String algorithms I & II
Prof. Manolis Kellis

Lecture 11

String algorithms

- Today: String matching I
  - The exact string matching problem
  - Naïve algorithm
  - Preprocessing the query:
    - Fundamental pre-processing
    - Knuth-Morris-Pratt algorithm
    - Boyer-Moore algorithm
    - Z algorithm algorithm
  - Semi-numerical string matching
    - Rabin-Karp algorithm
- Wednesday: String matching II
  - Finite state machines
  - Suffix-trees
  - Inexact matching

The exact matching problem

- Inputs:
  - a string \( P \), called the pattern
  - a longer string \( T \), called the text
- Output:
  - Find all occurrences, if any, of pattern \( P \) in text \( T \)
- Example
  \[
  P = \text{aba} \\
  T = \text{baabacabab}
  \]

Basic string definitions

- A string \( S \) – Ordered list of characters
  - Written contiguous from left to write
- A substring \( S[i..j] \) – all contiguous characters from \( i \) to \( j \)
  - Example: \( S[3..7] = \text{abaxa} \)
- A prefix is a substring starting at 1
- A suffix is a substring ending at \( |S| \)
- \( |S| \) denotes the number of characters in string \( S \)

The naïve string-matching algorithm

- NAÏVE STRING MATCHING
  1. \( n \leftarrow \text{length}[T] \)
  2. \( m \leftarrow \text{length}[P] \)
  3. for shift \( \in \{0 \ldots n\} \) do
     if \( P[1..m] == T[\text{shift}+1 \ldots \text{shift}+m] \)
       then print “Pattern occurs with shift” \( \text{shift} \)
  4. \( \text{Running time: } O(n) \) \( \text{O}(m) \)
- Where the test operation in line 4:
  - Tests each position in turn
  - If match, continue testing
  - Else: stop
- Running time – number of comparisons
  number of shifts (with one comparison each)
  + number of successful character comparisons

Comparisons made with naïve algorithm

- Worst case running time:
  - Test every position
  - \( P = \text{aaaa}, T = \text{aaaaaaaaaaaa} \)
- Best case running time:
  - Test only first position
  - \( P = \text{bbbb}, T = \text{aaaaaaaaaaaa} \)

Can we do better?
Key insight: make bigger shifts!

• If all characters in the pattern are the same:

```
<table>
<thead>
<tr>
<th>a a a a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a a a</td>
</tr>
<tr>
<td>a a a a</td>
</tr>
</tbody>
</table>
```

Information gathered at every comparison

Knowledge of the internal structure of P

Number of comparisons: O(n)

Key insight: make bigger shifts!

• If all characters in the pattern are different:

```
<table>
<thead>
<tr>
<th>a b c d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c d</td>
</tr>
<tr>
<td>a b c d</td>
</tr>
</tbody>
</table>
```

Number of comparisons:
- At most n matching comparisons
- At most n non-matching comparisons

⇒ Number of comparisons: O(n)

Key insight: make bigger shifts!

• Special case:
  - If all characters in the pattern are the same: O(n)
  - If all characters in the pattern are different: O(n)

• General case:
  - Learn internal redundancy structure of the pattern
  - Pattern pre-processing step

• Methods:
  - Fundamental pre-processing
  - Knuth-Morris-Pratt
  - Finite State Machine

Fundamental pre-processing

• Learning the redundancy structure of a string S

```
S = a a b c | a a | b x | a a a
Z = 0 1 0 0 3 1 0 2 2 1
```

```
Z-box = a a b c | a a | b x | a a a
```

```
r = a a b c | a a | b x | a a a
```

```
l = a a b c | a a | b x | a a a
```

```
Z1 Z2 Z3 … Zk-1 Zk
```

Computing Zk given Z1 … Zk-1

• Case 1: k is outside a Z-box: simply compute Zk

```
Can we compute Z, r, l in linear time O(|S|)?
```

• Case 2: k is inside a Z-box: Look up Zk

⇒ Case 2a: Zk < r-k
⇒ Case 2b: Zk >= r-k
Computing $Z_k$ given $Z_1 \ldots Z_{k-1}$

Case 2a: $Z_k < r-k$

Case 2b: $Z_k \geq r-k$

Explicitly compare starting at $r+1$

Correctness of $Z$ computation

Case 1: $k$ is outside a $Z$-box: explicitly compute $Z_k$

Case 2a: Inside $Z$-box and $Z_k < r-k$: set $Z_k = Z_k$

Case 2b: Inside $Z$-box and $Z_k \geq r-k$: explicitly compute starting at $r+1$

What's so fundamental about $Z$?

- Learning the redundancy structure of a string $S$

- $Z_i$ = fundamental property of internal redundancy structure

- Most pre-processings can be expressed in terms of $Z$
  - Length of the longest prefix starting/ending at position $i$
  - Length of the longest suffix starting/ending at position $i$

Putting it all together

- FUNDAMENTAL-PREPROCESSING($S$):
  
  ```
  for k in 2..n:
    if k > r:
      $Z_k,l,r = \text{explicitly compare } S[1..] \text{ with } S[k..]$
    if k <= r:
      if $Z_k < (r-k)$:
        Set $Z_k = Z_k$
      else:
        Set $Z_k = \text{explicitly compare } S[r+1..] \text{ with } S[(r-k)+1..]$
        $l = k$
        $r = l + Z_k$
  ```

Running time of $Z$ computation

Case 1: $k$ is outside a $Z$-box: explicitly compute $Z_k$

Case 2a: Inside $Z$-box and $Z_k < r-k$: set $Z_k = Z_k$

Case 2b: Inside $Z$-box and $Z_k \geq r-k$: explicitly compute starting at $r+1$

Back to string matching

- Given the fundamental pre-processing of pattern $P$
  - Compare pattern $P$ to text $T$
  - Shift $P$ by larger intervals based on values of $Z$

- Three algorithms based on these ideas
  - Knuth-Morris-Pratt algorithm
  - Boyer-Moore algorithm
  - $Z$ algorithm
Knuth-Morris-Pratt algorithm

- Pre-processing:
  - $S_p(P)$ = length of longest proper suffix of $P[1..i]$ that matches a prefix of $P$

- No other than the right-hand-side of the Z-boxes

Knuth-Morris-Pratt running time

- Number of comparisons bounded by characters in $T$
  - Every comparison starts at text position where last comparison ended
  - Every shift results in at most one extra comparison
  - At most $|T|$ shifts $\Rightarrow$ Running time bounded by $2|T|

Boyer-Moore algorithm

- Three fundamental ideas:
  1. Right-to-left comparison
  2. Alphabet-based shift rule
  3. Preprocessing-based shift rule

- Results in:
  - Very good algorithm in practice
  - Rule 2 results in large shifts and sub-linear time
  - Rule 3 ensures worst-case linear behavior
    - even in small alphabets, ex: DNA sequences

The Z algorithm

- The Z algorithm
  - Concatenate $P + \$$ + $T$
  - Compute fundamental pre-processing $O(m+n)$
  - Report all starting positions $i$ for which $Z_i = |P|

Semi-numerical string matching

Karp-Rabin algorithm

- Interpret strings as numbers
- Compute next number based on previous one
Computing the hash scores in linear time

- Use previous score to compute the next one

\[
\begin{align*}
14152 &= (31415-3\times10000)\times10^2 \pmod{13} \\
&= (7-3\times3)\times10^2 \pmod{13} \\
&= 8 \pmod{13}
\end{align*}
\]

- Other semi-numerical methods
  - Fast Fourier Transform
  - Shift-And method

String algorithms

- Today: String matching I
  - The exact string matching problem
  - Naïve algorithm
  - Preprocessing the query:
    - Fundamental pre-processing
    - Knuth-Morris-Pratt algorithm
    - Boyer-Moore algorithm
    - Z algorithm algorithm
  - Semi-numerical string matching
    - Rabin-Karp algorithm
- Wednesday: String matching II
  - Suffix-trees
  - Inexact matching
- Friday: Finite State Machines
  - String matching automata