More Exact Matching

(Following Gusfield Chapter 2)
Knuth-Morris-Pratt
Knuth-Morris-Pratt (KMP)

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

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

KMP Algorithm: Suppose mismatch at $i+1$ of $P$: $

\Rightarrow \text{can shift by: } i - spm_i$
$KMP$

\[
T: \quad \overbrace{\text{X}}^{c} \quad spm_{p-1} \quad y \quad new \; p \quad p
\]

\[\Rightarrow \text{can set new } p \text{ to } spm_{p-1} + 1\]

c = p = 1 \quad // \text{ptrs into } T \text{ and } P, \text{ respectively}

\[\textbf{while } c \leq |T| - |P| + p:\]

\[\textbf{while } P[p] = T[c] \textbf{ and } p \leq n: \quad // \text{compare } P \text{ and } T\]

\quad p++

\quad c++

\textbf{if } p = n + 1: \textbf{ print } “\text{Found at”, } c - n \quad // \text{if found}

\textbf{if } p = 1: \quad // \text{failure at start means inc } c

\quad c++

\textbf{else:}

\quad p = spm_{p-1} + 1 \quad // “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):

- In each iteration of the outer `while` loop, at most one character is compared that was compared in a previous iteration.
- Total comparisons: $\leq |T| + s$, where $s = \#$ of times through the outer `while` loop.
- $s \leq |T|$ since $P$ is shifted by $\geq 1$ each time.
- Therefore: $O(|T|)$ 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 $P$.

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

$f(j) =$ 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.**

$P[g(i) .. i] = P[1 .. Zg(i)]$ 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

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

```
thequickbrownfox
x
```

crown

- 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 $i$:

  $$ R_i(a) = R_i(T[k]) $$

  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:
  - \( \text{Occur}[x] = \) 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 \( O(n - i) \) time, since if mismatch occurred at position \( i \) then there can be at most \( n - i \) items on the list that are \( \geq i \).

- Only call this routine after matching \( O(n - i) \) characters, so at most doubles the running time.
Good Shift Rule

Apply these cases in order:

Case (A): If the rightmost occurrence of a matched suffix with different preceding character is aligned to matched part of $T$.

Case (B): The 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.
Processing the good suffix rule

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

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

- 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.
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: \( N_j(P) = Z_{n-j+1}(P^r) \), where \( P^r \) is \( P \) reversed. Can compute in \( O(n) \) time using Z-algorithm on \( P^r \).
Computing $L(i)$, continued

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

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

$\Rightarrow$ $L(i) =$ largest index $j$ 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 $i$.

\[
\text{for } j = 1 \text{ to } n - 1: \\
\quad i = n - N_j[P] + 1 \\
\quad L[i] = j
\]
Boyer-Moore

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

- 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.