EFFICIENT COMPUTATION OF REGULARITIES IN STRINGS AND APPLICATIONS

EFFICIENT COMPUTATION OF REGULARITIES IN STRINGS AND APPLICATIONS

BY

MUNINA YUSUFU, M.Eng. B.Sc.

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
MCMASTER UNIVERSITY
HAMILTON, ONTARIO, CANADA

MCMASTER UNIVERSITY

Title: Efficient Computation of Regularities in Strings and Applications

Author: Munina Yusufu

Department: Computing and Software

Supervisor: Dr. William F. Smyth

Degree: Doctor of Philosophy

Number of Pages: xiii, 122

Abstract

Regularities in strings model many phenomena and thus form the subject of extensive mathematical studies. Perhaps the most conspicuous regularities in strings are those that manifest themselves in the form of repeated subpatterns, that is, repeats, multi-repeats, repetitions, runs and others. Regularities in the form of repeating substrings were the basis of one of the earliest and still widely used compression algorithms and remain central in more recent approaches. Repeats and repetitions of lengthy substrings in DNA and protein sequences are important markers in biological research.

A large proportion of the available algorithms for computing regularities in strings depends on the prior computation of a suffix tree, or, more recently, of a suffix array. The design of new algorithms for computing regularities should emphasize conceptual simplicity, as well as both time and space efficiency.

In this thesis, we investigate mathematical and algorithmical aspects of the computation of regularities in strings.

The first part of the thesis is the development of space and time efficient nonextendible (NE) and supernonextendible (SNE) repeats algorithms RPT, shown to be more efficient than previous methods based on tests using different real data sets. In particular, we describe four variants of a new fast algorithm RPT1 that, based on suffix array construction, computes all the complete NE repeats in a given string $\boldsymbol{x}$ whose length (period) $p \geq p_{min}$, where $p_{min} \geq 1$ is a user-specified minimum. RPT1 uses $5n$ bytes of space directly, but requires the LCP array, whose construction needs $6n$ bytes. The variants RPT1–3 and RPT1–4 execute in $O(n)$ time independent of alphabet size and are faster than the two other algorithms previously proposed for

this problem. To provide a basis of comparison for RPT1, we also describe a straightforward algorithm RPT2 that computes complete NE repeats without any recourse to suffix arrays and whose total space requirement is only $5n$ bytes; however, this algorithm is slower than RPT1. Furthermore, we describe new fast algorithms RPT3 for computing all complete SNE repeats in $\boldsymbol{x}$. Of these, RPT3-2 executes in $\Theta(n)$ time independent of alphabet size, thus asymptotically faster than the methods previously proposed. We conclude with a brief discussion of applications to bioinformatics and data compression.

The second part of the thesis deals with the issue of finding the NE multirepeats in a set of N strings of average length n under various constraints. A multirepeat is a repeat that occurs at least m times ($m \geq 2$) in each of at least $q \geq 1$ strings in a given set of strings. We show that RPT1 can be extended to locate the multirepeats based on the investigation of the properties of the multirepeats and various strategies. We describe algorithms to find complete NE multirepeats, first with no restriction on "gap length" (that is, the gap between occurrences of the multirepeat), then with bounded gaps. For the first problem, we propose two algorithms with worst-case time complexities $O(Nn+\alpha \log_2 N)$ and $O(Nn+\alpha)$ that use $9Nn$ and $10Nn$ bytes of space, respectively, where α is the alphabet size. For the second problem, we describe an algorithm with worst-case time complexity $O(RNn)$ that requires approximately $10Nn$ bytes, where R is the number of multirepeats output. We remark that if we set the *min* and *max* constraints on gaps equal to zero in this algorithm, we can find all repetitions (tandem repeats) in arbitrary subsets of a given set. We demonstrate that our algorithms are faster, more flexible and much more space efficient than algorithms recently proposed for this problem.

Finally, the third part of the thesis provides a convenient framework for comparing the LZ factorization algorithms which are used in the computation of regularities in strings rather than in the traditional application to text compression. LZ factorization is the computational bottleneck in numerous string processing algorithms, especially in regularity studies, such as computing repetitions, runs, repeats with fixed gap,

branching repeats, sequence alignment, local periods, and data compression. Since 1977, when Ziv and Lempel described a kind of string factorization useful for data compression, there has been a succession of algorithms proposed for computing "LZ factorization". In particular, there have been several recent algorithms proposed that extend the usefulness of LZ factorization, especially to the computation of runs in a string x. We choose these algorithms and analyze each algorithm separately, and remark on them by comparing some of their important aspects, for example, additional space required and handling mechanism. We also address their output format differences and some special features. We then provide a complete theoretical comparison of their time and space efficiency. We conduct intensive testing on both time and space performance and analyze the results carefully to draw conclusions in which situations these algorithms perform best. We believe that our investigation and analysis will be very useful for researchers in their choice of the proper LZ factorization algorithms to deal with the problems related to computation of the regularities in strings.

Acknowledgements

I am most thankful to my supervisor Dr. William F. Smyth. Your great expertise and understanding of algorithms, devotion to research, constant encouragement and enormous support of my research and the thesis writeup has been the key for all the publications and the outcome of my current thesis. You are such an inspiring and patient advisor; working with you is always a pleasure.

I would like to thank the members of my supervisory committee Dr. Michael Soltys and Dr. Ned Nedialkov, for your help and advice over these years. Thank you also Dr. Ned Nedialkov, for having agreed to participate in my defence via teleconference. I am grateful to my external examiner Dr. Andrew Turpin, for reviewing my thesis thoroughly and providing many valuable remarks. I would like to thank Dr. Jeffery Zucker, for having agreed to review this work and attended my defence.

I have been privileged to work with other excellent researchers with whom I have shared many interesting moments in science. In particular, I would like to mention Dr. Simon J. Puglisi. I cannot recall the numbers of E-mails we wrote to each other, but I remember all those useful suggestions you sent to me. Thank you Dr. Costas S. Iliopoulos, for believing in me on the ideas of computing multirepeats and for working with us. Thank you all, my colleagues and friends, for your support and friendship.

I am deeply grateful to my parents Zibaida and Yusufu, my brother Alibiyati and sister Gulina. Thank you for your endless support, encouragement and love. Thank you, my daughter Michelle, for your unconditional love and generous spirit as I devoted much of my time to this work over these years. I am extremely lucky to have you all in my life.

Declaration

I declare that my thesis contains the following published or to be published materials of which I was one of the co-authors and my contributions have also been stated. These work are all done during my Ph.D. research and closely related to the context of computation of regularities in strings; therefore, they are all integral components of this thesis. The copyright holder has agreed to grant an irrevocable, non-exclusive licence to McMaster University and the National Library of Canada to reproduce the material as part of the thesis.

Simon J. Puglisi, William F. Smyth, and Munina Yusufu, Fast optimal algorithms for computing all the repeats in a string, *Prague Stringology Conference* (preliminary version), Jan Holub and Jan Zdarek (eds.) (2008) 161–169.

Simon J. Puglisi, William F. Smyth, and Munina Yusufu, Fast optimal algorithms for computing all the repeats in a string, submitted for publication (2009).

Contribution by me: I proposed the ideas of the new algorithms PSY1–1, PSY1–2, and PSY1–3 for computing NE repeats and PSY3–1 for computing SNE repeats and I was heavily involved in designing all the algorithms. I was also responsible for the implementing and testing of these new algorithms against existing competitors. I wrote the draft of the paper for the conference and rewrote it for the journal paper.

William F. Smyth and Munina Yusufu, Computing regularities in strings, *Proc. Second IEEE International Conference on Computer Science and Information Technology* (2009) 298–302.

Contribution by me: I proposed the ideas and wrote the draft of the paper.

Costas S. Iliopoulos, William F. Smyth, and Munina Yusufu, Faster algorithms for computing maximal multirepeats in multiple sequences, *Fundamenta Informaticae*, Special StringMasters Issue (2009) to appear.

Contribution by me: I proposed to extend our algorithms PSY1 in the first two papers to locate the multirepeats in a set of strings under various constraints and I was involved in designing efficient algorithms for two different problems. I also proposed many of the new data structures which we used in the algorithms. I wrote the draft of the paper.

Anisa Al-Hafeedh, Maxime Crochemore, Lucian Ilie, Jenya Kopylov, William F. Smyth, German Tischler, and Munina Yusufu, A comparison of Lempel-Ziv LZ77 factorization algorithms, submitted for publication (2009).

Contribution by me: I finished the most of survey work by providing much of the bibliography, and wrote the draft of the paper. I also did more than 50% of the testing of the algorithms and supervised the testing throughout the project process.

Munina Yusufu and Gulina Yusufu, Comparison of software specification methods using a case study, *Proc. 2008 International Conference on Computer Science and Software Engineering* (2008) 784–787.

Contribution by me: I proposed the ideas by discussing the properties of five formal specification methods theoretically, and wrote the most of the specifications by designing a particular part of the ABM system using each method. I wrote the draft of the paper. This study was beneficial to my research in the way that the formal specification greatly increases the ease with which programs can be written because it provides detailed and precise definitions of the desired functions.

Munina Yusufu, Computing complete repeats using suffix array, *Presented at WISE (Women in Science & Engineering) Initiative International Women's Day Conference* (2008).

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Various forms of regularity are central to the recognition of important patterns in performing retrieval from massive data sets. In this thesis, we investigate mathematical and algorithmical aspects of regularities in strings. In particular, we have developed novel algorithms for computing regularities which are both time and space efficient.

1.1 Motivation

1.1.1 Applications

The study of strings began a little over 100 years ago with a mathematical study of periodicity [81], the simplest form of regularity. When, 40 years ago, scientists and engineers began to understand that previously unimagined quantities of data would require efficient string *algorithms*, methods for recognizing periodicity in strings were among the first to be proposed [52, 59].

Today algorithms for computing regularities have myriad applications:

- **Data Compression.** Regularities in the form of repeating substrings were the

basis of `gzip`, one of the earliest and still widely-used compression algorithms [56, 91], and remain central in more recent approaches [14].

- **Computational Biology.** Repeats and repetitions of lengthy substrings in DNA and protein sequences are important markers in biological research [10, 82].

- **Information Security.** Spam, the electronic equivalent of junk mail, affects over 600 million users worldwide. Some methods for detecting spam are mainly based on similarity calculations on strings [40, 86].

- **Data Mining.** Various forms of regularity are central to the recognition of important patterns in retrieval from massive data sets [38]. [36] applies a particular pattern mining algorithm to find both ordered and unordered phrases.

- **Analysis of Musical Texts.** The identification of melodies and rhythms in huge musical databases depends heavily on algorithms for computing string regularities, approximate and exact [25, 26, 27].

- **Software Engineering.** The identification of approximate clones of methods or classes in very large software systems is of fundamental importance to effective software maintenance; modern methods depend heavily on algorithms that identify regularities in source or object code [8].

Apart from expected benefits in application areas discussed above, there should also be spin-off benefits of this research within the general scientific/technological

area of combinatorial algorithms.

1.1.2 Time and Space Efficiency

This research has concentrated on the design and development of highly efficient algorithms that permit applications to be dealt with quickly even when the problem size n is extremely large — billions or more. Thus time efficiency is of critical importance. Space efficiency is also essential for fast algorithms on large data sets. In the design of new algorithms for computing regularities, we have therefore tried to achieve conceptual simplicity, time efficiency, and space efficiency.

1.2 Computing Regularities

In this section we provide informal definitions of the regularities that are most important to this thesis. More formal definitions are given in Chapter 2.

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 the following string

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\boldsymbol{x} =$	a	b	a	a	b	a	b	a	a	b	a	a	b	a

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

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

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

Some repeats are more interesting than others. For example, we observe that 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 with the repeats of ba: 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.

Also interesting in a biological context are ***supernonextendible*** (SNE) repeats; that is, NE repeats that are not substrings of any other repeat in $\boldsymbol{x}$. In the above example, $abaaba$ is the only SNE repeat.

Multirepeats

A ***multirepeat*** is a repeat of minimum length p_{min} that occurs at least m_{min} times ($m_{min} \geq 2$) in each of at least $q \geq 1$ strings in a given set of strings. Consider, for example, the two strings

$$s_1 = ACGTACGACG,\ s_2 = ATACGTGACGACG.$$

Given $p_{min} = 3$, $m_{min} = 2$ and $q = 2$, we see that ACG of length p_{min} occurs at least m_{min} times in each of the q strings s_1 and s_2. Therefore ACG is a multirepeat in s_1 and s_2.

Repetitions

A ***repetition*** is a repeat of adjacent substrings. For example,

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

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

It was shown 25 years ago [16] that over all strings of length n, the maximum number of repetitions is $\Theta(n \log n)$ — achieved for example by Fibonacci strings of length n. Therefore it would seem that to output $\Theta(n \log n)$ repetitions must take $\Theta(n \log n)$ time. However, as we discover in the next subsection, repetitions can actually be computed in $\Theta(n)$ time.

Runs

A ***run*** is a periodicity that cannot be extended, either left or right. For example, in

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

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

Using the LZ factorization, it was shown 10 years ago [58] that the runs in any string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ can be computed in $\Theta(n)$ time, and thus essentially, since every repetition is part of a run, the repetitions. It turns out that, for computing runs and repetitions, the LZ factorization is of central importance.

LZ Factorization

Roughly speaking, the LZ factorization of a string identifies either leftmost occurrences of each letter or else longest substrings that occur at least twice in the string. For example, for a string

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

the LZ factorization is $\boldsymbol{f} = \boldsymbol{w_1 w_2 w_3 w_4 w_5 w_7}$, where $\boldsymbol{w_i}$ are the underlined segments.

LZ can be computed in linear time. The LZ factorization is widely used for compression (`gzip`, for example). But, as we have seen LZ is multipurpose: it is also important for computing runs and repetitions.

1.3 Major Contributions

In this thesis we explore theoretical and algorithmic aspects of the computation of regularities in strings. Specifically, we describe the following results in subsequent chapters.

1. **NE/SNE repeats:** We have proposed four variants of a new fast algorithm RPT1 for computing NE repeats and two variants of RPT3 for computing SNE repeats in $\boldsymbol{x}$ of length $p \geq p_{\min}$, using only LCP and BWT arrays, thus they are space efficient. We have also demonstrated for the first time that both NE and SNE complete repeats can be computed in $O(n)$ time independent of alphabet size. We also describe a straightforward algorithm RPT2 that computes

complete NE repeats without any recourse to suffix arrays and whose total space requirement is only $5n$ bytes. Our experimental results have shown that RPT1-4 is the best on overall strings tested, while the RPT1-1 and RPT1-2 are better for non-highly periodic strings. Moreover, RPT1 algorithms are faster than the two other algorithms previously proposed for this problem [33, 67]. We have briefly discussed some applications of our RPT1 and RPT2 algorithms to bioinformatics and data compression.

2. **Multirepeats:** We have also formulated two problems related to multirepeats in sets of strings with various restrictions and extended our RPT1 to present three efficient algorithms. We have demonstrated that our algorithms are faster, more flexible and much more space efficient than algorithms recently proposed for this problem [6]. They are also easier to implement than previous approaches. Among these algorithms, the first two, MultiRep-1 and MultiRep-2, are for multirepeats with arbitrary gaps, while the last one, MultiRepG, applies to the bounded gaps problem. Extending the algorithms of [6], our three algorithms output only repeats whose occurrences are substrings of length at least p_{min} (user-specified), thus eliminating trivial outputs. We remark that if we set the *min* and *max* constraints on gaps equal to zero in the algorithm MultiRepG, we can find all repetitions in arbitrary subsets of a given set S.

3. **LZ factorization:** The final part of our work provides a convenient framework

for comparing the LZ factorization algorithms which are used in the computation of regularities in strings rather than in the traditional application to text compression. We analyzed each algorithm separately and remarked on them by comparing some of their important aspects, for example, additional space required and handling mechanism. We also addressed their output format differences and some special features. We then provided a complete theoretical comparison of their time and space efficiency. We conducted intensive testing on both time and space performance and analyzed the results carefully to draw conclusions in which situations these algorithms perform best. We believe that our investigation and analysis will be very useful for researchers in their choice of the proper LZ factorization algorithms to deal with the problems related to computation of the regularities in strings.

1.4 Thesis Outline

The remainder of this thesis consists of the following chapters.

Chapter 2 introduces the related mathematical background, including all the definitions, such as complete repeats, NE/SNE repeats and multirepeats, and the data structures used in the proposed algorithms, such as suffix array, LCP array and BWT array.

Chapter 3 introduces several new, more efficient algorithms for computing complete NE and SNE repeats, and compares them with existing algorithms by conducting intensive testing using various data sets.

Chapter 4 describes more efficient new algorithms for computing multirepeats with various constraints, and compares them with existing algorithms by a theoretical analysis.

Chapter 5 discusses the main LZ factorization algorithms and provides a theoretical and experimental comparison of them.

Chapter 6 gives some concluding remarks and suggestions for future work.

Chapter 2

Preliminaries

In this chapter, we introduce the notation and terminology of strings which will be used in this thesis as well as the definitions of various regularities. Then we discuss and describe the data structures will be used in our algorithms.

Basic string terminology in this thesis follows [77]. The material in this chapter appears also in [80].

2.1 Basic Definitions

We identify a finite set A called an ***alphabet***, whose elements are ***letters***. The ***cardinality*** of an alphabet denoted by $\alpha = |A|$ is the number of distinct letters in the alphabet.

Usually we consider problems in the context of three kinds of alphabet A:

- ***general alphabet:*** $\forall \lambda, \mu \in A$, it is decidable in constant time whether $\lambda = \mu$ (for example, A is a set of Chinese ideographs).

- ***ordered alphabet:*** a general alphabet in which $\forall \lambda, \mu \in A$, it is decidable in constant time whether $\lambda < \mu$ for some order relation $<$ (for example, A is a set of English-language words).

- ***indexed alphabet:*** $\forall \lambda \in A$, it is possible to declare an array T such that T[λ] is accessible in constant time (for example, A is a subset of the ASCII characters).

A ***string*** $\boldsymbol{x}$ is a sequence of elements drawn from A. In this thesis we represent $\boldsymbol{x}$ as an array $\boldsymbol{x}[1..n]$ of $n \geq 0$ letters, where $n = |x|$ is called the ***length*** of the string. We say that $\boldsymbol{x}$ has n elements $\boldsymbol{x}[1], \boldsymbol{x}[2], ..., \boldsymbol{x}[n]$, and also we say that $\boldsymbol{x}$ has n ***positions*** while position 1 is at ***leftmost*** side of $\boldsymbol{x}$ and position n is at ***rightmost*** side of $\boldsymbol{x}$. The ***empty string*** is denoted by $\boldsymbol{\epsilon}$ which corresponds to an empty array and has length 0.

Corresponding to any pair of integers i and j that satisfy $1 \leq i \leq j \leq n$, we define a ***substring*** $\boldsymbol{x}[i..j]$ of $\boldsymbol{x}$ as follows:

$$\boldsymbol{x}[i..j] = \boldsymbol{x}[i]\boldsymbol{x}[i+1]...\boldsymbol{x}[j].$$

If $i = j$, then $\boldsymbol{x}[i..j] = \boldsymbol{x}[i]$; if $i > j$, then by convention $\boldsymbol{x}[i..j] = \epsilon$. We say that $\boldsymbol{x}[i..j]$ ***occurs*** at position i of $\boldsymbol{x}$ and that it has length $j - i + 1$. If $j - i + 1 < n$, then $\boldsymbol{x}[i..j]$ is called a ***proper substring***.

There are two special kinds of substring $\boldsymbol{x}[i..j]$ which are of particular importance.

For any integer $j \in 0..n$, we say that $\boldsymbol{x}[1..j]$ is a ***prefix*** of $\boldsymbol{x}$; if in fact $j < n$, $\boldsymbol{x}[1..j]$ is called a ***proper prefix***. Similarly, for any integer $i \in 1..n+1$, we say that $\boldsymbol{x}[i..n]$ is a ***suffix*** of $\boldsymbol{x}$, and a ***proper suffix*** if $i > 1$. Note that these definition include ϵ as both a prefix and a suffix.

An ordered alphabet induces an ordering on strings, called lexicographic order: given two strings $\boldsymbol{x}[1..n]$, $\boldsymbol{y}[1..m]$, we say that $\boldsymbol{x} < \boldsymbol{y}$ if and only if one of the following conditions holds:

- $\boldsymbol{x}$ is a proper prefix of $\boldsymbol{y}$;
- for some unique integer $i \in 1..\min\{m, n\}$, $\boldsymbol{x}[1..i-1] = \boldsymbol{y}[1..i-1]$, and $\boldsymbol{x}[i] < \boldsymbol{y}[i]$.

Informally, lexicographic order can be thought of as "dictionary order".

2.2 Definitions of Various Regularities

In this section we define the various regularities in strings that we deal with in this thesis.

2.2.1 Repeats

Intuitively, a ***repeat*** is a collection of repeating substrings, not necessarily adjacent. More formally, a ***repeat*** in $\boldsymbol{x}$ is a tuple

$$\boldsymbol{M_{x,u}} = (p; i_1, i_2, ..., i_e),$$

where $e \geq 2, 1 \leq i_1 < i_2 < ... < i_e \leq n$, and

$$\boldsymbol{u} = \boldsymbol{x}[i_1..i_1 + p - 1] = \boldsymbol{x}[i_2..i_2 + p - 1] = ... = \boldsymbol{x}[i_e..i_e + p - 1].$$

Note that it may happen, for some $j \in 1..e-1$, that $i_{j+1} - i_j = p$ or that $i_{j+1} - i_j < p$ – that is, the substrings of a repeat may be adjacent or even overlap. We call $\boldsymbol{u}$ the ***generator***, $\boldsymbol{p}$ the ***period***, and $\boldsymbol{e}$ the ***exponent*** of $M_{x,u}$. $M_{x,u}$ is called a ***square*** if $e = 2$; this extends the usual definition of a square in which the repeating substrings are required to be adjacent.

We say that $M_{x,u}$ is ***complete*** if for every

$$i \in 1..n \text{ and } i \notin \{i_1, i_2, ..., i_e\},$$

we are assured that $\boldsymbol{x}[i..i + p - 1] \neq \boldsymbol{u}$. We say that $M_{x,u}$ is ***left-extendible*** (LE) if

$$(p; i_1 - 1, i_2 - 1, ..., i_e - 1)$$

is a repeat; in this case, $(p + 1; i_1 - 1, i_2 - 1, ..., i_e - 1)$ is a repeat whose suffixes of length p are specified by $M_{x,u}$. Similarly, $M_{x,u}$ is ***right-extendible*** (RE) if

$$(p; i_1 + 1, i_2 + 1, ..., i_e + 1)$$

is a repeat; in this case, $(p + 1; i_1, i_2, ..., i_e)$ is a repeat whose prefixes of length p are specified by $M_{x,u}$. If $M_{x,u}$ is neither LE nor RE, we say that it is ***nonextendible*** (NE). In $\boldsymbol{x} = abaababa$, the NE repeats are

$$M_{x,a} = (1; 1, 3, 4, 6, 8) \text{ and } M_{x,aba} = (3; 1, 4, 6);$$

since every occurrence of b is both preceded and followed by a, there are no others.

Of particular interest are repeating substrings $\boldsymbol{u}$ such that M_{x,v_1uv_2} is a repeat if and only if $\boldsymbol{v_1} = \boldsymbol{v_2} = \epsilon$, the empty string – in other words, $\boldsymbol{u}$ is not a proper substring of any other repeating substring. We call such repeats ***supernonextendible*** (SNE). The repeating substring $\boldsymbol{u}$ in an SNE repeat $M_{x,u}$ is in some sense the longest in a class of repeating substrings that are substrings of $\boldsymbol{u}$. In the above example, (3; 1, 4, 6), identifying *aba* at positions 1, 4, 6, is the unique SNE repeat.

2.2.2 Multirepeats

In this thesis we consider and solve the multirepeats problem with various constraints.

A repeat $M_{x,u}$ of ***multiplicity*** m is the occurrence of the generator $\boldsymbol{u}$ in the string $\boldsymbol{x}$ m times. We define the ***quorum*** q to be the minimum number of strings in a set of strings such that a nonextendible multirepeat must occur, in order to be considered valid.

If we extend the repeats locating problem from a single string to a set of strings, considering several constraints, such as minimum periods, multiplicity, and quorum, then we give the following definition:

A ***multirepeat*** is a repeat of minimum length (period) p_{min} that occurs at least $m_{min} \geq 2$ times (multiplicity) in each of at least $q \geq 1$ strings (quorum) in a given set of strings. Consider, for example, the three strings in Figure 2.1.

Given $p_{min} = 3$, $m_{min} = 2$, and $q = 2$, we see that ACG of length p_{min} occurs

at least m_{min} times in $3 > q$ strings; that is, ACG satisfies all the constraints including minimum period, minimum multiplicity, and quorum. Therefore ACG is a multirepeat in $\boldsymbol{s}_1$, $\boldsymbol{s}_2$, and $\boldsymbol{s}_3$.

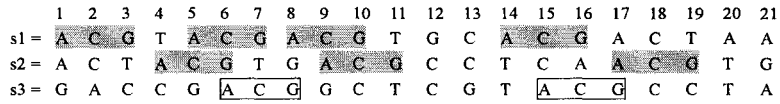

Figure 2.1: Multirepeats in a set of three strings $\boldsymbol{s}_1$, $\boldsymbol{s}_2$, and $\boldsymbol{s}_3$

Assuming that $\boldsymbol{u}$ occurs twice in a string $\boldsymbol{x}$ at positions i_1 and i_2, then the number of symbols between them is called a ***gap*** and it is equal to $g_1 = |i_2 - i_1| - p$. In the case that $g_1 = 0$, then $M_{x,u}$ is called a ***tandem repeat***; if $g_1 < 0$, then it is called ***overlapping***.

If restrictions are posed on the gaps between occurrences of $\boldsymbol{u}$, then the gap g_i between the ith and $(i+1)$th occurrence of $\boldsymbol{u}$ is bounded as follows: $d_{min_i} \leq g_i \leq d_{max_i}$, where d_{min_i} and d_{max_i} are lower and upper bounds on the gap size, respectively. Thus, in this case a repeat $M_{x,u}$ is represented by the pair (u, d), where d is a tuple $((d_{min_1},d_{max_1}), (d_{min_2},d_{max_2}),\ldots,(d_{min_{m-1}},d_{max_{m-1}}))$.

If we add the gap restriction to the above example, choosing $d_{min_i} = 1$ and $d_{max_i} = 5$ for all i, so that $1 \leq g_i \leq 5$, then ACG is a multirepeat only in $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ (shaded occurrences).

2.2.3 Repetitions

A ***repetition*** is a sequence of adjacent repeating substrings. More precisely, a repetition in a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a substring $\boldsymbol{x}[i..i+pe-1] = \boldsymbol{u}^e$, where $|\boldsymbol{u}| = p$ and $e \geq 2$. If moreover $\boldsymbol{u}$ itself is not a repetition, then $\boldsymbol{u}^e$ is said to be ***irreducible***.

Analogous to a repeat, we call $\boldsymbol{u}$ the ***generator***, $\boldsymbol{p}$ the ***period***, and $\boldsymbol{e}$ the ***exponent*** of the repetition $\boldsymbol{u}^e$. Note that a repetition is completely specified by the triple (i, p, e). We say that a repetition $(i, p, e) = \boldsymbol{u}^e$ is ***left-extendible*** (LE) if there exists a repetition at position $i-p$ of $\boldsymbol{x}$ that is also of period p. If no such repetition exists, we say that (i, p, e) is NLE. Similarly, if a repetition $(i, p, e) = \boldsymbol{u}^e$ is ***right-extendible*** (RE) if there exists a repetition at position $i+p$ of $\boldsymbol{x}$ that is also of period p. If no such repetition exists, we say that (i, p, e) is NRE. If (i, p, e) is both NLE and NRE, it is said to be NE.

In the string

	1	2	3	4	5	6	7	8
$\boldsymbol{x} =$	a	b	a	a	b	a	a	a

the repetitions are $(1,3,2) = (aba)^2, (3,1,2) = a^2, (6,1,2) = a^2$, $(6,1,3) = a^3$, and $(7,1,2) = a^2$. Among them, $(1,3,2) = (aba)^2$, $(3,1,2) = a^2$, and $(6,1,3) = a^3$ are NE, $(6,1,2) = a^2$ is RE, and $(7,1,2) = a^2$ is LE.

About a quarter-century ago, three algorithms were discovered [5, 16, 60] that employed widely different approaches to computing all the repetitions in a given string $\boldsymbol{x}[1..n]$ in $\Theta(n \log n)$ time; of these algorithms, two were based on a form of suffix tree

calculation ([5] explicitly, [16] implicitly), while the third used a divide-and-conquer technique. In [16] it was moreover shown that the Fibonacci string $\boldsymbol{f_K}$:

$$\boldsymbol{f_0} = b,\ \boldsymbol{f_1} = a;\ \boldsymbol{f_k} = \boldsymbol{f_{k-1}f_{k-2}},\ \boldsymbol{k} = 2, 3, ..., K$$

actually contains $\Theta(|f_K|\log|f_K|)$ repetitions. Hence these algorithms were regarded as asymptotically optimal, a concept that as we shall see depends heavily on what is accepted as a sufficient specification of a repetition.

2.2.4 Runs

Intuitively, a ***run*** is a nonextendible sequence of overlapping repetitions of the same period.

It was mentioned in the Introduction that the maximum number of repetitions in a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is $\Theta(n \log n)$. But this is a count of repetitions that are both nonextendible and irreducible. If instead we were asked to output the distinct squares $\boldsymbol{u^2}$ without these restrictions, we would find that $\boldsymbol{x} = a^n$, for example, would require $\lfloor n^2/4 \rfloor$ – that is, $\Theta(n^2)$ – outputs to specify squares

$$\boldsymbol{x}[1..2], \boldsymbol{x}[2..3], ..., \boldsymbol{x}[n-1..n], \boldsymbol{x}[1..4], \boldsymbol{x}[2..5], ..., \boldsymbol{x}[n-3..n],$$

and so on. Thus in restricting the output to nonextendible irreducible repetitions, we ***encode*** the output, by tacit agreement with the user, so as to reduce its quantity, hence the asymptotic complexity of the algorithm. For $\boldsymbol{x} = a^n$, this encoding dramatically reduces the output to a single repetition $(1, 1, n)$.

We now describe another encoding of repetitions that further reduces the quantity of output required to $\Theta(n)$. Given an NLE repetition (i, p, e), denote by t the greatest integer such that, for every $j \in 0..t$, $(i + j, p, e)$ is a repetition. Note that since (i, p, e) is nonextendible, therefore $t \in 0..p - 1$. We call t the ***tail*** of (i, p, e). Then a ***run (nonextendible periodicity)*** (it is also called ***maximal periodicity*** in some literature) is a 4-tuple (i, p, e, t), where (i, p, e) is an NLE repetition of tail t.

The idea of a run was first introduced by Main in [58], where also an algorithm was proposed to compute the leftmost occurrence of every distinct run in $\boldsymbol{x}[1..n]$. Given the suffix tree ST and the Lempel-Ziv factorization [56] of $\boldsymbol{x}$ (computable in linear time from ST), Main's algorithm computes all the leftmost runs in $\Theta(n)$ time. In [43] Kolpakov & Kucherov showed that the maximum number $\rho(n)$ of runs in any string of length n satisfies

$$\rho(n) < k_1 n - k_2 \sqrt{n} \log_2 n \tag{1}$$

for some pair of universal positive constants k_1 and k_2. They also extended Main's algorithm to compute all the runs in $\boldsymbol{x}$ in time proportional to their number; thus by (1), given ST, all the runs in $\boldsymbol{x}$, and so in effect all the repetitions, could be computed in $\Theta(n)$ time.

The exact bound of $\rho(n)$ is a subject of intense current research. It is known that $\rho(n) \geq 0.944565n$ [62] and $\rho(n) \leq 1.029n$ [22].

Optimal linear time algorithms for computing all runs exist based on suffix trees

[43, 58] or suffix arrays [3, 23] together with Lempel-Ziv factorization [56, 91].

2.3 Data Structures used in the Algorithms

For storage of data, we make throughout the assumption that string length $n \leq 2^{32}-1$, so that each position in $\boldsymbol{x}$ requires at most one computer word (four bytes) for storage. This is not really a serious restriction, because for computers with a 64-bit word length, we would suppose $n \leq 2^{64} - 1$ with 8 bytes (only the double of 4 bytes) required for storage of each position. This value of n is much larger than the length of any string that could be processed in practice.

Similarly we assume alphabet size $\alpha \leq 256$, the usual case; for larger α, $\lceil \log \alpha \rceil$ bits would be the minimum requirement for each letter in $\boldsymbol{x}$.

2.3.1 Suffix Tree (ST)

The suffix tree is historically one of the most important data structures in string processing. The "traditional" approach to LZ computation for $\boldsymbol{x}$ was based on prior construction of the suffix tree ST of $\boldsymbol{x}$; that is, a compacted trie on all the suffixes of $\boldsymbol{x}$ [37, 77].

The suffix tree for $\boldsymbol{x}$ of length n is defined as a rooted tree such that [87]:

- Edges of the tree are labeled with substrings of $\boldsymbol{x}$. Thus paths from the root are labeled with concatenations of edge labels.
- The path from the root to any terminal node (leaf) is a suffix of the string $\boldsymbol{x}$.

Each of the n suffixes of the string is in the tree, so it has n terminal nodes, which are labeled i, identifying suffix $\boldsymbol{x}[i..n]$;

- Each internal node has at least 2 children. Every internal node I identifies the length of the ***least common prefix*** of all the terminal nodes of the subtree rooted at I.

- The substrings labelling each edge out of a node must all begin with different letters.

Since such a tree does not exist for all strings, $\boldsymbol{x}$ is padded with a terminal symbol not seen in the string (usually denoted \$). This ensures that no suffix is a prefix of another, and that there will be n leaf nodes, one for each of the n suffixes of $\boldsymbol{x}$. Since all internal non-root nodes are branching, there can be at most $n-1$ such nodes, and $n+(n-1)+1 = 2n$ nodes in total.

In a suffix tree, the children of every node are displayed left to right in lexicographical order, and so the leaf nodes are also displayed left to right in lexicographical order. Figure 2.2 shows the suffix tree of

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

In order to construct and search a suffix tree, each internal node must contain a data structure (search tree, sorted array) that enables the correct downward path to be selected based on the next letter. Thus, especially for large alphabets, internal

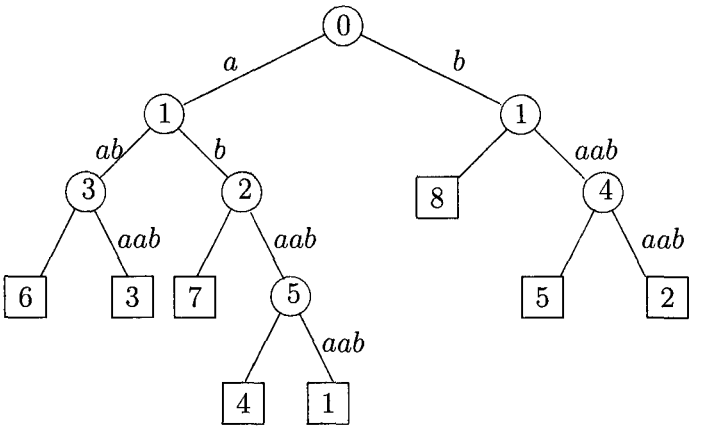

Figure 2.2: The suffix tree of $\boldsymbol{x} = abaabaab$

nodes require substantial storage and so, even though suffix tree storage is linear in n, it is also large, typically 20–40n bytes [77, p. 138]. For large n, this may exceed available main memory capacity.

Suffix trees can be computed in $O(n \log \alpha)$ time [61, 87], where $\alpha \in O(n)$, and online [85] with the same time complexity; on an integer alphabet, $\Theta(n)$-time efficiency is possible [30], but the algorithm is not practical for long strings. Kolpakov & Kucherov [42] have implemented [85] very efficiently so as to compute LZ on-line for $\alpha \leq 4$ (see Section 5.2.1, Algorithm KK).

2.3.2 Suffix Array (SA)

Consider a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ defined on an ordered alphabet A of size α (where if there is no explicit bound on alphabet size, we suppose $\alpha \leq n$). We refer to the suffix $\boldsymbol{x}[i..n]$, $i \in 1..n$, simply as ***suffix*** i. Then the ***suffix array*** SA is an array $[1..n]$ in which $\mathrm{SA}[j] = i$ iff suffix i is the j^{th} in lexicographical order among all the suffixes

of $\boldsymbol{x}$.

The SA array of the string

$$\begin{array}{rcccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{x} = & a & b & a & a & b & a & a & b, \end{array}$$

is shown in column 2 of Figure 2.3, and the corresponding suffixes in column 3.

i	SA[i]	$\boldsymbol{x}$[SA[i]..n]	LCP[i]	LPF[i]	QSA[i]	BWT[i]
1	6	*aab*	0	0	0	*b*
2	3	*aabaab*	3	0	0	*b*
3	7	*ab*	1	1	1	*a*
4	4	*abaab*	2	5	3	*a*
5	1	*abaabaab*	5	4	2	\$
6	8	*b*	0	3	4	*a*
7	5	*baab*	1	2	6	*a*
8	2	*baabaab*	4	1	5	*a*

Figure 2.3: SA, LCP, LPF, BWT, and QSA arrays of $\boldsymbol{x} = abaabaab$

SA can be computed in linear time [45, 49], though various supralinear methods [63, 66] are certainly much faster, as well as more space efficient, in practice [71], in some cases requiring space only for $\boldsymbol{x}$ and SA itself, which requires only $4n$ bytes (4 bytes per input character) in its basic form, compared to 20–40n bytes for the corresponding suffix tree. Probably the best overall SA construction code available is libdivsufsort maintained at the website [65]; we use this code for our tests in Chapter 5. In [3] an enhanced suffix array (ESA) is introduced, consisting of the suffix array together with an "lcp-interval tree" (see Section 5.2.1, Algorithm AKO).

2.3.3 Longest Common Prefix (LCP) Array

Another important data structure that is often used with the suffix array is the Longest Common Prefix (LCP) array. Let us denote the length of the longest common prefix of suffixes i and j by $\mathbf{lcp(i,j)}$. Then, the LCP array contains the lengths of the longest common prefixes between successive suffixes of SA. That is,

$$\mathrm{LCP}[\boldsymbol{i}] = lcp(\mathrm{SA}[i-1], \mathrm{SA}[i]),$$

for $1 < i \leq n$.

The LCP values are closely related to the LCP values given in the internal nodes of the suffix tree. For example, in Figure 2.2, suffixes 4 and 1 have longest common prefix 5. In the corresponding suffix array, suffixes 4 and 1 would occur in positions $i-1$ and i, respectively, with $\mathrm{LCP}[i] = 5$.

Given $\boldsymbol{x}$ and SA, LCP can also be computed in $\Theta(n)$ time [47, 48, 64, 74]: the algorithm described in [64] requires $9n$ bytes of storage and is almost as fast in practice as that of [47], which requires $13n$ bytes. However the algorithm in [74] is generally faster and requires only about $6n$ bytes of storage for its execution, since it overwrites the suffix array. The algorithm very recently proposed in [48] is also very fast, especially applied to highly repetitive strings, but again requires 13 bytes. Thus we use the algorithm of [74] for LCP calculations in this thesis, as the best balance of time and space efficiency. The fourth column of Figure 2.3 gives the LCP array of the string *abaabaab*.

2.3.4 Longest Previous Factor (LPF) Array

The Longest Previous Factor (LPF) array was introduced in [17], but also appears as the **prefix array** π in [34]. For any position i in a string $\boldsymbol{x}$, $\mathbf{LPF}[\boldsymbol{i}]$ is defined to be the length of the longest factor of $\boldsymbol{x}$ starting at position i that occurs previously in $\boldsymbol{x}$. Formally, [17] defines LPF[i] as follows:

$$\text{LPF}[i] = \max\big(\{\ell \mid x[i..i+\ell-1] \text{ is a factor of } x[0..i+\ell-2]\} \cup \{0\}\big)$$

Two LPF calculation algorithms are given in [17], another in [20], all based on prior construction of SA; in [19] an on-line LPF algorithm is described, as well as one that is space-optimal in the sense that only constant storage is required in addition to SA and LCP. For an example of LPF, see column 5 of Figure 2.3. It is shown in [17] that LPF is a permutation of LCP.

2.3.5 Quasi Suffix Array (QSA)

In order to implement the RPT2 algorithm in Chapter 3, we introduce a new data structure, modified from [34], called Quasi Suffix Array (QSA).

For increasing lengths $p = 1, 2, \ldots$ (periods of the repeats) and decreasing positions $i = n, n-1, \ldots, 1$, RPT2 computes QSA[i] $\leftarrow j$, where j is the largest integer less than i such that

$$\boldsymbol{x}[j..j+p-1] = \boldsymbol{x}[i..i+p-1];$$

QSA[i] $\leftarrow 0$ if no such j exists.

QSA (when $p = 1$) is illustrated in column 6 of Figure 2.3. For use in Chapter 5, we reformulate the definition of QSA using LPF as follows:

For every $i \in 1..n$, $\mathrm{QSA}[i] = 0$ iff $\mathrm{LPF}[i] = 0$; otherwise, $\mathrm{QSA}[i] = j$ for some $j \in 1..i-1$ such that

$$\boldsymbol{x}\big[j..j+\mathrm{LPF}[i]-1\big] = \boldsymbol{x}\big[i..i+\mathrm{LPF}[i]-1\big].$$

2.3.6 Burrows-Wheeler Transform (BWT) Array

We define the Burrows-Wheeler Transform BWT of $\boldsymbol{x}$ [14]: for $\mathrm{SA}[j] > 1$, $\mathrm{BWT}[j] = \boldsymbol{x}\big[\mathrm{SA}[j]-1\big]$, while for j such that $\mathrm{SA}[j] = 1$, $\mathrm{BWT}[j] = \$$, a sentinel letter not equal to any other in $\boldsymbol{x}$. In some algorithms it is useful also to define $\mathrm{BWT}[n+1] = \$$. BWT can clearly be computed in linear time from SA; some of our algorithms and LZ algorithms [69] use the BWT array since it occupies only n rather than $4n$ bytes. BWT is illustrated in column 7 of Figure 2.3.

Chapter 3

NE/SNE Repeats

3.1 Introduction

In [37, p. 147] an algorithm is described that, given the suffix tree ST of $\boldsymbol{x}$, computes all the NE (called "maximal") *pairs of repeats* in $\boldsymbol{x}$ in time $O(\alpha n+q)$, where q is the number of pairs output. [12] uses similar methods to compute all NE pairs $(p; i_1, i_2)$ such that $i_2-i_1 \geq g_{min}$ (or $\leq g_{max}$) for user-defined ***gaps*** g_{min}, g_{max}. [3] shows how to use the suffix array SA of $\boldsymbol{x}$ to compute the NE pairs in time $O(\alpha n+q)$. Since it may be that $\alpha \in O(n)$, all of these algorithms require $O(n^2)$ time in the worst case. [33] uses the suffix arrays of both $\boldsymbol{x}$ and its reversed string $\overline{\boldsymbol{x}} = \boldsymbol{x}[n]\boldsymbol{x}[n-1]\cdots\boldsymbol{x}[1]$ to compute all the complete NE repeats in $\boldsymbol{x}$ in $\Theta(n)$ time. More recently, [67] describes suffix array-based $\Theta(n)$-time algorithms to compute all ***substring equivalence classes*** — including the complete NE repeats — in $\boldsymbol{x}$.

In this chapter, following [72, 73, 88], we present four variants of a new fast algorithm RPT1 that computes all the complete NE repeats in a given string $\boldsymbol{x}$ whose

length (period) $p \geq p_{min}$, where $p_{min} \geq 1$ is a user-specified minimum. RPT1 uses $5n$ bytes of space directly, but requires the LCP array, whose construction needs $6n$ bytes. Of the RPT1 algorithms, RPT1-1 is fastest in practice, while the variants RPT1–3 and RPT1–4 execute in $O(n)$ time independent of alphabet size. To provide a basis of comparison for RPT1, we also describe a straightforward algorithm RPT2 that computes complete NE repeats without any recourse to suffix arrays and whose total space requirement is only $5n$ bytes; however, this algorithm is an order of magnitude slower than RPT1.

Finally, we describe two versions of a new linear time algorithm RPT3 to compute all the SNE repeats in $\boldsymbol{x}$. Both versions are fast, but the second, RPT3–2, executes in time $\Theta(n+\alpha)$. This improves on the algorithm described in [37, p. 146] that does the same calculation (of "supermaximal" repeats) in time $O(n \log \alpha)$ using a suffix tree, as well as on the algorithm described in [3, p. 59] that uses a suffix array and requires $O(n+\alpha^2)$ time. For $\alpha \in O(n)$ these times become $O(n \log n)$ and $O(n^2)$, respectively, whereas RPT3–2 remains $\Theta(n)$.

In Section 3.2 we describe our algorithms. Section 3.3 summarizes the results of experiments that compare the algorithms with each other and with existing algorithms. Section 3.4 discusses these results, including the strategy of computing complete (NE and SNE) repeats in the context of applications to bioinformatics and data compression.

3.2 Description of the Algorithms

The RPT1 and RPT3 algorithms described below make direct use of LCP and BWT (but not of SA or of $\boldsymbol{x}$ itself), and therefore require only $5n$ bytes of storage (plus relatively small stack space in the case of RPT1). For these algorithms, then, the $6n$ bytes needed for LCP construction provide an upper bound on the overall space requirement. The RPT2 algorithm uses a quasi suffix array construction modified from [34], thus avoiding altogether the calculation of SA/LCP/BWT. This "direct" approach requires only $5n$ bytes overall, but the resulting algorithm turns out to be an order of magnitude slower than SA/LCP/BWT calculation plus RPT1. On the other hand, RPT2 outputs positions in $\boldsymbol{x}$ directly, whereas RPT1 and RPT3 output ranges $i..j$ of positions in SA that specify complete repeats (NE for RPT1, SNE for RPT3). See Section 3.4 for further discussion of the postprocessing of RPT1/RPT3 output.

3.2.1 RPT1

In this section we introduce the basic methodology of the RPT1 family of algorithms, illustrated with the example

$$\boldsymbol{x}[1..22] = abcaabcabaccaabcacbaac.$$

Figure 3.1 displays the SA, LCP, and BWT arrays for this string. Figure 3.2 gives a graphical representation of the SA and LCP values:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
x	*a*	*b*	*c*	*a*	*a*	*b*	*c*	*a*	*b*	*a*	*c*	*c*	*a*	*a*	*b*	*c*	*a*	*c*	*b*	*a*	*a*	*c*
SA	4	13	20	8	1	5	14	21	17	10	19	9	2	6	15	22	3	12	7	16	18	11
LCP	0	5	2	1	2	4	4	1	2	2	0	2	1	3	3	0	1	6	2	2	1	1
BWT	*c*	*c*	*b*	*c*	*$*	*a*	*a*	*a*	*c*	*b*	*c*	*a*	*a*	*a*	*a*	*a*	*b*	*c*	*b*	*b*	*a*	*a*

Figure 3.1: SA, LCP, and BWT arrays of $\boldsymbol{x} = abcaabcabaccaabcacbaac$

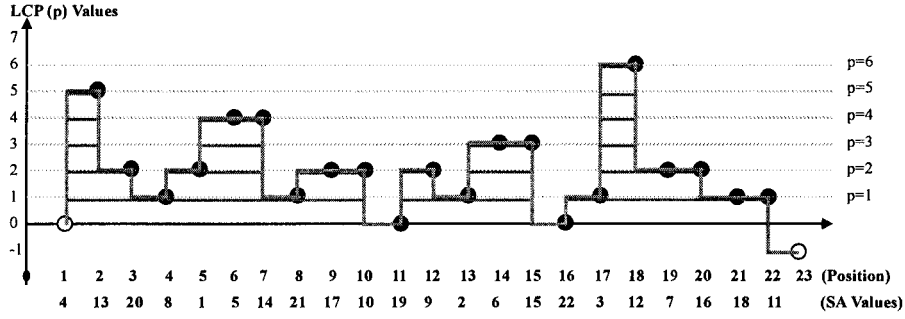

Figure 3.2: LCP and SA arrays with graphical illustration for indicating repeat patterns

- The horizontal axis gives the SA positions $1..n$ with the ending value $n + 1$ added for processing convenience.

- At each position $i = 1, 2, .., n$, both i and the corresponding SA[i] values are shown.

- The vertical axis gives LCP[i] values for pairs of suffixes SA[$i - 1$] and SA[i], $i \geq 2$, with LCP[1] $= 0$ and LCP[$n+1$] $= -1$, again for processing convenience.

- The heavy dots ($\bullet$) in Figure 3.2 identify the LCP value p at each position i.

- The vertical lines in the figure identify increases and decreases in the LCP values p as i ranges from 1 to n.

- Each horizontal line corresponding to some $p > 0$ identifies a repeat in $\boldsymbol{x}$ of length p.

- Each horizontal line can be specified by a tuple $(p; i, j)$ where i is the left endpoint of the line, j is the right endpoint of the line at height p; then $(p; i, j)$ specifies a repeat in $\boldsymbol{x}$, moreover, a complete repeat. For example, since there are 19 such tuples in Figure 3.2, there exist 19 complete repeats in the string $\boldsymbol{x}[1..22]$. For instance, within it, there are 3 repeats of length 4, that is, $(4; 1, 2)$, $(4; 5, 7)$, and $(4; 17, 18)$, corresponding to $(4; 4, 13) = aabc$, $(4; 1, 5, 14) = abca$, and $(4; 3, 12) = caab$, respectively.

  The notation $(p; i, j)$ used here, that identifies a range $i..j$ in SA provides a mechanism for compressing the reporting of repeats; in terms of positions in $\boldsymbol{x}$, for example, the repeat $(4; 5, 7)$ would need to be reported as $(4; 1, 5, 14)$. See below for the explanation.

- Some horizontal lines occur in a specific ***segment*** $(p; i, j)$, $p = p', p' + 1, .., q$; that is, with the same range $i..j$, but with different heights p. In such cases,

the ***peak*** tuple $(q; i, j)$ represents a longest repeat for positions i and j:

$$\boldsymbol{x}[\mathrm{SA}[i]..\mathrm{SA}[i]+q-1]$$

$$\boldsymbol{x}[\mathrm{SA}[i+1]..\mathrm{SA}[i+1]+q-1]$$

$$\vdots$$

$$\boldsymbol{x}[\mathrm{SA}[j]..\mathrm{SA}[j]+q-1]$$

For example, the peak tuple $(5; 1, 2)$ identifies the longest repeat

$$\boldsymbol{x}[\mathrm{SA}[1]..\mathrm{SA}[1]+4] = \boldsymbol{x}[4..8] = \boldsymbol{x}[\mathrm{SA}[2]..\mathrm{SA}[2]+4] = \boldsymbol{x}[13..17] = aabca.$$

The other tuples in this segment are $(3; 1, 2)$ corresponding to aab and $(4; 1, 2)$ corresponding to $aabc$. We observe that both aab and $aabc$ are RE repeats, but $aabca$ is NRE, because it is at the peak (it can not be extended).

Note that $(2; 4, 7) = ab$ is also a longest repeat according to this definition; here $p' = q = 2$, indicating that this segment only includes one tuple, thus ab is NRE.

The following two lemmas express more formally the observations made above:

Lemma 3.1.1 (Completeness) *Suppose there is a tuple $(p; i, j)$ as defined above. Let $u = \boldsymbol{x}[SA[i]..SA[i]+p-1]$. Then $(p; i, j)$ identifies a complete repeat.*

Proof Since p is the longest common prefix of the suffixes

$$\mathrm{SA}[i], \mathrm{SA}[i+1], .., \mathrm{SA}[j],$$

and $i < j$, therefore the prefixes of length p of these suffixes certainly identify a repeat

$$(p; \mathrm{SA}[i], \mathrm{SA}[i+1], .., \mathrm{SA}[j])$$

of $\boldsymbol{x}$. If $(p; i, j)$ is not a complete repeat, then there must exist k, such that $\boldsymbol{x}[k..k+p-1] = u$, for $k \in \{1, 2, .., n\} \wedge k \notin \{\mathrm{SA}[i], \mathrm{SA}[i+1], ..\mathrm{SA}[j]\}$. But since for some t, $\mathrm{SA}[t] = k$, it follows that $t \in \{i, i+1, .., j\}$; that is, $k \in \{\mathrm{SA}[i], \mathrm{SA}[i+1], .., \mathrm{SA}[j]\}$, so $M_{x,u} = (p; \mathrm{SA}[i], \mathrm{SA}[i+1], .., \mathrm{SA}[j])$ must be a complete repeat of $\boldsymbol{x}$. □

This lemma, together with the graphical presentation exemplified in Figure 3.2, motivates representation of repeats by $(p; i, j)$ where $i..j$ is a range in SA. This is the approach adopted by the RPT1 family of algorithms.

Lemma 3.1.2 (NRE repeat) *The peak tuple $(q; i, j)$ which represents a longest repeat within a specific segment $(p; i, j)$, where $p = p', p'+1, .., q$, must be a NRE repeat.*

Proof If $(q; i, j)$ were not NRE, there would be a horizontal line $q+1$ above q in the segment, a contradiction. □

Note, however, that a peak tuple may be LE; for example $(3; 13, 15)$ is a peak tuple in Figure 3.2, corresponding to the repeating substring

$$\boldsymbol{x}[2..4] = \boldsymbol{x}[6..8] = \boldsymbol{x}[15..17] = bca,$$

which is LE to

$$\boldsymbol{x}[1..4] = \boldsymbol{x}[5..8] = \boldsymbol{x}[14..17] = abca.$$

RPT1–1

According to the methodology we discussed above, we now present the first algorithm.

Figures 3.3 and 3.5 show pseudocode for the brute force (but fast) version RPT1–1.

```
  — Preprocessing: compute SA, BWT & LCP
  — in Θ(n) time (LCP overwrites SA).
j ← 0; p ← −1; q ← 0; prevNE ← 0; push(LB; 0, 0)
while j < n do
    repeat
        j ← j+1; p ← q; q ← LCP[j+1]
        if q > p and q ≥ p_min then push(LB; j, q)
    until p > q
    repeat
        if top(LB).lcp > 0 then
            (i, p) ← pop(LB)
            if prevNE ≥ i then output(p; i, j)
            elsif NLE(i, j) then prevNE ← i; output(p; i, j)
    until top(LB).lcp ≤ q
    if top(LB).lcp < q and q ≥ p_min then
        push(LB; i, q)
```

Figure 3.3: Algorithm RPT1–1 — compute all NE repeats of period $p \geq p_{min}$ as ranges in SA using one stack

RPT1–1 performs a single left-to-right scan of LCP. It uses a single stack LB (Left Boundary) containing pairs (position, LCP value), that identifies leftmost positions i at which there is an increase in the LCP value. More precisely, in the typical case, entries (j, q) are pushed onto LB at every position j at which the LCP value increases (to q) and popped whenever the LCP value decreases. Thus each pop, together with

the current j value, identifies a complete repeat $(p; i, j)$ as proved in Lemma 3.1.1, that must be NRE according to Lemma 3.1.2, but that may or may not be NLE.

But a push onto LB may also occur after a sequence of pops has been processed (see the second *repeat* loop in Figure 3.3). This case arises when the current LCP value $q = \text{LCP}[j + 1]$ (not pushed onto LB) turns out to be strictly greater than the LCP value at the top of the stack; thus for the current leftmost position i, the pair (i, q) needs to be pushed onto LB.

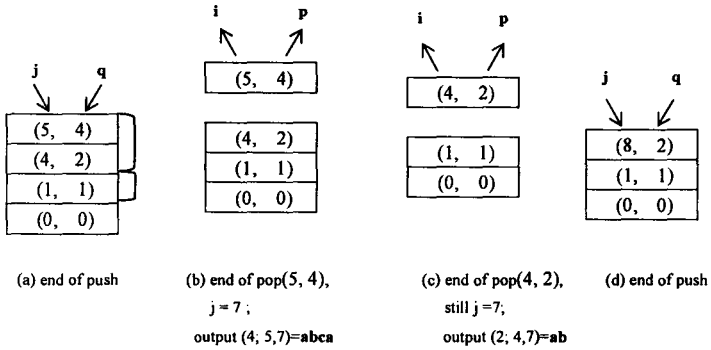

Figure 3.4: Snapshots of the stack LB in RPT1–1

Figure 3.4 (a) shows these two cases. For $j = 7$, the stack contains four pairs. The first of them is the initial element $(0, 0)$, followed by $(1, 1)$ that is produced by line 16 of the code (Figure 3.3), recording the fact that there is a candidate repeat starting at $i = 1$ with period 1, which is strictly greater than the LCP value 0 at the top of the stack LB. The next two pairs, $(4, 2)$ and $(5, 4)$, are produced by line 7 of

the code. In contrast, Figure 3.4 (d) shows a typical case only, where $(j, q) = (8, 2)$ was pushed onto the stack by line 7. In between snapshot (a) and snapshot (d), two NE repeats were generated, $(4; 5, 7) = M_{x,abca}$ and $(2; 4, 7) = M_{x,ab}$ by popping the pairs $(5, 4)$ and $(4, 2)$.

Since the expected maximum length of a repeating substring in a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet of size α is $2\log_\alpha n$ [46], this quantity is also the expected maximum number of entries in LB; in the worst case ($\boldsymbol{x} = a^n$), the maximum could be $\Theta(n)$. Note that since there is at most one output for each pop of LB, the number of repeats, thus in particular the number of complete NE repeats in $\boldsymbol{x}$ is $O(n)$ (at most the number of internal nodes in the suffix tree).

Lemma 3.1.3 (NE repeat) *Suppose $M_{x,u} = (p; i, j)$ is the complete repeat output after a "pop". Then $M_{x,u}$ is a complete NE repeat iff $M_{x,u}$ is NLE.*

Proof According to Lemma 3.1.1 and Lemma 3.1.2, $M_{x,u}$ must be a complete NRE repeat, since only those repeats which are peak tuples have been pushed and thus popped in RPT1–1. If $M_{x,u}$ is also NLE, $M_{x,u}$ is a complete NE repeat. On the other hand, if $M_{x,u}$ is a complete NE repeat, then $M_{x,u}$ must be NLE. □

Referring to Figure 3.2, RPT1–1 will never push $(j, q) = (5, 3)$ and then pop the complete repeat $(3; 5, 7) = M_{x,abc}$ to check if it is NLE, since it is not a peak tuple, thus not NRE; instead it pushes $(j, q) = (5, 4)$ onto the stack LB, then pops the

complete NRE repeat $(4; 5, 7) = M_{x,abca}$ to check if it is NLE. If it is, then it must be NE.

We use function NLE to check whether a complete NRE repeat is NLE. In this function, we use the BWT array instead of the SA array to reduce the space complexity from $4n$ bytes to n bytes.

function NLE(i, j)
— *Range is LE only if all preceding letters are identical.*
$\lambda \leftarrow$ BWT$[i]$; $i' \leftarrow i+1$
while $i' \leq j$ **and** $\lambda =$ BWT$[i']$ **do** $i' \leftarrow i'+1$
return $(i' \leq j)$

Figure 3.5: Determine whether the repeat $(p; i, j)$ is NLE (one stack)

Since by definition BWT$[i] = \boldsymbol{x}$[SA$[i] - 1$], BWT provides information about the letter to the left of the NRE repeat. Once an NRE repeat $M_{x,u}$ is popped, we only need to check that the preceding letters of each occurrence of the repeating substring u are identical; if they are, then it is LE, otherwise NLE. Referring to the above example, if we check $(4; 5, 7) = M_{x,abca}$ for the NLE property, since

$$\lambda = \text{BWT}[5] = \$,\ \text{BWT}[i'] = \text{BWT}[6] = a,$$

the NLE function will return *true*, indicating that $(4; 5, 7) = M_{x,abca}$ is an NLE repeat.

Now we discuss a very important feature of NE repeats and how we implement it in RPT1–1; that is, the use of the variable *prevNE*.

Observe that *prevNE* is originally set in line 13 of Figure 3.3 to be the lefthand boundary i in SA of the range $i..j$ of the NE repeat most recently output. Thereafter,

as shown in line 12, every NRE repeat subsequently found whose lefthand boundary i is not to the right of $prevNE$ ($i \leq prevNE$) is accepted to be also NLE and therefore output. This situation is illustrated in Figure 3.4, where the NE repeat $(4; 5, 7) = abca$ will be identified and output first, causing $prevNE$ to be set to 5; then subsequently, the NRE repeat $(2; 4, 7) = ab$ is considered; since $i = 4 \leq prevNE = 5$, $(2; 4, 7)$ is NLE and therefore output. Thus in this case the function NLE does not need to be invoked.

To explain why this is so, we state and prove two lemmas.

Lemma 3.1.4 *Given two distinct complete NE repeats* $M_{x,u} = (p; i_1, j_1)$ *and* $M_{x,\widetilde{u}} = (\widetilde{p}, i_2, j_2)$, *if* $i_2 \leq i_1 < j_1 \leq j_2$, *then* $\widetilde{p} < p$.

Proof Let $u = \boldsymbol{x}[\mathrm{SA}[h_1]..\mathrm{SA}[h_1]+p-1]$, $i_1 \leq h_1 \leq j_1$, $\widetilde{u} = \boldsymbol{x}[\mathrm{SA}[h_2]..\mathrm{SA}[h_2]+\widetilde{p}-1]$, $i_2 \leq h_2 \leq j_2$. Therefore $\widetilde{u} = \boldsymbol{x}[\mathrm{SA}[h_1]..\mathrm{SA}[h_1]+\widetilde{p}-1]$, $i_1 \leq h_1 \leq j_1$, and either $u = \widetilde{u}$, u is a proper prefix of $\widetilde{u}$, or $\widetilde{u}$ is a proper prefix of u. If $u = \widetilde{u}$, then $p = \widetilde{p}$, and since the two repeats are complete and NE, they are therefore not distinct, a contradiction. Suppose therefore that u is a proper prefix of $\widetilde{u}$, so that $p < \widetilde{p}$. Hence $u = \boldsymbol{x}[\mathrm{SA}[h_2]..\mathrm{SA}[h_2]+p-1]$, $i_2 \leq h_2 \leq j_2$, and $(p; i_2, j_2)$ is a repeat, contradicting the assumption that $(p; i_1, j_1)$ is complete. It follows that $\widetilde{u}$ is a proper prefix of u, hence that $p > \widetilde{p}$. □

Lemma 3.1.5 *Given two distinct complete NE repeats* $M_{x,u} = (p; i_1, j_1)$ *and* $M_{x,\widetilde{u}} =$

$(\widetilde{p}, i_2, j_2)$. *Suppose that* $M_{x,u}$ *is an NE repeat,* $M_{x,\widetilde{u}}$ *an NRE repeat. If* $i_2 \leq i_1 < j_1 \leq j_2$, *then* $M_{x,\widetilde{u}}$ *must be an NE repeat.*

Proof Since $i_2 \leq i_1 < j_1 \leq j_2$, $\widetilde{u}$ must be a proper prefix of u, and since $M_{x,u}$ is NE, therefore NLE, so $M_{x,\widetilde{u}}$ must be NLE; since $M_{x,\widetilde{u}}$ is also NRE, so $M_{x,\widetilde{u}}$ must be an NE repeat. □

The importance of $prevNE$ derives from Lemma 3.1.5; thus if an NE repeat $(p; i_1, j_1)$ has been output, we set $prevNE \leftarrow i_1$. Then for any subsequent NRE repeat $(\widetilde{p}; i_2, j_2)$ such that $\widetilde{p} \leq p$, $i_2 \leq i_1 < j_1 \leq j_2$, we know without further testing that $(\widetilde{p}; i_2, j_2)$ is NLE and can therefore be output; this is implemented in line 12 by checking whether $i_2 < prevNE$. Thus storing $prevNE$ enables us to reduce testing when a current range falls within the subsequent one.

In practice, this heuristic greatly reduces the time requirement. It is interesting that, apart from highly periodic strings (that rarely occur in practice), RPT1–1 is the fastest of the four variants on the strings used for testing. The variable $prevNE$ is also used in the other RPT1 algorithms, but its effect is diminished in those cases by other features of those algorithms.

However, RPT1–1 is not linear in string length n in the worst case. For integer $k \geq 1$, $n = 8k+2$, $\boldsymbol{x} = (ab)^{n/2}$, every repeat of $b, bab, \ldots, b(ab)^{n/2-2}$ is LE, requiring $n/2, n/2–1, \ldots, n/2–(n/2-2)$ positions of SA, respectively, to be checked by function NLE, a total of

$$\sum_{i=0}^{n/2-2}(n/2-i) = (n/2)(n/2-1) - \sum_{i=0}^{n/2-2} i$$

$$= (n/2)\ (n/2 - 1) - ((n/2 - 1)\ (n/2 - 2))/2$$

$$= (n/2 + 2)(n/2 - 1)/2 \in O(n^2)$$

letter comparisons.

RPT1–2

To avoid repetitive checking of LE repeats, we introduce two integer variables, $leftLE$ and $rightLE$, that identify the left and right boundaries, respectively, of the repeat (range in SA) most recently found to be LE in the left-to-right scan of LCP. In the event that $leftLE..rightLE$ is a subrange of a range $i..j$ whose LE status needs to be checked, this change allows the LE subrange $leftLE..rightLE$ to be skipped.

Assume that a particular peak tuple $M_{x,u'} = (p; i'..j')$ is LE. Then

$$\boldsymbol{x}[\mathrm{SA}[i'] - 1] = \boldsymbol{x}[\mathrm{SA}[i' + 1] - 1] = ... = \boldsymbol{x}[\mathrm{SA}[j'] - 1].$$

If we set such i' to $leftLE$, j' to $rightLE$, then for the subsequent complete NRE repeat $M_{x,u} = (\widetilde{p}; i, j)$ such that $i \leq i' < j' \leq j$, we also need to check whether or not it is NLE, that is, to check whether or not there exists at least one pair in the letters

$$\boldsymbol{x}[\mathrm{SA}[i] - 1], \boldsymbol{x}[\mathrm{SA}[i + 1] - 1], ..., \boldsymbol{x}[\mathrm{SA}[j] - 1]$$

are different from each other. But since $i \leq i' < j' \leq j$, we only need to check that

$$((\widetilde{p}; i..leftLE) \text{ is NLE}) \vee ((\widetilde{p}; rightLE..j) \text{ is NLE}).$$

From this analysis, we know that we can eliminate unnecessary character comparisons. The revised algorithm RPT1–2 is shown in Figure 3.6.

```
— Preprocessing: compute SA, BWT & LCP
— in Θ(n) time (LCP overwrites SA).
j ← 0; p ← −1; q ← 0; prevNE ← 0; push(LB; 0, 0)
while j < n do
    repeat
        j ← j+1; p ← q; q ← LCP[j+1]
        if q > p and q ≥ p_min then push(LB; j, q)
    until p > q
    leftLE ← j; rightLE ← j
    repeat
        if top(LB).lcp > 0 then
            (i, p) ← pop(LB)
            if prevNE ≥ i then output(p; i, j)
            elsif rightLE < j and NLE(rightLE, j) then
                prevNE ← rightLE; output(p; i, j)
            elsif i < leftLE and NLE(i, leftLE) then
                prevNE ← i; output(p; i, j)
            else
                leftLE ← i; rightLE ← j
    until top(LB).lcp ≤ q
    if top(LB).lcp < q and q ≥ p_min then
        push(LB; i, q)
```

Figure 3.6: Algorithm RPT1–2 — compute all NE repeats of period $p \geq p_{min}$ as ranges in SA using one stack and LE range variables $leftLE, rightLE$

We shall find in Section 3.3 that RPT1–2 greatly speeds up for highly periodic strings in practice over RPT1–1, mainly due to the use of these two variables; but in practice it runs the same or slightly slower for other strings, indicating that not many cases exist in these strings such that the subranges of a range are LE, therefore the

functionality of $leftLE$ and $rightLE$ actually is not very useful.

Moreover, RPT1–2 is still not linear in the worst case, a result that is not unexpected, but that turns out to be rather more difficult to establish than for RPT1–1. Consider a string $\boldsymbol{x}$ in which the following substrings occur:

$$\begin{aligned} \boldsymbol{u}^{(k)} &= \mu\lambda_1 \cdots \lambda_k \quad (k \text{ times}) \\ \boldsymbol{u}^{(k-1)} &= \mu\lambda_1 \cdots \lambda_{k-1} \quad (2 \text{ times}) \\ &\vdots \\ \boldsymbol{u}^{(1)} &= \mu\lambda_1 \quad (2 \text{ times}) \end{aligned}$$

where $\lambda_1 < \lambda_2 < \cdots < \lambda_k < \mu$. For example, let

$$\boldsymbol{x} = \boldsymbol{u}^{(k)}\boldsymbol{u}^{(1)}\boldsymbol{u}^{(1)}\boldsymbol{u}^{(k)}\boldsymbol{u}^{(2)}\boldsymbol{u}^{(2)} \cdots \boldsymbol{u}^{(k)}\boldsymbol{u}^{(k-1)}\boldsymbol{u}^{(k-1)}\boldsymbol{u}^{(k)}$$

of length $n = 2(k^2+k-1)$, and observe that for every λ_i, $1 \leq i \leq k$, there exist exactly $k-i+1$ distinct substrings

$$\lambda_i\lambda_{i+1} \cdots \lambda_k < \lambda_i\lambda_{i+1} \cdots \lambda_{k-1} < \cdots < \lambda_i,$$

all of which are NRE and LE repeats, with the lexicographically least occurring k times, the others twice. It follows that during the execution of RPT1–2, the function NLE will need to perform letter comparisons on the substring $\lambda_i\lambda_{i+1} \cdots \lambda_k$ of length $k-i+1$ a total of $k-i+1$ separate times. Then at least

$$\sum_{i=1}^{k}(k-i+1)^2 = \sum_{i=1}^{k} i^2 = k(k+1)(2k+1)/6 \in \Theta(k^3)$$

letter comparisons are required. Since $n \in \Theta(k^2)$, we see that the number of letter comparisons is $O(n\sqrt{n})$. We conjecture that this is the worst case for RPT1–2.

RPT1–3

To guarantee worst-case linear time, we introduce another stack PREVRANGES, thus creating a third variant RPT1–3 (see Figures 3.7 and 3.8).

```
 — Preprocessing: compute SA, BWT & LCP
 — in Θ(n) time (LCP overwrites SA).
j ← 0; p ← −1; q ← 0; prevNE ← 0
push(LB; 0, 0); push(PREVRANGES; 0, 0, $)
while j < n do
    repeat
        j ← j+1; p ← q; q ← LCP[j+1]
        if q > p and q ≥ p_min then push(LB; j, q)
    until p > q
    repeat
        if top(LB).lcp > 0 then
            (i, p) ← pop(LB)
            if prevNE ≥ i then output(p; i, j)
            elsif NLE(i, j, PREVRANGES) then
                setempty(PREVRANGES)
                push(PREVRANGES; 0, 0, $)
                prevNE ← i; output(p; i, j)
    until top(LB).lcp ≤ q
    if top(LB).lcp < q and q ≥ p_min then
        push(LB; i, q)
```

Figure 3.7: Algorithm RPT1–3 — compute all NE repeats of period $p \geq p_{min}$ as ranges in SA using two stacks

If the current repeat processed by function NLE (Figure 3.8) is found to be left-extendible, its range boundaries i, j are pushed onto PREVRANGES together with the letter λ that precedes each suffix in the range $i..j$. Since each range must be

either disjoint from, or a proper subrange of, subsequent ranges identified during the scan (reflecting the subtree structure of the suffix tree of $\boldsymbol{x}$), these ranges allow us to efficiently determine the left-extendibility of subsequent ranges without duplicating letter comparisons already made, based on the following simple observations:

```
function NLE(i, j, PREVRANGES)
 — Range is NLE if any subrange is NLE.
λ ← BWT[i]; I ← i
while top(PREVRANGES).j' > i do
    (i', j', λ') ← pop(PREVRANGES)
    if λ ≠ $ then
        if λ = λ' then I ← j−1
        else λ ← $
if λ = $ then return TRUE
else
    λ ← CHECK(I+1, j, λ)
    if λ = $ then return TRUE
    else push(PREVRANGES; i, j, λ); return FALSE

function CHECK(min, max, λ)
j' ← min
while j' ≤ max and BWT[j'] = λ do j' ← j'+1
if j' > max then
    return λ
else
    return $
```

Figure 3.8: Determine whether the repeat $(p; i, j)$ is NLE (two stacks)

- every subrange of an LE range must also be LE (that is, a single NLE subrange ensures that the enclosing range is also NLE);
- moreover, the letter $\lambda \neq \$$ that establishes left-extendibility must be identical over all the subranges found in PREVRANGES.

Figure 3.9 shows an example of how the stacks of LB and PREVRANGES work for the NRE repeats $(5;1,2)$ and $(2;1,3)$. After function NLE establishes that $(5;1,2)$ is not NLE (that is, BWT[1] = BWT[2] = c), it will set $\lambda = c$, and push $(1,2,c)$ onto stack PREVRANGES (Figure 3.9(b)). Thus when $(1,2)$ is popped into (i,p) (Figure 3.9(a)), where $j = 3$, function NLE will check whether or not $(2;1,3)$ is NLE; since range 1..2 is in range 1..3, it will only compare BWT[3] (b in the example) with the current λ value, by popping out $(1,2,c)$ into (i',j',λ') (Figure 3.9(b)) and calling function CHECK; since $c \neq b$, this indicates that $(2;1,3)$ is NLE. Thus the algorithm avoids unnecessary letter comparisons.

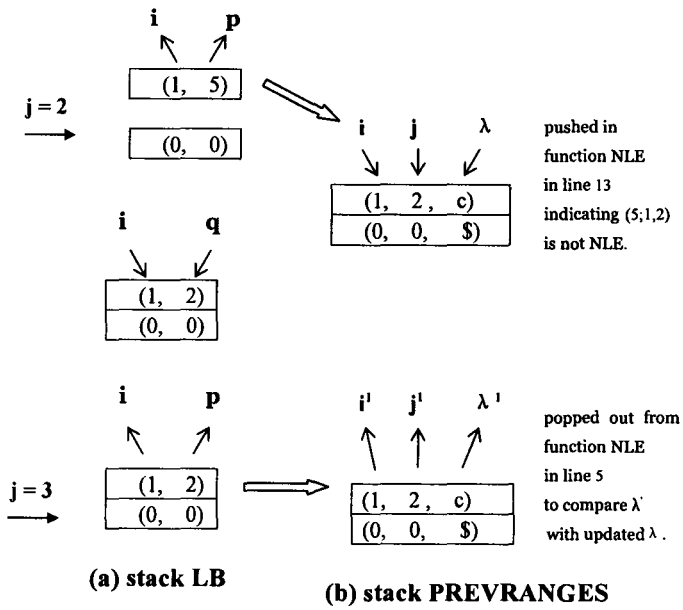

Figure 3.9: Snapshots of the stacks of LB and PREVRANGES in RPT1–3

Since every LE range is pushed onto PREVRANGES with a λ value that is not a sentinel \$, therefore subsequent NRE repeats, which include the previous ranges as subranges, will only check the ranges that are unmarked with λ values; thus in the worst case, the number of letter comparisons is $O(n)$.

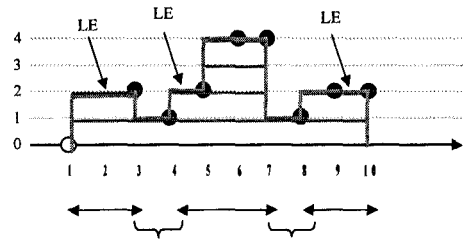

Figure 3.10: A bad case for RPT1–3

One of the bad cases for RPT1–3 is illustrated in Figure 3.10; if NRE repeats $(2;1,3)$, $(2;4,7)$ and $(2;8,10)$ are already identified by function NLE as LE, when an NRE repeat $(1;1,10)$ is checked by function NLE, since ranges 1..3, 4..7 and 8..10 were already checked and pushed into stack PREVRANGES with λ values, therefore function NLE will yield these λ values. As a result, function NLE will only check whether or not

$$\text{BWT}[3] = \text{BWT}[4] \text{ and } \text{BWT}[7] = \text{BWT}[8].$$

Note that, if NRE repeat $(4;5,7)$ was identified to be LE, then when $(2;4,7)$ is checked by function NLE, it will only check whether or not BWT[4] = BWT[5]. Thus each position in the whole range 1..10 will only be checked once by the NLE function.

If an NRE repeat in any subrange of 1..10 was checked to be NLE (thus NE), then $(1; 1, 10)$ will be directly output without invoking the NLE function because of the functionality of the $prevNE$ variable.

RPT1–4

To guarantee worst-case linear time, and still use one stack, we create the fourth variant RPT1–4. As shown in Figure 3.11, RPT1–4 performs a single left-to-right scan of LCP, inspecting each position j from 1 to n. During the scan, whenever a position lb (initially $lb = j$) is found for which the LCP value increases, an entry is pushed onto a stack LB. As before, LB specifies the Left Boundary lb and period p of a repeat that must be NRE, but that may or may not be NLE: lb marks the leftmost occurrence in SA of a repeating substring of length $p = \text{LCP}[lb{+}1] > \text{LCP}[lb]$, thus the left boundary of a repeat. In fact, a triple (p, lb, bwt) is pushed onto the stack, where bwt is a letter that determines the left-extendibility of the repeat: initially bwt equals the sentinel letter \$ if $\text{BWT}[lb] \neq \text{BWT}[lb{+}1]$, and otherwise equals $\text{BWT}[lb]$. This is the calculation performed repeatedly by the function `LEletter`. Thus $bwt = \$$ if the repeat is NLE (and so eventually should be output), but assumes a regular letter value if the repeat (so far at least) is LE.

Since the pushes to LB occur in increasing order of position lb, the pops occur in decreasing order of lb: the most recently pushed triple is popped when a position j is reached for which $\text{LCP}[j{+}1] < \texttt{top}(LB).lcp$. Then j is the right boundary for

```
— Preprocessing: compute SA, BWT & LCP
— in Θ(n) time and 6n bytes of space.
lcp ← LCP[1]; lb ← 1; bwt1 ← BWT[1]
push(LB; lcp, lb, bwt1)
for j ← 1 to n do
    lb ← j; lcp ← LCP[j+1]
     — Compute LEletter of BWT[j] and BWT[j+1].
    bwt2 ← BWT[j+1]; bwt ← LEletter(bwt1, bwt2); bwt1 ← bwt2
    while top(LB).lcp > lcp do
        pop(LB; p, i, prevbwt)
        if prevbwt = $ and p ≥ p_min then
            output(p; i, j)
        lb ← i
        top(LB).bwt ← LEletter(prevbwt, top(LB).bwt)
        bwt ← LEletter(prevbwt, bwt)
    if top(LB).lcp = lcp then
        top(LB).bwt ← LEletter(top(LB).bwt, bwt)
    else
        push(LB; lcp, lb, bwt)

function LEletter(ℓ1, ℓ2)
if ℓ1 = $ or ℓ1 ≠ ℓ2 then return $
else return ℓ1
```

Figure 3.11: Algorithm RPT1–4: compute all NE repeats of period $p \geq p_{min}$ as ranges in SA

the popped triple $(p, i, prevbwt)$ and a repeat $(p; i, j)$ is identified. Observe that this repeat is NRE: if the same letter followed each occurrence of the repeating substring of length p, then p could not be maximum, contradicting the definition of LCP.

It remains to determine whether or not the popped triple is NLE. For this the popped value *prevbwt* needs to be inspected to determine whether it is \$ — that is, whether the repeat is NLE, whether it should be output. To ensure that top(LB).*bwt* is maintained correctly, we use a simple property of ranges of repeats: two ranges are

either disjoint (empty common prefix) or else one range contains the other (common prefix over the longer range). It follows that if top(LB).*bwt* = \$ for a contained range, then for every range that encloses it, we must also have top(LB).*bwt* = \$. Moreover, if for some letter $\lambda \in A$, a contained range is LE with *bwt* = λ, then the enclosing range will be LE only if every other contained range also has *bwt* = λ. In RPT1–4 the correct *bwt* value for the enclosing range is maintained by invoking `LEletter` to update top(LB).*bwt* whenever LCP$[j+1] \leq$ top(LB).*lcp*. For LCP$[j+1] <$ top(LB).*lcp*, `LEletter` is used again to update the current *bwt* based on the *prevbwt* just popped.

In view of this discussion, we claim the correctness of RPT1–4. Execution time is $\Theta(n)$, since the number of executions of the *while* loop is at most the number of triples pushed onto LB, thus $O(n)$.

Space required directly for the execution of all RPT1 variants is $5n$ bytes plus maximum stack storage: 8-byte entries in LB and 9-byte entries in PREVRANGES. The largest number of entries in both of these stacks will be the maximum depth of the suffix tree — thus $O(n)$ in the worst case — but expected depth on an alphabet of size $\alpha > 1$ is $2\log_\alpha n$ [46]. Thus even for $\alpha = 2$, expected space for LB is $18\log_\alpha n$ bytes — if $n = 2^{20}$, 360 bytes. On strings arising in practice, LB requires negligible space (Section 3.4).

Figure 3.12 shows how the stack LB works for NRE repeats $(5; 1, 2)$ and $(2; 1, 3)$. Since BWT[1] = BWT[2] = c, the triple $(5, 1, c)$ will be pushed onto stack; when it

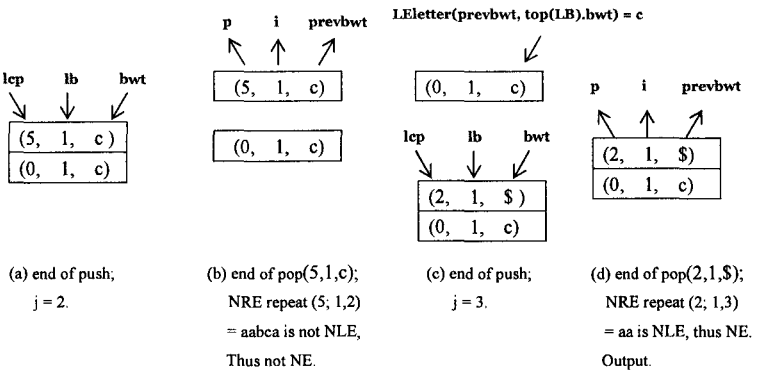

Figure 3.12: Snapshots of the stack LB in PRT1–4

popped into variables $(p, i, prevbwt)$, NRE repeat $(5; 1, 2)$ will not be output since $prevbwt \neq \$$; then `LEletter` is used to update the current `top`(LB).*bwt* (to c) and *bwt* (to \$) based on the *prevbwt* just popped. So in the next step, $(2, 1, \$)$ will be pushed onto stack; then when it is popped into variables $(p, i, prevbwt)$, NRE repeat $(2; 1, 3)$ will be output since $prevbwt = \$$. We observe that the *bwt* value plays the key role in determining the left extendibility of the NRE repeats.

3.2.2 RPT2

As outlined in Figure 3.13 and 3.14, Algorithm RPT2 avoids altogether the calculation of SA/BWT/LCP and instead uses a QSA computation modified from [34] to compute all the NE repeats of period at least p_{min} that are separated in $\boldsymbol{x}$ by gaps of at most g_{max}. Unlike RPT1, RPT2 is therefore unable to output repeats as a simple range in

SA — it must output each position in $\boldsymbol{x}$ at which a repeating substring occurs.

Moreover, RPT2 is certainly not linear in its behaviour in the worst case (see below for a discussion of complexity). On the other hand, RPT2 has the advantage of being easily adjustable to cases when p_{min} and g_{max} are important — it is fast in practice for values of p_{min} that are not too small, and it uses just over $5n$ bytes of storage ($\boldsymbol{x}$, QSA, and the bit vector PROC).

```
— Fast sort: initialize QSA for p = 1 by setting QSA[i] ← j iff j is the
— first position left of i such that x[i] = x[j]; QSA[i] ← 0 if none such.
— Initialize n' ← number of zero positions in QSA.
count[1..α] ← 0^α; n' ← 0
for i ← 1 to n do
    QSA[i] ← count[x[i]]
    if QSA[i] = 0 then
        n' ← n' + 1
    count[x[i]] ← i
```

Figure 3.13: Algorithm RPT2 — Initialize QSA array and n'

For increasing lengths $p = 1, 2, \ldots$ and decreasing positions $i = n, n-1, \ldots, 1$, RPT2 computes QSA$[i] \leftarrow j$, where j is the largest integer less than i such that

$$\boldsymbol{x}[j..j+p-1] = \boldsymbol{x}[i..i+p-1];$$

QSA$[i] \leftarrow 0$ if no such j exists. For some largest $i = i_1$, therefore, the result is a ***chain***: $i_1, i_2, \ldots, i_k$, of length $k \geq 1$, where for every $h \in 1..k-1$, QSA$[i_h] = i_{h+1}$ and QSA$[i_k] = 0$. Thus for $k \geq 2$ each such chain (p, i) defines a complete repeat of substrings of $\boldsymbol{x}$ of length p. As shown in Figure 3.14, RPT2 classifies every new chain according to its extendibility:

```
p ← 1; max ← n
while n′ < n do
    PROC[1..max] ← FALSE^max
    for i ← max downto 2 do
        if not PROC[i] then
            PROC[i] ← TRUE
            q ← QSA[i]
— For a new chain, determine whether LE, RE, or NE (output required).
            if q > 0 then
— Determine whether current chain LE/RE; if NRE, find first break j.
                (LE, j, j′) ← checkchain(p, i, q)
                if LE then
— For every position h in chain (p, i) such that QSA[h] ≠ 0,
— set QSA[h] ← 0 and n′ ← n′+1.
                    (QSA, n′) ← setzero(p, i, q, n′)
                elsif j′ = 0 then
— Current chain is RE: it can be skipped for the current p.
                    PROC ← oldchain(p, i, q)
                else
                    outchain(p, i, q, j, p_min, g_max)
                    if j = i then q ← −q
— In the interior of a chain already processed, prepare for p+1.
            if q < 0 then
                (QSA, n′) ← splitchain(p, i, q)
    p ← p+1; max ← max−1
```

Figure 3.14: Algorithm RPT2 — output all NE repeats of period $\geq p_{min}$ with gaps $\leq g_{max}$

- If the chain is LE, then the positions $i = i_1, i_2, \ldots, i_k$ will not be reported as part of any NLE repeat, and so we set QSA$[i] \leftarrow 0$ for each such i.

- If the chain is NLE and RE, it will be output for some $p' > p$, and so each position in the chain can be marked PROC for the current p.

- If the chain is NLE and NRE, then it may need to be output. At the same time,

some advantage may be taken of information available from function checkchain that specifies the positions j, j' in the chain at which it ceases to be RE.

```
function checkchain(p, i, q)
h ← i; h' ← q; LE ← TRUE; RE ← TRUE
while h' > 0 and (LE or RE) do
    if LE and x[h−1] ≠ x[h'−1] then
        LE ← FALSE
    if RE then
        if x[h+p] ≠ x[h'+p] then
            RE ← FALSE; t ← h; t' ← h'
    h ← h'; h' ← QSA[h']
if h' = 0 and RE then
    return (LE, h, h')
else
    return (LE, t, t')
```

Figure 3.15: Algorithm RPT2 — function checkchain

In the function outchain, if $p \geq p_{min}$, output subchains consisting of elements of chain (p, i) whose occurrences are within g_{max} of each other. For positions $pos > j$ in the chain, set PROC[pos] ← TRUE; set QSA[j] ← −QSA[j].

For interior positions in chains of period p, identified by negative QSA values, it is necessary only to specify the nearest position to the left (or zero, if none such) that is a member of the $(p+1)$-chain. This is the task of function splitchain.

Now we analyze the time complexity of the algorithm RPT2. RPT2 uses three functions, checkchain, oldchain and splitchain, that are called within the internal for loop. Any call of any one of the three functions could require $O(n)$ time. Observe that the for loop of RPT2 is called for the values $max = n, n-1, n-2, ..., n-k$,

```
function oldchain(p, i, q)
t ← q
while t > 0 do
    PROC[t] ← TRUE; t ← QSA[t]

function splitchain(p, i, q)
i' ← i+p
if i' > n then
    QSA[i] ← 0; n' ← n'+1
else
    t ← −q; λ ← x[i']
    while t > 0 and λ ≠ x[t+p] do t ← QSA[t]
    QSA[i] ← t
    if t = 0 then n' ← n'+1
t ← −q; QSA[t] ← −QSA[t]
```

Figure 3.16: Algorithm RPT2 — functions oldchain & splitchain

where k depends on the setting of n'. Thus the for loop can be executed $O(n^2)$ times, and because of the three functions, each execution could require $O(n)$ time. Thus RPT2 required $O(n^3)$ time. This is the worst case; on some strings RPT2 can be quite fast. We also note that the execution time of RPT2 is related to the input of p_{min} and g_{max}.

3.2.3 RPT3

In this section we introduce two algorithms for computing SNE repeats.

According to Lemma 3.1.2, the peak tuple $(q; i, j) = M_{x,u}$ must be a NRE repeat, and such tuples are candidates for SNRE repeats. Since each range must be either disjoint from, or a proper subrange of, subsequent ranges identified during the scan, then we have the following observations:

- If the range $i..j$ of a peak tuple $(q; i, j)$ is disjoint from subsequent ranges, the peak tuple $(q; i, j)$ must be an SNRE repeat.

- Otherwise, since the repeating substring u will be a proper prefix of subsequent repeating substrings, therefore according to the definition of SNE repeat (given in Section 2.2.1, that is, u is not a proper substring of any other repeating substring), it is not SNRE.

From these observations, we know that for the string $\boldsymbol{x}[1..22]$, the following NRE repeats must be SNRE.

$$(5; 4, 13) = M_{x,aabca}$$

$$(4; 1, 5, 14) = M_{x,abca}$$

$$(2; 21, 17, 10) = M_{x,ac}$$

$$(2; 19, 9) = M_{x,ba}$$

$$(3; 2, 6, 15) = M_{x,ca}$$

$$(6; 3, 12) = M_{x,caabca}$$

We observe that only those peak tuples which are in the "top" positions are SNRE repeats. Figure 3.17 illustrates this pattern.

The next step is to check whether or not these SNRE repeats are also SNLE repeats. For this purpose, we introduce more formal definitions of SNLE and SNRE repeats.

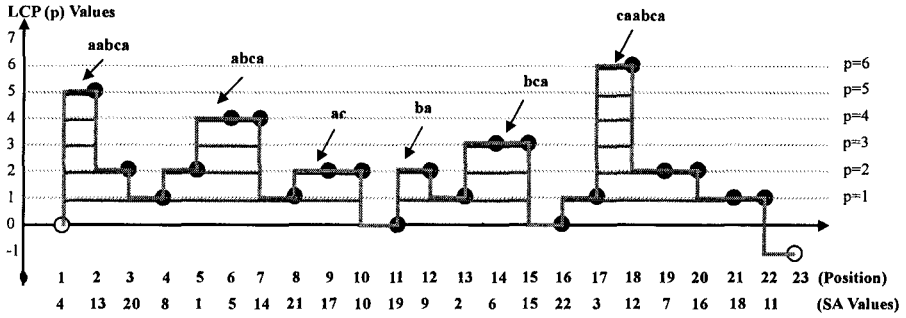

Figure 3.17: LCP and SA arrays with graphical illustration for indicating SNRE repeat patterns

Definition 3.1.1 (SNLE repeat) *Suppose* $M_{x,u} = (p; i, j)$ *is a complete repeat of* $\boldsymbol{x}$. *If* $\boldsymbol{x}[SA[i]-1], \boldsymbol{x}[SA[i+1]-1], .., \boldsymbol{x}[SA[j]-1]$ *are pairwise distinct, then* $M_{x,u} = (p; i, j)$ *is a complete SNLE repeat.*

Definition 3.1.2 (SNRE repeat) *Suppose* $M_{x,u} = (p; i, j)$ *is a complete repeat of* $\boldsymbol{x}$. *If* $\boldsymbol{x}[SA[i]+p], \boldsymbol{x}[SA[i+1]+p], .., \boldsymbol{x}[SA[j]+p]$ *are pairwise distinct, then* $M_{x,u} = (p; i, j)$ *is a complete SNRE repeat.*

As for NE repeats, we draw the following conclusion for SNE repeats:

Lemma 3.1.6 (SNE repeat) *Suppose* $M_{x,u} = (p; i, j)$ *is a complete SNRE repeat, then* $M_{x,u}$ *is SNE iff it is SNLE.*

RPT3–1

The SNE repeats algorithm described in [3] does not deal explicitly with the problem of determining whether or not a complete SNRE repeat is also SNLE. This determination requires that the left extensions (BWT values) of the k positions in the repeat be pairwise distinct as Definition 3.1.1. The straightforward approach to this problem requires at most $\binom{k}{2}$ letter comparisons, where k can possibly be order n. However, we make two observations:

- The cardinality k of an SNE repeat cannot exceed the alphabet size α. Thus a single test suffices to eliminate candidate SNRE repeats of cardinality greater than α, and the straightforward algorithm can then compute SNE repeats in time $O(n+z\alpha^2)$, where $z \in O(n)$ is the number of SNRE repeats in $\boldsymbol{x}$.

- Use of a bit map $B = B[1..\alpha]$ can reduce to $\Theta(\alpha)$ the time required to determine whether or not each SNRE repeat is also SNLE, thus reducing worst-case time to $\Theta(\alpha n)$.

Figure 3.18 gives details of the algorithm RPT3–1 suggested by these remarks. Since RPT3–1 requires no stacks, storage is reduced to $5n$ bytes plus α bits (again with up to $6n$ bytes for LCP construction).

```
 — Preprocessing: compute SA, BWT & LCP
 — in Θ(n) time (LCP overwrites SA).
j ← 0; p ← −1; q ← 0
while j < n do
    high ← 0
    repeat
        j ← j+1; p ← q; q ← LCP[j+1]
        if q > p then high ← q; start ← j
    until p > q
    if high > 0 and SNLE(start, j) then
        output(p; start, j)

function SNLE(start, end)
k ← end−start+1
if k > α then return FALSE
else
    B[1..α] ← FALSE^α
    for h ← start to end do
        λ ← BWT[h]
        if B[λ] then return FALSE
        else B[λ] ← TRUE
    return TRUE
```

Figure 3.18: Algorithm RPT3–1 — compute all SNE repeats as ranges in SA using a bit array B

RPT3–2

A theoretical and also practical disadvantage of RPT3–1 is its need to perform $\Theta(\alpha)$-time processing in function SNLE in order to clear the bit array B, a task that may be repeated $O(n)$ times. We now describe a more sophisticated approach that reduces worst-case complexity to $\Theta(n+\alpha)$ at the cost of a slight increase in actual processing time.

Instead of BWT, we compute an array LAST = LAST[1..n] in which for every

$j \in 1..n$, LAST[j] measures the offset between the BWT letter corresponding to the current position j in SA and the position $jprev$ of the rightmost previous occurrence in SA of the same BWT letter — if $jprev$ does not exist or if $j - jprev \geq \alpha$, then LAST[j] $\leftarrow \alpha - 1$. However, if $jprev$ exists and satisfies $j - jprev < \alpha$, we set

$$\text{LAST}[j] \leftarrow j - jprev - 1,$$

so that LAST[j] takes values in the range $0..\alpha$–2. Then when function SNLE processes a possibly supernonextendible repeat consisting of $end - start + 1$ substrings of $\boldsymbol{x}$, for every position $h \in start+1..end$, the value of BWT[h] will be unique within the range if and only if $h - \text{LAST}[h] > start$. Given LAST, the function SNLE simplifies as shown in Figure 3.19.

```
function SNLE(start, end, LAST)
k ← end−start+1
if k > α then return FALSE
else
    for h ← start+1 to end do
        if h−LAST[h] > start then return FALSE
    return TRUE
```

Figure 3.19: Algorithm RPT3–2 — the simplified SNLE function using LAST

In general it is possible that the offsets stored in LAST could be integers of size $O(n)$. But offsets of magnitude greater than α–1 need not be stored, since as remarked above the interval $start..end$ consists of at most α positions. Thus LAST requires the same amount of storage as BWT, which stores letters that are also restricted to

be at most $\alpha - 1$ in magnitude. The method can be implemented for any finite α, but with the usual convention that each letter in the alphabet is confined to a single byte ($\alpha \leq 256$), the array LAST becomes an array of bytes, just like BWT. The calculation of LAST is shown in Figure 3.20.

```
 — Initialize an array storing rightmost positions of each letter.
for ℓ ← 1 to α do
    lastpos[ℓ] ← 0
 — Compute LAST in a single left-to-right scan of SA.
α' ← α−1
for j ← 1 to n do
    i ← SA[j]−1
    if i ← 0 then
        LAST[j] ← α'
    else
        letter ← x[i]; jprev ← lastpos[letter]
        if jprev = 0 or j−jprev ≥ α then
            LAST[j] ← α'
        else
            LAST[j] ← j−jprev−1
        lastpos[letter] ← j
```

Figure 3.20: Preprocessing for Algorithm RPT3–2 — computing LAST

In fact, in order to take advantage of the CPU cache, our implementation of this algorithm actually computes BWT first, then makes a pass over BWT to convert it into LAST — an approach that turns out to be 2–3 times faster than a straightforward implementation of the preprocessing algorithm.

3.2.4 The Output of RPT1 and RPT3

The RPT1 and RPT3 algorithms described above output NE and SNE repeats in the compact form $(p; i, j)$ (slightly different as $(p; start, j)$ in RPT3) rather than

$$(p; \mathrm{SA}[i], \mathrm{SA}[i+1], ..., \mathrm{SA}[j]).$$

Thus repeats are keyed to a range $i..j$ in SA rather than to a collection of $j - i + 1$ distinct positions in $\boldsymbol{x}$. In order to key each repeat to $\boldsymbol{x}$, we need to access SA and perform processing such as the following in Figure 3.21.

```
— Input: SA[1..n]
for every (p; i, j)
  output (p)
  for h ← i to j do
    output (SA[h])
```

Figure 3.21: Change the output form

Thus in such cases, the RPT1 and RPT3 algorithms constitute the first step of a two-step process. The second step shown in Figure 3.21 requires $O(max(n, \eta))$ time and $O(n)$ space, where η is the number of occurrences of NE/SNE repeats in $\boldsymbol{x}$; also, it requires that SA be available, either in main memory or in secondary storage. This two-step approach does not increase asymptotic complexity, either for space or time, while retaining some flexibility for handling user requirements. Also, it has enabled us to focus on achieving true linear-time performance for the first step.

3.3 Experimental Results

We have implemented and tested eight algorithms: four RPT1 versions, two RPT3 versions, RPT2, and the algorithm of [67].

Some algorithms require SA and LCP arrays as input. For SA construction the KS algorithm was used [49] — the fastest such algorithm is perhaps MP2 [63] that, based on experiments documented in [63, 71], would perform 5–10 times faster on average, using about $5.2n$ bytes of storage. For LCP construction the algorithm of [74] was used, as we discussed in Section 2.3.

All programs were written in C++, using techniques such as function inlining, avoidance of superfluous array references in order to yield efficient code and we are confident that all implementations tested are of high quality.

Platform

Hardware All tests were conducted on a SUN X4600 M2 Server with four 2.6 GHz Dual-Core AMD Opteron(tm) 8218 Processor (total of eight processor cores), 32GB of RAM and four 146GB SAS disks.

Software The operating system is Redhat Linux 5.3 running kernel 2.6.18. All implementations were in C++, compiled using GNU g++ at the -O3 optimization level.

Timing

Running times are the minimum of ten runs and do not include time spent reading input files. Times were recorded with the C++ standard library function **clock**.

Test Data

Experiments were conducted on a diverse selection of files chosen from the collection at `http://www.cas.mcmaster.ca/~bill/strings/` and listed in Table 3.1.

File Type	Name	No. Bytes	Description
highly periodic	fibo35	9,227,465	Fibonacci
	fibo36	14,930,352	Fibonacci
	fss9	2,851,443	run-rich [35]
	fss10	12,078,908	run-rich [35]
random	rand2	8,388,608	$\alpha = 2$
	rand21	8,388,608	$\alpha = 21$
DNA	ecoli	4,638,690	*escherichia coli* genome
	chr22	34,553,758	human chromosome 22
	chr19	63,811,651	human chromosome 19
Genbank protein database	prot-a	16,777,216	sample
	prot-b	33,554,432	doubled sample
English	bible	4,047,392	King James bible
	howto	39,422,105	Linux howto files
	mozilla	51,220,480	Mozilla source code

Table 3.1: Description of the strings used in experiments

These files are of five main types:

- highly periodic strings – strings that do not occur often in practice, containing many repetitions (Fibonacci strings, binary strings constructed in [35]);

- strings with very few runs (random strings on small and fairly large alphabets).
- DNA strings on alphabet $\{a, c, g, t\}$;
- protein sequences on an alphabet of 20 letters;
- strings on large alphabets (English-language, ASCII characters).

Test Results

We give in Table 3.2 the preprocessing times for the various data structures required by the RPT1, RPT3, and [67] algorithms; specifically, the SA, LCP, BWT, and LAST arrays.

Test results of algorithms are shown in Table 3.3 and 3.4. In Table 3.3, times for the RPT1 implementations and [67] include times required for SA, LCP, and BWT computation; RPT2 times include time for QSA initialization. In Table 3.4, times for the RPT3–1 include times required for SA, LCP, and BWT computation; while RPT3–1 include times required for SA, LCP, and LAST computation.

In both Table 3.3 and 3.4, averages within file type are not weighted by file size, and the final AVERAGE is a simple average of the "microseconds per letter" ratios for each of the 14 test files. Note that for each program tested, the number of microseconds per letter is generally stable within each file type and not highly variable overall. Tests of RPT1 used $p_{min} = 1$; as expected, for larger p_{min} the run time decreased, usually to about half the starting value.

File	SA	LCP	BWT	LAST
fibo35	0.898	0.142	0.025	0.031
fibo36	0.886	0.601	0.027	0.033
fss9	0.826	0.561	0.026	0.031
fss10	0.958	0.576	0.025	0.032
periodic AVG	0.892	0.470	0.026	0.032
rand2	0.947	0.144	0.026	0.031
rand21	1.135	0.112	0.025	0.031
random AVG	1.041	0.128	0.025	0.031
ecoli	1.413	0.116	0.025	0.031
chr22	1.635	0.146	0.035	0.040
chr19	1.873	0.160	0.044	0.053
DNA AVG	1.754	0.141	0.035	0.041
prot-a	1.778	0.142	0.027	0.032
prot-b	1.971	0.159	0.034	0.039
protein AVG	1.874	0.151	0.030	0.036
bible	1.417	0.111	0.024	0.030
howto	1.912	0.178	0.035	0.039
mozilla	1.815	0.135	0.032	0.036
English AVG	1.417	0.141	0.030	0.035
AVERAGE	1.390	0.235	0.029	0.035

Table 3.2: Microseconds per letter used by each run for SA, LCP, BWT, and LAST arrays

The vertical lines in Table 3.3 separates the algorithms RPT1 with RPT2 and the algorithm of [67]. In each section of both Table 3.3 and 3.4, we underline in bold the quantity that achieves the best result for the current test case.

To assess the effect of stack use on the space requirements of the RPT1 family, a record was kept of maximum stack usage for each of the files tested, as shown in Table 3.5. The files `prot-a` and `prot-b` consumed by far the most stack space, totalling in each case a little less than 125K bytes. (The maximum size of the LB

File	RPT1–1	RPT1–2	RPT1–3	RPT1–4	RPT2	[67]
fibo35	1.119	1.084	1.084	**1.077**	–	1.126
fibo36	1.570	1.533	1.534	**1.526**	–	1.570
fss9	1.462	1.434	1.434	**1.427**	–	1.470
fss10	1.614	1.582	1.580	**1.572**	–	1.618
periodic AVG	1.442	1.409	1.408	**1.401**	–	1.446
rand2	**1.131**	1.133	1.134	1.134	1.688	1.169
rand21	**1.280**	1.281	1.282	1.284	2.352	1.285
random AVG	**1.205**	1.207	1.207	1.209	2.020	1.226
ecoli	**1.566**	1.569	1.569	1.569	26.715	1.585
chr22	**1.830**	1.832	1.832	1.832	18.422	1.863
chr19	**2.091**	2.093	2.099	2.093	33.938	2.128
DNA AVG	**1.943**	1.946	1.949	1.946	26.358	1.973
prot-a	**1.958**	1.959	1.960	1.960	65.479	1.975
prot-b	**2.176**	2.177	2.178	2.177	–	2.201
protein AVG	**2.067**	2.068	2.069	2.068	–	2.088
bible	1.568	1.569	1.569	**1.567**	8.239	1.601
howto	**2.140**	2.141	2.143	2.141	–	2.185
mozilla	**1.993**	**1.993**	1.995	1.995	–	2.014
English AVG	**1.602**	1.603	1.604	1.603	–	1.635
AVERAGE	1.678	1.670	1.671	**1.668**	–	1.699

Table 3.3: Microseconds per letter used by each run for RPT1, RPT2, and the algorithm of [67]

stack was as expected identical for all four RPT1 algorithms.)

3.4 Discussion

We make the following observations:

(1) Except RPT2, the other six new algorithms tested are very fast, especially on strings that arise in practice: even if SA were to execute 10 times faster, as it might if MP2 were used, still each algorithm would require 5% or less of the

File	RPT3–1	RPT3–2
fibo35	**1.070**	1.075
fibo36	**1.519**	1.524
fss9	**1.419**	1.422
fss10	**1.565**	1.571
periodic AVG	**1.394**	1.398
rand2	**1.127**	1.130
rand21	**1.282**	1.286
random AVG	**1.204**	1.208
ecoli	**1.564**	1.568
chr22	**1.826**	1.829
chr19	**2.087**	2.094
DNA AVG	**1.940**	1.944
prot-a	**1.957**	1.960
prot-b	**2.174**	2.177
protein AVG	**2.065**	2.069
bible	**1.562**	1.566
howto	**2.134**	2.137
mozilla	**1.990**	1.993
English AVG	**1.597**	1.600
AVERAGE	**1.663**	1.667

Table 3.4: Microseconds per letter used by each run for RPT3 algorithms total SA/LCP time.

(2) From the 6th and the last line in Table 3.3, we observe that the RPT1–4 is behaving very well for highly periodic strings and is the fastest one of the four variants over the complete range of strings tested.

(3) Contrary to the averages given in the last line of Table 3.3, RPT1–1 and RPT1–2 are actually slightly faster than RPT1–3 and RPT1–4 on strings that arise in practice (that is, strings that are not highly periodic): the average microseconds

File	LB	PREVRANGES
fibo35	33	30
fibo36	34	32
fss9	33	36
fss10	37	40
random2	24	8
random21	8	5
ecoli	24	14
chr22	64	35
chr19	249	50
prot-a	6701	7448
prot-b	6701	7448
bible	23	112
howto	91	292
mozilla	2772	2750

Table 3.5: Maximum number of stack entries required by RPT1

per letter for RPT1–1 on such files is 1.773, the best of all.

(4) We have computed maximum stack size for each of the test files: only for `prot-a` and `prot-b` did the maximum size of the LB stack exceed three digits — for `prot-a` (the worst case) the total maximum storage for LB and PREVRANGES was 0.1% of the $5n$ bytes required for LCP and BWT storage.

(5) The algorithm of [67] was originally designed to provide more comprehensive output than that of our RPT1 algorithms, and so is not directly comparable. However, the authors have modified their code to reduce processing requirements. Using this version for Table 3.3, the modified version appears to execute around 2–3 times slower than RPT1 on real-world files.

(6) We conducted no experiments on the algorithm of [33] because it needs to compute SA/LCP twice and will therefore be very slow.

(7) Even though, in addition to its asymptotic advantage, RPT3–2 runs around 30% faster than RPT3–1, nevertheless it does not overcome the disadvantage of the additional preprocessing time required for LAST compared to BWT (see both Table 3.2 and 3.4): RPT3–1 together with BWT runs consistently slightly faster than RPT3–2 with LAST.

(8) RPT2 seems to provide possible advantage only for random strings.

The output of RPT1 and RPT2 can be used in various ways and for various purposes. For offline data compression the output can be used for phrase selection [4, 51, 83]. It is also useful for duplicate text/document detection [11]. If the user requires positions in $\boldsymbol{x}$ to be output, this can trivially be achieved, since SA is available, by postprocessing that replaces $i..j$ by $\mathrm{SA}[i], \mathrm{SA}[i+1], \ldots, \mathrm{SA}[j]$. In applications to protein sequences, such as the detection of low-complexity regions, the use of either RPT1 or RPT2 will provide significant algorithmic speed-up over currently-proposed methods [75] that are effective but slow. In the context of genome analysis the post-processing of interest may be to compute NE pairs as in [3, 12, 37]. Assuming an integer alphabet $1..\alpha$, this can be accomplished as follows for each range $i..j$. Introduce a new array $\mathrm{BWT}' = \mathrm{BWT}'[1..n]$, where for $\mathrm{SA}[h] < n$, $\mathrm{BWT}'[h] = \boldsymbol{x}[\mathrm{SA}[h]+1]$, otherwise $\mathrm{BWT}'[h] = \$$.

(1) Perform a radix sort on the pairs

$$(\mathrm{BWT}[i], \mathrm{BWT}'[i]),\ (\mathrm{BWT}[i+1], \mathrm{BWT}'[i+1]),\ \ldots,\ (\mathrm{BWT}[j], \mathrm{BWT}'[j])$$

into bins that are accessed from an array $\mathrm{B} = \mathrm{B}[1..\alpha, 1..\alpha]$. As a byproduct of the sort, positions in a Boolean array $\mathrm{E} = \mathrm{E}[1..\alpha]$ are set: $\mathrm{E}[b]$ = `TRUE` if and only if row b of B is empty.

(2) For every nonempty row b_1 of B, and for every $b_2 \in 1..\alpha$, perform the following simple processing:

for $h_1 \leftarrow b_1 + 1$ to α **do**

  if *not* $\mathrm{E}[h_1]$ **then**

    for $h_2 \leftarrow (1$ to $b_2 - 1)$ and $(b_2 + 1$ to $\alpha)$ **do**

      output all pairs $\mathrm{B}(b_1, b_2)$ with $\mathrm{B}(h_1, h_2)$

This approach requires checking at most $\alpha^2(\alpha-1)^2/2$ positions in B for each range processed; in the DNA case with $\alpha = 4$, this amounts to at most 72 (that is, $\alpha^3+2\alpha$) positions, but will for most ranges be much less. Otherwise the time required is proportional to the number of pairs output. Due to CPU cache effects, we believe this will be an efficient algorithm for computing NE pairs: it depends only on i, j, BWT and BWT'.

Chapter 4

Multirepeats

4.1 Introduction

In this chapter, we propose efficient algorithms for finding the nonextendible (NE) multirepeats in a set of strings under various constraints. The problem of finding common regularities among a set of strings is very important [37]. In biological sequences (DNA, RNA, or protein) the problem of locating repeats in a set of strings (*multiple repeats*, or *multirepeats*) arises in many contexts, such as database searching and sequence alignment [6]. It is also important in data mining [32, 50].

In Chapter 3, we discussed several existing algorithms for computing different kinds of NE repeats under some restrictions, and we proposed several fast algorithms for similar problems. But they are all only for a single string. To compute repeats in a set of strings (multirepeats), there exists only one algorithm [6]. This algorithm is not space efficient since it uses suffix trees, one for each string in the set plus a "generalized" suffix tree for all of them. Thus it is not easy to implement. In addition,

it has high time complexity.

If gaps (for definition, see Section 2.2.2) are unrestricted, the algorithm of [6] requires $O(\alpha N^2 n + \eta)$ time; If gaps are required to fall in a range of length c, it requires $O\big((c^2+\alpha^2)mN^2 n \log(Nn) + \eta\big)$ time. Here α is the alphabet size, N the number of strings, n the average length of the N strings, m the *multiplicity* (number of occurrences) of the multirepeat, and η the total number of occurrences of all reported repeats. While n may be quite large (millions), in applications N is generally a small integer (at most two digits). Similarly, we may suppose that the number R of reported repeats is $o(n)$.

In this chapter, following [39], we extend our work in Chapter 3 to the problems considered in [6], proposing algorithms that are more time efficient, as well as being easier to implement and using much less space. We describe algorithms to find complete NE multirepeats that occur at least m_{min} times in each of at least q strings in a given set S of N strings, first with no restriction on gap length, then with bounded gaps. For the first problem, we propose two algorithms with worst-case time complexities $O(Nn + \eta \log_2 N)$ and $O(Nn + \eta)$ that use $9Nn$ and $10Nn$ bytes of space, respectively. For the second problem, we describe an algorithm with worst-case time complexity $O(RNn)$ that requires approximately $10Nn$ bytes. Note that all times are independent of alphabet size. Extending the algorithms of [6], our three algorithms output only repeats whose occurrences are substrings of length at least p_{min}

(user-specified), thus eliminating trivial outputs.

The remainder of the chapter is organized as follows. In Section 4.2, we formulate the problems. In Section 4.3, we give details of the three algorithms noted above. Finally, in Section 4.4 we give the conclusions.

4.2 Formulation of Problems

We define two problems:

Unconstrained Multirepeats (abbreviated *MultiRep*):

Given a set $S = \{s_1, s_2, ..., s_N\}$ of strings, where each string s_k, $1 \leq k \leq N$, has length n (if the lengths of the strings vary, n represents their average length), and a tuple of positive integers $D = (p_{min}, q, m_{min})$, where $p_{min} \geq 1$, $q \in 1..N$, $m_{min} \geq 2$, we output all NE multirepeats of period at least p_{min} that occur at least m_{min} times in each of at least q strings of S. Following [6], we call q the *quorum* and m_{min} the *minimum multiplicity*.

Example 1 (see Figure 4.1): Given a set of three strings $S = \{s_1, s_2, s_3\}$, with $D = (3, 2, 2)$, we find an NE repeat ACG of length $p_{min} = 3$ that occurs at least $m_{min} = 2$ times in all $3 \geq q = 2$ of the strings. Thus the repeat would be output. However, for $D = (3, 3, 3)$, s_3 would not satisfy $m \geq 3$ and so only $2 < q = 3$ of the strings would have the minimum number of occurrences; in this case no output would occur.

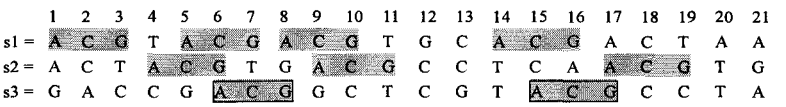

Figure 4.1: Multirepeats without constrained gaps

Our second problem considers restrictions on the gaps as follows: if for $i \in 1..\mu-1$, where $\mu = m_{min}$, g_i is the gap between the ith and $(i+1)$th occurrences of $\boldsymbol{u}$, then we require $d_{min_i} \leq g_i \leq d_{max_i}$, lower and upper bounds on g_i. Collectively, these restrictions are represented by a $(\mu-1)$-tuple

$$d = \big((d_{min_1}, d_{max_1}), (d_{min_2}, d_{max_2}), \ldots, (d_{min_{\mu-1}}, d_{max_{\mu-1}})\big). \tag{4.2.1}$$

Multirepeats with Constrained Gaps (abbreviated *MultiRepG*):
In addition to S and D, we are given a tuple of gap constraints (4.2.1). We compute all the repeats that satisfy (4.2.1) at least q times as well as the constraints D. More precisely, in each individual string $\boldsymbol{s}_k \in S$ that contains $m \geq m_{min}$ occurrences of the repeating substring, we look for a sequence of $\mu = m_{min}$ consecutive occurrences that satisfies (4.2.1); if such a sequence exists in at least q strings, we output all m occurrences in every $\boldsymbol{s}_k$ for which (4.2.1) is satisfied.

Example 2 (see Figure 4.2): Given the same set S and $D = (3, 2, 2)$ as in Example 1, we introduce the constraint $(d_{min_i}, d_{max_i}) = (0, 5)$ for every $i \in 1..\mu-1$. Because the gap between $\boldsymbol{s}_3[6..8]$ and $\boldsymbol{s}_3[15..17]$ exceeds 5, ACG does not satisfy the gap

constraints in s_3, but continues to do so in s_1 and s_2, thus at least $q = 2$ times. Thus occurrences of ACG only in s_1 and s_2 are output.

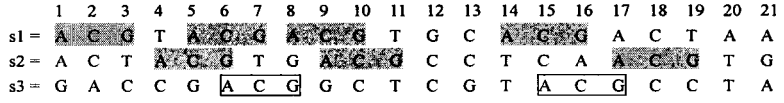

Figure 4.2: Multirepeats with constrained gaps

4.3 Description of the Algorithms

The overall strategy for both problems MultiRep and MultiRepG is the same:

- form a single string s from the given set S of N strings;
- in a preprocessing phase, compute the suffix array SA, the longest common prefix array LCP and the Burrows-Wheeler transform BWT for s;
- use Algorithm RPT1 introduced in Chapter 3 to compute all NE repeats of period $p \geq p_{min}$ in s;
- output the repeats that satisfy D (MultiRep) and both D and d (MultiRepG).

4.3.1 No Constraints on Gaps

Algorithm MultiRep-1

From the set $S = \{s_1, s_2, \ldots, s_N\}$ of strings, form $s = s_1\$_1 s_2\$_2 s_3\$_3 ... \$_{N-1} s_N\$$, as shown in Figure 4.3, where the end-of-string sentinels $\$_j$, $1 \leq j \leq N-1$, and $\$$

are distinct symbols less in lexicographic order than any of the letters in the s_k, $1 \leq k \leq N$, and that moreover satisfy $\$ < \$_1 < \$_2 < ... < \$_{N-1}$. Let $s_k = s_k[1..n_k]$.

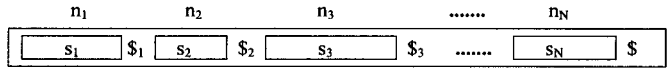

Figure 4.3: Form a new string using end-of-string sentinels

The preprocessing computes the SA, LCP and BWT arrays for s using standard algorithms as described in Section 2.3 of Chapter 2: in these algorithms the $\$_j$ are treated as normal letters, while \$ just marks the end of s and is not included in calculations.

Example 3 (see Figure 4.4): Given $S = \{s_1, s_2, s_3\}$, where s_1 = AAGTCAG, s_2 = AGAG, s_3 = CAGTAGC, we form $s = s_1\$_1 s_2\$_2 s_3\$$ and preprocess.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
s	A	A	G	T	C	A	G	$\$_1$	A	G	A	G	$\$_2$	C	A	G	T	A	G	C	$
SA	8	13	1	6	11	9	18	15	2	20	5	14	7	12	10	19	16	3	17	4	
LCP	-1	0	0	1	2	2	2	2	3	0	1	3	0	1	1	1	1	2	0	1	-1
BWT	G	G	$	C	G	$\$_1$	T	C	A	G	T	$\$_2$	A	A	A	A	A	A	G	G	$

Figure 4.4: Form a new string and compute its SA, LCP, and BWT arrays

RPT1 makes use of the preprocessed arrays to compute NE repeats, each one a triple $(p; i, j)$ specifying a period $p \geq p_{min}$ and a range $i..j$ in SA such that for every $h \in i..j$, suffix $SA[h]$ has an identical prefix of length p, while suffixes $SA[i-1]$ and

$SA[j+1]$ (if they exist) do not. If we are given $p_{min} = 2$ in Example 3, RPT1 would output only one NE repeat for $\boldsymbol{u} = \text{AG}$ in the form $(p; i, j) = (2; 4, 9)$ with period $p = 2$, where the range 4..9 identifies $\text{SA}[4] = 6, \text{SA}[5] = 11, \text{SA}[6] = 9, \text{SA}[7] = 18, \text{SA}[8] = 15, \text{SA}[9] = 2$. Thus the NE repeat occurs in positions $6, 11, 9, 18, 15, 2$ of $\boldsymbol{s}$ as shown by the shading in Example 3.

Given an output $(p; i, j)$ from RPT1, we need to determine if the conditions specified by the tuple D are satisfied. Our first task is to use the suffix array SA to convert this output into the form $M = \big(p; \text{SA}[i], \text{SA}[i+1], \ldots, \text{SA}[j]\big)$ keyed to positions in $\boldsymbol{s}$ rather than SA: over all repeats found by RPT1, this will require $O(\eta)$ time. We then make use of two arrays, *divpts* and *count*. Array *divpts* specifies the starting points of each substring $\boldsymbol{s}_k$ of $\boldsymbol{s}$ — this permits a binary search to be done to determine in which substring $\boldsymbol{s}_k$ the current repeating substring is located. More precisely:

$$divpts[1..N+1] = [1, n_1+2, n_1+n_2+3, \ldots, \sum_{k=1}^{N-1}(n_k+1)+1, \sum_{k=1}^{N}(n_k+1)+1].$$

The array $count = count[1..N]$ just maintains a count of the number of repeating substrings that have so far been found to lie within each of the N strings $\boldsymbol{s}_k$.

Using these arrays, it is straightforward to determine in time $O\big((j-i)\log N\big)$ whether the repeat $(p; i, j)$ occurs at least m_{min} times in each of at least q substrings of $\boldsymbol{s}$, as shown in MultiRep-1 (see Figure 4.5). Note that if $j-i+1 < m_{min}q$, this condition cannot be satisfied and so no tests are required. The function BinarySearch returns the index k indicating that position $\text{SA}[h]$ in $\boldsymbol{s}$ occurs in substring $\boldsymbol{s}_k$.

Input: an NE multirepeat $M = (p; \mathrm{SA}[i], \mathrm{SA}[i+1], \ldots, \mathrm{SA}[j])$ of $\boldsymbol{s}$,
together with integers $m_{min} \geq 2$, $q \geq 1$.
Output: M if and only if its repeating substring occurs
at least m_{min} times in each of at least q substrings of $\boldsymbol{s}$.
— *Preprocessing: compute* $divpts[1..N+1]$.
$r \leftarrow j - i + 1$
if $r \geq qm_{min}$ **then**
  $count[1..N] \leftarrow 0^N$; $qtotal \leftarrow 0$
  for $h \leftarrow i$ **to** j
    $k \leftarrow \mathrm{BinarySearch}(divpts, \mathrm{SA}[h])$
    $count[k] \leftarrow count[k] + 1$
    if $count[k] = m_{min}$ **then**
      $qtotal \leftarrow qtotal + 1$
  if $qtotal \geq q$ **then**
    output(M)

Figure 4.5: Algorithm MultiRep-1: check multiplicity & quorum

In Example 3, *divpts* will be $[1, 9, 14, 22]$ and the output repeat will be $(2; 6, 11, 9, 18, 15, 2)$. After binary search we find that $count[1] = 2$ (positions 6 and 2), $count[2] = 2$ (11 and 9), and $count[3] = 2$ (18 and 15): the repeat occurs at least twice in each of the three substrings. Thus for $m_{min} = 2, q = 3$, the repeat satisfies the constraints specified by D.

Now we analyze the time and space complexity of the algorithm MultiRep-1. For construction of SA there are algorithms linear in string length ℓ [45, 49], though in practice algorithms with worst-case $O(\ell^2 \log \ell)$ time requirement are several times faster [71]. To compute LCP from SA there are several linear time algorithms [47, 64], and the easy calculation of BWT from SA is also linear. Given LCP and BWT, RPT1 executes in linear time (Chapter 3). In our case $\ell = N(n+1)$, and so all the repeats

$(p; i, j)$ in $\boldsymbol{s}$ can be computed in time $O(Nn)$. For each of $O(R)$ repeats, the array *count* must be cleared at a cost of $O(N)$ time. In addition, for each of at most η occurrences of repeating substrings in $\boldsymbol{s}$, the time required is at most $O(\log_2 N)$ for the binary search. Thus to compute all the repeats satisfying constraint D, the worst-case time complexity is $O(Nn+RN+\eta \log_2 N)$.

However, the asymptotic time complexity of MultiRep-1, though not perhaps the expected running time in practice, can be slightly reduced, as we now explain. Instead of performing $count \leftarrow 0^N$ as part of the algorithm, execute it only once as preprocessing over all invocations of MultiRep-1. Introduce into MultiRep-1 a list L, initially empty, to which each value k computed by BinarySearch is added; then at the end of MultiRep-1 introduce a new loop that removes from L each entry k and performs $count[k] \leftarrow 0$. The resulting algorithm executes in time $O(Nn+\eta \log_2 N)$, independent of R.

Preprocessing for a string of length ℓ requires as few as 5ℓ bytes for SA [71], 6ℓ for LCP [74] and 6ℓ for BWT, thus at most $6N(n+1)$ bytes for $\ell = N(n+1)$. RPT1 itself requires only 5ℓ bytes for its execution, plus a further 4ℓ for storage of SA (since each range $i..j$ in SA needs to be converted into a sequence $\mathrm{SA}[i], \mathrm{SA}[i+1], \ldots, \mathrm{SA}[j]$ in $\boldsymbol{s}$). Since *divpts* and *count* are arrays $1..N$ of integer, their total space requirement is $8N$ bytes, and so the total is $N(9n+17)$ bytes, in simple terms $9Nn$.

The algorithm MultiRep-1 outputs all of the repeats M. It may instead be required

to output only those positions in M that occur in the $\boldsymbol{s}_k$ for which $m \geq m_{min}$. One way to accomplish this is to introduce a Boolean array $mok = mok[1..N]$ (similar in its role to the array *gapsok* described below for MultiRepG) — mok records for each $k \in 1..N$ whether or not $\boldsymbol{s}_k$ contains at least m_{min} occurrences of M. Then a straightforward processing of M, again using BinarySearch, produces the required output, using the same asymptotic time and space.

Algorithm MultiRep-2

We briefly describe a strategy to avoid the binary search of MultiRep-1, at a cost of an additional $N(n+1)$ bytes of storage (based on the assumption that N is small — less than 256). In the preprocessing stage we introduce an array *pos* of byte such that, for each $i \in 1..N(n+1)$, $pos[i] = k$ iff i is a position in $\boldsymbol{s}_k$, while otherwise $pos[i] = 0$ ($\boldsymbol{s}[i]$ is a sentinel). Thus for every i, $pos[i] \in 0..N$. Example 3 augmented with *pos* is

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
s	A	A	G	T	C	A	G	$1	A	G	A	G	$2	C	A	G	T	A	G	C	$
sa	8	13	1	6	11	9	18	15	2	20	5	14	7	12	10	19	16	3	17	4	
lcp	-1	0	0	1	2	2	2	2	3	0	1	3	0	1	1	1	1	2	0	1	-1
pos	0	0	1	1	2	2	3	3	1	3	1	3	1	2	2	3	3	1	3	1	
bwt	G	G	$	C	G	$1	T	C	A	G	T	$2	A	A	A	A	A	A	G	G	$

Figure 4.6: Compute *pos* array

shown in Figure 4.6. Using *divpts*, *pos* can easily be computed in $\Theta(Nn)$ time. Then, in order to determine, for each position h in SA which substring $\boldsymbol{s}_k$ the position SA[h] occurs in, it is necessary only to compute $k \leftarrow pos[\text{SA}[h]]$. This $O(1)$ computation

replaces BinarySearch in MultiRep-1, reducing processing time to $O(Nn+\eta)$, thus asymptotically optimal. The algorithm MultiRep-2 is shown in Figure 4.7.

Input: an NE multirepeat $M = (p; \mathrm{SA}[i], \mathrm{SA}[i+1], \ldots, \mathrm{SA}[j])$ of $\boldsymbol{s}$,
together with integers $m_{min} \geq 2$, $q \geq 1$.
Output: M if and only if its repeating substring occurs
at least m_{min} times in each of at least q substrings of $\boldsymbol{s}$.
— *Preprocessing: compute divpts*$[1..N+1]$, *and then pos array of* $\boldsymbol{s}$.
$r \leftarrow j - i + 1$
if $r \geq qm_{min}$ **then**
  $count[1..N] \leftarrow 0^N$; $qtotal \leftarrow 0$
  for $h \leftarrow i$ **to** j
    if $pos[h] = k$ **then**
      $count[k] \leftarrow count[k] + 1$
      if $count[k] = m_{min}$ **then**
        $qtotal \leftarrow qtotal + 1$
  if $qtotal \geq q$ **then**
    output(M)

Figure 4.7: Algorithm MultiRep-2: check multiplicity & quorum

4.3.2 Restricted Gaps (MultiRepG)

In this section, we introduce the algorithm MultiRepG, for which the input is an NE multirepeat $(p; i, j)$ of $\boldsymbol{s}$ satisfying constraints $D = (p_{min}, q, m_{min})$ and the output consists of the elements of $(p; i, j)$ that satisfy the gap constraints d in at least q substrings $\boldsymbol{s}_k$ of $\boldsymbol{s}$.

In order to satisfy constraints d in addition to those specified by D, we need to introduce a bit vector $loc = loc[1..N(n+1)]$. In a single preprocessing stage every position in loc is set `FALSE` in time $O(Nn/w)$, where w is the computer word length, and the precondition $loc[h] =$ `FALSE` for all h is maintained thereafter. Then for *each*

NE repeat $(p; i, j)$ of period $p \geq p_{min}$, the positions $loc[\mathrm{SA}[h]], h = i, i+1, \ldots, j$, are set TRUE, so that a left-to-right scan of loc will yield in increasing order the positions of the repeating substrings in $\boldsymbol{s}$. Such a scan is shown in Figure 4.8, used to determine which of the substrings $\boldsymbol{s}_k$ in $\boldsymbol{s}$ satisfy the gap constraints. A Boolean array $gapsok = gapsok[1..N]$ is used to record the values $k \in 1..N$ for which $\boldsymbol{s}_k$ satisfies d.

```
 — Precondition: loc = FALSE^{N(n+1)}.
for h ← i to j do loc[sa[h]] ← TRUE
 — First Phase: Checking
q' ← 0; gapsok[1..N] ← FALSE^N
k ← 1; m ← 0; r ← j−i+1; r' ← 0; h ← 1
while r' < r do
    if loc[h] then
        r' ← r'+1
        if h < divpts[k+1] then m ← m+1; occ[m] ← h
        else
            if m ≥ m_min and check(p, occ, m, d, m_min) then
                q' ← q'+1; gapsok[k] ← TRUE
            m ← 1; occ[1] ← h
            repeat k ← k+1 until h < divpts[k+1]
    h ← h+1
if m ≥ m_min and check(p, occ, m, d, m_min) then
    q' ← q'+1; gapsok[k] ← TRUE
 — Second Phase: Output
if q' ≥ q then
    for k ← 1 to N do
        if gapsok[k] then
            m ← 0
            for h ← divpts[k] to divpts[k+1]−1 do
                m ← m+1; occ[m] ← h
            output(p, k, occ)
for h ← i to j do loc[sa[h]] ← FALSE
```

Figure 4.8: Algorithm MultiRepG: for each substring s_k of s, if occ contains a sequence of length $\mu = m_{min}$ that satisfies (4.2.1), then output occ

Algorithm MultiRepG executes in two phases, a checking phase and an output phase.

In the checking phase, *divpts* is used to compute for each s_k an array *occ* of candidate positions. The function *check*, described below, actually applies the constraints d to *occ* — its total time usage over all invocations is $O(r)$, where $r = j-i+1 < Nn$; also, the positions inspected in *divpts* and *gapsok* for each repeat are at most N. Thus for each candidate repeat, the time required to evaluate the constraints d is $O(Nn)$. For R such repeats, the overall time requirement of the first phase is therefore $O(RNn)$. We note that since $\eta \leq RNn$ (the total number η of occurrences of repeats cannot exceed the number R of repeats times the overall string length Nn), therefore $O(RNn)$ in fact represents the total time required both for MultiRep-2 and the checking phase. For cases that arise in practice, a corresponding statement holds also for MultiRep-1.

In the output phase, there is no action if less than q substrings of s contain repeats satisfying the constraints d. Otherwise, *occ* is recomputed for each s_k that satisfies d and the repeat is then output. For the strings and gap constraints of Example 2, described above, the output of the algorithm MultiRepG would be $(p, k, occ) = (3, 1; 1, 5, 8, 14)$ and $(3, 2; 4, 9, 17)$. The overall time requirement of the output phase is again $O(RNn)$.

The Boolean function *check*, shown in Figure 4.9, slides a window of width m_{min}

```
function check(p, occ, m, d, m_min) : boolean
    I_0 ← I ← 1
    while m−I ≥ m_min−1 do
        J ← 1
        while J < m_min and d[J,1] ≤ occ[I+1] − occ[I] − p ≤ d[J,2] do
            I ← I+1; J ← J+1
        if J = m_min then
            return TRUE
        else
            I_0 ← I ← I_0+1
    return FALSE
```

Figure 4.9: Function *check*: given an array *occ* of m occurrences of a repeating substring in s_k, determine whether *occ* contains a subarray of length $\mu = m_{min}$ that satisfies the constraints d

over the $m \geq m_{min}$ entries in *occ*, corresponding to the substring s_k, shifting right by one position at each step. For each window, *check* determines whether its entries satisfy the constraints d; if so, *check* returns TRUE, causing the m repeating substrings of *occ* that occur in s_k to be output. If no window of *occ* satisfies d, *check* returns FALSE. The constraints d are accessed as a two-dimensional array $d[1..m_{min}-1, 1..2]$. The outer *while* loop of *check* is executed $(m - m_{min} + 1)$ times in the worst case, and the inner *while* loop is executed at most m_{min} times; thus the execution time of *check* at each invocation is $O(m_{min}(m - m_{min} + 1)) = O(m)$. Here we assume that the specified input value m_{min} is constant over the execution of the algorithm. Over all invocations, therefore, the execution time of *check* is $O(r)$.

We note that the corresponding algorithm described in [6] requires that the differences between the maximum and minimum gaps specified in (4.2.1) should all be

bounded by a small constant c. The methodology described here requires no such bound, and its effectiveness does not depend on such differences. Note also that MultiRepG can easily be modified, with the same asymptotic complexity and usage of space, to output only those ranges of *occ* that satisfy d, omitting those entries that do not.

The additional storage required for MultiRepG consists of the $4Nn/w$ bytes for *loc* plus up to $4n$ bytes for the integer array *occ*, a total of $4n(N/w+1)$. For $w = 32$, this amounts to $n(N/8+4)$, perhaps as much as an additional Nn bytes on top of the $9Nn$ used by MultiRep-1.

Table 4.1 compares the algorithms described here with those proposed in [6].

Problem	Algorithm	Time	Space
MultiRep	[6]	$O(\alpha N^2 n + \eta)$	linear but large
	MultiRep-1	$O(Nn+\eta \log_2 N)$	$9Nn$ bytes
	MultiRep-2	$O(Nn+\eta)$	$10Nn$ bytes
MultiRepG	[6]	$O((c^2 + \alpha^2)mN^2 n \log(Nn)) + \eta)$	$O(c^2 Nnm)$
	MultiRepG	$O(RNn)$	$10Nn$ bytes

Table 4.1: Comparison of algorithms

Note that even though the suffix tree storage is linear, the large amount of information in each edge and node makes the suffix tree very expensive (see Section 2.3). In [6], the algorithm MultiRep uses suffix trees, one for each string in the set plus a "generalized" suffix tree for all of them, therefore the memory usage would be very large.

4.4 Discussion

We have formulated two problems related to multirepeats in sets of strings with various restrictions and presented efficient algorithms with lower time complexity and less memory consumption compared to previously proposed algorithms. We remark that if in Algorithm MultiRepG we set the *min* and *max* constraints on gaps equal to zero, we can find all tandem repeats (repetitions) in arbitrary subsets of S.

Chapter 5

LZ Factorization

5.1 Introduction

In this chapter, following [1], we discuss the main LZ factorization algorithms and provide both theoretical and practical comparisons of their time and space efficiency.

More than 30 years ago, in three very influential papers [56, 91, 92], Lempel and Ziv proposed methods for factoring a string $\boldsymbol{x}$ into substrings (factors) in such a way as to facilitate encoding the string into a compressed form (lossless text compression). The first two of these papers dealt with what is today generally called LZ77 factorization, the third with a variant called LZ78.

Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ be a string of length n on an ordered alphabet A of size α. In general terms, an LZ factorization of $\boldsymbol{x}$ is a decomposition of $\boldsymbol{x}$ into nonempty factors: $\boldsymbol{x} = \boldsymbol{w_1 w_2 \cdots w_k}$. There are numerous forms of LZ factorization; for our purposes we use the following definition:

Definition 5.1.1 (lz77) *A factorization of $\boldsymbol{x} = \boldsymbol{w_1 w_2 \cdots w_k}$ is **LZ** if and only if each $\boldsymbol{w_j}$, $j \in 1..k$, is*

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

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

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

$$\begin{array}{rcccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{x} = & a & b & a & a & b & a & a & b, \end{array} \tag{5.1.1}$$

$\boldsymbol{w_1} = a$, $\boldsymbol{w_2} = b$, $\boldsymbol{w_3} = a$, $\boldsymbol{w_4} = abaab$.

The factorization of $\boldsymbol{x}$ can be reported in many ways. In its original form, ***LZ77 factorization*** [91] reports each factor $\boldsymbol{w_j}$ as a triple $(\text{POS}, \text{LEN}, \lambda)$, where

- POS is the location of a previous occurrence of $\boldsymbol{w_j}$ in $\boldsymbol{x}$ or the location of $\boldsymbol{w_j}$ if no previous occurrence exists;
- LEN is the length (possibly zero) of the matching previous occurrence;
- λ is the "letter of mismatch": for $j < k$, $\lambda = \boldsymbol{x}[|\boldsymbol{w_1 w_2 \cdots w_{j-1}}| + \text{LEN} + 1]$, while for $j = k$, $\lambda = \$$, an arbitrary sentinel.

Thus LZ77 would report the factorization of (5.1.1) as

$$10a, 20b, 11a, 15\$.$$

It is noteworthy that this (in general, compressed) encoding of $\boldsymbol{x}$ permits the original string to be reconstituted (decoded) with no need for an explicit dictionary; more

precisely, the dictionary is dynamic, derived from the reported triples. Essentially, ***LZ78 factorization*** [92] removes LEN from the output, thus compressing the text further, but introducing the need for a separate dictionary in order to recover the original text.

For most of the last 30 years, LZ factorization has been used primarily for text compression, and many LZ variants have been proposed and computed, including factorization of infinite words [7]. Useful surveys are available at [28, 68, 90]. In the context of compression, LZ algorithms generally operate not on the string as a whole, but only on a sliding window of length N (usually $N = 4096$ or 8192), with a long prefix that has already been factored and a short (typically 18 letters) as-yet-unfactored suffix F: the next factor $\boldsymbol{w_j}$ is the longest prefix of F that matches a preceding substring within the window. Once $\boldsymbol{w_j}$ has been determined, the window is shifted right by $|\boldsymbol{w_j}|$ positions. It has been found that in practice the use of the sliding window provides compression as good as using the entire string would yield, and of course processing time is greatly reduced. Many sliding-window algorithms have been proposed, of which several are described in [9, 70, 79] and the surveys noted above. LZ is the basis of the `gzip` (Unix), `winzip` and `pkzip` compression techniques.

More recently LZ factorization has found application in the computation of various "regularities" in strings: repetitions [15], runs (maximal periodicities) [3, 17, 20, 23, 24, 43], repeats with fixed gap [44], branching repeats [78], sequence alignments [21],

and local periods [29]. For these applications, the factorization of the entire string, not merely a window, is required. On the other hand, the output can be simplified, since the given string $\boldsymbol{x}$ is available and does not need to be reconstituted. In particular, the letter λ is not required in the output, and so the triple $(\mathrm{POS}, \mathrm{LEN}, \lambda)$ can be reduced to a pair $(\mathrm{POS}, \mathrm{LEN})$. The output for (5.1.1) thus becomes

$$(\mathrm{POS}, \mathrm{LEN}) = (1,0), (2,0), (1,1), (1,5). \tag{5.1.2}$$

Indeed, further simplification is possible. For text compression it was important to be able to identify a position POS to the left of each factor $\boldsymbol{w_j}$ such that $\boldsymbol{x}[\mathrm{POS}..\mathrm{POS}+|\boldsymbol{w_j}|-1] = \boldsymbol{w_j}$. However, for computing regularities, it usually suffices only to ensure that each $\boldsymbol{w_j}$ does in fact provide a maximum-length match over all substrings to its left in $\boldsymbol{x}$. The position of match is not as a rule required. Thus it is necessary only to give the sequence of occurrences of each $\boldsymbol{w_j}$: the factorization $\boldsymbol{x} = \boldsymbol{w_1 w_2 \cdots w_k}$ is specified by a sequence

$$i_1, i_2, \ldots, i_k, \tag{5.1.3}$$

where for $j \in 1..k-1$, $\boldsymbol{w_j} = \boldsymbol{x}[i_j..i_{j+1}-1]$, while $\boldsymbol{w_k} = \boldsymbol{x}[i_k..n]$. For our example (5.1.1), instead of (5.1.2), the sequence $(1,2,3,4)$ is sufficient, together with $\boldsymbol{x}$, to identify all the LZ factors.

In Section 5.2 we give an overview of LZ algorithms that yield results suitable for the computation of regularities in strings. In addition we provide for each a characterization of asymptotic time and space efficiency. As we shall see, in the last

few years there have been many algorithms proposed, remarkable for their diversity of approach. Section 5.3 presents the results of experiments that compare the time and space requirements of the algorithms applied to a varied selection of strings. Section 5.4 summarizes our main conclusions.

5.2 Overview of LZ Algorithms

In this section we provide an overview of algorithms that compute the LZ factorization (as discussed above) of an entire string $\boldsymbol{x}$. For the data structures they used, see Section 2.3.

5.2.1 The LZ Algorithms

In this subsection we provide an overview of the main LZ algorithms currently available for the computation of regularities, especially maximal periodicities [58], in strings. Figure 5.1 shows schematically the various data structures used by these algorithms that we have denoted KK, AKO, CPS, CI, CIS, CII, and OS.

These algorithms all operate on the entire string $\boldsymbol{x}$ and produce outputs that are comparable, though in some cases differing slightly. In the figure we include for completeness a generic LZ77 algorithm used for text compression: as mentioned in the introduction, there are actually several variants of this approach (for example, [9, 70, 79]), all using a sliding window typically of length $N = 4096$ with an as-yet-unfactored suffix typically of length $F = 18$. When scaled up so as to be useful for

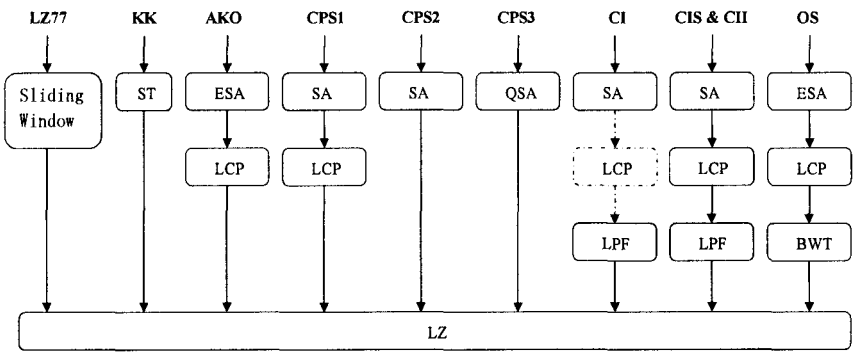

Figure 5.1: The steps and main data structures used by each algorithm

regularities ($N = n$, the length of $\boldsymbol{x}$, and F equal to the full length of the unfactored suffix), these algorithms become uncompetitive. We therefore provide no further description of them; the other algorithms are outlined below.

Algorithm KK

At [42] Kolpakov & Kucherov make available an implementation of Ukkonen's ST algorithm that computes LZ on-line for small alphabets ($\alpha \leq 4$). For an overview, see [77, pp. 175–178]. Even though approaches based on suffix trees normally require more time, and particularly more space, than suffix array approaches, nevertheless the KK algorithm is consistently faster [24] applied to DNA files ($\alpha = 4$) than other LZ algorithms, as well as being highly space efficient. On the other hand, for $\alpha = 2$, it appears not to be competitive [24]. We therefore do not test this algorithm further. Asymptotically, the KK implementation uses $\Theta(n)$ time and space.

Algorithm AKO

In [3] Abouelhoda, Kurtz & Ohlebusch show how to compute LZ using an ***enhanced suffix array*** (ESA): a suffix array SA that is augmented by an ***lcp-interval tree*** consisting of the internal nodes I of the corresponding ST additionally labelled with a range $lb..rb$ of positions in SA. This range specifies in left-to-right order the leaf nodes in the subtree rooted at I: SA[h], $h = lb, lb{+}1, \ldots, rb$. Here lb and rb stand for left and right boundary, respectively, and each internal node I is called an ***lcp-interval.*** Thus the label of each lcp-interval takes the form $(lcp; lb, rb)$. For our example (5.1.1), the internal node in Figure 2.2 labelled 1 that corresponds to a would be labelled $(1; 1, 5)$, since the leaf nodes are

$$\text{SA}[1] = 6,\ \text{SA}[2] = 3,\ \text{SA}[3] = 7,\ \text{SA}[4] = 4,\ \text{SA}[5] = 1.$$

For the LZ calculation, the lcp-interval tree is virtual — not actually implemented as a tree. In fact the lcp-intervals in the tree are computed with the help of a stack, the elements of which are lcp-intervals represented by tuples $\langle lcp, lb, rb, childList, min\rangle$, where lcp, lb and rb are as defined above, $childList$ is a list of the current lcp-interval's child intervals, and min is the minimum of the leaf nodes

$$\text{SA}[lb], \text{SA}[lb{+}1], \ldots, \text{SA}[rb].$$

This stack is used to simulate a bottom-up traversal of ST that generates all lcp-intervals (all internal nodes I). Every time an lcp-interval is generated, a procedure

is called to update the POS and LEN arrays as exemplified in (5.1.2). Thus the factorization can be computed in linear time and space.

The AKO algorithm was the first to compute the LZ factorization based only on SA and LCP.

The CPS1 Algorithms

In [23, 24] Chen, Puglisi & Smyth describe a collection of algorithms CPS1 for LZ factorization, based on SA/LCP computation, that execute consistently faster and with lower space requirements than AKO.

The execution of CPS1 is based on three pointers i_1, i_2, i_3 to positions in SA for which the invariant $1 \leq i_1 < i_2 < i_3 \leq n+1$ is maintained:

- i_1 marks the leftmost position in SA of a sequence of suffixes with lcp of length at least LCP$[i_1+1]$; each such value i_1 is pushed onto a stack S.
- i_2 marks the end of at least one sequence of suffixes sharing the same lcp; S is then popped until a stacked position i is found for which LCP$[i] \leq$ LCP$[i_2+1]$.
- i_3 marks the next position to the right of i_2 that has not yet been processed.

The basic algorithm CPS1–1 locates, in a left-to-right traversal of SA, a next position $i_2 > i_1$ such that LCP$[i_2] >$ LCP$[i_3]$ for some least $i_3 > i_2$; it then backtracks (using the stack S) from i_2, setting POS$[p_2] \leftarrow p1$ or POS$[p_1] \leftarrow p2$ depending on whether

$p_1 = \mathrm{SA}[i_1] < \mathrm{SA}[i_2] = p_2$ or not, until the LCP value for the position i_1 popped from S falls below $\mathrm{LCP}[i_2]$.

It turns out that none of the position pointers i_1, i_2, i_3 will ever point to any position i in SA such that $\mathrm{POS}[\mathrm{SA}[i]]$ has been previously set. This observation allows the storage for SA and LCP to be dynamically reused so as to specify the location and contents of the array POS, thus saving $4n$ bytes of storage. POS can then be computed by a straightforward in-place compactification of SA and LCP into LCP (now redefined as POS). This second algorithm is called CPS1–2.

But more storage can be saved by removing all reference to LEN from CPS1–2, so that it computes only POS and in particular allocates no storage for LEN. This third version is called CPS1–3.

Since at least one position in POS is set at each stage of the main **while** loop, it follows that the execution time of CPS is linear in n. For CPS1–1 space requirements total $17n$ bytes (for $\boldsymbol{x}$, SA, LCP, POS & LEN) plus a stack of maximum size $4s$ bytes — at most the maximum depth of ST. For $\boldsymbol{x} = a^n$, $s = n$, but in the expected case, $s \in O(\log_\alpha n)$ [46]; for strings that arise in practice, s is usually negligible and seems never to exceed $n/5$. For CPS1–2 the space required is $13n$ bytes plus stack. CPS1–3 gives rise to two variants:

- CPS1–3a that is slightly the faster of the two while using exactly $13n$ bytes of storage including stack;

- CPS1–3b that computes LEN[i] only on demand, without the need to store a LEN array, and that therefore uses only $9n$ bytes plus stack.

For large strings, saving space will generally be more important, and so in such cases CPS1–3b would likely be the algorithm of choice among all the CPS1 algorithms, even though it is the slowest by 20% or so compared with the fastest. To give the range of capability for the CPS1 algorithms, in Section 5.3 we test the fastest, CPS1–2, using $13n$ bytes plus stack, and the slowest, CPS1–3b, using $9n$ bytes plus stack.

All variants of CPS1 require $\Theta(n)$ time in the worst case, and, like AKO, all produce output (POS, LEN) of type (5.1.2) that is usable for text compression as well as for the computation of regularities.

Algorithms CPS2 & CPS3

In [24] two other LZ algorithms, CPS2 and CPS3, are also described. Both of these algorithms are supralinear in asymptotic worst-case execution time, and in practice often (though not always) execute considerably more slowly than the CPS1 algorithms. On the other other hand, both CPS2 and CPS3 use much less storage space than any of the CPS1 algorithms, comparable in fact to Algorithm OS (see below). We therefore provide brief descriptions of these algorithms here.

As shown in Figure 5.2, CPS2 is a very simple algorithm that makes use of a function `lzfactor` to compute the position and length of the LZ factor beginning at the current position i in $\boldsymbol{x}$. For its execution it requires only $\boldsymbol{x}$ and SA together with a

data structure $\mathrm{RMQ}_{\mathrm{SA}}$ that supports range minimum queries **rmq** on SA: in constant time and $O(n)$ space [31], $\mathrm{SA}[\mathrm{RMQ}_{SA}(lb, rb)]$ uses **rmq** to compute the minimum of $\mathrm{SA}[lb..rb]$ (a least position in $\boldsymbol{x}$). At a cost of $O(\log n)$ time per calculation, the minimum is repeatedly computed for narrower and narrower ranges $lb..rb$ until the longest substring beginning at i is identified that matches a previous position in $\boldsymbol{x}$.

output $(1, 1)$
$i \leftarrow 2$
while $i \leq n$ **do**
    $(\mathrm{POS}, \mathrm{LEN}) \leftarrow$ `lzfactor`$(\boldsymbol{x}, \mathrm{SA}, i)$
    output $(\mathrm{POS}, \mathrm{LEN})$
    $i \leftarrow i + \mathrm{LEN}$

Figure 5.2: Algorithm CPS2

CPS3 is another simple algorithm that combines the idea of a QSA with that of a q-gram [84] — that is, a substring of length q. In a preprocessing phase, CPS3 builds a kind of QSA, called QSA_q, in which only matches between substrings of length at most q are considered at positions i and $\mathrm{QSA}_q[i]$ of $\boldsymbol{x}$. In addition to $4n$ bytes for QSA_q, the preprocessing uses $4\alpha^q$ bytes, and so the parameter q is chosen so that $4\alpha^q \approx n$; that is, $q \approx \log_\alpha n - \log_\alpha 4$. QSA_q can be thought of as an inverted file for $\boldsymbol{x}$ based on t-grams, $t \in 1..q$. At positions i where an LZ factor begins, CPS3 extends the partial LPF information provided by QSA_q to determine the correct $(\mathrm{POS}, \mathrm{LEN})$ values required by the LZ factorization.

The CI Algorithms

In [17] Crochemore & Ilie describe two algorithms to compute LPF, then give the simple pseudocode shown in Figure 5.3 to compute LZ from LPF.

```
LZ[1] ← 1; i ← 1
while LZ[i] < n do
    LZ[i+1] ← LZ[i]+max(1, LPF[LZ[i]])
    i ← i+1
return LZ
```

Figure 5.3: Given LPF for a string $\boldsymbol{x}$, compute LZ

This pseudocode outputs LZ in the form (5.1.3) not suitable for compression, but the CI algorithms, as well as the CIS and CII algorithms described below, can all be easily modified to output an array called PrevOcc, identical to the QSA array defined in Section 2.3. Using QSA and slightly modified processing, LZ can also be computed from LPF in the form (5.1.2). Regardless of the form in which LZ is provided, no extra space needs to be allocated for the LZ computation by these algorithms, because arrays used during the LPF phase can be deallocated and reused.

Both CI algorithms compute LPF based on prior computation of SA. The basic idea of both is to search in the neighbourhood (to both left and right) of each position j in SA, where $i = \mathrm{SA}[j]$, in order to compute a position $\mathrm{QSA}[i] = i^* < i$ in $\boldsymbol{x}$ such that

$$\mathrm{lcp}(i^*, i) = \max_{i' < i} \mathrm{lcp}(i', i).$$

The first algorithm, CI1, makes use of two $1..n$ arrays prev_L and prev_R that for

every position j in SA identify the nearest position in SA, to left and to right of j, respectively, such that $\mathrm{SA}[prev_L] < \mathrm{SA}[j]$ and $\mathrm{SA}[prev_R] < \mathrm{SA}[j]$. After the calculation of SA, prev_L and prev_R are computed from SA in a $\Theta(n)$-time preprocessing stage; CI1 itself executes in $\Theta(n)$ time and requires $13n$ bytes of storage (LPF, prev_L, prev_R, $\boldsymbol{x}$). In the pseudocode given in [17], CI1 actually uses two additional arrays, but these are not necessary.

The second algorithm, CI2, instead uses SA and LCP directly to search the neighbourhoods of positions j in SA. Also required for this search is a stack S containing entries (j, LEN), where j is a position in SA and LEN is the lcp of suffix $\boldsymbol{x}[\mathrm{SA}[j]..n]$ and suffix $\boldsymbol{x}[\mathrm{SA}[j']..n]$, where j' is the position for the preceding entry in S. CI2 controls pushes and pops to S so as to compute LPF. CI2 executes in $\Theta(n)$ time using $12n$ bytes (LPF, SA, LCP), but it also requires space for S. Like the stack for CPS1–1, S may in the worst case ($\boldsymbol{x} = a^{n-1}b$) contain n entries (thus $8n$ bytes), but the expected size is only $16 \log_\alpha n$ bytes, since the expected maximum LPF value is $2 \log_\alpha n$ [46]. For $\alpha = 2$ and $n = 10^7$, the expected size of S is only 372 bytes.

Algorithm CIS

Like CI2, CIS computes LPF using SA and LCP in $\Theta(n)$ time, but it does not need to access $\boldsymbol{x}$; and even though CIS also uses a stack S, the entries are only single integers and there are at most $2\sqrt{2n}$ of them. Thus the worst-case space requirement of CIS is $12n+8\sqrt{2n}$ bytes. As for CI1 and CI2, LZ can also easily be computed in the form

(5.1.3) suitable for compression.

The basis of CIS is the observation that $\mathrm{LPF}[\mathrm{SA}[j]]$, $j = 1, 2, \ldots, n$, can be immediately computed in the following two cases (a third case symmetric to (b) does not arise in left-to-right processing of SA):

(a) if $\mathrm{SA}[j] > \max\big(\mathrm{SA}[j-1], \mathrm{SA}[j+1]\big)$, then

$$\mathrm{LPF}[\mathrm{SA}[j]] = \max\big(\mathrm{LCP}[j], \mathrm{LCP}[j+1]\big);$$

(b) if $\mathrm{SA}[j-1] < \mathrm{SA}[j] < \mathrm{SA}[j+1]$, then $\mathrm{LPF}[\mathrm{SA}[j]] = \mathrm{LCP}[j]$.

What CIS does is to scan SA left-to-right, applying case (a) or (b) whenever possible, and stacking for subsequent iterations entries $i = \mathrm{SA}[j]$ that do not immediately yield an LPF value.

Algorithm CII

We have noted above that the CI and CIS algorithms use very little more than the $12n$ bytes of storage required for input (SA and LCP) and output (LPF). However, in [19], Crochemore, Ilie, Iliopoulos et al. show how to compute LPF from SA and LCP using only constant additional space.

As shown in Figure 5.4, CII actually calls three LPF algorithms, each of which computes a part of the final LPF array:

- Algorithm CII-ON-LINE is a descendant of CIS that partially computes LPF

```
— Suppose n ≥ 8.
K ← ⌊n−2√(2n)⌋
procedure CII-ON-LINE (SA, LCP, K)
procedure CII-NAIVE (SA, LCP, K, n)
procedure CII-ANCHORED (SA, LCP, K, n)
return LPF
```

Figure 5.4: Three-stage calculation of LPF using only constant additional space

on-line with respect to suffix ordering in SA, while making use of unused positions in LPF to store the CIS stack values. As we have seen, for length n, the CIS stack contains at most $2\sqrt{2n}$ entries; thus $K = \lfloor n-2\sqrt{2n} \rfloor$ positions in LPF can be safely computed, leaving enough positions available in LPF to accommodate the stack. This algorithm computes values LPF[i], $i = \text{SA}[1], \text{SA}[2], \ldots, \text{SA}[K]$, in time $\Theta(K)$.

- For $i = \text{SA}[K+1], \text{SA}[K+2], \ldots, \text{SA}[n]$, Algorithm CII-NAIVE computes LPF[i], performing a straightforward search of LCP to the left and right of each position i in order to locate the corresponding LPF value. These searches are equivalent to computing the $prev_L$ and $prev_R$ values in Algorithm CI1. Each of these $n-K$ searches may require $O(n-K)$ time, yielding a total $O\big((n-K)^2\big) = O(n)$, since $n-K = 2\sqrt{2n}$.

- Since the previous two algorithms perform their processing independent of each other, it may be that some values LPF[i] may not be fully computed for positions $i = \text{SA}[j]$, where j is close to K. Thus CII-ANCHORED completes the

calculation by appropriate updates to these LPF values based on inspection of the LCP values to the left and right of each position. This simple algorithm also requires $O(n)$ time in the worst case.

The algorithm of Figure 5.4 is of theoretical interest, but slow in practice. Moreover, its space requirement is actually more than that of CPS1-3b (see Table 5.1) that computes LZ directly without LPF. Of more interest for testing, therefore, is the variant of CIS, Algorithm CIS-ON-LINE, executed using parameter n rather than K. This is the "CII" algorithm for which test results are reported in Tables 5.4 and 5.5, but in Table 5.1 we show properties of both the minimum-space and on-line versions.

Algorithm OS

In [69] Okanohara & Sadakane describe an LZ algorithm that is

- succinct — it stores SA in compressed form and uses **rank/select** operations [54] and range minimum queries **rmq** [31] to access and update array entries;
- on-line — it reads $\boldsymbol{x}$ a letter at a time and immediately reports the corresponding LZ value;
- compatible with implementation on a sliding window — because it is on-line, OS can adjust string "history" to any desired length n simply by removing at each step a prefix equal in length to the suffix added.

Algorithm OS executes in $O(n \log^3 n)$ time, using $n \log n + o(n \log \alpha) + O(n)$ bits of space, where n is the length of the string or of the sliding window.

OS maintains succinct representations of versions of ESA, LCP and BWT that are defined on the reverse $\overline{\boldsymbol{x}}$ of $\boldsymbol{x}[n_0..n]$, where $n_0 \geq 1$ is the selected lefthand position of the window, and n is the current position being processed on-line. Processing the reverse string facilitates update of the data structures, because the number of updates is $o(n)$. Figure 5.5 shows the modified arrays for our example string (5.1.1).

i	**SA**$[i]$	$\overline{\boldsymbol{x}}$[SA$[i]..\boldsymbol{n}$]	**LCP**$[i]$	**BWT**$[i]$
1	8	*a*	0	*b*
2	5	*aaba*	1	*b*
3	2	*aabaaba*	4	*b*
4	6	*aba*	1	*a*
5	3	*abaaba*	3	*a*
6	7	*ba*	0	*a*
7	4	*baaba*	2	*a*
8	1	*baabaaba*	5	\$

Figure 5.5: SA, LCP, and BWT arrays of the reversed string $\overline{\boldsymbol{x}} = baabaaba$

LCP is stored using only $2n$ bits [76], including a balanced search tree used for **rmq** operations in the LCP update; **rank/select** operations are used for the update of BWT [57].

5.2.2 Theoretical Comparison

In Table 5.1 we summarize some of the information given above for the LZ algorithms. The upper portion of the table covers the linear-time algorithms that generally have

larger space requirements; the lower portion displays supralinear algorithms that make use of "succinct" data structures to reduce space.

The space estimates are a mixture of theoretical calculations based on the data structures used and tests performed on a variety of standard files (see Section 5.3). A "+" after the space estimate indicates some allocation for stacks — except for certain AKO cases generally negligible or small (up to $2n$ additional bytes). The nature of the output refers to the forms mentioned earlier: specifying the previous occurrence of each factor $\boldsymbol{w_j}$ (5.1.2) or else giving only the current position (5.1.3).

Algorithm	Asymptotic Worst-Case Time	Space (bytes)	Output	Special Feature
KK	$\Theta(n)$	$\sim 11n$	(5.1.2)	$\alpha = 4$
AKO	$\Theta(n)$	$17n$+	(5.1.2)	
CPS1–1	$\Theta(n)$	$17n$+	(5.1.2)	
CPS1–2	$\Theta(n)$	$13n$+	(5.1.2)	
CPS1–3a	$\Theta(n)$	$13n$	(5.1.2)	
CPS1–3b	$\Theta(n)$	$9n$+	(5.1.2)	
CI1	$\Theta(n)$	$13n$	(5.1.3)	
CI2	$\Theta(n)$	$12n$+	(5.1.3)	
CIS	$\Theta(n)$	$12n$+	(5.1.3)	
CII	$\Theta(n)$	$12n$+	(5.1.3)	on-line
	$\Theta(n)$	$12n$	(5.1.3)	
CPS2	$O(n \log n)$	$6n$	(5.1.2)	
CPS3	$O(n^2)$	5–$7n$	(5.1.2)	
OS	$O(n \log^3 n)$	3–$7n$	(5.1.2)	on-line

Table 5.1: Theoretical comparison of LZ algorithms

5.3 Experimental Results

5.3.1 Implementation

We have tested nine algorithms – AKO, CPS1-2, CPS1-3b, CI1, CI2, CIS, CII (online); CPS2, CPS3 – that fall naturally into two groups. The first seven of these algorithms use full data structures such as SA and LCP in order to execute as quickly as possible; they typically require $8n - 16n$ bytes of storage. The final two however use limited or succinct data structures that reduce space usage to about $4n$ bytes, while generally making a substantial sacrifice in terms of execution time.

As indicated in Figure 5.1, many of these algorithms in a preprocessing stage require computation of SA and LCP arrays; as discussed in Section 2.3, for SA construction we use libdivsufsort [65] and for LCP construction the algorithm in [74]. Since the preprocessing is usually a major component of the overall time for the algorithm, we show these times separately in Table 5.3, where for comparison we include also times for the LCP construction algorithm described in [48].

We used the testing data sets as in Table 5.2. For each string we give its length in letters (bytes) and alphabet size α. Also displayed (from [24]) are the number of factors in each string's LZ factorization together with the length of the maximum factor.

All tests were conducted on a SUN X4600 M2 Server with four 2.6 GHz Dual-Core AMD Opteron(tm) 8218 Processor (total of eight processor cores), 32GB of

String	Length	α	No. Factors	Max Factor	Description
fibo36	14930352	2	35	5702887	36th Fibonacci string
fss10	12078908	2	44	5158310	10th run rich string of [35]
random2	8388608	2	385232	42	Random string
random21	8388608	21	1835235	9	Random string
chr22	34553758	4	2554184	1768	Human Chromosome 22
chr19	63811651	4	4411679	3397	Human Chromosome 19
prot-a	16777216	23	2751022	6699	Small Protein dataset
bible	4047392	62	337558	549	King James Bible
howto	39422105	197	3063929	70718	Linux Howto files
mozilla	51220480	256	3823511	41323	Mozilla binaries

Table 5.2: Description of the strings used in experiments

RAM and four 146GB SAS disks. The operating system is Redhat Linux 5.3 running kernel 2.6.18. All implementations were in C++, compiled using GNU g++ (gcc version 4.1.2) at the -O3 optimization level, and carefully tested.

Running times are the minimum of four runs and do not include time spent reading input files. Times were recorded with the C++ standard library function **clock**. Memory usage was recorded with the **memusage** command available with most Linux distributions.

5.3.2 Test Results

As noted above, we give in Table 5.3 the preprocessing times for the various data structures required by the various LZ factorization algorithms; specifically, the SA, LCP, and RMQ arrays.

Tables 5.4 and 5.5 give the total runtime (in seconds) and peak memory usage

String	SA	LCP[74]	LCP[48]	RMQ
fibo36	4.89	10.23	1.07	0.88
fss10	3.89	7.87	0.82	0.70
random2	1.55	1.48	0.93	0.51
random21	2.34	1.16	1.06	0.51
chr22	7.88	5.97	4.60	2.11
chr19	15.93	11.93	10.76	3.91
prot-a	4.79	2.79	1.99	1.02
bible	0.59	0.54	0.36	0.24
howto	7.96	7.91	4.38	2.32
mozilla	8.45	8.46	5.39	3.19

Table 5.3: Runtime in seconds for SA, LCP, and RMQ arrays

(in bytes per input symbol), respectively, for each of the LZ algorithms tested. Both tables take full account of the contribution made by the preprocessing. The vertical lines in these tables separates the algorithms that use full data structures from "succinct" algorithms that generally use less than eight bytes per input symbol. In each section of the table we underline in bold the quantity that achieves the best result for the current test case.

We make the following observations:

(1) Of the seven algorithms that use full data structures, CPS1–2 and CIS execute fastest on all the files that are not highly periodic, and in fact differ by no more than 5% in any of the test cases. Since CIS uses on average about 10% less space than CPS1–2, it must therefore be regarded as the algorithm of choice between the two.

String	AKO	CPS1–2	CPS1–3b	CI1	CI2	CIS	CII	CPS2	CPS3
fibo36	22.10	15.81	16.73	**7.61**	15.95	15.93	16.20	9.06	**2.47**
fss10	17.20	12.27	13.11	**6.26**	12.48	12.38	12.80	7.19	**1.93**
random2	6.77	**3.46**	4.00	3.69	3.60	**3.46**	3.75	15.84	**2.55**
random21	5.36	**3.91**	4.42	4.51	4.07	3.94	4.20	23.46	**4.64**
chr22	28.15	16.40	20.10	19.63	16.73	**16.15**	17.26	103.84	**31.49**
chr19	55.73	33.04	41.30	39.03	34.30	**32.94**	35.09	208.01	**133.23**
prot_a	12.37	8.55	9.92	10.07	9.46	**8.52**	9.03	**45.85**	145.20
bible	2.12	**1.33**	1.65	1.42	1.39	**1.33**	1.45	5.54	128.26
howto	29.00	**18.13**	23.09	18.80	19.14	18.44	19.47	74.26	–
mozilla	28.97	**19.38**	25.35	30.85	21.13	20.37	21.23	88.18	–

Table 5.4: Total runtime in seconds for each LZ factorization algorithm

String	AKO	CPS1–2	CPS1–3b	CI1	CI2	CIS	CII	CPS2	CPS3
fibo36	23.32	15.52	**11.52**	13.00	12.00	12.00	13.00	6.00	**5.56**
fss10	23.08	15.12	**11.12**	13.00	12.00	12.00	13.00	6.00	**5.68**
random2	22.80	13.00	**9.00**	13.00	12.00	12.00	13.00	6.00	**6.00**
random21	10.24	13.00	**9.00**	13.00	12.00	12.00	13.00	6.00	**5.52**
chr22	16.64	13.00	**9.00**	13.00	12.00	12.00	13.00	6.00	**5.64**
chr19	16.84	13.00	**9.00**	13.00	12.00	12.00	13.00	6.00	**5.36**
prot_a	12.12	13.20	**9.20**	13.00	12.00	12.00	13.00	6.00	**5.12**
bible	14.28	13.00	**9.00**	13.00	12.00	12.00	13.00	6.00	**5.28**
howto	14.72	13.00	**9.00**	13.00	12.00	12.00	13.00	**6.00**	–
mozilla	10.52	13.72	**9.72**	13.00	12.00	12.00	13.00	**6.00**	–

Table 5.5: Peak memory usage in bytes per input symbol for the algorithms

(2) However, the execution times of all six CPS and CI algorithms are consistently very close, with CPS1–3b being in every case the slowest. Over the strings that are not highly periodic, CPS1–3b runs at most 20% slower than the fastest of the six. On the other hand, CPS1–3b uses at least 20% less space than any other of the other six algorithms. Since space is often a more important criterion, especially for large files, CPS1–3b seems therefore to be advantageous in many practical situations.

(3) On highly periodic strings that rarely occur in practice, and over non-succinct algorithms, CI1 is the surprising winner in terms of execution time. This phenomenon occurs because, as noted in Figure 5.1, CI1 is independent of LCP construction, and we note from Table 5.3 that the LCP algorithm [74] used in our tests is 10 times slower on highly periodic strings than the algorithm [48], while otherwise being very competitive. The advantage of CI1 on highly periodic strings disappears if the other LCP algorithm is used; CPS1–2 again becomes marginally the fastest. Note that CI1 uses 13 bytes of space.

(4) For CPS3, Tables 5.4 and 5.5 give results for the choice $q = 2$; for larger q, specifically $q = 3$, either time or storage requirements become unacceptably large. Over all algorithms, succinct or not, CPS3 ($q = 2$) is dominant on binary strings, both highly periodic and random: it is both fastest overall and least space-consuming. It is competitive in terms of time, and superior in terms of

space usage, also on random strings on an alphabet of size 21, but for the other strings tested, it turns out to be very slow.

(5) CPS2 seems to provide possible advantage only for natural language strings (e.g., the Bible). This observation perhaps provides a basis for more detailed research.

(6) A comparison of Tables 5.3 and 5.4 reveals that 80% or more of the overall runtime is generally used in computing the data structures, especially SA and LCP. Once these structures are in place, the algorithm-specific processing is extremely efficient.

5.4 Conclusion

In this chapter we have surveyed a collection of LZ construction algorithms that have been proposed by several authors over the last few years for use in the calculation of regularities in strings rather than in the traditional application to text compression. These algorithms fall into two categories: those that use full data structures, and those that are "succinct". The succinct methods typically use about one computer word per text symbol, but are several times slower in execution speed. In view of the importance of minimizing the usage of space, the challenge for the future seems to be to devise succinct algorithms whose time efficiency competes with that of algorithms that make use of full data structures. Alternatively, in view of the large proportion

of overall runtime spent on computing these data structures, and in the light of of recent research [18, 35] showing that regularities in strings are usually sublinear in string length, perhaps it is time to intensively investigate methods that avoid these computations altogether.

Chapter 6

Summary and Future Work

6.1 Summary

In this thesis, we studied several problems related to the computation of regularities in strings.

We proposed four RPT1 and one RPT2 algorithms for computing NE repeats and two RPT3 algorithms for computing SNE repeats. Among them, RPT1-3, RPT1-4 and RPT3-2 execute in linear time independent of alphabet size. We believe that quality software is produced by the thoughtful application of good engineering methods and tools throughout the specification [89], design, coding, and testing stages. Thus, in order to evaluate the efficiency of our algorithms in practice as well as to compare them with existing algorithms for similar problems, we have conducted comprehensive experimental work, analyzed the results carefully and drawn conclusions about the situations in which these algorithms perform best on our results. Our experimental results have shown that RPT1-4 is the best on overall strings tested, while

the RPT1-1 and RPT1-2 are better for non-highly periodic strings. Moreover, RPT1 algorithms are faster than the two other algorithms previously proposed for this problem. These RPT algorithms have several practical applications in data compression, computational biology and data mining.

We have also formulated two problems related to multirepeats in sets of strings with various restrictions and extended our RPT1 to present three efficient algorithms with lower time complexity and less memory consumption compared to previously proposed algorithms. Among these algorithms, two versions are for multirepeats with arbitrary gaps, with worst-case time complexities $O(Nn+\alpha \log_2 N)$ and $O(Nn+\alpha)$ that use $9Nn$ and $10Nn$ bytes of space respectively, while the third one applies to the bounded gaps problem, with worst-case time complexity $O(RNn)$ that requires approximately $10Nn$ bytes, where R is the number of multirepeats output. In biological sequences (DNA, RNA, or protein) the problem of locating multirepeats arises in many contexts, such as database searching and sequence alignment. It is also important in data mining. We observed that if we set the *min* and *max* constraints on gaps equal to zero in the algorithm MultiRepG, we can find all repetitions in arbitrary subsets of S.

In the final part of the thesis, we investigated the recently proposed LZ factorization algorithms which are used in the computation of regularities in strings rather

than in the traditional application to text compression. We first provided a theoretical comparison of their time and space efficiency, followed by the experimental results on both time and space testing. We observed that these algorithms fall into two categories: those that use full data structures, and those that are "succinct". The succinct methods typically use about one computer word per text symbol, but are several times slower in execution speed. Of the seven algorithms that use full data structures, CPS1–2 and CIS execute fastest on all the files that are not highly periodic, and in fact differ by no more than 5% in any of the test cases. Since CIS uses on average about 10% less space than CPS1–2, it must therefore be regarded as the algorithm of choice between the two. In view of the importance of minimizing the usage of space, the challenge for the future seems to be to devise succinct algorithms whose time efficiency competes with that of algorithms that make use of full data structures. Alternatively, in view of the large proportion of overall runtime spent on computing these data structures, perhaps it is time to intensively investigate methods that avoid these computations altogether.

6.2 Future Work

There are several lines along which future research could proceed.

1. In the main work of this thesis in Chapter 3, we reported all complete non-extendible repeats with user defined minimum period, but they are reported in order of positions in the SA array rather than in $\boldsymbol{x}$. So the question remains: can we

efficiently reorganize the output NE repeats in the order of positions in $\boldsymbol{x}$? If so, we believe that we could compute complete repetitions in an efficient way.

2. As we can report all the complete NE repeats including tandem, overlapping, and split repeats in $O(n)$ time and $6n$ space, we wonder if we can make use of all the advantages of the RPT1 algorithms to report all quesiperiodicities from our algorithm as a by-product? Then we could solve the problem that the BP algorithm [13] proposed by using SA arrays.

3. There are many studies of locating approximate repeats or weighted repeats in a string, but not multirepeats in sets of strings. Future work, then could include detection of degenerate (approximate) multirepeats and weighted multirepeats.

4. In [2] the algorithm for locating common motifs with gaps by using suffix trees was proposed. The authors consider that the problem of finding common motifs with gaps is similar to finding the NE repeats present in strings with contiguous don't care symbols between them, which form the gaps. If we consider this problem as for a particular NE repeat that appears in each string with fixed (or arbitrary) gaps with other NE repeats which must also appear in each string, then we could extend our RPT1 algorithm to find fixed gap or arbitrary gap motifs without using suffix trees, which would be more space efficient.

Bibliography

[1] Anisa Al-Hafeedh, Maxime Crochemore, Lucian Ilie, Jenya Kopylov, William F. Smyth, German Tischler, and Munina Yusufu, A comparison of Lempel-Ziv LZ77 factorization algorithms, submitted for publication (2009).

[2] Pavlos Antoniou, Maxime Crochemore, Costas S. Iliopoulos, and Pierre Peterlongo, Application of suffix trees for the acquisition of common motifs with gaps in a set of strings, *Proc. 1st International Conference on Language and Automata Theory and Applications* (2007) 57–66.

[3] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch, Replacing suffix trees with enhanced suffix arrays, *J. Discrete Algs. 2* (2004) 53–86.

[4] Alberto Apostolico and Stefano Lonardi, Off-line compression by greedy textual substitution, *Proc. IEEE 88–11* (2000) 1733–1744.

[5] Alberto Apostolico and Franco P. Preparata, Optimal off-line detection of repetitions in a string, *Theoret. Comput. Sci. 22* (1983) 297–315.

[6] A. Bakalis, Costas S. Iliopoulos, Christos Makris, Spyros Sioutas, Evangelos Theodoridis, Athanasios K. Tsakalidis, and Kostas Tsichlas, Locating maximal multirepeats in multiple strings under various constraints, *The Computer Journal 50–2* (2007) 178–185.

[7] Jean Berstel and Alessandra Savelli, Crochemore factorization of Sturmian and other infinite words, *Proc. 31st Internat. Symp. Math. Foundations of Computer Sci.*, Ratislav Kralovic and Pawel Urzyczyn (eds.), LNCS 4162, Springer-Verlag (2006) 157–166.

[8] Hamid Abdul Basit, Simon J. Puglisi, William F. Smyth, Andrew Turpin, and Stan Jarzabek, Efficient token based clone detection with flexible tokenization, *Proc. 6th Joint Meeting: European Software Engineering Conference & ACM SIGSOFT Symposium on Software Engineering* (2007) 513–516.

[9] Timothy C. Bell, Better OPM/L text compression, *IEEE Trans. Communications COM-34 (12)* (1986) 1176–1182.

[10] Gary Benson, Tandem repeats finder: A program to analyze DNA sequences, *Nucleic Acids Research 27-2* (1999) 573–580.

[11] Yaniv Berstein and Justin Zobel, Accurate discovery of co-derivative documents via duplicate text detection, *Information Systems 31* (2006) 595–609.

[12] Gerth S. Brodal, Rune B. Lyngso, Christian N. S. Pederesen, and Jens Stoye, Finding maximal pairs with bounded gap, *J. Discrete Algs. 1* (2000) 77–103.

[13] Gerth S. Brodal and Christian N.S. Pedersen, Finding maximal quasiperiodicities in strings, *Lecture Notes in Computer Science 1848* (2000) 397–411.

[14] Michael Burrows and David J. Wheeler, A block-sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation (1994).

[15] Maxime Crochemore, Transducers and repetitions, *Theoret. Comput. Sci. 45-1* (1986) 63–86.

[16] Maxime Crochemore, An optimal algorithm for computing the repetitions in a word, *Inform. Process. Lett. 12-5* (1981) 244–250.

[17] Maxime Crochemore and Lucian Ilie, Computing longest previous factor in linear time and applications, *Inform. Proc. Lett. 106* (2008) 75–80.

[18] Maxime Crochemore and Lucian Ilie, Maximal repetitions in strings, *J. Computer & Sys. Sciences 74-5* (2008) 796–807.

[19] Maxime Crochemore, Lucian Ilie, Costas S. Iliopoulos, Marcin Kubica, Wojciech Rytter, and Tomasz Waleń, LPF computation revisited, *Proc. 20th Internat. Workshop on Combinatorial Algs.*, Jan Kratochvil and Mirka Miller (eds.), LNCS, Springer-Verlag (2009) to appear.

[20] Maxime Crochemore, Lucian Ilie, and William F. Smyth, A simple algorithm for computing the Lempel–Ziv factorization, *Proc. 18th Data Compression Conference (DCC'08), J. A. Storer and M. W. Marcellin (eds.)* (2008) 482–488.

[21] Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson, A sub-quadratic sequence alignment algorithm for unrestricted cost matrices, *Proc. 12th ACM-SIAM Symp. Discrete Algs.* (2002) 679–688.

[22] Maxime Crochemore, Lucian Ilie, and Liviu Tinta, The "runs" conjecture, submitted for publication (2009).

[23] Gang Chen, Simon J. Puglisi, and William F. Smyth, Fast and practical algorithms for computing all runs in a string, *Proc. 18th Annual Symposium on Combinatorial Pattern Matching* (2007) 307–315.

[24] Gang Chen, Simon J. Puglisi, and William F. Smyth, Lempel-Ziv factorization using less time & space, *Mathematics in Computer Science 1-4*, Joseph Chan and Maxime Crochemore (eds.) (2008) 605–623.

[25] Tim Crawford, Costas S. Iliopoulos, and Rajeev Raman, String matching techniques for musical similarity and melodic recognition, *Computing in Musicology 11* (1998) 73–100.

[26] Manolis Christodoulakis, Costas S. Iliopoulos, M. Sohel Rahman, and William F. Smyth, Identifying rhythms in musical texts, *Internat. J. Foundations of Computer Sci. 19-1* (2008) 37–52.

[27] Emilios Cambouropoulos, Maxime Crochemore, Costas S. Iliopoulos, Laurent Mouchard, and Yoan J. Pinzon, Algorithms for computing approximate repetitions in musical sequences, R. Raman and J. Simpson editors, *Proc. of the 10th Australasian Workshop on Combinatorial Algorithms* (1999) 129–144.

[28] Michael Dippersein, *LZSS (LZ77) Discussion and Implementation*,

`http://michael.dipperstein.com/lzss/`

[29] Jean-Pierre Duval, Roman Kolpakov, Gregory Kucherov, Thierry Lecroq, and Arnaud Lefebvre, Linear-time computation of local periods, *Theoret. Comput. Sci. 326(1-3)* (2004) 229–240.

[30] Martin Farach, Optimal suffix tree construction with large alphabets, *Proc. 38th IEEE Symp. Found. Computer Science* (1997) 137–143.

[31] Johannes Fischer and Volker Heun, A new succinct representation of rmq-information and improvements on the enhanced suffix array, *Proc. ESCAPE 2007*, Bo Chen, Mike Paterson and Guochuan Zhang (eds.), LNCS 4614, Springer-Verlag (2007) 459–470.

[32] Johannes Fischer, Volker Heun, and Stefan Kramer, Optimal string mining under frequency constraints, *Proc. 10th European Conf. on Principles and Practice of Knowledge Discovery in Databases*, LNCS 4213, Springer-Verlag (2006) 139–150.

[33] Frantisek Franek, William F. Smyth, and Yudong Tang, Computing all repeats using suffix arrays, *J. Automata, Languages & Combinatorics 8-4* (2003) 579–591.

[34] Frantisek Franek, Jan Holub, William F. Smyth, and Xiangdong Xiao, Computing quasi suffix arrays, *J. Automata, Languages & Combinatorics 8-4* (2003) 593–606.

[35] Frantisek Franek, Jamie Simpson, and William F. Smyth, The maximum number of runs in a string, *Proc. 14th Australasian Workshop on Combinatorial Algs.*, Mirka Miller and Kunsoo Park (eds.) (2003) 36–45.

[36] Ryoichi Fujino, Hiroki Arimura, and Setsuo Arikawa, Discovering unordered and ordered phrase association patterns for text mining, *Proc. 4th Pacific-Asia Conf. Knowledge Discovery and Data Mining* (2000) 281–293.

[37] Dan Gusfield, *Algorithms on Strings, Trees and Sequences*, Cambridge University Press (1997) 534 pp.

[38] Jiawei Han and Micheline Kamber, *Data Mining: Concepts and Techniques*, 2nd edition, Morgan Kaufmann (2006).

[39] Costas S. Iliopoulos, William F. Smyth, and Munina Yusufu, Faster algorithms for computing maximal multirepeats in multiple sequences, *Fundamenta Informaticae*, Special StringMasters Issue (2009) to appear.

[40] Kazuyuki Narisawa, Hideo Bannai, Kohei Hatano, and Masayuki Takeda, Unsupervised spam detection based on string alienness measures, Technical report. Department of Informatics, Kyushu University (2007)

[41] Stefan Kurtz, Reducing the space requirement of suffix trees, *Software, Practice & Experience 29-13* (1999) 1149–1171.

[42] Roman Kolpakov and Gregory Kucherov, *Mreps*,

`http://bioinfo.lifl.fr/mreps/`

[43] Roman Kolpakov and Gregory Kucherov, Finding maximal repetitions in a word in linear time, *Proc. 40th Annual IEEE Symp. Found. Computer Science* (1999) 596–604.

[44] Roman Kolpakov and Gregory Kucherov, Finding repeats with fixed gap, *Proc. 7th International Symposium on String Processing and Information Retrieval* (2000) 162–168.

[45] Pang Ko and Srinivas Aluru, Space efficient linear time construction of suffix arrays, *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, R. Baeza-Yates, E. Chávez, and M. Crochemore (eds.), LNCS 2676, Springer-Verlag (2003) 200–210.

[46] S. Karlin, G. Ghandour, F. Ost, S. Tavare, and L. J. Korn, New approaches for computer analysis of nucleic acid sequences, *Proc. Natl. Acad. Sci. USA 80* (1983) 5660–5664.

[47] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, *Proc. 12th Annual Symp. Combinatorial Pattern Matching*, LNCS 2089, Springer-Verlag (2001) 181–192.

[48] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi, Permuted longest-common-prefix array, *Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM'09)* (2009) 181–192.

[49] Juha Kärkkäinen and Peter Sanders, Simple linear work suffix array construction, *Proc. 30th Internat. Colloq. Automata, Languages & Programming*, LNCS 2719, Springer-Verlag (2003) 943–955.

[50] Sau Dan Lee and Luc De Raedt, An efficient algorithm for mining string databases under constraints, *Proc. KDID*, LNCS 3377, Springer-Verlag (2005) 108–129.

[51] Jesper Larsson and Alistair Moffat, Offline dictionary-based compression, *Proc. Data Compression Conference (DCC'99)* (1999) 296–305.

[52] André Lentin and Marcel P. Schützenberger, A combinatorial problem in the theory of free monoids, *Combinatorial Mathematics & Its Applications*, R. C. Bose and T. A. Dowling (eds.), University of North Carolina Press (1969) 128–144.

[53] Glen G. Langdon, A note on the Ziv-Lempel model for compressing individual sequences, *IEEE Trans. Inform. Theory, IT-29* (1983) 284–287.

[54] Sunho Lee and Kunsoo Park, Dynamic rank-select structures with applications to run-length encoded texts, *Proc. 18th Annual Symp. Combinatorial Pattern Matching*, Bin Ma and Kaizhong Zhang (eds.), LNCS 4580, Springer-Verlag (2007) 95–106.

[55] Jesper Larsson and Kunihiko Sadakane, Faster suffix sorting, Tech. Rep. LU-CS-TR:99-214 [LUNFD6/(NFCS-3140)], Department of Computer Science, Lund University, Sweden (1999).

[56] Abraham Lempel and Jacob Ziv, On the complexity of finite sequences, *IEEE Trans. Information Theory 22* (1976) 75–81.

[57] Ross A. Lippert, Clark M. Mobarry, and Brian Walenz, A space-efficient construction of the Burrows-Wheeler transform for genomic data, *J. Computational Biology 12-7* (2005) 943–951.

[58] Michael G. Main, Detecting leftmost maximal periodicities, *Discrete Applied Maths. 25* (1989) 145–153.

[59] Michael G. Main and Richard J. Lorentz, An $O(n \log n)$ algorithm for recognizing repetition, Tech. Rep. CS-79–056, Computer Science Department, Washington State University (1979).

[60] Michael G. Main and Richard J. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, *J. Algs. 5* (1984) 422–432.

[61] Edward M. McCreight, A space-economical suffix tree construction algorithm, *J. Assoc. Comput. Mach. 32-2* (1976) 262–272.

[62] Wataru Matsubara, Kazuhiko Kusano, Akira Ishino, Hideo Bannai, and Ayumi Shinohara1, New lower bounds for the maximum number of runs in a string, *Proc. Prague Stringology Conference 2008*, Jan Holub and Jan Zdarek (eds.) (2008) 140–144.

[63] Michael Maniscalco and Simon J. Puglisi, Faster lightweight suffix array construction, *Proc. 17th Australasian Workshop on Combinatorial Algs.*, J. Ryan and Dafik (eds.) (2006) 16–29.

[64] Giovanni Manzini, Two space-saving tricks for linear time LCP computation, *Proc. 9th Scandinavian Workshop on Alg. Theory*, LNCS 3111, T. Hagerup and J. Katajainen (eds.), Springer-Verlag (2004) 372–383.

[65] Yuta Mori, *DivSufSort (2005)*

`http://www.homepage3.nifty.com/wpage/software/libdivsufsort.html`

[66] Giovanni Manzini and Paolo Ferragina, Engineering a lightweight suffix array construction algorithm, *Algorithmica 40* (2004) 33–50.

[67] Kaziyuki Narisawa, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda, Efficient computation of substring equivalence classes with suffix arrays, *Proc. 18th Annual Symp. Combinatorial Pattern Matching* (2007) 340–351.

[68] Mark Nelson and Jean loup Gailly, *The Data Compression Book*, M&T Books (1995) 541 pp.

[69] Daisuke Okanohara and Kunihiko Sadakane, An online algorithm for finding the longest previous factors, *Proc. 16th Annual European Symp. on Algs.*, Dan Halperin & Kurt Melhorn (eds.), LNCS 5193, Springer-Verlag (2008) 696–707.

[70] Pawel Pylak, Efficient modification of LZSS compression algorithm, *Annales UMCS Informatica AI 1* (2003) 61–72.

[71] Simon J. Puglisi, William F. Smyth, and Andrew Turpin, A taxonomy of suffix array construction algorithms, *ACM Computing Surveys 39–2* (2007) 1–31.

[72] Simon J. Puglisi, William F. Smyth, and Munina Yusufu, Fast optimal algorithms for computing all the repeats in a string, *Prague Stringology Conference* (preliminary version), Jan Holub and Jan Zdarek (eds.) (2008) 161–169.

[73] Simon J. Puglisi, William F. Smyth, and Munina Yusufu, Fast optimal algorithms for computing all the repeats in a string, submitted for publication (2009).

[74] Simon J. Puglisi and Andrew Turpin, Space-time tradeoffs for longest-common-prefix array computation, *Proc. 19th Internat. Symp. Algs. & Computation*, S.-H. Hong, H. Nagamochi & T. Fukunaga (eds) (2008) 124–135.

[75] Sung W. Shin and Sam M. Kim, A new algorithm for detecting low-complexity regions in protein sequences, *Bioinformatics 21-2* (2005) 160–170.

[76] Kunihiko Sadakane, Succint representations of lcp information and improvements in the compressed suffix arrays, *Proc. 13th ACM-SIAM Symp. Discrete Algs.* (2002) 225–232.

[77] Bill Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) 423 pp.

[78] Jens Stoye and Dan Gusfield, Simple and flexible detection of contiguous repeats using a suffix tree, *Theoret. Comput. Sci. 279-1/2* (2002) 843–850.

[79] James A. Storer and Thomas G. Szymanski, Data compression via textual substitution, *J. Assoc. Comput. Mach. 29–4* (1982) 928–951.

[80] William F. Smyth and Munina Yusufu, Computing regularities in strings, *Proc. Second IEEE International Conference on Computer Science and Information Technology* (2009) 298–302.

[81] Axel Thue, Über unendliche zeichenreihen, *Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana 7* (1906) 1–22.

[82] Tatsuhiko Tsunoda, Masao Fukagawa, and Toshihisa Takagi, Time and memory efficient algorithm for extracting palindromic and acid sequences, *Pacific Symposium on Biocomputing* (1999) 202-213.

[83] Andrew Turpin and William F. Smyth, An approach to phrase selection for offline data compression, *Proc. 25th Australasian Computer Science Conference*, Michael Oudshoorn (eds.) (2002) 267-273.

[84] Esko Ukkonen, Approximate string-matching with q-grams and maximal matches, *Theoret. Comput. Sci. 92–1* (1992) 191–211.

[85] Esko Ukkonen, On-line construction of suffix trees, *Algorithmica 14* (1995) 249–260.

[86] Tanguy Urvoy, Thomas Lavergne, and Pascal Filoche, Tracking Web spam with hidden style similarity, *AIRWEB 2006* (2006) 25–31.

[87] Peter Weiner, Linear pattern matching algorithms, *Proc. 14th Annual IEEE Symp. Switching and Automata Theory* (1973) 1–11.

[88] Munina Yusufu, Computing complete repeats using suffix array, *Presented at WISE (Women in Science & Engineering) Initiative International Women's Day Conference* (2008).

[89] Munina Yusufu and Gulina Yusufu, Comparison of software specification methods using a case study, *Proc. 2008 International Conference on Computer Science and Software Engineering* (2008) 784–787.

[90] Christina Zeeh, *The Lempel-Ziv Algorithm* (2003)

`http://tuxtina.de/files/seminar/LempelZiv.pdf`

[91] Jacob Ziv and Abraham Lempel, A universal algorithm for sequential data compression, *IEEE Trans. Information Theory 23* (1977) 337–343.

[92] Jacob Ziv and Abraham Lempel, Compression of individual sequences via variable-rate coding, *IEEE Trans. Information Theory 24* (1978) 530–536.