**Exact String Matching Problem**

Perhaps the most basic string problem of all:

Given pattern \( P \) and target \( T \), find all occurrences of \( P \) in \( T \) (that is, substrings equal to \( P \)).

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

\[
\begin{align*}
i & : 123456789012 \\
T & : \text{bbabaxababay}
\end{align*}
\]

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 \( T[1..m] \):

\[
\begin{align*}
\text{for } i & = 1 \text{ to } m - n + 1 \text{ do} \\
& \quad \text{if } T[i] = P[1] \text{ then} \\
& \quad \quad \text{if } l = 1 \text{ and } T[i+l] = P[l+1] \text{ do } l := l + 1; \\
& \quad \quad \text{if } l = n \text{ then Report a match at } i; \\
& \quad \text{endif} \\
& \quad \text{endif}; \\
& \text{endfor};
\end{align*}
\]

**Drawback:** \( n(m - n + 1) = \Theta(mn) \) 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:

\[
\begin{align*}
T & : \text{xabcdabcdabcx} \\
P & : \text{abcdabcx} \\
& \quad \text{abcdabcx} \quad \text{(AHA: } P[1] \text{ doesn't occur until a shift by 4) }
\end{align*}
\]

\( \sim \) total of 17 comparisons

**Ideas for Speed-up II**

II: Avoid comparisons known to succeed:

\[
\begin{align*}
T & : \text{xabcdabcdabcx} \\
P & : \text{abcdabcx} \\
& \quad \text{abcdabcx} \quad \text{ababcdcx} \quad \text{abcx}
\end{align*}
\]

From earlier comparisons, we know the prefix \( \text{"abc"} \) 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**

Developed 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 $S[1 \ldots 11] = aabcazbzbaz$,  

$$Z_2 = 1, Z_3 = Z_4 = 0$$  

$$Z_5 = 3 (\equiv S[5 \ldots 11] = aabcazbaz)$$  

$$Z_6 = 2, Z_{10} = 1, Z_{11} = 0$$

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 $\Theta(n^2)$ time $O(n)$.

**Definitions** for a linear time solution:

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

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

---

**Example of $Z$-boxes**

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

| a | b | b | a | b | a | | x | a | b | a |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

Then  

$Z_2 = 0, r_2 = l_2 = 0$  

$Z_3 = 5, r_3 = l_3 = 3$  

$Z_4 = 0, r_4 = l_4 = 3$  

$Z_5 = 3, r_5 = l_5 = 3$ (or 3)  

$Z_6 = 0, r_6 = l_6 = 7$ (or 3, or 5)

---

**Fundamental Preprocessing in Linear Time**

**Basic method:** a single scan of positions $k = 2, \ldots, n$ in $S$, utilizing $Z_i$ values already computed (2 \leq k < n);  

Variables $l$ and $r$ for the most recent $l_i$ and $r_i$. (That is, $r_i$ is the right-most end of any $Z$-box seen so far)

To begin, $Z_2$ 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 $l_{120} = 101$; we’re inside $Z$-box $S[101 \ldots 131] = S[1 \ldots 31]$. Thus $S[121 \ldots 131] = S[21 \ldots 31]$. (Draw a picture!)

Now if $Z_{2i}$ is, say, 9, we know that $Z_{2i+1} = 9$ (without examining any characters).

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

the $Z$ algorithm:

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

Then compute $Z_i$ for each $k = 2, \ldots, n$ as follows:

---

**The Z Algorithm**

for $k := 2, \ldots, n$ either case 1 or case 2 applies:

1. if $k > r$ then  

   $Z_k = \max\{j \leq n - k + 1 | S[1 \ldots j] = S[k \ldots k + j - 1]\}$  
   
   If $Z_k > 0$, set $l := k + Z_k - 1$;

2. if $k \leq r$,

   we’re inside $Z$-box $S[k \ldots r] = S[1 \ldots Z_k]$, and thus $S[k \ldots r] = S[k \ldots r]$ for $k' = k - 1 + t$.

   (Draw a picture!)

   Let $t := S[k \ldots r]$;

   (a) If $Z_k < k$, we know to set $Z_k := Z_k$;

   (b) Otherwise $S[k \ldots r] = S[k' \ldots Z_k] = S[1 \ldots t]$. Find  

      $j = \max\{j \leq n-r | S[r+1 \ldots r+j] = S[t+1 \ldots t+j]\}$  
      
      and set $Z_k := t + j, r := r + j$, and $i := k'$.  

---

**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 $\leq |S|$.

---

**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 $S := P$ (where $S$ appears in neither $P$ nor $T$);  

Compute $Z_i(S)$ for $i = 2, \ldots, m + n + 1$;  

This takes time $O(n + m)$.

Because of $S$ 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 $i - (n + 1)$.  

---

*BSA Lecture 2: Exact string matching – p.13/18*
Space Complexity

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

Computed \( Z_0 \) 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 \( k' \leq Z_l \leq n \), and thus it suffices to store \( Z_l \) values for \( i \leq n \), i.e., to use \( O(|P|) \) 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’ve got a simple linear-time matching algorithm. Why to study others?

1. Boyer-Moore algorithm is very efficient in practice ("sub-linear time")
2. Knuth-Morris-Pratt generalizes to matching a set of patterns in linear time --- Aho-Corasick algorithm
3. suffix trees support, after \( O(|T|) \) time preprocessing, matching in time \( O(|P|) \) (and have many other applications)