More Exact Matching

(Following Gusfield Chapter 2)
Knuth–Morris–Pratt

\[ |P| = n \]

\[ |T| = m \]
Knuth-Morris-Pratt (KMP)

- Shift by more than 1 place, if possible, upon mismatch.

**Def.** \(spm_i(P)\) = the length of the longest substring of \(P\) that ends at \(i > 1\) and matches a prefix of \(P\) and such that \(P[i+1] \neq P[spm_i + 1]\). ("spm" stands for suffix, prefix, mismatch.)

KMP Algorithm: Suppose mismatch at \(i+1\) of \(P\):

\[
\begin{align*}
P: & \quad \overbrace{\text{spm}_i}^{x} \quad \overbrace{\text{spm}_i}^{y} \\
T: & \quad \overbrace{\text{spm}_i}^{x} \\
\end{align*}
\]

\(\Rightarrow\) can shift by: \(i - spm_i\)
KMP

$c = p = 1$  // pointers into $T$ and $P$, respectively

while $c \leq |T| - |P| + p$:

    while $P[p] = T[c]$ and $p \leq n$:  // compare $P$ and $T$
        $p++$
        $c++$

    if $p = |P| + 1$: print "Found at", $c - |P|$  // if found

    if $p = 1$: // failure at start means inc $c$
        $c++$

    else:
        $p = spm_{p-1} + 1$  // "shift" by $n - spm_{p-1}$ (even if $p=n+1$)
KMP Running Time

Pseudocode runs in $O(|T|)$ time (making at most $2|T|$ comparisons):

- Every character is *matched* at most once (might be *mismatched* more than once) [Proof: $c$ is never decremented.]

- At any time $t$, there are $q_t \leq |P|$ characters that have been compared in $P$ and are currently matching.

- Each mismatch can be "charged" to a some shift of $P$ (because when there is a mismatch, we shift).

- When we shift $|P|$, we shift it by $\leq q_t$ so a mismatch can be "charged" so some matches we already performed.

- So total # of mismatches $< |T|$.

- Therefore: $O(2|T|)$ for the pseudocode on previous page.
Recall: Fundamental Preprocessing

**Def.** $Z_i(P)$ = the length of the longest substring of $P$ that starts at $i > 1$ and matches a prefix of $P$.

- $P = \text{“aardvark”}$: $Z_2 = 1, Z_6 = 1$
- $P = \text{“alfalfa”}$: $Z_4 = 4$
- $P = \text{“photophosphorescent”}$: $Z_6 = Z_{10} = 3$
Computing $spm_i$ for KMP

$f(j) = \text{the right end of the Z-box (if any) that starts at } j.$

$g(i) = \min \{ j : f(j) = i \}$ or 0 if empty set.

**Thm.** $spm_i = Zg(i)$ if $g(i) > 0$ otherwise 0

**Proof.**

$P[g(i) \ldots i] = P[1..Zg(i)]$ by the definition of $Z$.

Also, $P(i+1) \neq P[Zg(i)+1]$, otherwise $Zg(i)$ would be bigger.

So, $spm_i \geq Zg(i)$. But it can’t be longer, because otherwise $g(i)$ would be smaller.
Boyer–Moore
Boyer-Moore Main Ideas

1. For a given shift, compare $P$ to $T$ from right to left.

2. Two rules for shifting:
   1. Bad Character Rule
   2. Good Suffix Rule
Bad Character Rule

**Def.** $R_i(x) =$ position of the rightmost occurrence of character $x$ before position $i$.

- When a mismatch occurs at pattern position $i$:
  
  $T$: \[ T[1, k] \]
  
  $P$: \[ a \]
  
  $R_i(a) = R_i(T[k])$

  shift by $i - R_i(T[k])$ characters so that the next occurrence of $T[k]$ in the pattern is underneath position $k$ in $T$.

  (Called the “bad character rule” because it fires on a mismatch, but really it shifts so that the next *good* character matches.)
Computing \( R_i(x) \)

**Def.** \( R_i(x) = \) position of the rightmost occurrence of character \( x \) before position \( i \).

- Array \( R[i,x] \) would depend on the size of the alphabet, which is undesirable.
- Better to use a collection of lists (total size < \( O(|P|) \)):
  - \( \text{Occur}[x] = \) positions where \( x \) occurs in \( P \) in decreasing order.
- To find \( R_i(x) \):
  - scan down list \( x \) until you find first index < \( i \)
- **Time:** at most \( O(|P| - i) \) time, since if mismatch occurred at position \( i \) then there can be at most \( |P| - i \) items on the list that are \( \geq i \).
- Only call this routine after matching \( O(|P| - i) \) characters, so at most doubles the running time.
**Good Shift Rule**

**Case (A):** Shift so that the rightmost occurrence of matched suffix with different preceding character is aligned to matched part of $T$. 

**Case (B):** $\beta$ longest proper prefix of $P$ that matches a suffix of $\alpha$. Shift so that the prefix $\beta$ matches the suffix $\beta$ that was matched to $T$. 

**Case (C):** If not (A) or (B), shift $|P|$ places.

Apply these cases in order:
Processing the good suffix rule

**Def.** $L(i) =$ largest index such that $P[i..n]$ matches suffix of $P[1..L(i)]$ and $P[i-1] \neq$ the character preceding that suffix (0 if no such index exists).

**Def.** $l(i) =$ size of largest suffix of $P[i..n]$ that equals some *prefix* of $P$ (0 if none exists).

- Case (A): shift by $n - L(i)$.
- Case (B): if $L(i) = 0$: shift by $n - l(i)$ places.
- If match: shift by $n - l(2)$ places.
Computing L(i)

**Def.** $N_j(P) =$ length of longest suffix of $P[1..j]$ that is also a suffix of $P$.

Recall: **Def.** $Z_i(P) =$ the length of the longest substring of $P$ that starts at $i > 1$ and matches a prefix of $P$.

$N_j(P)$ and $Z_i(P)$ are reverses of each other:

\[ N_j(P) = Z_{n-j+1}(P^r) \]

\[ P^r \] is $P$ reversed.

Can compute in $O(n)$ time using Z-algorithm on $P^r$. 
Computing $L(i)$, continued

- $L(i) = \text{largest index } j \text{ such that } P[i..n] \text{ matches suffix of } P[1..L(i)] \text{ and } P[i-1] \neq \text{ the character preceding that suffix.}$

- $N_j(P) = \text{length of longest suffix of } P[1..j] \text{ that is also a suffix of } P.$

$$\Rightarrow L(i) = \text{largest index } j \text{ such that } N_j(P) = |P[i..n]| = n - i + 1$$

- $x \neq y$ because otherwise $N_j(P)$ would be longer.

Compute $N_j(P)$ via Z-Algorithm for all $j$.
Initialize $L[i] = 0$ for all $i$.

for $j = 1$ to $n - 1$:
  
  $i = n - N_j(P) + 1$
  
  $L[i] = j$
Boyer-Moore

\[ k = 1 \]
\[ \textbf{while} \ k < |T| - |P| + 1: \]
\[ \quad \text{Compare } P \text{ to } T[k..|P|] \text{ from right to left.} \]
\[ \quad s = \max \{ \text{bad character rule, good suffix rule, 1} \} \]
\[ \quad k += s \]

- Worst case running time = \( O(nm) \) since might shift by 1 every time.
- Despite this, Boyer-Moore often the best choice in practice because on real texts the running time is often sublinear (since the heuristics allow skipping a lot of characters).
- Extensions exist that guarantee \( O(|P| + |T|) \) running time.