**String algorithms I & II**

Prof. Manolis Kellis

Lecture 11

---

**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**

\[
P = a b a
\]

\[
T = b a a b a c a b a b a d
\]

---

**String algorithms**

- **Today:** String matching I
  - The exact string matching problem
  - Naïve algorithm
  - Preprocessing the query:
    - Fundamental preprocessing
    - 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 naïve string-matching algorithm**

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

---

**Basic string definitions**

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

---

**Comparisons made with naïve algorithm**

- **Worst case running time:**
  - Test every position
  - \( P=aaaa, T=aaaaaaaaaaa \)
- **Best case running time:**
  - Test only first position
  - \( P=bbbb, T=aaaaaaaaaa \)

---

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

• If all characters in the pattern are the same:

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:

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

$\Rightarrow$ 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 $S$

$S = a b c d e f g h i j k$ $Z = z_0 z_1 z_2 z_3 z_4 z_5 z_6 z_7 z_8 z_9 z_{10}$

$Z_{\text{box}} = 1 2 3 4 5 6 7 8 9 10 11$
$r = a b c d e f g h i j k$
$l = a b c d e f g h i j k$

$Z_1 Z_2 Z_3 \ldots Z_{k-1} Z_k$

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

$S = a b c d e f g h i j k$ $Z_k = z_{11}$ $Z_{k'}$

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

Computing $Z_k$ given $Z_1 \ldots Z_{k-1}$

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

$S = a b c d e f g h i j k$ $Z_k = z_{11}$ $Z_{k'}$

$\Rightarrow$ Case 2a: $Z_k < r-k$
$\Rightarrow$ Case 2b: $Z_k \geq r-k$
Computing $Z_k$ given $Z_1 \ldots Z_{k-1}$

Case 2a: $Z_k < r-k$

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

Correctness of Z computation

Running time of Z computation

Putting it all together

FUNDAMENTAL-PREPROCESSING(S):

$$Z_{2,1,r} = \text{explicitly compare } S[1..] \text{ with } S[2..]$$

for $k$ in 2..n:

- if $k > r$: $Z_{k,1,r}$ = explicitly compare $S[1..]$ with $S[k..]$
- if $k < r$:
  - if $Z_k < (r-k)$: $Z_k = Z_k'$
  - else:
    - $Z_k = \text{explicitly compare } S[r+1..] \text{ with } S[(r-k)+1..]$

$$l = k$$
$$r = l + Z_k$$

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 $P$

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 $Z_i = |P|$
Computing the hash scores in linear time

- Use previous score to compute the next one

\[ 14152 = (31415-3*10000)*10+2 \mod 13 \]
\[ = (7-3*3)*10+2 \mod 13 \]
\[ = 8 \mod 13 \]

- 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