CS481: Bioinformatics Algorithms

Can Alkan
EA509
calkan@cs.bilkent.edu.tr

http://www.cs.bilkent.edu.tr/~calkan/teaching/cs481/
EXACT STRING MATCHING
The problem of String Matching

Given a string ‘t’, the problem of string matching deals with finding whether a pattern ‘p’ occurs in ‘t’ and if ‘p’ does occur then returning position in ‘t’ where ‘p’ occurs.
Brute force ($O(\text{mn})$)

\begin{verbatim}
n <- |t|
m <- |p|
i <= 1

while i < n
  if p == t[i, i+m-1]
    return i;
  else
    i = i + 1;
\end{verbatim}
## SimpleStringSearch

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

| Y    | Y    | Y    | Y    | Y    | N    |
SimpleStringSearch

\[
\begin{array}{cccccccccccc}
A & B & C & E & F & G & A & B & C & D & E \\
A & B & C & D \\
N
\end{array}
\]
SimpleStringSearch

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

N
SimpleStringSearch

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>
```

\text{N}
## SimpleStringSearch

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

N
### SimpleStringSearch

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

N
**SimpleStringSearch**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

Y Y Y Y Y
Straightforward string searching

- **Worst case:**
  - Pattern string always matches completely except for last character
  - Example: search for XXXXXXXY in target string of XXXXXXXXXXXXXXXXXXXXXXXX
  - Outer loop executed once for every character in target string
  - Inner loop executed once for every character in pattern
  - $O(mn)$, where $m = |p|$ and $n = |t|$

- **OK if patterns are short, but better algorithms exist**
### SimpleStringSearch

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

| Y      | Y      | Y      | Y      | N      |
Knuth-Morris-Pratt

- $O(m+n)$
- Key idea:
  - if pattern fails to match, slide pattern to right by as many boxes as possible without permitting a match to go unnoticed
The KMP Algorithm - Motivation

- Knuth-Morris-Pratt's algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$

.. $a$ $b$ $a$ $a$ $b$ $x$ .. .. .. ..

$a$ $b$ $a$ $a$ $b$ $a$

No need to repeat these comparisons

Resume comparing here
**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 $P[1..j]$.

- Knuth-Morris-Pratt’s algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j-1)$.

<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>$P[j]$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>$F(j)$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

![Diagram](image-url)
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 $2n$ iterations of the while-loop.
- Thus, KMP’s algorithm runs in optimal time $O(m + n)$.

Algorithm $KMPMatch(T, P)$

```
F ← failureFunction(P)
i ← 0
j ← 0
while i < n
    if $T[i] = P[j]$
        if $j = m - 1$
            return $i - j \{ \text{match} \}$
        else
            $i ← i + 1$
            $j ← j + 1$
    else
        $j ← F[j-1]$
    else
        $i ← i + 1$
        $j ← 0$
return $-1 \{ \text{no match} \}$
```
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 $2m$ iterations of the while-loop.

**Algorithm failureFunction(P)**

```plaintext
F[0] ← 0
i ← 1
j ← 0
m ← length(P)
while i < m
  if P[i] = P[j]
    {we have matched $j + 1$ chars}
    F[i] ← j + 1
    i ← i + 1
    j ← j + 1
  else if j > 0 then
    {use failure function to shift P}
    j ← F[j - 1]
  else
    F[i] ← 0  {no match}
    i ← i + 1
```
Example

\[
\begin{array}{cccccc}
  a & b & a & c & a & b \\
  1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]

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

\[
\begin{array}{cccccc}
  a & b & a & c & a & b \\
  7 & 8 & 9 & 10 & 11 & 12 \\
\end{array}
\]

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

\[
\begin{array}{cccccc}
  j & 0 & 1 & 2 & 3 & 4 & 5 \\
  P[j] & a & b & a & c & a & b \\
  F(j) & 0 & 0 & 1 & 0 & 1 & 2 \\
\end{array}
\]

\[
\begin{array}{cccccc}
  a & b & a & c & a & b \\
  13 & 14 & 15 & 16 & 17 & 18 & 19 \\
\end{array}
\]
The Boyer-Moore Algorithm

- Similar to KMP in that:
  - Pattern compared against target
  - On mismatch, move as far to right as possible

- Different from KMP in that:
  - Compare the patterns from right to left instead of left to right

- Does that make a difference?
  - Yes – much faster on long targets; many characters in target string are never examined at all
There is no E in the pattern: thus the pattern can’t match if any characters lie under t[3]. So, move four boxes to the right.
Boyer-Moore example

Again, no match. But there is a B in the pattern. So move two boxes to the right.
Boyer-Moore example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

Y Y Y Y Y Y
Problem: determine $d$, the number of boxes that the pattern can be moved to the right.

$d$ should be smallest integer such that $t[k+m-1] = p[m-1-d]$, $t[k+m-2] = p[m-2-d]$, ..., $t[k+i] = p[i-d]$
We said:

- d should be smallest integer such that:
  - $T[k+m-1] = p[m-1-d]$  
  - $T[k+m-2] = p[m-2-d]$  
  - $T[k+i] = p[i-d]$

Reminder:

- $k =$ starting index in target string  
- $m =$ length of pattern  
- $i =$ index of mismatch in pattern string

Problem: statement is valid only for $d \leq i$

Need to ensure that we don’t “fall off” the left edge of the pattern
**Boyer-Moore: another example**

<table>
<thead>
<tr>
<th>$t[k]$</th>
<th>$t[k+5]$</th>
<th>$t[k+8]$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c</td>
<td>X</td>
</tr>
<tr>
<td>Y</td>
<td>Z</td>
<td>W</td>
</tr>
</tbody>
</table>

If $c == W$, then $d$ should be 3

If $c == R$, then $d$ should be 7
Bad Character Rule

Suppose that $P_1$ is aligned to $T_s$ now, and we perform a pair-wise 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 Character Matching Rule.
- For any character $x$ in $T$, find the nearest $x$ in $P$ which is to the left of $x$ in $T$. 
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 $P$, move $P$ to the right of $x$
Ex: Suppose that P1 is aligned to T6 now. We compare pairwise between T and P from right to left. Since T16,17 = P11,12 = “CA” and T15 = “G” ≠ P10 = “T”. Therefore, we find the rightmost position c=7 in the left of P10 in P such that Pc is equal to “G” and we can move the window at least (10-7=3) positions.

\[s=6\]

\(\begin{array}{cccccccccccccccc}
\end{array}\)

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

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

\(\text{direction of the scan}\)

\(\text{mismatch}\)

\(c\)

\(j=10\)

\(m=12\)
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).
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 \( P \);
Suppose that $P_1$ is aligned to $T_6$ now. We compare pair-wise 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.
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.
**Unique Substring Rule**

- The substring $u$ appears in $P$ exactly once.
- If the substring $s$ 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.
The Suffix to Prefix Rule

- 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 $B(a)$ be the rightmost position of $a$ in $P[1..j]$. The function will be used for applying *bad character rule*.

We can move our pattern right at least $j-B[j](T_{s+j-1})$ position by above $B$ function.

Move at least $10-B[10](G) = 10$ 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)\n
• \( gs_2(j) \) 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.)\n
• \( Gs(j) = m - \max\{gs_1, gs_2\} \), if \( j = m \), \( Gs(j) = 1 \).

<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>( )</td>
<td>( ) )</td>
<td>( ) )</td>
<td>( ) )</td>
<td>( X )</td>
<td>( X )</td>
<td>( X )</td>
<td>( X )</td>
<td>( X )</td>
<td>( ) )</td>
<td>( ) )</td>
<td>( ) )</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>

\( gs_1(7) = 9 \)

\( \because P_{8,12} \) is a suffix of \( P_{1,9} \) and \( P_4 \neq P_7 \)

\( gs_2(7) = 4 \)

\( \because P_{1,4} \) is a suffix of \( P_{8,12} \)
\[
\begin{array}{cccccccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
  gs_1 & 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 6 & 1 & 0 & \\
  gs_2 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 1 & 1 & 1 & 0 \\
  Gs & 8 & 8 & 8 & 8 & 8 & 8 & 3 & 8 & 11 & 6 & 11 & 1 \\
\end{array}
\]

\[s=6\]

\[s+j-1\]

\[m=12\]

\[j'=6\]

\[j=10\]
Time Complexity

- Use good character or bad suffix rule
  - The one that skips more
- The preprocessing phase in $O(m+\Sigma)$ complexity
- If you are searching for ALL matches, worst case:
  - $O(mn)$ when $P$ is in $T$ at all positions
    - $T=$AAAAAAAAAAAAAAA; $P=$AAAA
  - $O(m+n)$ when $P$ is not in $T$
BRUTE FORCE – EXAMPLE #2
Brute force

T = ABABABCABABABCABABAC
P = ABABAC              Comparisons: 6
    x

T = ABABABCABABABCABABAC
P = ABABAC              Comparisons: 1
    x

T = ABABABCABABABCABABAC
P = ABABAC              Comparisons: 5
    x

T = ABABABCABABABCABABAC
P = ABABAC              Comparisons: 1
    x

T = ABABABCABABABCABABAC
P = ABABAC              Comparisons: 3
    x
Brute force

\[
\begin{align*}
T &= \text{ABABABCABABABABCABABAC} \\
P &= \text{ABABAC} \\
\text{Comparisons: 1} \\
T &= \text{ABABABCABABABABCABABAC} \\
P &= \text{ABABAC} \\
\text{Comparisons: 1} \\
T &= \text{ABABABCABABABABCABABAC} \\
P &= \text{ABABAC} \\
\text{Comparisons: 6} \\
T &= \text{ABABABCABABABABCABABAC} \\
P &= \text{ABABAC} \\
\text{Comparisons: 1} \\
T &= \text{ABABABCABABABABCABABAC} \\
P &= \text{ABABAC} \\
\text{Comparisons: 5}
\end{align*}
\]
Brute force

T = ABABABCABABABCABABAC
P = ABABAC
Comparisons: 1

T = ABABABCABABABCABABAC
P = ABABAC
Comparisons: 3

T = ABABABCABABABCABABAC
P = ABABAC
Comparisons: 1

T = ABABABCABABABCABABAC
P = ABABAC
Comparisons: 1

T = ABABABCABABABCABABAC
P = ABABAC
Comparisons: 6

match

Total comparisons: 41
KMP – EXAMPLE #2
Knuth-Morris-Pratt

T = ABABABCABABABCABABAC
P = ABABAC

Reminder: \( F(j) \) is defined as the size of the largest prefix of \( P[0..j] \) that is also a suffix of \( P[1..j] \)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Algorithm `failureFunction(P)`

\[
\begin{align*}
F[0] & \leftarrow 0 \\
i & \leftarrow 1 \\
j & \leftarrow 0 \\
m & \leftarrow \text{length}(P) \\
\text{while } i < m \\
& \quad \text{if } P[i] = P[j] \\
& \quad \quad \{ \text{we have matched } j + 1 \text{ chars} \} \\
& \quad \quad F[i] \leftarrow j + 1 \\
& \quad \quad i \leftarrow i + 1 \\
& \quad \quad j \leftarrow j + 1 \\
& \quad \text{else if } j > 0 \text{ then} \\
& \quad \quad \{ \text{use failure function to shift } P \} \\
& \quad \quad j \leftarrow F[j - 1] \\
& \quad \text{else} \\
& \quad \quad F[i] \leftarrow 0 \{ \text{ no match } \} \\
& \quad i \leftarrow i + 1
\end{align*}
\]
Knuth-Morris-Pratt

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

T = ABABABABCABABABCABABAC
P = ABABAC

\[ T[i] = ABABABCABABABCABABAC \]
\[ P[j] = ABABAC \]

\[ j = 5, \ i = 5 \]

New j = F[4] = 3 (shift 5-3 = 2)
New i = 5 (no change)

\[ T[i] = ABABABCABABABCABABAC \]
\[ P[j] = ABABAC \]

Algorithm **KMPMatch**(T, P)

\[ F \leftarrow \text{failureFunction}(P) \]
\[ i \leftarrow 0 \]
\[ j \leftarrow 0 \]

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

\[ i \leftarrow i + 1 \]
\[ j \leftarrow 0 \]

return \(-1\) { no match }

Comparisons = 6
Algorithm \textbf{KMPMatch}(T, P)

\[
\begin{align*}
F & \leftarrow \text{failureFunction}(P) \\
& \text{if } T[i] = P[j] \\
& \quad \text{if } j = m - 1 \\
& \quad \quad \text{return } i - j \{ \text{match} \} \\
& \quad \text{else} \\
& \quad \quad i \leftarrow i + 1 \\
& \quad \quad j \leftarrow j + 1 \\
& \text{else} \\
& \quad \quad \text{if } j > 0 \\
& \quad \quad \quad j \leftarrow F[j-1] \\
& \quad \quad \text{else} \\
& \quad \quad \quad i \leftarrow i + 1 \\
& \quad \quad \quad j \leftarrow 0 \\
& \text{return } -1 \{ \text{no match} \}
\end{align*}
\]

\text{T = ABABABCABABABCABABACABABAC}
\text{P = ABABAC}

\text{j = 4, i = 6}

\text{New } j = F[3] = 2 \text{ (shift 4-2 = 2)}
\text{New } i = 6 \text{ (no change)}

\text{T = ABABABCABABABCABABAC}
\text{P = ABABAC}

\text{Comparisons = 6+5 = 11}
Knuth-Morris-Pratt

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

T = ABABABCABABABCABABABCABABAC
P = ABABAC

j = 2, i = 6

New j = F[1] = 0 (shift 2-0 = 2)
New i = 6 (no change)

T = ABABABCABABABCABABABCABABAC
P = ABABAC

Algorithm \textit{KMPMatch}(T, P)

\[
\begin{align*}
F & \leftarrow \text{failureFunction}(P) \\
i & \leftarrow 0 \\
j & \leftarrow 0 \\
\text{while } i < n & \\
& \quad \text{if } T[i] = P[j] \\
& \quad \quad \text{if } j = m - 1 \\
& \quad \quad \quad \text{return } i - j \{ \text{match} \} \\
& \quad \quad \text{else} \\
& \quad \quad \quad i \leftarrow i + 1 \\
& \quad \quad \quad j \leftarrow j + 1 \\
& \quad \text{else} \\
& \quad \quad \text{if } j > 0 \\
& \quad \quad \quad j \leftarrow F[j - 1] \\
& \quad \quad \text{else} \\
& \quad \quad \quad i \leftarrow i + 1 \\
& \quad \quad \quad j \leftarrow 0 \\
& \text{return } -1 \{ \text{no match} \}
\end{align*}
\]

Comparisons = 11 + 3 = 14
Algorithm KMPMatch(T, P)

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

\[
i \leftarrow 0
\]

\[
j \leftarrow 0
\]

while \( i < n \)

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

\( \text{if } j = m - 1 \)

\( \text{return } i - j \{ \text{match} \} \)

\( \text{else} \)

\( i \leftarrow i + 1 \)

\( j \leftarrow j + 1 \)

\( \text{else} \)

\( \text{if } j > 0 \)

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

\( \text{else} \)

\( i \leftarrow i + 1 \)

\( j \leftarrow 0 \)

\( \text{return } -1 \{ \text{no match} \} \)

Comparisons = 14+1 = 15
Algorithm **KMPMatch**(*T*, *P*)

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

\[ i \leftarrow 0 \]

\[ j \leftarrow 0 \]

while \( i < n \)

\[
\begin{align*}
\text{if } T[i] &= P[j] \\
\quad &\text{if } j = m - 1 \\
\quad &\quad \text{return } i - j \{ \text{match} \} \\
\quad &\text{else} \\
\quad &\quad i \leftarrow i + 1 \\
\quad &\quad j \leftarrow j + 1 \\
\text{else} \\
\quad &\text{if } j > 0 \\
\quad &\quad j \leftarrow F[j - 1] \\
\quad &\text{else} \\
\quad &\quad i \leftarrow i + 1 \\
\quad &\quad j \leftarrow 0 \\
\text{return } -1 \{ \text{no match} \}
\end{align*}
\]

**T** = ABABABCABABABABCABABACABABAC

**P** = ABABAC

\( j = 5, \ i = 12 \)

New \( j = F[4] = 3 \) (shift 5-3=2)

New \( i = 12 \) (no change)

**T** = ABABABCABABABABCABABACABABAC

**P** = ABABAC

\( i=12 \)

\( j=3 \)

Comparisons = 15 + 6 = 21
Algorithm \textit{KMPMatch}(T, P)

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

\[ i \leftarrow 0 \]

\[ j \leftarrow 0 \]

\[ \text{while } i < n \]

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

\[ \text{if } j = m - 1 \]

\[ \text{return } i - j \{ \text{match} \} \]

\[ \text{else} \]

\[ i \leftarrow i + 1 \]

\[ j \leftarrow j + 1 \]

\[ \text{else} \]

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

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

\[ \text{else} \]

\[ i \leftarrow i + 1 \]

\[ j \leftarrow 0 \]

\[ \text{return } -1 \{ \text{no match} \} \]

\[ \]
Algorithm **KMPMatch**(T, P)

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

\[
i \leftarrow 0
\]

\[
j \leftarrow 0
\]

\[\text{while } i < n\]

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

\[\text{if } j = m - 1\]

\[\text{return } i - j \{ \text{ match} \}\]

\[\text{else}\]

\[i \leftarrow i + 1\]

\[j \leftarrow j + 1\]

\[\text{else}\]

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

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

\[\text{else}\]

\[i \leftarrow i + 1\]

\[j \leftarrow 0\]

\[\text{return } -1 \{ \text{ no match} \}\]

---

**Knuth-Morris-Pratt**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

\[F \rightleftharpoons 0 \ 0 \ 1 \ 2 \ 3 \ 0\]

\[T = \text{ABABABCABABABCABABAC}\]

\[P = \text{ABABAC}\]

\[j = 2, \ i = 13\]

New \(j = F[1] = 0\) (shift 2-0=0)

New \(i = 13\) (no change)

\[T = \text{ABABABCABABABCABABAC}\]

\[P = \text{ABABAC}\]

\[i=13\]

\[j=0\]

**Comparisons** = 26+3 = 29
Algorithm KMPMatch(T, P)

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

\[ i \leftarrow 0 \]
\[ j \leftarrow 0 \]

while \( i < n \)

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

\[ \text{if } j = m - 1 \]

\[ \text{return } i - j \{ \text{match} \} \]

\[ \text{else} \]

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

\[ \text{else} \]

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

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

\[ \text{else} \]

\[ i \leftarrow i + 1 \]
\[ j \leftarrow 0 \]

\[ \text{return } -1 \{ \text{no match} \} \]
BOYER-MOORE – EXAMPLE

#2
Boyer-Moore

T = ABABABCABABABCABABAC
P = ABABAC

Comparison: 1

T = ABABA

P = ABABAC

Bad character rule

T = ABABABCBABABABCABABAC
P = ABABAC

Good suffix rule
Note: no suffix matches in the previous step!!!

Pick bad character rule shift:

T = ABABABCBABABABCABABAC
P = ABABAC
Boyer-Moore

T = ABABABCABABABABCABABAC
P = ABABAC

Comparison: 1

T = ABABABC
P = ABABAC

x

T = ABABABCABABABCABABAC
P = ABABAC

Bad character rule

T = ABABABABCABABABCABABAC
P = ABABAC

Good suffix rule
Note: no suffix matches in the previous step!!!

Pick either shift:

T = ABABABCABABABABCABABAC
P = ABABAC
Boyer-Moore

Comparison: 1

T = ABABABCABABABABCABABAC
P = ABABAC

X

T = ABABABCABABABABCABABAC
P = ABABAC

Bad character rule

T = ABABABCABABABABCABABAC
P = ABABAC

Good suffix rule
Note: no suffix matches in the previous step!!!

Pick bad character rule shift:

T = ABABABCABABABABCABABAC
P = ABABAC
Boyer-Moore

Comparison: 1

T = ABABABCABABABABCABABAC
P = ABABAC

T = ABABABCABABABCABABAC
P = ABABAC

Bad character rule

T = ABABABCDEFGABABABCABABAC
P = ABABAC

Good suffix rule
Note: no suffix matches in the previous step!!!

Pick bad character rule shift:

T = ABABABCDEFGAABABCABABAC
P = ABABAC

T = ABABABCDEFGABABABCABABAC
P = ABABAC

Pick bad character rule shift:
**Boyer-Moore**

\[ T = \text{ABABABCABABABCABABAC} \]
\[ P = \text{ABABAC} \]

Comparison: 1

\[ T = \text{ABABABCABABA}_B \text{CABABAC} \]
\[ P = \text{ABA}_B \text{BAC} \]

Bad character rule

\[ T = \text{ABABABCABABABCABABAC} \]
\[ P = \text{ABABAC} \]

Good suffix rule

Note: no suffix matches in the previous step!!!

Pick bad character rule shift:

\[ T = \text{ABABABCABABABCABABAC} \]
\[ P = \text{ABABAC} \]
Boyer-Moore

T = ABABABCABABABABCABABAC
P = ABABAC

Comparison: 1

T = ABABABCABABABABC
P = ABAB

Bad character rule

T = ABABABCABABABABCABABAC
P = ABABAC

Good suffix rule

Pick either:

T = ABABABCABABABABCABABAC
P = ABABAC
Boyer-Moore

T = ABABABCABABABCABABAC 
P = ABABAC 

Comparison: 1

T = ABABABCABABABCA 
P = ABAC

Bad character rule

T = ABABABCABABABCABABAC 
P = ABABAC

Good suffix rule

Pick bad character rule shift:

T = ABABABCABABABCABABAC 
P = ABABAC
Boyer-Moore

\[ T = \text{ABABABCABABABCABABAC} \quad \text{Comparison: 1} \]
\[ P = \text{ABABAC} \]
\[ x \]

\[ T = \text{ABABABCABABABCABABCABABAC} \quad \text{Bad character rule} \]
\[ P = \text{ABABAC} \]

\[ T = \text{ABABABCABABABCABABAC} \quad \text{Good suffix rule} \]
\[ P = \text{ABABAC} \]

Pick bad character rule shift and match:

\[ T = \text{ABABABCABABABCABABAC} \quad \text{Comparison: 6} \]
\[ P = \text{ABABAC} \]

Total comparisons: 14
BOYER-MOORE – EXAMPLE
#3
Boyer-Moore

\[ T = \text{ABABABCABABABCABCABCBAB} \]
\[ P = \text{ABCBAB} \]

Comparison: 4

\[ T = \text{ABABABCABABABCABCABCBAB} \]
\[ P = \text{ABCBAB} \]

\[ T = \text{ABABABCABABABCABCABCBAB} \]
\[ P = \text{ABCBAB} \]

Bad character rule

\[ T = \text{ABABABCABABABCABCABCBAB} \]
\[ P = \text{ABCBAB} \]

Good suffix rule 2

\[ \text{Suffix (BAB)} = \text{prefix}(P) = \text{AB} \]

Pick good suffix rule 2:

\[ T = \text{ABABABCABABABCABCABCBAB} \]
\[ P = \text{ABCBAB} \]
Boyer-Moore

\[ T = ABABABCABABABCBAB \quad \quad \text{Comparison: 1} \]

\[ P = ABCBAB \]

\[ x \]

\[ T = ABABABCABABABCBAB \quad \quad \text{Bad character rule} \]

\[ P = ABCBAB \]

\[ T = ABABABCABABABCBAB \quad \quad \text{Good suffix rule} \]

\[ P = ABCBAB \]

\[ \quad \text{No suffix match in previous step} \]

Pick either:

\[ T = ABABABCABABABCBAB \]

\[ P = ABCBAB \]
Boyer-Moore

T = ABABABCA\textcolor{red}{BABABABABCACBCBAB} \quad \text{Comparison: 4}
\text{P} = \textcolor{red}{ABCBAB} \quad \text{x}

T = ABABABCA\textcolor{red}{ABABABABCACBCBAB} \quad \text{Bad character rule}
\text{P} = \textcolor{red}{ABCBAB}

T = ABABABCA\textcolor{red}{BABABABCACBCBAB} \quad \text{Good suffix rule 2}
\text{P} = \textcolor{red}{ABCBAB}
\text{Suffix (BAB) = prefix(P) = AB}

Pick good suffix rule 2:

T = ABABABC\textcolor{red}{ABABABABCACBCBABABAB\textcolor{red}{ABCACBCBAB}}
\text{P} = \textcolor{red}{ABCBAB}
Boyer-Moore

Comparison: 1

T = ABABABCABABABABCABCBAB
P = ABCBAB

x

T = ABABABCABABABABCABCBAB
P = ABCBAB

Bad character rule

T = ABABABCABABABABCABCBAB
P = ABCBAB

Good suffix rule
No suffix match in previous step

Pick either:

T = ABABABCABABABABCABCABCBAB
P = ABCBAB
Boyer-Moore

Comparison: 3

Bad character rule

Good suffix rule 1 or 2
Suffix (AB) = prefix(P) = AB

Total comparisons: 19