Biosequence Algorithms, Spring 2007
Lecture 2

Pekka Kilpeläinen

University of Kuopio
Department of Computer Science
I: Exact String Matching
(tarkka (merkkijono)hahmon sovitus)

1. the **naive method**

2. a **linear-time method** based on “fundamental preprocessing”
Perhaps the most basic string problem of all:

Given **pattern** $P$ (*hahmo*) and **target** $T$ (*kohde*), find all **occurrences** of $P$ in $T$ (that is, substrings equal to $P$)

**Example:** Pattern $P = \text{"aba"}$ occurs in text

\[
i: \quad 123456789012 \\
T: \quad \text{bbabaxababay}
\]

at locations $i = 3$, $i = 7$, and $i = 9$

Multiple applications: word processing, file searching (Unix grep), information searching on the Net, *sequence databases*
For practical word-processing the problem can be considered solved.

Why then study exact matching?

- Efficient solutions relevant for sequence DBs. (For example, *over 4 hour* search in *Genbank* for a 30 char pattern using a popular interface (GCG) vs *a few minutes* using the Boyer-Moore algorithm.)

- Used as a *subtask* for more complex searches

- Basic ideas possibly applicable to new and less understood problems.
Naive Pattern Matching

Compare $P[1 \ldots n]$ char-by-char against each $n$-length substring of $T[1 \ldots m]$:

for $i := 1$ to $m - n + 1$ do
    if $T[i] = P[1]$ then
        $l := 1$; // chars matched
        while $l < n$ and $T[i + l] = P[1 + l]$ do $l := l + 1$
        if $l = n$ then Report a match at $i$
    endif;
endfor;

**Drawback:** $n(m - n + 1) = \Theta(nm)$ comparisons in the worst case; Rare in word processing, but probable if small alphabet and lots of repetitions in strings (as in bio-sequences)
Naive method “shifts” $P$ by one position along the target:

T: xabcdabcdabcx
P: abcdabcx

abcdabcx
abcdabcx
abcdabcx
abcdabcx
abcdabcx

(Legend: successful and unsuccessful comparison);
20 comparisons in total
Ideas for Speed-up I

I: Use longer shifts that avoid comparisons known to fail:

T: xabcdabcdabcx
P: abcdabcx

abcdabcx (AHA: P[1] doesn’t occur
abcdabcx until a shift by 4)

≈ total of 17 comparisons
Ideas for Speed-up II

II: Avoid comparisons known to succeed:

\[ T: \text{xabcdabcdabcx} \]
\[ P: \text{abcdabcx} \]

From earlier comparisons, we know the prefix “abc” to match; \( \sim \) total of 14 comparisons

Next: Preprocessing the pattern to implement these ideas

\( \sim \) linear-time \( (O(|P| + |T|)) \) pattern matching algorithms
Fundamental Preprocessing

By Gusfield, to explain diverse classical algorithms;
Also leads to simple linear time matching

Given a string $S[1 \ldots n]$ and $i \in \{2, \ldots, n\}$, define $Z_i$ to be the length of the longest common prefix of $S$ and $S[i \ldots n]$

Example: For $S[1 \ldots 11] = aabcaabxaaz$

$Z_2 = 1$, $Z_3 = Z_4 = 0$
$Z_5 = 3$ (← $S[5 \ldots 11] = abxaaz$)
$\vdots$
$Z_9 = 2$, $Z_{10} = 1$, $Z_{11} = 0$

If $S$ is not clear from context, we write $Z_i(S)$ instead of $Z_i$
How to compute the $Z_i$ values?

A direct approach:

\[
\begin{align*}
\text{for } & i := 2 \text{ to } n \text{ do} \\
& l := 0; \\
& \text{while } i + l \leq n \text{ and } S[i + l] = S[1 + l] \text{ do} \\
& \quad l := l + 1; \\
& \quad Z[i] := l; \\
& \text{endfor}
\end{align*}
\]

Time complexity $\Theta(\sum_{i=2}^{n} Z_i)$,
in the worst case $\Theta(\sum_{l=1}^{n-1} l) = \Theta(n^2)$

Need something more clever for linear time
For $Z_i > 0$, let the $Z$-box at $i$ be $S[i \ldots i + Z_i - 1]$ (occurrence of a maximal non-empty prefix starting at $i$).

For each $i \geq 2$, let $r_i$ be the right-most of endpoints of any $Z$-box at $j \leq i$. (If there is no such, let $r_i = 0$)

If $r_i > 0$, let $l_i$ be the startpoint $j$ of a $Z$-box $S[j \ldots r_i]$ occurring at $j \leq i$. (Otherwise $l_i = 0$.)
Example of $Z$-boxes

Example: (with $Z$-boxes surrounded by brackets, and indices below):

\[
\begin{array}{cccc|cccc|c}
  1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11
\end{array}
\]

Then

\[
\begin{align*}
  Z_2 &= 0, r_2 = l_2 = 0 \\
  Z_3 &= 5, r_3 = 7, l_3 = 3 \\
  Z_4 &= 0, r_4 = 7, l_4 = 3 \\
  Z_5 &= 3, r_5 = 7, l_5 = 5, (\text{or 3}) \\
  Z_8 &= 0, r_8 = 7, l_8 = 7, (\text{or 3}, \text{ or 5})
\end{align*}
\]
Basic strategy: a single scan of positions \( k = 2, \ldots, n \) in \( S \), utilizing \( Z_i \) values already computed (\( 2 \leq i < k \));

Variables \( l \) and \( r \) for the most recent \( l_i \) and \( r_i \); (That is, \( r \) is the right-most end of any \( Z \)-box seen so far, and \( l \) its start position)

First \( Z_2 \) is computed by comparing \( S[1 \ldots n] \) and \( S[2 \ldots n] \) explicitly, until the first mismatch
How to use computed $Z_i$ values?

**Example:** Suppose that $k = 121$, $r_{120} = 131$ and $l_{120} = 101$; position $k$ is inside the $Z$-box $S[101 \ldots 131] = S[1 \ldots 31]$. Thus $S[121 \ldots 131] = S[21 \ldots 31]$. (Draw a picture!)

Now if $Z_{21}$ is, say, 9, we know that $Z_{121} = 9$ (without examining any characters).

**General method** for computing $Z_2, \ldots, Z_n$,

the **$Z$ algorithm**:

Initialize: $l := 0; r := 0$;

Then compute $Z_k$ for each $k = 2, \ldots, n$ as follows:
for $k := 2, \ldots, n$ either case 1 or 2 applies:

1. if $k > r$ then // compute $Z_k$ directly:
   $Z_k := \max\{j \leq n - k + 1 \mid S[1 \ldots j] = S[k \ldots k + j - 1]\}$;
   If $Z_k > 0$, set $l := k$ and $r := k + Z_k - 1$;

2. if $k \leq r$, we’re inside $Z$-box $S[l \ldots r] = S[1 \ldots Z_l]$, and
   thus $S[k \ldots r] = S[k' \ldots Z_l]$ for $k' = k - l + 1$. (Draw it!)
   Let $t = |S[k \ldots r]| = r - k + 1$;
   (a) If $Z_{k'} < t$, we can set $Z_k := Z_{k'}$.
   (b) Otherwise $S[k \ldots r] = S[k' \ldots Z_l] = S[1 \ldots t]$. Find
      $j := \max\{j \leq n-r \mid S[r+1 \ldots r+j] = S[t+1 \ldots t+j]\}$;
      and set $Z_k := t + j$, $r := r + j$, and $l := k$;
**Correctness and Complexity**

**Theorem 1.4.1** Algorithm $Z$ is correct.

**Proof.** Straight-forward inspection.

**Theorem 1.4.2** Algorithm $Z$ works in time $O(|S|)$.

**Proof.** Each of the $|S| - 1$ iterations takes, besides the character comparisons (resulting in a match or a mismatch), constant time. Out of the character comparisons...

- each *mismatch* ends an iteration $\rightarrow$ number of them $< |S|$
- each *match* increments the value of $r$ at least by 1 $\rightarrow$ number of successful comparisons $\leq |S|$
The $Z$ algorithm provides a linear-time matching algorithm, which is perhaps the simplest of all:

Given $P[1\ldots n]$ and $T[1\ldots m]$, let $S := P$\$T$ (where $\$ appears in neither $P$ nor $T$);
Compute $Z_i(S)$ for $i = 2, \ldots, |S|$;
This takes time $O(n + m)$

Because of ’$’ each $Z_i \leq n$.

Now each position $i > n + 1$ with $Z_i = n$ (and only such) indicates an occurrence of $P$ in $T$ at position $i - (n + 1)$. 
Space Complexity

How much space do the \( Z \) values need?

Computed \( Z_{k'} \) values are used in Case 2 of Algorithm \( Z \).
There we have \( k \leq r \) and \( S[k \ldots r] = S[k' \ldots Z_l] \).
Therefore \( k' \leq Z_l \leq n \).
Thus we need to store \( Z_i \) values for \( i \leq n \) only,
using \( O(n) = O(|P|) \) space

NB After the preprocessing, algorithm \( Z \) performs exactly
the comparisons shown on Slide “Ideas for Speed-up II”
btw characters of \( P \) and \( T \)
Why Continue?

We have a simple linear-time matching algorithm. Why to study others?

- The *Boyer-Moore* algorithm is efficient in practice ("sub-linear time")

- *Knuth-Morris-Pratt* generalizes to matching a set of patterns in linear time \( \leadsto \text{Aho-Corasick} \) algorithm

- *suffix trees* support, after \( O(|T|) \) time preprocessing, matching in time \( O(|P|) \) (and have many other applications)