a b x a b a a c a}}
\end{array}$

$P = \begin{array}{cccc}
1 & 2 & 3 & 4 \\
{\text{a b a a c}}
\end{array}$

$U(x) = \begin{pmatrix}
0 \\
0 \\
0 \\
0
\end{pmatrix}$

$\text{BitShift}(0) \& U(T(1)) = \begin{pmatrix}
1 \\
0 \\
0 \\
0
\end{pmatrix} \& \begin{pmatrix}
0 \\
0 \\
0 \\
0
\end{pmatrix} = \begin{pmatrix}
0 \\
0 \\
0 \\
0
\end{pmatrix}$
An example $j = 2$

$T = x a b x a b a a c a$

$P = a b a a c$

$U(a) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$

$BitShift (1) \& U(T(2)) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \& \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$
An example $j = 3$

$T = x \ a \ b \ x \ a \ b \ a \ a \ c \ a$

$P = a \ b \ a \ a \ c$

$$
U(b) = \begin{pmatrix}
0 \\
1 \\
0 \\
0 \\
0
\end{pmatrix}
$$

$$
\text{BitShift} (2) \& U(T(3)) = \begin{pmatrix}
1 & 0 & 0 \\
1 & 1 & 1 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{pmatrix}
$$
An example $j = 8$

$T = x\ a\ b\ x\ a\ b\ a\ a\ c\ a$

$1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10$

$P = a\ b\ a\ a\ c$

$1\ 2\ 3\ 4\ 5$

$\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}$

$U(a) = \begin{pmatrix}
1 \\
0 \\
1 \\
1 \\
0
\end{pmatrix}$

$\text{BitShift (7) \& } U(T(8)) = \begin{pmatrix}
1 \\
1 \\
0 \\
1 \\
0
\end{pmatrix} \& \begin{pmatrix}
1 \\
0 \\
1 \\
1 \\
0
\end{pmatrix} = \begin{pmatrix}
1 \\
0 \\
0 \\
1 \\
0
\end{pmatrix}$
Correctness

- For $i > 1$, Entry $M(i,j) = 1$ iff
  1) The first $i-1$ characters of $P$ match the $i-1$ characters of $T$ ending at character $j-1$.
  2) Character $P(i) \equiv T(j)$.

- 1) is true when $M(i-1,j-1) = 1$.
- 2) is true when the $i$'th bit of $U(T(j)) = 1$.

- The algorithm computes the and of these two bits.
Correctness

$T = x \ a \ b \ x \ a \ b \ a \ a \ c \ a$

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

- $M(4,8) = 1$, this is because $a \ b \ a \ a$ is a prefix of $P$ of length 4 that ends at position 8 in $T$.
- Condition 1) – We had $a \ b \ a$ as a prefix of length 3 that ended at position 7 in $T \leftrightarrow M(3,7) = 1$.
- Condition 2) – The fourth bit of $P$ is the eighth bit of $T \leftrightarrow$ The fourth bit of $U(T(8)) = 1$. 
How much did we pay?

- Formally the running time is $\Theta(mn)$.
- However, the method is very efficient if $n$ is the size of a single or a few computer words.
- Furthermore only two columns of $M$ are needed at any given time. Hence, the space used by the algorithm is $O(n)$. 
Search in keyword trees

- Naïve threading in keyword trees do not *remember* the partial matches
- $P = \{\text{apple, appropos}\}$
- $T = \text{appappropos}$
- When threading
  - *app* is a partial match
  - But naïve threading will go back to the root and re-thread *app*
- Define *failure links*
v: a node in keyword tree K
L(v): the label on v, that is, the concatenation of characters on the path from the root to v.
lp(v): the length of the longest proper suffix of string L(v) that is a prefix of some pattern in P. Let this substring be $\alpha$.

Lemma. There is a unique node in the keyword tree that is labeled by string $\alpha$. Let this node be $n_v$. Note that $n_v$ can be the root.

The ordered pair $(v, n_v)$ is called a failure link.
Failure Link

\[ P = \{\text{potato, tattoo, theater, other}\} \]
Failure Link

Failure link computation is $O(n)$
Failure Link
Failure Link

\[ l = c - l_p(w) = 8 - 3 = 5 \]

\[ c = 8 \]
Failure Link

How to construct failure links for a keyword tree in a linear time?

Let \( d \) be the distance of a node \((v)\) from the root \(r\).

- When \( d \leq 1\), i.e., \(v\) is the root or \(v\) is one character away from \(r\), then \(n_v = r\).

Suppose \(n_v\) has been computed for every node \((v)\) with \(d \leq k\), we are going to compute \(n_v\) for every node with \(d = k + 1\).

- \(v\): parent of \(v\), then \(v\) is \(k\) characters from \(r\), that is \(d = k\) thus the failure link for \(v\) has been computed. \(n_v\).
- \(x\): the character on edge \((v', v)\)
(1) If there is an edge $(n_v, w)$ out of $n_v$, labeled with $x$, then $n_v = w$. 
Failure Link
(2) If such an edge does not exist, examine $n_{n_y}$ to see if there is an edge out of it labeled with $x$. Continue until the root.
(2) If such an edge does not exist, examine $n_{nv}$ to see if there is an edge out of it labeled with $x$. Continue until the root.
Failure Link
Failure Link
Output: calculate $n_v$ for $v$

Algorithm $n_v$

$v`$ is the parent of $v$ in $K$
$x$ is the character on edge $(v`, v)$

$w = n_v$

while there is no edge out of $w$ labeled with $x$ and $w \neq r$

$w = n_w$

If there is an edge $(w, w`) out of $w$ labeled $x$ then

$n_v = w`$

else $n_v = r$
Aho-Corasick Algorithm

Input: Pattern set \( P \) and text \( T \)
Output: all occurrences in \( T \) any pattern from \( P \)

Algorithm AC

\( l=1; \)
\( c=1; \)
\( w=\)root of \( K \)
Repeat
  while there is an edge \((w, w')\) labeled with \( T(c) \)
    if \( w' \) is numbered by pattern \( i \) then
      report that \( p_i \) occurs in \( T \) starting at \( l; \)
      \( w=w'; c++; \)
    \( w=n_w \) and \( l=c-lp(w); \)
Until \( c>m \)
Slides from Tolga Can

SUFFIX ARRAYS
Suffix arrays were introduced by Manber and Myers in 1993.

More space efficient than suffix trees.

A suffix array for a string $x$ of length $m$ is an array of size $m$ that specifies the lexicographic ordering of the suffixes of $x$. 
## Suffix arrays

Example of a suffix array for `acaacatat$`

<table>
<thead>
<tr>
<th>Index</th>
<th>Suffix</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>aaacatat$</code></td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td><code>aacatat$</code></td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td><code>acaacatat$</code></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td><code>acatat$</code></td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td><code>atat$</code></td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td><code>at$</code></td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td><code>caaacatat$</code></td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td><code>catat$</code></td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td><code>tat$</code></td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td><code>t$</code></td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td><code>$</code></td>
<td>11</td>
</tr>
</tbody>
</table>
Suffix array construction

- Naive in place construction
  - Similar to insertion sort
  - Insert all the suffixes into the array one by one making sure that the new inserted suffix is in its correct place
  - Running time complexity:
    - $O(m^2)$ where $m$ is the length of the string

- Manber and Myers give a $O(m \log m)$ construction.
Suffix arrays

- $O(n)$ space where $n$ is the size of the database string
- Space efficient. However, there’s an increase in query time
- Lookup query
  - Based on binary search
  - $O(m \log n)$ time; $m$ is the size of the query
  - Can reduce time to $O(m + \log n)$ using a more efficient implementation
Searching for a pattern in Suffix Arrays

find(Pattern P in SuffixArray A):
    i = 0
    lo = 0, hi = length(A)
    for 0<=i<length(P):
        Binary search for x,y

        lo<=x<=j<y<=hi
        lo = x, hi = y
    return {A[lo],A[lo+1],...,A[hi-1]}
Search example

Search *is* in *mississippi*$

Examine the pattern letter by letter, reducing the range of occurrence each time.

First letter *i*: occurs in indices from 0 to 3

So, pattern should be between these indices.

Second letter *s*: occurs in indices from 2 to 3

Done.

Output: *issippi*$ and *ississippi*$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>i$</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>ippi$</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>issippi$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>ississippi$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>mississippi$</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>pi$</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>ppi$</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>sippi$</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>sissippi$</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>ssippi$</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>ssissippi$</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>$</td>
</tr>
</tbody>
</table>
Suffix Arrays

- It can be built very fast.
- It can answer queries very fast:
  - How many times ATG appears?
- Disadvantages:
  - Can’t do approximate matching
  - Hard to insert new stuff (need to rebuild the array) dynamically.