Combinatorics on words began more than a century ago with a demonstration that an infinitely long string with no repetitions could be constructed on an alphabet of only three letters. Computing all the repetitions (such as \( \cdots TTT \cdots \) or \( \cdots CGACGA \cdots \)) in a given string \( x \) of length \( n \) is one of the oldest and most important problems of computational stringology, requiring \( \mathcal{O}(n \log n) \) time in the worst case. About a dozen years ago, it was discovered that repetitions can be computed as a by-product of the \( \Theta(n) \)-time computation of all the maximal periodicities or runs in \( x \). However, even though the computation is linear, it is also brute force: global data structures, such as the suffix array, the longest common prefix array and the Lempel–Ziv factorization, 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 the string length. In this paper, 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.

1. Introduction

Repeating substrings (‘repetitions’) are fundamental to the twin disciplines of combinatorics on words and string algorithms. The very first paper in this area, by Axel Thue in 1906 [1], showed how to construct an infinitely long string entirely free of repetitions on an alphabet \( \{a, b, c\} \) of only three letters. With the advent of the electronic computer a half-century later, algorithms to compute all the repetitions in a given string were among the first to be studied—by coincidence it was at about this same time that Watson & Crick [2] first described the structure of DNA (a ‘string’ billions of
letters long on a four-letter alphabet \{A, C, G, T\} in which repeating substrings were of great genetic significance. Since that time, with the huge expansion of textual data—in computer files, on the Internet, in massive data capture from space—computing repetitions has found many other applications in areas such as text compression, cryptography and data mining.

In this paper, I seek to provide, for the non-expert, a survey of the past and present of algorithms that compute repetitions; an understanding of their limitations, especially when applied to massive data (terabytes and beyond) and a glimpse into the possibilities for their dramatic future improvement. I give many references to relevant work; it is my hope that some readers may be motivated to do research in this intriguing, subtle and challenging field.

Section 2 defines the terminology and notation necessary for an understanding of stringological discussions and briefly introduces some of the basic data structures used in algorithms on strings. Section 3 characterizes the existing algorithms that compute repetitions, while §4 reflects on the nature of these algorithms and puts forward the view that though they are efficient enough on a few gigabytes of data, much more combinatorial insight is required to enable them to function efficiently on the massive datasets of the future. Section 5 describes new repetition algorithms that are more efficient, both in their use of time and space, by an order of magnitude over those currently available. Finally, §6 proposes future research directions.

2. Preliminaries

(Usage generally follows [3,4].) A string or word is a finite sequence of symbols (letters) drawn from some finite set \(\Sigma\) called the alphabet. The alphabet size is \(\sigma = |\Sigma|\). Usually the alphabet is ordered, thus inducing lexicorder (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,

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]

\(x = a\ b\ a\ a\ b\ a\ a\ b\)

(2.1)

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

If \(x = uv\vDash w\), then \(u\) is said to be a prefix, \(v\) a substring (or factor) and \(w\) a suffix of \(x\); proper in each case if \(uv \neq \varepsilon\), \(uv \neq \varepsilon\), \(uv \neq \varepsilon\), respectively. If \(x = uv = \vDash w\) 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 (2.1) has borders \(aba\) and \(ab\), hence corresponding periods 5 and 8, respectively. Note that as every string \(x\) has an empty border, it therefore also has a period \(x\).

If \(x = uv^e w\), with integer \(e > 1\) and \(u\) neither a suffix of \(v\) nor a prefix of \(w\) (\(e\) is maximum), then \(uv^e\) 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

\[
x = \underline{abaaba}babaab,
\]

(2.2)

there are repetitions \(a^2\) (twice), \((ab)^2\) and \((ba)^2\), \((aba)^2\) and \((abab)^2\). Each of these repetitions is a square \((e = 2)\). In general, every repetition has a square prefix. An interesting string is the Fibonacci string \(f_k\)

\[
f_0 = b, \quad f_1 = \underline{a}; \quad k \geq 2 \Rightarrow f_k = f_{k-1}f_{k-2}.
\]

It contains \(O(f_i \log f_k)\) repetitions [5–7].

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 \(x\) is said to be a maximal periodicity or run in \(x\) [8] and \(v\) is said to have exponent \(e = |v/u|\) and tail \(t = v \bmod u\). When \(t = 0\), the run is also a repetition. All of the repetitions in (2.2) 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.
There are several important and well-known data structures that are useful for the computation of runs or repetitions in a string $x$ (figure 1):

— the suffix tree $ST_x$: a compacted trie [9,10] on the suffixes of $x$, where each leaf node is labelled with the starting position of a suffix and the ordering at internal nodes is lexicographic;
— the suffix array $SA_x$, in which for every $j \in 1..x$, $SA[j] = i$, where $i$ is the starting position of the $j$th suffix in lexorder; and
— the longest common prefix array $LCP_x$, where $LCP[1] = 0$ and for every $j \in 2..x$, $LCP[j] = \text{lcp}\{x[SA[j-1]..x], x[SA[j]..x]\}$, with $\text{lcp}\{u,v\}$ the length of the longest common prefix of $u$ and $v$.

The suffix tree was introduced by Weiner [11], who described an $O(x \log \sigma)$-time algorithm for its computation, improved somewhat in [12] and further by an online algorithm in [13]. A $\Theta(x)$-time algorithm, independent of $\sigma$ (though restricted to an integer or ‘indexed’ alphabet [3]), was proposed in [14]; although not practical for large strings, this approach provided the model for practical linear-time SA algorithms (see below). As early as 1985, the suffix tree was praised for its ‘myriad virtues’ [15]: fast construction time, linear space requirement, support for computation of repetitions and for pattern matching in time proportional only to pattern length. Its Achilles’ heel was the space required, especially for larger $\sigma$: even though linear, the constant of proportionality became large due primarily to the need to store a search structure at each node. For smaller $\sigma$, with careful implementation, the use of the ST could still sometimes be competitive with more recently discovered data structures, such as the SA [16].

In 1990, Manber & Myers [17,18] introduced the suffix array and proposed $O(x \log x)$-time algorithms for its construction. As documented in [19], over the next 15 years numerous SACAs (suffix array construction algorithms) were proposed, including three $\Theta(x)$-time contributions [20–22] in 2003. There are currently three fast algorithms available that achieve either linear time or lightweight memory use (only constant additional storage) or both:

— the algorithm of [23] achieves $\Theta(n)$ execution time with $0.25n$ bytes of additional space;
— the implementation of [24] executes in $O(n \log n)$ time but uses only $O(1)$ additional memory; and
— the recent algorithm of [25] executes in linear time with constant additional space on a fixed alphabet.

In view of methods discovered for effective use of the suffix array [26], it has become clear that for almost all applications SA is superior to ST. Often used in conjunction with SA is LCP, shown to be
computable in linear time in 2001 [27], then later also with reduced space requirements [28–30]. Currently, the preferred approach to LCP construction is Fischer’s algorithm [31] introduced as a modification of the SACA of Nong et al. [23].

It should be emphasized that these data structures are global; that is, they express relationships among suffixes that are widely separated within the given string $x$; therefore, changes to $x$ affect them in ways difficult to predict.

I conclude this section with the introduction of the fourth global data structure that, we shall see, is central to the efficient calculation of runs:

A factorization $x = w_1w_2 \cdots w_k$ is LZ (for Lempel–Ziv [32,33]) if and only if each $w_j$, $j \in 1..k$, is

(a) a letter that does not occur in $w_1w_2 \cdots w_{j-1}$; or otherwise
(b) the longest substring that occurs at least twice in $w_1w_2 \cdots w_j$.

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 $w_1 = a$, $w_2 = b$, $w_3 = a$, $w_4 = aba$ and $w_5 = ba$.

Historically, the LZ factorization was widely used in text compression [34], but it has turned out also to be central to computing runs in strings, as made clear in a recent survey [35], where as illustrated in figure 2 the various ways of computing LZ are compared.1 More recently, LZ factorization has been studied by various authors [36–40], with consequent improvement in efficiency and practicality. For our purposes, there are two pertinent facts:

— LZ, itself a global structure, can be efficiently computed (in time linear in the string length) only based on other global data structures and
— every known algorithm for computing runs whose time requirement is linear in the string length depends on prior computation of LZ.

---

1The left-hand side calculation shown in the figure is used only for compression. ESA and QSA are variants of the suffix array, LPF and BWT are global data structures—all are computable in linear time.
3. Computing repetitions and runs

In the early 1980s, three $O(x \log x)$-time algorithms were proposed to compute all the repetitions in a given string $x$, all of them based on the use of global data structures:

— Crochemore [5] describes a method of successive refinement that identifies all equal substrings of lengths 1, 2, ... until for some length $\ell$ every substring is unique. As remarked in [3], his method is essentially an algorithm for suffix tree construction. As noted earlier, Crochemore also showed that the Fibonacci string $f_k$ contains $O(f_k \log f_k)$ repetitions—thus all these algorithms are optimal.

— Apostolico & Preparata [41] use suffix trees plus auxiliary data structures.

— Main & Lorentz [42] use a divide-and-conquer approach based on prior computation of $LZ_x$.

The computation of runs has a more recent history:

— In 1989, Main [8] showed how to compute all ‘leftmost’ runs, again from $LZ_x$, in linear time.

— In 1999, Kolpakov & Kucherov [43,44] showed how to compute all runs in $x$ 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

$$\rho(n) \leq k_1 n - k_2 \sqrt{n \log_2 n}$$

for some universal positive constants $k_1$ and $k_2$.

— They provided computational evidence (up to $n = 60$) that in fact $\rho(n) \leq n$—this was their conjecture.

— Based on work by many authors over the last 10 years, it has been shown that, for sufficiently large $n$, $0.944575 < \rho(n)/n < 1.029$—where the upper bound is primarily a computational result. The current lower bound is owing to [45], the upper to [46].

Thus, the computation of runs, essentially a local phenomenon in a string, depends, as we have seen, on heavy preprocessing of global data structures—linear-time, but certainly with a large constant of proportionality. Moreover, Puglisi & Simpson [47] have shown that in some sense runs occur sparsely in strings; the expected number of runs in a string of length $n$ is

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

There should be a better way: if runs are sparse and occur locally, why must heavy global methods be used to compute them?

4. Combinatorial insight and brute force

So far, the view has been put forward that current methods for computing runs are not so far from brute force, even though the time requirement is linear. More precisely, it is shown in [35] that, using full data structures, 12 bytes per input letter (thus 36 gigabytes for the human genome) are required; using compressed data structures, this reduces to about 5 bytes per input letter (15 GB for the human genome), but linearity is lost: processing is an order of magnitude slower. Perhaps 36 GB is acceptable usage of memory, but what about plant genomes that are several times the size of the human genome, or 50 GB/500 GB strings? In such cases, supercomputers
aside, secondary storage would surely be required, thus slowing the computation time by several orders of magnitude. On the other hand, if the input string $x$ could be processed sequentially, using only $O(1)$ additional storage, and performing only calculations local to the current position in the string, then at most 1 byte per input symbol (just two bits for DNA) and a fraction of the global time requirement would be required to compute all the runs in $x$.

Even though we know that there are at most $1.029^n$ runs in a string, we know little of their possible arrangements—we do not have sufficient combinatorial insight to be able to make intelligent decisions based on analysis of the local situation in the string. Until very recently, the only theoretical result that shed light on this matter was the ‘Three Squares Lemma’, published almost 90 years after Thue’s first treatment of squares:

**Lemma 4.1 (Crochemore & Rytter [48]).** Suppose $u$ is not a repetition, and suppose $v \neq u^e$ for any $e \geq 1$. If $u^2$ is a prefix of $v^2$, which is in turn a proper prefix of $w^2$, then $w \geq u + v$.

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

$$f = abaaba babaabaabaab abababaabaab$$

This lemma is a step in the right direction, but in order to be able to analyse runs at each position $i$, we need to know much more: in particular, what happens when squares (runs) begin in some neighbourhood of each other—when they overlap. In §5, we describe recent theoretical results that begin to make sense of this kind of situation. As we know that the maximum number of runs is about equal to the string length, it follows that if two squares (runs) begin at the same position, then there must as a result be a position somewhere at which no square (run) begins—there must be combinatorial restrictions on the ways in which squares in a string can overlap. Referring back to the above Fibonacci example, observe that $f$ breaks down into overlapping substrings of (small) period 3

$$f = (aba)^2 \land (aba)^3 \land (aba)^2 \land (aba)^2ab.$$

We shall find that this behaviour is characteristic also in more general cases, thus perhaps algorithmically useful as a signal of overlapping runs.

## 5. Overlapping squares

This section provides a bird’s-eye view of research into overlapping squares that dates back to 2005 [4,49–55]; especially, the more recent papers will be helpful to anyone intending to pursue further research in this area.

The bulk of the research considers two squares $u^2$ and $v^2$, $u < v < 2u$, so that $u$, but not $u^2$, is a prefix of $v$. There are two cases, whose analysis is quite different, but whose results are qualitatively the same, a breakdown of the string into runs of small period

(C1) $v \leq 3u/2$ and

(C2) $v > 3u/2$.

The details are complicated, but the main results are as follows:

**Theorem 5.1 (C1).** If $x = v^2$ with prefix $u^2$, $u < v \leq 3u/2$, then

$$x = u_1^m u_2 u_1^{m+1} u_2 u_1,$$

where $u_1 = v - u \leq u/2$, $u_2 = u \mod u_1 \geq 0$, $m = \lfloor u/u_1 \rfloor \geq 2$ and $u_2$ is a proper prefix of $u_1$. Moreover, $x$ contains no runs of period $\geq u_1$ other than specific identifiable ones described in [4].
Theorem 5.2 (C2). Suppose \( u^2 \) and \( v^2 \), \( 3u/2 < v < 2u \), occur at the same position \( i \) in \( x \). Then \( v = u_1 u_2 u_1 u_1 u_2 \), where \( u_1 = 2u - v \) and \( u_2 = 2v - 3v \). If moreover the third square \( w^2 \) occurs at position \( i + k \), where \( v - u < w < v, w \neq u, 0 \leq k < v - u \), then \( x[i..i + 2v - 1] \) breaks down into runs of small period according to 14 well-defined subcases \([4,54]\).

I confess that it is an exaggeration to call the latter result a ‘theorem’—two of the 14 subcases have been only partly proved \([52,54]\). Nevertheless, there is convincing evidence from extensive computer simulations \([4]\) that the incomplete cases do satisfy the stated constraint. The subcases are defined by the sizes of \( k \) and \( w \) relative to \( u \) and \( v \): figure 3 shows Subcase 5, for example, where it is known \([4]\) that \( v = d v/d \), with \( d \) a prefix of \( v \) of length \( d = \gcd(u, v, w) \).

Clearly, quite apart from the two missing subcases, there is much more work to be done:

— What can be said when \( w > v \) (as in the case of the Three Squares Lemma), but with \( k > 0 \)?
— What if \( u^2 \) and \( v^2 \) are not coincident?
— What if \( w^2 \) occurs to the left of \( u_2 \)—or somewhere in between \( u_2 \) and \( v_2 \)? and
— Above all, we need a general case that puts together

\[
\begin{array}{cccc}
& u & u \\
k_1 & & \\
\end{array}
\quad
\begin{array}{cccc}
& v & v \\
& & \\
\end{array}
\quad
\begin{array}{cccc}
& v & v \\
& w & w \\
& k_2 & \quad
\end{array}
\]

In fact, a start has recently been made in this direction \([55]\), but an analysis of the combinatorial possibilities requires consideration of many more subcases.

### 6. Future directions

In this paper, I have argued that it ‘should’ be possible to compute runs (and, by the way, perhaps other ‘regularities’ as well \([56]\)) in a manner consistent with their occurrences in strings: both sparse and local. We have looked at the methods currently available, all heavily dependent on global data structures; it is clear that if local methods could be used algorithmically, in a left-to-right scan of the string, there would be order-of-magnitude savings in both time and space requirements, thus extending the applicability of the computation of runs to much longer strings. It should be emphasized that what is proposed is an improvement in software, without any hardware upgrade at all.

Of course what ‘should’ be true does not always turn out to be actuality. But we have presented evidence that greater combinatorial insight, though complicated and onerous, may indeed show how to apply methods that are ultimately simpler than the brute force of global data structures. Indeed, even without the potential applications, the contribution to combinatorics on words, the enhanced understanding of periodicity in strings, would be important and significant.

**Acknowledgements.** The material presented here is based on an invited talk given to the Theo Murphy International Scientific Meeting on ‘Storage & Indexing of Massive Data’, The Royal Society at Chicheley Hall, 7–8 February 2013.
Funding statement. This work was supported in part by the Natural Sciences & Engineering Research Council of Canada.

References

40. Main MG, Lorentz RJ. 1984 An $O(n \log n)$ algorithm for finding all repetitions in a string. J. Algorithms 5, 422–432. (doi:10.1016/0196-6774(84)90021-X)


