I: Exact String Matching

1. the naive method
2. a linear-time method based on “fundamental preprocessing”

Exact String Matching Problem

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 \( P = \) “aba” occurs in text

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

at locations \( i = 3 \), \( i = 7 \), and \( i = 9 \)

Multiple applications: word processing, file searching (Unix grep), information searching on the Net, sequence databases

Naive Pattern Matching

Compare \( P[1...n] \) char-by-char against each \( n \)-length substring of \( T[1...m] \):

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

Ideas for Speed-up I

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

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

\[ \text{abcdbcx} \]

(ANA: \( P[1] \) doesn’t occur \( \text{abcdabcx} \) until a shift by 4)

\( \approx \) total of 17 comparisons

Ideas for Speed-up II

II: Avoid comparisons known to succeed:

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

\[ \text{abcdabcx} \]

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

Next: Preprocessing the pattern to implement these ideas
\( \approx \) linear-time \( \Theta(|P| + |T|) \) 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 $S[1 \ldots 11] = aabcaabxaaz$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$Z_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

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:

- for $i := 2$ to $n$ do
  - $l := 0$
  - while $i + l \leq n$ and $S[i + l] = S[l]$ do
    - $l := l + 1$
  - $Z_i := l$

**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…

**Example of $Z$-boxes**

Numbers $3, 6, 7, 9$ give elements of the $Z$-box, and $121$ is, say, the last element of the $Z$-box.

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>l</th>
</tr>
</thead>
</table>
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

Then

$Z_2 = 0, Z_3 = 2, Z_4 = 5, Z_5 = 7, Z_6 = 3, Z_7 = 3, Z_8 = 3, Z_9 = 5, Z_9 = 5, Z_9 = 5$.

**How to use computed $Z_i$ values?**

**Example**: Suppose that $k = 121, l_{120} = 131$ 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 $Z_{21} = 9$ (without examining any characters).

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

the $Z$ algorithm:

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

Then compute $Z_i$ 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
   - $Z_k := \max \{j \leq n - k + 1 | S[1 \ldots j] = S[k \ldots k + j - 1]\}$
   - 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 $Z_r \leq l$, we can set $Z_r := Z_r$.
     - (b) Otherwise $S[l \ldots r] = S[k' \ldots Z_l] = S[1 \ldots t]$. Find
       - $j := \max \{j \leq n - r | S[r + 1 \ldots r + j] = S[t + 1 \ldots t + j]\}$
       - and set $Z_r := t + j, r := r + j, and l := k$.
Simplest Linear-Time Matching

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

Because of '$\approx$' 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 $(n + 1)$.

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 $\rightarrow$ Aho-Corasick algorithm
- suffix trees support, after $O(|T|)$ time preprocessing, matching in time $O(|P|)$ (and have many other applications)

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 $S[k \ldots r] = S[r' \ldots Z_i]$.
Therefore $r' \leq Z_i \leq n$.
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"