Computing Patterns in Very Long Strings

Bill Smyth

Algorithms Research Group, Department of Computing & Software
and
School of Computational Science & Engineering
McMaster University, Hamilton, Canada

School of Mathematics & Statistics,
University of Western Australia, Perth, Australia

email: smyth@mcmaster.ca

McMaster University, 13 November 2013
Abstract

Combinatorics on words began more than 100 years ago with a demonstration that an infinitely-long string with NO repetitions could be constructed on an alphabet of only three letters [T06]. Computing ALL the repetitions (such as \( \cdots TTT \cdots \) or \( \cdots CGACGA \cdots \)) in a string is one of the oldest and most important problems of computational stringology. About a dozen years ago it was discovered [KK00] that repetitions can be computed as a byproduct of the \( \Theta(n) \)-time computation of all the maximal periodicities or runs in a string \( x \) of length \( n \). However, even though the computation is linear [CPS07], it is also brute force: global data structures such as the suffix array [PST07], the longest common prefix array [KMP09], and the Lempel-Ziv factorization [ACIKSTY13] need to be computed in a preprocessing phase. Furthermore, all of this effort is required despite the fact that the expected number of runs in a string is generally a small fraction of string length [PS08]. In this talk I explore the possibility that repetitions (perhaps also other regularities in strings) can be computed in a manner commensurate with the size of the output.
Outline

1. Preliminaries, Terminology & Data Structures
2. Computing Repetitions & Runs
3. Combinatorial Insight vs. Brute Force
4. Three Squares
Preliminaries

- A string is a finite sequence of symbols (letters) drawn from some finite or infinite set \( \Sigma \) called the alphabet. The alphabet size is \( \sigma = |\Sigma| \). Usually the alphabet is ordered, thus inducing lexorder (dictionary order) on the strings.

- We write a string \( x \) in mathbold, and we represent it as an array \( x[1..n] \) for some \( n \geq 0 \). We call \( n = x \) the length of \( x \). For example,

\[
x = a \ b \ a \ a \ b \ a \ b \ a \ a \ b
\]

is a string of length \( x = 10 \) on \( \Sigma = \{a, b\} \). For \( x = 0 \), \( x = \varepsilon \), the empty string.

- If \( x = uvw \), then \( u \) is said to be a prefix, \( v \) a substring and \( w \) a suffix of \( x \); if \( vw \neq \varepsilon \), \( uw \neq \varepsilon \), \( uv \neq \varepsilon \), respectively, then \( u \), \( v \), \( w \) is, respectively, a proper prefix, proper substring, proper suffix of \( x \).

- If \( x = uv = wu \) for \( u < x \), then \( u \) is a border of \( x \), and \( x \) has period \( p = x - u \); that is, for every \( i \in 1..u \), \( x[i] = x[i+p] \). The string (1) has borders \( abaab \) and \( ab \), hence corresponding periods 5 and 8, respectively.
Terminology

- If $x = vu^ew$, with integer $e > 1$ and $u$ neither a suffix of $v$ nor a prefix of $w$ ($e$ is maximum), then $u^e$ is said to be a repetition in $x$. The integers $u$ and $e$ are the period and exponent, respectively, of the repetition.

- For example, in $x = abaababaab$, there are repetitions $a^2$ (twice), $(ab)^2$ and $(ba)^2$, $(aba)^2$, and $(abaab)^2$. Each of these repetitions is a square ($e = 2$). In general, every repetition has a square prefix.

- If $v = x[i..j]$ has period $u$, where $v/u \geq 2$, and if neither $x[i-1..j]$ nor $x[i..j+1]$ (whenever these are defined) has period $u$, then $v$ is said to be a maximal periodicity or run in $x$ [M89] and $v$ is said to have exponent $e = \lfloor v/u \rfloor$ and tail $t = v \mod u$. When $t = 0$, the run is also a repetition.

- All of the repetitions in (2) are runs except for $(ab)^2$ and $(ba)^2$: these are prefix and suffix, respectively, of the run $v = ababa$.

- In general, every repetition is a substring of some run; thus computing all the runs implicitly computes all the repetitions.
Global Data Structures: ST, SA, LCP

|x| = |a| b| a| a| b| a| b| a|

SA\(x\) = 8 3 6 1 4 7 2 5
LCP\(x\) = 0 1 1 3 3 0 2 2

Figure: Suffix tree, suffix array, LCP array
ST Checklist

- leaf nodes in lexorder
- internal nodes give lcp
- $O(x \log \sigma)$ construction time, where $\sigma$ is the alphabet size
- $O(x)$ space
- for construction and search: data structure at each node
- pattern-matching in time proportional to pattern-length $m$
- useful for repeats, repetitions, LZ

Marvellous!

- but too much space; replaced by SA/LCP since 2004 [AKO04]
Global Data Structures: LZ

A factorization \( x = w_1 w_2 \cdots w_k \) is **LZ** (for Lempel-Ziv [LZ76, ZL77]) if and only if each \( w_j, j \in 1..k \), is

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

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

We observe that \( w_1 = x[1] \), further that a factor \( w_j \) may overlap with its previous occurrence in \( x \). For the string

\[
x = abaababa
\]

the factorization \( LZ_x \) is given by \( w_1 = a, w_2 = b, w_3 = a, w_4 = aba, w_5 = ba \).

All of these data structures can be computed in time linear in \( x \):

- ST [W73, M76, U95, F97],
- SA [MM90, MM93, PST07, NZC09, M09],
- LCP [KLAAP01, M04, PT08, KMP09].

Then LZ can be computed from some combination of them:
Computing LZ

Figure: From [ACIKSTY13]
Computing Repetitions

In the early 1980s three \(O(x \log x)\)-time algorithms were proposed to compute all the repetitions in a given string \(x\):

- Crochemore [C81] describes a method of successive refinement that identifies all equal substrings of lengths 1, 2, \ldots until for some length \(\ell\) every substring is unique. As remarked in [S03], his method is essentially an algorithm for suffix tree construction. Crochemore also showed that a string \(x\) can contain as many as \(O(x \log x)\) repetitions — thus all these algorithms are optimal.

- Apostolico & Preparata [AP83] use suffix trees plus auxiliary data structures.

- Main & Lorentz [ML84] use a divide-and-conquer approach based on prior computation of \(LZ_x\).

*** THUS ALL USE GLOBAL DATA STRUCTURES ***
Computing Runs

- In 1989 Main [M89] showed how to compute all “leftmost” runs, again from LZ\(_X\), in linear time.
- In 1999 Kolpakov & Kucherov [KK99, KK00] showed how to compute all runs from the leftmost ones, also in linear time.
- To establish linearity, they proved that the maximum number \(\rho(n)\) of runs over all strings of length \(n\) satisfies

\[
\rho(n) \leq k_1 n - k_2 \sqrt{n \log_2 n} \tag{3}
\]

for some universal positive constants \(k_1\) and \(k_2\).
- They provided computational evidence (up to \(n = 60\)) that \(\rho(n) \leq n\) — this was their conjecture.
- Based on work by many authors over the last 10 years, it has been shown that \(0.944575 < \rho(n)/n \leq 1.029\) — primarily a computational rather than a combinatorial result.
The K&K method still requires **BRUTE FORCE**, the computation of global data structures:

- SA/ST
- LCP
- LZ

while the expected number of runs in a string of length $n$ is small (Puglisi & Simpson [PS08]):

- $0.41n$ runs for alphabet size $\sigma = 2$;
- $0.25n$ runs for DNA ($\Sigma = \{A, C, G, T\}$);
- $0.04n$ for protein ($\sigma = 20$);
- $0.01n$ for English-language text.

Runs (hence repetitions) in most strings are **SPARSE**!
Combinatorial Insight vs. Brute Force

- Using full data structures, linear-time processing (with a high constant of proportionality) is achieved using 12 bytes of storage per input symbol: 36 gigabytes for the human genome (3GB long) [ACIKSTY13].
- Using compressed data structures, space can be reduced to about 5 bytes per input symbol (15GB for the human genome), but linearity is lost (order of magnitude slower) [ACIKSTY13].
- Maybe this is acceptable using current main memory capacity, but what about processing plant DNA (15-20GB) or 50GB/500GB strings? Only possible using secondary storage and slowing the computation by several orders of magnitude.
- All this to compute something that is generally sparse and occurs locally in the string: we want to use at most one byte per input symbol (only two bits for DNA!) and process an order of magnitude faster.

So we need to understand why the maximal periodicities are restricted — why there are restrictions on their overlaps — so that we can process a string from left to right in a controlled way, outputting the runs as we go: we need combinatorial insight!
A Tiny Idea

If $\rho(n)/n$ is limited to be near one, it means that on average there is about one run starting at each position. So ... if TWO runs start at some position, then there must be some other position, probably nearby, at which NO runs start.

Runs always start with squares — what do we know about squares that begin at about the same position? What COMBINATORIAL INSIGHT do we have into the restrictions that might be imposed upon occurrences of overlapping squares? Until recently, very little:
What We Knew (After 90 Years!)

Lemma (Crochemore & Rytter [CR95])

Suppose \( u \) is not a repetition, and suppose \( v \neq u^j \) for any \( j \geq 1 \). If \( u^2 \) is a prefix of \( v^2 \), in turn a proper prefix of \( w^2 \), then \( w \geq u + v \).

The Fibonacci string demonstrates that this result is best possible (squares ending at positions 6, 10, 16 = 6 + 10, 26 = 10 + 16):

\[
\begin{array}{cccccccccccccccccccc}
x = a & b & a & a & b & a & b & a & a & b & a & a & b & a & a & b & a & a & b & a & a & b & a & a & b & a & a & b
\end{array}
\]

The Three Squares Lemma is a result of great insight: it tells us that if three squares occur at the same position, then one of them has to be “large”. But we want to know much more: what if the three squares just overlap, just occur in the same neighbourhood? What then???
New Ideas (since 2005)

We paraphrase the accumulated results of [FSS05, PST05, S05, FPST06, S07, KS12, FFSS12]:

Suppose $u^2$ and $v^2$, $u < v < 2u$, occur at the same position $i$ in $x$, with a third square $w^2$ occurring at position $i + k$, where $v - u < w < v$, $w \neq u$, $0 \leq k < v - u$. Then $x[i..i+2v-1]$ is a repetition of small period.

(In other words, two coincident squares, followed by a third one — are impossible!)
The General Case

But of course there are many other cases:

- What if $u^2$ and $v^2$ are not coincident?
- What if $w^2$ occurs somewhere to the left of $u^2$ — or somewhere in between $u^2$ and $v^2$?

We need a general case that puts together

$$
\begin{array}{ccc}
  & u & u \\
  k_1 & v & v \\
\end{array}
$$

and

$$
\begin{array}{ccc}
  v & v \\
  k_2 & w & w \\
\end{array}
$$

This would cover all possible cases!
Putting It All Together

The good news: a characterization of the General Case has been proposed in a very recent paper [BS13]: it treats the overlap of two squares in terms of five distinct cases that depend on the relative sizes of \((k_1, u, v)\) — or \((k_2, v, w)\).

The bad news: there are \(25 = 5 \times 5\) cases to analyze in order to determine with precision what is possible, what not. A lot of work!

Then:

- with new combinatorial knowledge, perhaps we can classify the possible periodic structures at each position in a string,
- so that a computer program can do a left-to-right scan to compute all the repetitions using an order of magnitude less time and space than present algorithms;
- and thus deal with terabytes tomorrow the way we process gigabytes today,
- based on an advance in software rather than hardware!
A faint hope?

A mirage??

A pipe-dream???

Pie in the sky????

Maybe ...

Maybe not ...

... but at least we will understand strings better!


Yuta Mori, libdivsufsort: http://code.google.com/p/libdivsufsort/


