The Knuth-Morris-Pratt algorithm

- Best known
  - For linear time for exact matching
- Preprocessing pattern P
- Compares from Left to Right
The KMP Ideas

- Shift more than one space
- Reduce comparison

The KMP Algorithm - Motivation

- When a mismatch occurs, what is the most we can shift the pattern?
- Answer: the largest prefix of $P[0..j-1]$ that is a suffix of $P[1..j-1]$.

KMP Failure Function

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.
- The failure function $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$. 
KMP Failure Function

- Knuth-Morris-Pratt’s algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.
- The failure function $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$.

The failure function $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$.

$$f: \{0, 1, 2, 3, 4, 5\} \rightarrow \{a, b, a, a, a, b\}$$

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[f]$</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>$F[f]$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

The KMP Algorithm

```
Algorithm KMPMatch(T, P)
F = failureFunction(P)
i = 0
j = 0
while i < n
    if $T[i] = P[j]$
        if $j = m - 1$
            return $i - j$ [match]
        else
            $i \leftarrow i + 1$
            $j \leftarrow j + 1$
    else
        if $j > 0$
            $j \leftarrow F[j - 1]$
        else
            $i \leftarrow i + 1$
    return -1 [no match]
```
**Example**

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P[i]</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>F[i]</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Algorithm KMPMatch(T, P):

\[ F \leftarrow \text{failureFunction}(P) \]

\[ i \leftarrow 0 \]

\[ j \leftarrow 0 \]

while \( i < n \) do

\[ \text{if } T[i] = P[j] \]

\[ i \leftarrow i + 1 \]

\[ j \leftarrow j + 1 \]

\[ \text{else if } j > 0 \]

\[ j \leftarrow F[j-1] \]

\[ \text{else} \]

\[ i \leftarrow i + 1 \]

\[ \text{return } i - j \] (no match)

The KMP Algorithm

- The failure function can be represented by an array and can be computed in \( O(m) \) time.
- At each iteration of the while-loop, either
  - \( i \) increases by one, or
  - the shift amount \( i - j \) increases by at least one (observe that \( F[j-1] < j \)).
- Hence, there are no more than \( 2m \) iterations of the while-loop.
- Thus, KMP’s algorithm runs in optimal time \( O(m + n) \).

Computing the Failure Function

- The failure function can be represented by an array and can be computed in \( O(m) \) time.
- The construction is similar to the KMP algorithm itself.
- At each iteration of the while-loop, either
  - \( i \) increases by one, or
  - the shift amount \( i - j \) increases by at least one (observe that \( F[j-1] < j \)).
- Hence, there are no more than \( 3m \) iterations of the while-loop.

Algorithm failureFunction(P):

\[ F[0] \leftarrow 0 \]

\[ i \leftarrow 1 \]

\[ j \leftarrow 0 \]

while \( i < m \) do

\[ \text{if } P[i-1] = P[j] \]

\[ \text{if } j > 0 \]

\[ \text{if } F[j-1] < j - 1 \]

\[ i \leftarrow i + 1 \]

\[ j \leftarrow j + 1 \]

\[ F[i] \leftarrow j + 1 \]

\[ \text{else} \]

\[ j \leftarrow F[j-1] \]

\[ \text{else if } P[j-1] > P[i-1] \]

\[ i \leftarrow i + 1 \]

\[ \text{else} \]

\[ F[i] \leftarrow 0 \] (no match)}
Longest Common Subsequence

Given a sequence \( X = x[1], x[2], ..., x[m] \), another sequence \( Z = z[1], z[2], ..., z[k] \) is a subsequence of \( X \) if there are indices \( i[1] < i[2] < i[3] < ... < i[k] \), such that for all \( j = 1, ..., k \), \( x[i[j]] = z[j] \).

**Algorithm**

Given two sequences \( X \) and \( Y \), a sequence \( Z \) is a common subsequence of \( X \) and \( Y \) if it is a subsequence of both \( X \) and \( Y \).

**Algorithm**

The longest common subsequence (LCS) problem:
Given two sequences \( X \) and \( Y \), find the longest common subsequence of both \( X \) and \( Y \).

Longest Common Subsequence

- Applications:
  - Molecular biology: When biologists find a new sequence, they typically want to know what other sequences it is most similar to.
  - File comparison: compare two different versions of the same file, to determine what changes have been made to the file. It works by finding a longest common subsequence of the lines of the two files.
- ...

Algorithm LCSLength(X, Y)

```python
if empty(X) or empty(Y):
    return 0
else if X[0] = Y[0]:
    return 1 + LCSLength(X[1..m-1], Y[1..n-1])
else:
    return max(LCSLength(X[1..m-1], Y), LCSLength(X, Y[1..n-1]))
```

Find a recursive solution
Calculate bottom-up and avoid recalculation
Longest Common Subsequence

Algorithm LCSLength(X, Y)

for i ← m downto 0 do A[i][n] ← 0
for j ← n downto 0 do A[m][j] ← 0
for i ← m-1 downto 0 do
  for j ← n-1 downto 0 do
    if X[i] = Y[j]
      A[i][j] ← 1 + A[i+1][j+1]
    else A[i][j] ← max(A[i+1][j], A[i][j+1])
  return A[0][0]
If $X[i] = Y[j]$ then $A[i][j] ← 1 + A[i+1][j+1]$
else $A[i][j] ← \max(A[i+1][j], A[i][j+1])$

\[i ← 8\]
\[j ← 8 \text{ down to } 0\]
\[
\begin{align*}
    & i \leftarrow 5 \\
    & j \leftarrow 8 \text{ downto } 0
\end{align*}
\]

\[
\begin{array}{cccccccccc}
    & E & M & P & T & Y & N & E & S & S \\
\hline
    N & 0 &  &  &  &  &  &  &  &  \\
    E &  &  &  &  &  &  &  &  &  \\
    M &  &  &  &  &  &  &  &  &  \\
    A &  &  &  &  &  &  &  &  &  \\
    T &  &  &  &  &  &  &  &  &  \\
    O & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 0 \\
    D & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 0 \\
    E & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 0 \\
    S & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
    & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

\[
\begin{align*}
    & i \leftarrow 4 \\
    & j \leftarrow 8 \text{ downto } 0
\end{align*}
\]

\[
\begin{array}{cccccccccc}
    & E & M & P & T & Y & N & E & S & S \\
\hline
    N & 0 &  &  &  &  &  &  &  &  \\
    E &  &  &  &  &  &  &  &  &  \\
    M &  &  &  &  &  &  &  &  &  \\
    A &  &  &  &  &  &  &  &  &  \\
    T & 3 & 3 & 3 & 2 & 2 & 2 & 1 & 1 & 0 \\
    O & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 0 \\
    D & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 0 \\
    E & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 0 \\
    S & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
    & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

\[
\begin{align*}
    & i \leftarrow 3 \\
    & j \leftarrow 8 \text{ downto } 0
\end{align*}
\]

\[
\begin{array}{cccccccccc}
    & E & M & P & T & Y & N & E & S & S \\
\hline
    N & 0 &  &  &  &  &  &  &  &  \\
    E &  &  &  &  &  &  &  &  &  \\
    M &  &  &  &  &  &  &  &  &  \\
    A & 3 & 3 & 3 & 2 & 2 & 2 & 1 & 1 & 0 \\
    T & 3 & 3 & 3 & 2 & 2 & 2 & 1 & 1 & 0 \\
    O & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 0 \\
    D & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 0 \\
    E & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 0 \\
    S & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
    & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]
<table>
<thead>
<tr>
<th>i ← 2</th>
<th>j ← 8 downto 0</th>
<th>E M P T Y N E S S</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>M</td>
<td>4 4 3 3 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>3 3 3 3 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>3 3 3 3 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2 2 2 2 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>2 2 2 2 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>2 2 2 2 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>1 1 1 1 1 1 1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

if $X[i] = Y[j]$ then $A[i][j] ← 1 + A[i+1][j+1]$ else $A[i][j] ← \max(A[i+1][j], A[i][j+1])$


<table>
<thead>
<tr>
<th>i ← 1</th>
<th>j ← 8 downto 0</th>
<th>E M P T Y N E S S</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>M</td>
<td>5 4 3 3 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>3 3 3 3 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>3 3 3 3 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2 2 2 2 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>2 2 2 2 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>2 2 2 2 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>1 1 1 1 1 1 1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>i ← 0</th>
<th>j ← 8 downto 0</th>
<th>E M P T Y N E S S</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>M</td>
<td>5 4 3 3 3 3 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>3 3 3 3 3 3 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>3 3 3 3 3 3 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2 2 2 2 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>2 2 2 2 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>2 2 2 2 2 2 2 2 1 1 0</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>1 1 1 1 1 1 1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
Longest Common Subsequence
The subsequence itself.
Algorithm LCSequence(X, Y, A)
i ← 0, j ← 0
while i < m and j < n do
  if X[i] = Y[j]
    add X[i] to the end of the sequence S
    increment i, increment j
  else if A[i+1][j] > A[i][j+1]
    increment i
  else
    increment j
return S
if $X[i] = Y[j]$ then add $X[i]$ to the end of $S$, increment $i$ and increment $j$

else $A[i+1][j] > A[i][j+1]$ then increment $i$ else increment $j$
if $X[i] = Y[j]$ then add $X[i]$ to the end of $S$, increment $i$ and increment $j$

else if $A[i+1][j] > A[i][j+1]$ then increment $i$
else increment $j$

$i ← 4$

$j ← 3$

if $X[i] = Y[j]$ then add $X[i]$ to the end of $S$, increment $i$ and increment $j$

else if $A[i+1][j] > A[i][j+1]$ then increment $i$
else increment $j$

$i ← 5$

$j ← 4$

if $X[i] = Y[j]$ then add $X[i]$ to the end of $S$, increment $i$ and increment $j$

else if $A[i+1][j] > A[i][j+1]$ then increment $i$
else increment $j$

$i ← 5$

$j ← 5$
if $X[i] = Y[j]$ then add $X[i]$ to the end of $S$, increment $i$ and increment $j$
else if $A[i+1][j] > A[i][j+1]$ then increment $i$ else increment $j$

\[
i \leftarrow 5\]
\[
j \leftarrow 6\]
\[
S
\]

\[
S
\]

\[
S
\]

\[
S
\]
if $X[i] = Y[j]$ then add $X[i]$ to the end of $S$, increment $i$ and increment $j$
else if $A[i+1][j] > A[i][j+1]$ then increment $i$ else increment $j$

if $X[i] = Y[j]$ then add $X[i]$ to the end of $S$, increment $i$ and increment $j$
else if $A[i+1][j] > A[i][j+1]$ then increment $i$ else increment $j$

Tries
Trie

- Takes its name from retrieval.
  - Pre-process the text so that searching is faster
  - Supports pattern matching queries in time proportional to the pattern size
- A trie is a compact data structure for representing a set of strings, such as all the words in a text
- Ideal if the text is large, immutable and searched for often
  - e.g., works by Shakespeare
- Often used for pattern matching and prefix matching
  - e.g., Find all words that start with ki

Tries

- Standard Tries
- Compressed Tries
- Suffix Tries

Standard Trie (1)

- The standard trie for a set of strings $S$ is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered
  - The paths from the external nodes to the root yield the strings of $S$
- Example: standard trie for the set of strings
  $S = \{ \text{bear, bell, bid, bull, buy, sell, stock, stop} \}$

- Standard Trie (1)
Word Matching with a Trie

- We insert the words of the text into a trie.
- Each leaf stores the occurrences of the associated word in the text.

Standard Trie (2)

- A standard trie, $\mathcal{T}$, has $s$ external nodes:
  - $s$ : the # of strings in $S$
- Every internal node in $\mathcal{T}$ has at most $d$ children:
  - $d$ : size of the alphabet
- Height of $\mathcal{T}$ is the length of the longest string in $S$

Standard Trie (3)

- A standard trie uses $O(n)$ space where $n$ is the total size of the strings in $S$
- supports searches, insertions and deletions in time $O(dm)$, where:
  - $n$: total size of the strings in $S$
  - $m$: size of the string parameter of the operation
  - $d$: size of the alphabet
Applications of Tries

- A standard trie supports the following operations on a preprocessed text in time $O(m)$, where $m = |X|
  - word matching: find the first occurrence of word $X$ in the text
  - prefix matching: find the first occurrence of the longest prefix of word $X$ in the text
- Each operation is performed by tracing a path in the trie starting at the root

Compressed Trie

- Also known as Patricia Tree.
- Each internal node has at least 2 children
- obtained from standard trie by compressing chains of "redundant" nodes

Compact Representation

- Compact representation of a compressed trie for an array of strings:
  - Stores at the nodes ranges of indices instead of substrings
  - Uses $O(n)$ space, where $n$ is the number of strings in the array
  - Serves as an auxiliary index structure
The suffix trie of a string $X$ is the compressed trie of all the suffixes of $X$. 

- **Suffix Trie (1)**
  - Compact representation of the suffix trie for a string $X$ of size $n$ from an alphabet of size $d$.
  - Uses $O(n)$ space.
  - Supports arbitrary pattern matching queries in $X$ in $O(dm)$ time, where $m$ is the size of the pattern.

**Tries and Internet**

- **Web Crawler**: A program that gathers information about web pages and stores them in a special dictionary called inverted file.
  - **Inverted File**: Dictionary storing key-value pair $(w, L)$, where $w$ is a searchable word and $L$ is a collection of references to pages (URLs) containing $w$.
    - $w$ is called *index term*
    - $L$ is called *occurrence list*
- **Web Search Engine**: Program that allows us to retrieve information from this database.
Tries and Internet

- The **index terms** are stored into a compressed trie.
- Each leaf of the trie is associated with a word and has the pointer to **occurrence list**.
- The trie is kept in internal memory.
- The occurrence lists are kept in external memory and are ranked by relevance.
- Boolean queries for sets of words (e.g., Java and coffee) correspond to set operations (e.g., intersection) on the occurrence lists.
- Additional **information retrieval** techniques are used, such as:
  - Stop-word elimination (e.g., ignore “the” “a” “is”)
  - Stemming (e.g., identify “add” “adding” “added”)
  - Link analysis (recognize authoritative pages)