Computing Patterns in Strings II: Generic Patterns

Bill Smyth[1,2,3]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada
email: smyth@mcmaster.ca

[2] Digital Ecosystems & Business Intelligence Institute
Curtin University, Perth, Western Australia
email: b.smyth@curtin.edu.au

[3] Department of Computer Science
King's College London, UK

DEBII 2008

Outline

1. Abstract
2. Applications
3. Five Generic Patterns
   - Pattern-Matching: the Ubiquitous Border Array
   - Compression: Lempel-Ziv Factorization
   - Repeats
   - Repetitions
   - A Beautiful Pattern: Lyndon Decomposition

Outline

1. Abstract
2. Applications
3. Five Generic Patterns
   - Pattern-Matching: the Ubiquitous Border Array
   - Compression: Lempel-Ziv Factorization
   - Repeats
   - Repetitions
   - A Beautiful Pattern: Lyndon Decomposition

Outline

1. Abstract
2. Applications
3. Five Generic Patterns
   - Pattern-Matching: the Ubiquitous Border Array
   - Compression: Lempel-Ziv Factorization
   - Repeats
   - Repetitions
   - A Beautiful Pattern: Lyndon Decomposition

Outline

1. Abstract
2. Applications
3. Five Generic Patterns
   - Pattern-Matching: the Ubiquitous Border Array
   - Compression: Lempel-Ziv Factorization
   - Repeats
   - Repetitions
   - A Beautiful Pattern: Lyndon Decomposition

Abstract

In this talk, the second of three, we consider generic patterns in strings — generally those that describe regularities: borders, repeating substrings, regular decompositions/factorizations of strings, periodicities.

Outline

1. Abstract
2. Applications
3. Five Generic Patterns
   - Pattern-Matching: the Ubiquitous Border Array
   - Compression: Lempel-Ziv Factorization
   - Repeats
   - Repetitions
   - A Beautiful Pattern: Lyndon Decomposition

Applications

- Facilitating skips or shifts of the pattern along the text in specific pattern-matching algorithms — the border array calculation.
- Data compression (`gzip` and others).
- Identifying cloned or near-cloned methods/classes in large software systems (e.g., the Wilderness that is Windows).
- Identifying repetitive or periodic segments (exact or approximate) in web pages, e-mail transmissions, encoded material.
- Finding repetitive or periodic segments of DNA (a result of transciption) indicating common functionality of genes or chromosomes.

Outline

1 Abstract

2 Applications

3 Five Generic Patterns
- Pattern-Matching: the Ubiquitous Border Array
- Compression: Lempel-Ziv Factorization
- Repeats
- Repetitions
- A Beautiful Pattern: Lyndon Decomposition

Border & Period

A string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ has period p if for every $i \in 1..n{-}p$, $\boldsymbol{x}[i] = \boldsymbol{x}[i{+}p]$. So

$$\begin{array}{cccccccccc} & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{t} & = & a & b & c & a & b & c & a & b \end{array}$$

has period 3.

A string $\boldsymbol{x}$ has border $\boldsymbol{u}$ if $\boldsymbol{x} = \boldsymbol{uv}$ and $\boldsymbol{x} = \boldsymbol{wu}$ ($\boldsymbol{u}$ both a prefix and a suffix of $\boldsymbol{x}$) and $len(\boldsymbol{u}) < len(\boldsymbol{x})$. Let $m = len(\boldsymbol{u})$. (We let the empty string be a border, so maybe $m = 0$.) Then for every $i \in 1..m$, $\boldsymbol{x}[i] = \boldsymbol{x}[i{+}(n{-}m)]$. A miracle — $\boldsymbol{x}$ has period $p = n{-}m$!!

Example string $\boldsymbol{t}$ has a border of length 5 and period $8{-}5 = 3$.

The Border Array

In the border array $\beta = \beta[1..n]$ of $\boldsymbol{x}$, for every $i \in 1..n$, $\beta[i]$ is the length of the longest border of $\boldsymbol{x}[1..i]$. Remember Fibonacci?

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{f} =$	a	b	a	a	b	a	b	a	a	b	a	a	b
$\beta =$	0	0	1	1	2	3	2	3	4	5	6	4	5

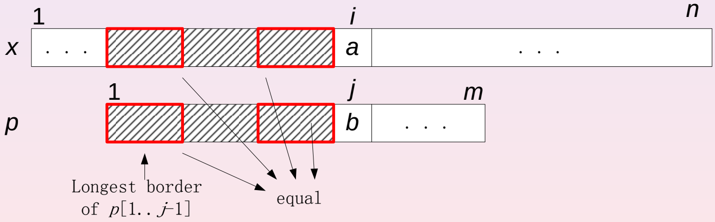
An amazing property:

- If $\boldsymbol{u}$ is the longest border of $\boldsymbol{x}$, and $\boldsymbol{v}$ is the longest border of $\boldsymbol{u}$, then $\boldsymbol{v}$ is the second longest border of $\boldsymbol{x}$!

So the border array gives all the borders (hence all the periods) of every prefix of $\boldsymbol{x}$. Furthermore, the border array can be computed in time proportional to n ($O(n)$ time) by a simple and elegant algorithm.

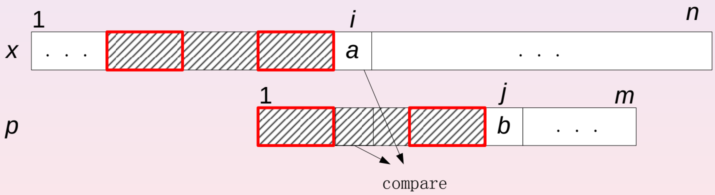
KMP Revisited I

The border array of $\boldsymbol{p}$ is precomputed in time proportional to m. So the longest border of every prefix is known.



KMP Revisited II

Thus when a mismatch occurs, the correct shift length is known that replaces the suffix of ***p*** with the prefix of ***p***.



Periodicity

We have looked at a very simple structure — the border array — that is easy to compute and describes all the periods of all the prefixes of a string.

The border array is used in dozens of algorithms to take advantage of the periodicity of a pattern in order to speed up processing.

The Periodicity Lemma

This is the mathematical foundation of stringology (combinatorics on words), often used in proofs of theorems required to show the correctness of algorithms:

Let p and q be two periods of $\boldsymbol{x} = \boldsymbol{x}[1..n]$, and let $d = \gcd(p, q)$. If $p + q \leq n + d$, then d is also a period of $\boldsymbol{x}$.

As an example, consider the string

$$
\begin{array}{ccccccccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\
\boldsymbol{x} = & a & b & a & a & a & b & a & a & a & b & a & a & a & b & a & a & a & b
\end{array}
$$

of length $n = 18$ and periods $q = 12$, $p = 8$: since $d = \gcd(p, q) = 4$ and $p + q = 20 < n + d = 22$, we conclude that $d = 4$ is also a period of $\boldsymbol{x}$.

Compression

- Mostly compression will achieve a 40-60% reduction in file size — if the file size is 10GB, this saves 4–6GB. Maybe worthwhile!
- The basic idea of lossless compression is to replace long repeating substrings with much shorter ones; for example, we can make the replacement

$$abcdgabcdabcd \longrightarrow AgAA$$

  if we have a "dictionary" to tell us that A is really $abcd$.
- This is the basic idea of the Lempel-Ziv (`gzip`) approach — probably used without your knowledge every time you send or receive an e-mail attachment. LZ compression goes back to 1976!

What is the LZ Factorization?

The LZ factorization $\mathrm{LZ}_{\boldsymbol{x}}$ of $\boldsymbol{x}$ is a factorization $\boldsymbol{x} = \boldsymbol{w_1 w_2} \cdots \boldsymbol{w_k}$ such that each $\boldsymbol{w_j}$, $j \in 1..k$, is

(a) a letter that does *not* occur in $\boldsymbol{w_1 w_2} \cdots \boldsymbol{w_{j-1}}$; or otherwise

(b) the longest substring that occurs at least twice in $\boldsymbol{w_1 w_2} \cdots \boldsymbol{w_j}$.

Forever Fibonacci! For

$$\boldsymbol{f} = \underline{a}\ \underline{b}\ \underline{a}\ \underline{a\ b\ a}\ \underline{b\ a\ a\ b\ a}\ \underline{a\ b}$$

the factorization is

$$\boldsymbol{w_1} = a,\ \boldsymbol{w_2} = b,\ \boldsymbol{w_3} = a,\ \boldsymbol{w_4} = aba,\ \boldsymbol{w_5} = baaba,\ \boldsymbol{w_7} = ab.$$

LZ Does the Job.

For long strings, LZ usually identifies long repeating substrings that form the basis of effective compression.

LZ can be computed in $O(n)$ time. Fast both for compression and decompression.

And LZ is multipurpose: it is the basis of the most efficient algorithm for computing all the repetitions in $\boldsymbol{x}$ (see below).

Since 1993 LZ has a worthy competitor: the Burrows-Wheeler transform (BWT).

Understanding Repeats

A repeat R is a collection of identical repeating substrings in $\boldsymbol{x}$; R is complete if it contains all of them.

So for (surprise!)

$$\begin{array}{cccccccccccccccc} & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ \boldsymbol{f} & = & a & b & a & a & b & a & b & a & a & b & a & a & b & a \end{array}$$

we can represent all the occurrences of *aba* by a complete repeat

$$R = (3;\ 1, 4, 6, 9, 12),$$

where $\ell = 3$ is the length of the repeating substring and $1, 4, 6, 9, 12$ are the positions at which it occurs.

Understanding Repeats II

In this string

$$\begin{array}{ccccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ \boldsymbol{f} = & a & b & a & a & b & a & b & a & a & b & a & a & b & a \end{array}$$

the repeats of *ab* are NOT interesting: they can all be right-extended with *a*, and so *aba* must occur at all the same locations. Similarly the repeats of *ba* are not interesting: they can all be left-extended with *a*.

However, *aba* is nonextendible (NE) and so interesting.

In general, we only need to output NE complete repeats!

Computing Repeats

All NE complete repeats in $\boldsymbol{x}$ of length $\ell \geq \ell_{\text{min}}$ can be computed in $O(n)$ time — fast. (Munira Yusufu's research.)

And supernonextendible repeats can be computed even faster: NE repeats that are not substrings of any other repeat in $\boldsymbol{x}$.

So to get an overview of repeats in any string (web pages, e-mail transmissions, long documents), only linear time is required — perhaps a useful tool of research and analysis.

What is a Repetition?

A repetition is a repeat of adjacent substrings:

$$\boldsymbol{x} = \cdots dabcabcabcad \cdots$$

contains three repetitions $(abc)^3$, $(bca)^3$, $(cab)^2$.

It was shown 25 years ago that over all strings of length n, the maximum number of repetitions is $O(n \log n)$ — achieved by Fibonacci strings of length n (of course).

To output $O(n \log n)$ repetitions must take $O(n \log n)$ time. Isn't this strange? — computing all the repeats only takes $O(n)$ time.

What is a Run?

Rescue! — the idea of a run or maximal periodicity: a periodicity that cannot be extended, either left or right: it is NE!

$$\boldsymbol{x} = \cdots d\underline{abcabcabca}d\cdots$$

The underlined segment is a run that represents the three repetitions $(abc)^3$, $(bca)^3$, $(cab)^2$.

Using the LZ decomposition, it was shown 10 years ago that the runs in any string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ can be computed in $O(n)$ time, and thus essentially the repetitions.

What is $O(n)$?

$O(n)$ means "proportional to n", so the number of runs in $\boldsymbol{x}$ is at most kn, where k is a constant. What if $k = 10^{10^{10}}$?!? How do we know it isn't? How big can k be? Recall the example

$$0010100101101001010010110100101 00 \cdots$$

A very exciting research question — to a mathematician!

But also practical — if the maximum number of runs is close to the length of the string, then maybe simpler (and faster) ways can be found to compute repetitions.

Currently we know (Jamie Simpson, Curtin University) that

$$0.944575712 < k < 1.048.$$

Five years of work by a few dozen mathematicians ...

Computing Repetitions

The fastest current method for computing all the runs is linear, but still it does a lot of work: it has to compute the suffix array (see intrinsic patterns), the longest common prefix array (another intrinsic pattern), then the LZ decomposition — and still it isn't done!

Knowing that the number of runs is not too large may well lead to a faster all-runs algorithm.

Lyndon Words

Suppose the letters in the alphabet are ordered: $a, b, c, \ldots$ or $1, 2, 3, \ldots$. This induces lexicographical order (lexorder or dictionary order) on strings: substitute $<$ substitution, $a^n b < a^{n-1} b$.

A Lyndon word $L(\boldsymbol{x})$ is the lex least rotation of $\boldsymbol{x}$:

$$\boldsymbol{x} = aba, \text{ rotations } aba, \; baa, \; aab.$$

So $L(aba) = aab$.

Lyndon Decomposition Theorem

Theorem (Chen, Fox, Lyndon: 1958) Every string $\boldsymbol{x}$ can be expressed as a unique decomposition

$$\boldsymbol{x} = \boldsymbol{w}_1 \boldsymbol{w}_2 \cdots \boldsymbol{w}_k$$

of Lyndon words $\boldsymbol{w}_i$, $1 \leq i \leq k$, where $\boldsymbol{w}_1 \geq \boldsymbol{w}_2 \geq \cdots \geq \boldsymbol{w}_k$.

For example,

$$\begin{aligned} \boldsymbol{x} &= d/d/c/c/ab/a \\ \boldsymbol{x} &= aad/aac/aab \\ \boldsymbol{f} &= ab/aabab/aab/aab/a \end{aligned}$$

Lyndon Decomposition Algorithm

Duval (1983): Compute $L(\boldsymbol{x})$

$h \leftarrow 0$
while $h < n$ **do**
    $i \leftarrow h{+}1$
    $j \leftarrow h{+}2$
    while $\boldsymbol{x}[j] \geq \boldsymbol{x}[i]$ **do**
        if $\boldsymbol{x}[j] > \boldsymbol{x}[i]$ **then** $i \leftarrow h{+}1$ **else** $i \leftarrow i{+}1$
        $j \leftarrow j{+}1$
    repeat
        $h \leftarrow h + (j{-}i)$; **output** h
    until $h \geq i$

Stringology's most elegant algorithm!