**Biosequence Algorithms, Spring 2005**  
**Lecture 3: Boyer-Moore Matching**

---

**Boyer-Moore Algorithm**

The Boyer-Moore algorithm (BM) is the practical method of choice for exact matching. It is especially suitable if
- the alphabet is large (as in natural language)
- the pattern is long (as often in bio-applications)

The speed of BM comes from shifting the pattern \( P[1 \ldots n] \) to the right in longer steps. Typically less than \( m \) chars (often about \( m/3 \) only) of \( T[1 \ldots n] \) are examined

BM is based on three main ideas:

---

**Boyer-Moore: Main Ideas**

Longer shifts are based on
- examining \( P \) right-to-left, in order \( P[n], P[n-1], \ldots \)
- “bad character shift rule” avoids repeating unsuccessful comparisons against a target character
- “good suffix shift rule” aligns only matching pattern characters against target characters already successfully matched

Either rule alone works, but they’re more effective together

---

**Bad Character Rule Formally**

For any \( x \in \Sigma \), let \( R(x) = \max \{ 0 \cup \{ i < n \mid P[i] = x \} \} \)

Easy to compute in time \( \Theta(\Sigma + |P|) \) (\( \Sigma \) is the alphabet)

**Bad character shift**: When \( P[i] \neq T[h] = x \), shift \( P \) to the right by \( \max(1, i - R(x)) \). This means:
- if the right-most occurrence of \( x \) in \( P[1 \ldots n-1] \) is at \( j < i \), chars \( P[j] \) and \( T[h] \) get aligned
- if the right-most occurrence of \( x \) in \( P[1 \ldots n-1] \) is at \( j > i \), the pattern is shifted to the right by one
- if \( x \) doesn’t occur in \( P[1 \ldots n-1] \), shift \( i \), and the pattern is next aligned with \( T[h + 1 \ldots h + n] \)

---

**Right-to-left Scan and Bad Character Rule**

The pattern is examined right-to-left:

\[
\begin{align*}
1 & & 2 & & 3 \\
123456789012345678901234567890 & & & & \\
T: & \text{maistuko kaima maisemaomaloma?} & & P: & \text{maisemaomalom} \text{ [legend: match, mismatch]}
\end{align*}
\]

**Bad character rule**: Shift the next-to-left occurrence of \( i \) below the mismatched \( `i' \) of \( T \) right

\[
\begin{align*}
1 & & 2 & & 3 \\
123456789012345678901234567890 & & & & \\
T: & \text{maistuko kaima maisemaomaloma?} & & P: & \text{maisemaomalom} \text{ [legend: match, mismatch]}
\end{align*}
\]

---

**(Strong) Good Suffix Rule**

Bad character rule is effective, e.g., in searching natural language text (because mismatches are probable)

If the alphabet is small, occurrences of any char close to the end of \( P \) are probable. Especially in this case, additional benefit can be obtained from considering the successfully matched suffix of \( P \)

We concentrate on the so called strong good suffix rule, which is more powerful than the (weak) suffix rule of the original Boyer-Moore method

---

**Good Suffix Rule: Illustration**

Consider a mismatch at \( P[n-2] \):

\[
\begin{array}{cccc}
1 & 2 & 3 \\
123456789012345678901234567890 & & & \\
T: & \text{maistuko kaima maisemaomaloma?} & & P: & \text{maisemaomalom} \text{ [legend: match, mismatch]}
\end{array}
\]

In an occurrence, \( T[12 \ldots 14] = \text{ima} \) must align with "xma", where \( x \) differs from \( P[n-2] = `o' \) right

\[
\begin{array}{cccc}
1 & 2 & 3 \\
123456789012345678901234567890 & & & \\
T: & \text{maistuko kaima maisemaomaloma?} & & P: & \text{maisemaomalom} \text{ [legend: match, mismatch]}
\end{array}
\]

---

**Good Suffix Rule Formally**

Suppose that \( P[i \ldots n] \) has been successfully matched against \( T \)

**Case 1**: If \( P[i-1] \) is a mismatch and \( P \) contains another copy of \( P[i \ldots n] \) which is not preceded by char \( P[i-1] \), shift \( P \) s.t. the closest-to-left such copy is aligned with the substring already matched by \( P[i \ldots n] \)

(See the previous slide for an example)

What if no preceding copy of \( P[i \ldots n] \) exists?

\( \leftarrow \) Case 2
Consider a mismatch at $P[n - 5]$:

```
  1 2 3
12345678901234567890123456789012
T: mahtava talomaisena omalomaiun
P: maisemaomaloma
```

No preceding occurrence of "aloma" in $P$, but a potential occurrence of $P$ begins at $T[13\ldots 14] = "ma"$ →

```
  1 2 3
12345678901234567890123456789012
T: mahtava talomaisena omalomaiun
P: maisemaomaloma
```

### Preprocessing for the Good Suffix Rule (Case 1)

For $i = 2, \ldots, n + 1$, define $L(i)$ as the largest position of $P$ that satisfies the following:

- $L(i)$ is the end position of an occurrence of $P[j \ldots n]$ that is not preceded by char $P[i - 1]$;
- if no such copy of $P[j \ldots n]$ exists in $P$, let $L(i) = 0$

**NB 1:** $0 \leq L(i) < n$; If $L(i) > 0$, it is the right endpoint of the closest-to-left copy of "good suffix" $P[j \ldots n]$, which gives the shift $n - L(i)$

**NB 2:** Since $P[n + 1 \ldots n] = r$, $L(n + 1)$ is the right-most position $j$ s.t. $P[j] \neq P[n]$ (or 0 if all chars are equal).

### Computing the $L'$ Values (1)

Define $N_j(P)$ to be the length of the longest common suffix of $P[1 \ldots j]$ and $P$ ($\Rightarrow 0 \leq N_j(P) \leq j$)

**Example:** For

```
  1
12345678901234
P: maisemaomaloma
```

$N_0(P) = N_1(P) = 0$, $N_2(P) = 2$, $N_3(P) = \cdots = N_9(P) = 0$, $N_{10}(P) = 2$, $N_{11}(P) = \cdots = N_{13}(P) = 3$, $N_{14}(P) = \cdots = N_{15}(P) = 0$, $N_{16}(P) = 14$

### Computing the $L'$ Values (2)

Now $N_j$ (longest common suffix) values and $Z_i$ (longest common prefix) values are reverses of each other, i.e.,

$N_j(P) = Z_{n - j + 1}(P^r)$

where $P^r$ is the reverse of $P$

**Example:**

```
  1
123 45678
P: aamunamum
```

```
  1
876 5432 1
P^r: umunamum
```

The $N_j$ values can be computed in time $O(|P|)$ by applying Algorithm $Z$ to the reversal of $P$

### Computing the L' Values (3)

How do the $N_j$ values help?

**Theorem 2.2.2** If $L(i) > 0$, it is the largest $j < n$ for which $N_j(P) = |P[j \ldots n]| = n - i + 1$

**Proof.** Such $j$ is the right endpoint of the closest-to-left copy of $P[j \ldots n]$ which is not preceded by $P[i - 1]$

$\implies$ The $L'(i)$ values can be computed in $O(n)$ time by locating the largest $j$ s.t. $N_j(P) = n - i + 1$

$\forall$ such $j$ in $L'(i)$ for $i = n - N_j(P) + 1$:

- for $i = 2$ to $n + 1$ do $L'(i) := 0$;
- for $j = 1$ to $n - 1$ do $L'(n - N_j(P) + 1) := j$;

### Preprocessing for Case 2 (1)

How to compute the smallest shift that aligns a matching prefix of $P$ with a suffix of the successfully matched substring of $T = P[i \ldots n]$?

For $i \geq 2$, let $l(i)$ be the length of the longest prefix of $P$ (that is, $P[1 \ldots l(i)]$) that is equal to a suffix of $P[i \ldots n]$

**Example:** For $P = P[1 \ldots 5] = "ababa"

$l(6) = 0$ ($\Rightarrow P[6 \ldots 5] = r$), $l(5) = l(4) = 1$ ("a"), and $l(3) = l(2) = 3$ ("aba")
**Preprocessing for Case 2 (2)**

Now the following theorem holds

**Theorem 2.2.4** \( l(i) = \max \{0 \leq j \leq |P[i \ldots n]| \mid N_j(P) = j \} \)

**Proof.** (Left as an exercise)

This allows us to compute the \( l(i) \) values in time \( O(|P|) \)

\( \leftarrow \text{Exercise} \)

---

**Shifts by the Good Suffix Rule**

When \( P[i-1] \) is a mismatch (after matching \( P[i \ldots n] \) successfully)

- (Case 1) if \( L'(i) > 0 \), shift the pattern to the right by \( n - L'(i) \) positions
- (Case 2) if \( L'(i) = 0 \), shift the pattern to the right by \( n - l(i) \) positions

**NB** If already \( P[n] \) fails to match, \( i = n + 1 \), which also gives correct shifts

When an occurrence of \( P \) has been found, shift \( P \) to the right by \( n - l(2) \) positions. Why? To align a prefix of \( P \) with the longest matching proper suffix of the occurrence

---

**Which Shift to Use?**

Since neither the bad character rule nor the good suffix rule misses any occurrence, we can use the maximum of alternative shift values

**Complete Boyer-Moore Algorithm:**

// Preprocessing:
Compute \( R(x) \) for each \( x \in \Sigma \);
Compute \( L'(i) \) and \( l(i) \) for each \( i = 2, \ldots, n + 1 \);

---

**Final Remarks**

The presented rules carefully avoid performing unnecessary comparisons that would fail

They can be shown to lead to linear-time behavior, but only if \( P \) does not occur in \( T \). Otherwise the worst-case complexity is still \( \Theta(nm) \)

A simple modification ("Galil rule"; Gusfield, Sect. 3.2.2) corrects this and leads to a provable worst-case linear time.

On natural language texts the running time is almost always sub-linear