Implementing Efficient Algorithms for Computing Runs

Chia-Chun (Jasper) Weng

Implementing Efficient Algorithms for Computing Runs

By

Chia-Chun Weng, B.Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (2011) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE:	Implementing Efficient Algorithms for Computing Runs
AUTHOR:	Chia-Chun Weng, B.Sc. (National Central University)
SUPERVISOR:	Dr. Frantisek Franek
NUMBER OF PAGES:	x, 55.

Abstract

In the first part of this thesis we present a C++ implementation of an improved $O(n \log n)$ algorithm to compute runs, number of primitively rooted distinct squares, and maximal repetitions, based on Crochemore's partitioning algorithm. This is a joint work with Mei Jiang and extends her work on the problem. In the second part we present a C++ implementation of a linear algorithm to compute runs based on the Main's, and Kolpakov and Kucherov's algorithms following the strategy:

1. Compute suffix array and LCP array in linear time;
2. Using the suffix array and LCP array, compute Lempel-Ziv factorization in linear time;
3. Using the Lempel-Ziv factorization, compute in linear time some of the runs that include all the leftmost runs following Main's algorithm;
4. Using Kolpakov and Kucherov's approach, compute in linear time the rest of all the runs.

For our linear time implementation, we partially relied on Jonathan Fischer's Java implementation [20].

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Frantisek Franek, for all of his guidance and support throughout my studies, and for giving me the opportunity to learn under his advisement. My special thanks go to the members of the examination committee: Dr. Antoine Deza and Dr. Emil Sekerinski. I also wish to gratefully acknowledge the help from Dr. Frantisek Franek, Dr. Antoine Deza and all the Advanced Optimization Laboratory members, Feng Xie, Mei Jiang, Min Jing Liu, Hongfeng Liang and Robert Fuller. Finally, I want to thank my lovely parents and my family for their constant support and encouragement.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Periodicity is a fundamental topic in research into algorithms on strings. Periodicity of a string allows often efficient processing and thus research of periodicities finds numerous applications in bioinformatics, data compression, text retrieval, text searching, data mining and many more. The basic element of periodicity in a string is a substring of the form uu, a so-called *square*. The expression uu indicating *concatenation* of the two substrings u is often abbreviated to u^2. More complex conglomerations of squares often investigated are *maximal repetitions* and even more complex *runs*. In this thesis, we are going to discuss two approaches to computations of runs. The first leads to an $O(n \log n)$ algorithm and is based on the idea of computing repetitions using refinement of certain classes of equivalences introduced by Crochemore. The C++ implementation presented here extends Crochemore's repetition algorithm and provides computations not only of the maximal repetitions, but also the number of primitively rooted distinct squares and of runs. An important goal of the implementation is lowering the memory requirement during processing.

The second approach presented here leads to a linear time implementation, based on the Main's, and the Kolpakov and Kucherov's algorithms. It amounts to implementing linear-time constructions of the suffix array, longest common prefix array, and Lempel-Ziv factorization first to get enough data for computing runs in linear time. After the essential information is computed, Main's algorithm calculates all the leftmost occurrences of the runs, and Kolpakov and Kucherov's algorithm then uses the leftmost runs to find all the other runs. Considering how the memory handling effects the speed of execution, we chose C++ for the implementation of this algorithm. We partially rely on Johannes Fischer's Java implementation of the same algorithm [20].

This thesis is structured in the following way: the rest of this chapter covers the basic notions and notations of Stringology. Chapter 2 introduces the original Crochemore's repetition algorithm, which separates the indices of the input string into many partitions (classes of equivalence) and computes the resulting squares and maximal repetitions in $O(n \log n)$ time. Section 2.2 introduces three $O(n \log n)$ extensions of this algorithms to compute runs due to Franek and Jiang. All of the three algorithms compute the runs by consolidating the maximal repetitions as computed by the Crochemore's algorithm. They differ in the strategy when the consolidation takes place and in the data structure used to collect the maximal repetitions. Section 2.3 introduces another $O(n \log n)$ extension due to Franek, Jiang and myself that reduces the working memory required and removes the overhead processing of the collection and consolidation. The final memory requirement of the overall algorithm was reduced from $O(n \log n)$ integers to $13n$ integers, even less than the most memory efficient implementation of the Crochemore's algorithm that is $15n$ integers.

Chapter 3 introduces Main's algorithm, then Kolpakov and Kucherov's algorithm, and the new C++ overall implementation. Sections 3.2 to 3.4 discuss the linear constructions of the suffix array, the longest common prefix array, and the Lempel-Ziv factorization. After those data structures are all computed, all the leftmost runs are computed by the Main's algorithm in Section 3.5, and using the leftmost runs to compute all the runs by Kolpakov and Kucherov's algorithm in Section 3.6. Finally, the conclusion of these implementations and the possible future work will be discussed in the last chapter. The source code can be downloaded from my website [19].

1.2 Basic definitions

Here we are going to introduce some key notions and definitions used in this thesis. Basically, most of the original definitions were given in [16] and [5].

1.2.1 Strings

A string is a finite collection of elements, and the elements have unique labels. For every element which is not the leftmost, we can find its predecessor $p(x)$; for every element which is not the rightmost, we can find its successor $s(x)$. For each element x, we can find x equal to $s(p(x))$ or $p(s(x))$ or both. For any two distinct elements x and y, we can also find a positive number k that satisfies $x = s^k(y)$ or $x = p^k(y)$. In this thesis, the strings we deal with are the *linear strings*. A linear string is a sequence (or an array) of characters. The set of the unique labels of the characters is called the *alphabet*. Thus, we consider a string x as an array $x[0..n-1]$, where $n \geq 0$ is the length of the string. The subarray $x[i..j]$ where $0 \leq i \leq n-1$ is called a *substring*. The substring $x[0..i]$

where $i \leq n-1$ is called a *prefix* of x. The substring $x[i..n-1]$ where $i \geq 0$ is called a *suffix* of x. The size of a string $x[0..n-1]$ is denoted by $|x|$.

The *equality of strings* can be defined recursively by the length of the strings: $x = y$ if and only if $|x| = |y| = n$ and $x[0] = y[0]$ and $x[1..n-1] = y[1..n-1]$ for $n > 1$. An *empty string* is denoted by ε and is considered to be a substring of any string.

For technical reasons we often consider string to be terminated by a special, so-called *sentinel character* \$ (a kin to NULL character terminating C strings) that is always considered as lexicographically smallest in the alphabet of the string.

1.2.2 Repetitions and squares

A collection of the same repeating substrings is called a *repeat*. A *tandem repeat* with adjacent repeating substrings is called a *repetition*. A repetition of $r \geq 2$ substrings u is often denoted as u^r. If $r = 2$ we talk of *squares*, if $r = 3$ we talk of *cubes*. If a string $x = u^r v$ where v is a proper prefix of u, then $p = |u|$ is called a *period* of x. For a repetition u^r, u is referred to as its *generator*, $|u|$ is its *period*, r is referred to as the *exponent* (or *power*). For example, $x = aabaabaabaab = (aab)^4$, aab is the generator, 3 is the period, 4 is the exponent, and the string x is a repetition.

A string is called *primitive* if is not a repetition. A repetition is *primitively rooted* if its generator is primitive. The motivation is quite clear – consider for instance $abababab$ it can be viewed as $(ab)^4$ or as $(abab)^2$. It is unnecessary to "compute" or "list" both, it is quite enough to compute the primitively rooted one $(ab)^4$ as the non-primitively rooted one $(abab)^2$ can be easily "deduced"

from $(ab)^4$.

Repetitions in a string x can be coded into triples (s, p, e), where s is the starting index (position) of the repetition, p is its period, and e is its exponent. For instance $x = x[0..9] = a\underline{ab}\underline{ab}bbaba$ has a repetition $(1, 2, 2)$ (underscored). Alternatively, a repetition can be coded as (s, p, d) where s is the starting position of the repetition, p is its period, and d is the ending position. The mutual transformations from one to the other coding are simple: $d = s + ep - 1$ and $e = d/p$ where / signifies the integer division.

A repetition (s, p, e) in a string $x = x[0..n-1]$ is *left-maximal* if "it cannot be extended to the left"; more precisely if either $s < p$ or $x[s-p..s-1] \neq x[s..s+p-1]$. Similarly, a repetition (s, p, e) is *right-maximal* "if it cannot be extended to the right"; more precisely if either $s > n+p$ or $x[s+(e-1)p..s+ep-1] \neq x[s+ep..s+(e+1)p-1]$. A repetition that is both left-maximal and right-maximal is referred to as *maximal*. The motivation is quite clear: consider for instance a string $x[0..7] = abababab$. This contains repetitions $(0, 2, 2)$, $(0, 2, 3)$, $(0, 2, 4)$, $(2, 2, 2)$, $(2, 2, 3)$, and $(4, 2, 2)$. The repetitions $(0, 2, 2)$, $(0, 2, 3)$, $(0, 2, 4)$ are left-maximal, while the repetitions $(0, 2, 4)$, $(2, 2, 3)$, and $(4, 2, 2)$ are right-maximal. Only $(0, 2, 4)$ is maximal and all the other repetitions can be "deduced" from it. It thus makes sense to focus computing on maximal repetitions.

Note that each maximal repetition is a conglomeration of primitively rooted squares. For instance, the above maximal repetition $(0, 2, 4)$ is a conglomeration of $(0, 2, 2)$, $(2, 2, 2)$ and $(4, 2, 2)$ primitively rooted squares. The squares are mutual "shifts" of the size equal to the period of the repetition. This fact is strongly used in the Crochemore's repetitions based algorithms to trace maximal repetitions from the computation of squares.

1.2.3 Runs

A *run* is in essence a repetition followed by a proper prefix of the generator of the repetition, it is another way of considering repetitions with non-integer exponents. Consider for instance a string $x = aabaabaa = (aab)^2aa = (aab)^{2+2/3}$ telling us that the generator repeats 2 times fully, and then 2 out of 3. We call the incomplete last repeat the *tail* of the repetition, so aa is the tail.

A repetition with a tail can be easily encoded by (s, p, e, t) where (s, p, e) encodes the repetition and t is the size of the tail ($0 \leq t < p$). Alternatively, to save a space we could encode a repetition with a tail by (s, p, d) where s is the starting position, p is the period, d is the ending position. Note that again we can easily transform from one notation to the other: $d = s + ep - 1 + t$ and $e = d/p$ and $t = d\%p$ where / signifies the integer division and % denotes the modulo function.

A primitively rooted repetition with tail (s, p, e, t) in a string $x = x[0..n-1]$ is a *run* if it cannot be extended to the left, nor to the right, i.e.

- either $s = 0$ or $x[s-1] \neq x[s+p-1]$
- either $s = n-1$ or $x[s+ep-t] \neq x[s+(e-1)p+t]$

Note that a run is a conglomeration of primitively rooted squares that are "shifted" just one position: for instance the run $(0, 3, 2, 2)$ in the string $x = x[0..7] = aabaabaa$ is a conglomeration of the following primitively rooted squares: $(0, 3, 2)$, $(1, 3, 1)$, and $(2, 3, 2)$. Again, this fact is strongly used in the Crochemore's repetitions based algorithms to trace runs from the computation of squares.

Chapter 2

A new implementation of Crochemore-repetitions-algorithm based algorithm for runs

Crochemore's partitioning algorithm introduced in 1981 was the first $O(n \log n)$ algorithm to compute maximal repetitions in a string, see [1]. An advantage of the algorithm is that fact that it is independent of the size of the alphabet of the input string; the only requirement is that the alphabet be indexed. However, an implementation of the algorithm requires complex and complicated data structures of about $20n$ integers, where n is the length of the input string. Note that the algorithm computes the maximal repetitions in levels, where level L consists of all maximal repetitions of period L. Franek, Smyth and Xiao introduced in 2003 a compact C/C++ implementation using only $15n$ integers of memory [7]. The data structures used allowed for all the required memory to be allocated prior to the execution as a single segment, which improves performance as no dynamic memory allocation or deallocation is performed.

Of course, the memory saving techniques were detrimental to the speed, but not to any significant degree.

In 2009, Franek and Jiang extended the original Crochemore's repetition algorithm to compute runs, [5, 6]. Their algorithm computes the maximal repetitions first, stores them in a data structure, and then consolidates them into runs. They used three different approaches: the algorithm A stores the maximal repetitions for a single level in a binary search tree, when the computation of the level is completed, it traverses the tree and consolidates the repetitions into runs. It takes $O(n \log^2 n)$ time and $O(n \log n)$ extra memory for the search tree. The algorithm B creates for each level a separate search tree and these are traversed and the maximal repetitions consolidated to runs only after all levels have been computed. This keeps the complexity of the algorithm same as the complexity of the underlying Crochemore's algorithm, i.e. $O(n \log n)$, however the memory requirement is boosted significantly. The algorithm C also collect the maximal repetitions from all levels, but they are stored in a different data structure - a linked list array of "buckets". The array is traversed after all the levels have been computed and the stored repetitions are consolidated into runs. This keeps the memory required comparable to program A (i.e. $O(n \log n)$ integers), while keeping the complexity to $O(n \log n)$. Moreover, extensive experiments (see [5, 6]) established the algorithm C as the fastest and most efficient.

In section 2.2.1 we will discuss the loose term "consolidate" we have used so far.

2.1 Crochemore's repetition algorithm

In order to be able to discuss the extensions of Crochemore's algorithm, we need a brief introduction of the algorithm and the relevant data structures.

In 1981, Crochemore's algorithm was the first $O(n \log n)$ algorithm to compute maximal repetitions, [1]. As it works through refinement – or partitioning – of certain classes of equivalence, it is also referred to as *Crochemore's partitioning algorithm*.

Let $x = x[0..n-1]$ be an input string. For a given $1 \leq p \leq n/2$, the equivalence $\underset{p}{\sim}$ is defined as follows: for $0 \leq i, j < n$, $i \underset{p}{\sim} j$ if and only if $x[i..i+p-1] = x[j..j+p-1]$. A class of equivalence $\underset{p}{\sim}$ thus consists of all indices that are starting positions for the same substring of length p. The classes of equivalence $\underset{p}{\sim}$ thus form a partition of the set of all indices $\{0, ..n-1\}$.

Level 1 consists of all the classes of equivalence of $\underset{1}{\sim}$, level 2 of all the classes of equivalence of $\underset{2}{\sim}$, etc. In general, level L consists of all the classes of equivalence of $\underset{L}{\sim}$.

Note that having all these classes of all the levels gives us complete information of all repeats in the string x, and hence all primitively squares in x: if $i \underset{p}{\sim} j$, and $j = i + p$, then there is a tandem repeat, i.e. a square at position i. If the elements of the classes are maintained in their natural order, if i is moreover an immediate predecessor of j in the same class, the square starting at position i must be primitively rooted.

As mentioned above, maximal repetitions are a conglomerations of primitively rooted squares and thus computing all the levels will allow us to determine all maximal repetitions.

The size of all levels is $O(n^2)$ classes, thus a naive algorithm would require

$O(n^3)$ processing, a bit smarter but straightforward algorithm could compute all the levels on $O(n^2)$ steps. The ingenuity of Crochemore's approach was in the way it can be computed in $O(n \log n)$ time.

The algorithm computes all the classes of equivalence $\underset{1}{\sim}$ by brute force; this is the 1st level. Level $L+1$ is computed by refinement of the classes from level L. If the algorithm used the information from the input string and every class of level L would be traversed, this would lead to $O(n^2)$ algorithm.

Instead, for refinement, other classes of the same level are used. Let us consider two classes C_1, C_2 of $\underset{L}{\sim}$: let $C_1 = \{i_1, ..., i_k\}$ and let $j_1, j_2 \in C_2$ and let $j_1 - 1, j_2 - 1 \in C_1$. Then at position $j_1 - 1$ is the same substring of length $L+1$ as at position $j_2 - 1$, i.e. the positions $j_1 - 1, j_2 - 1$ will be in the same class of equivalence of $\underset{L+1}{\sim}$. For illustration, in Figure 2.1, when refining class $C_1 = \{1, 4, 7, 10\}$ by class $C_2 = \{2, 3, 5, 6\}$ on level 1, we will put 1 and 4 into the same class since 2 and 5 are both located in C_2. On the other hand, 8 and 11 come from the other two classes, so we separate $\{7\}$ and $\{10\}$ into two individual classes and thus the class $\{1, 4, 7, 10\}$ is partitioned into the classes $\{7\}$, $\{10\}$, and $\{1, 4\}$.

However, if every class of the level L is refined using every class of the level L, the resulting processing would still be of $O(n^2)$ complexity. The trick is to use only the so-called small classes for the refining. All the classes of level $L+1$ that are a refinement of a class from level L form a family (i.e. they all have the same "parent" on level L). A largest (in size) of them is designed as *big*, all the other as *small*. It is clear that for the refinement process to proceed correctly, it is sufficient to use only the small classes for the refinement. Crochemore realized that, and in the paper he proved that there are at most $O(\log n)$ small classes.

As there are at most $O(n)$ levels, this leads to the complexity of $O(n \log n)$ steps.

Note that in this setting, once the level 1 is computed, the input string may be discarded as it is not used for any further processing.

To illustrate the notion of small classes: when talking about the classes $\{1, 4\}$, $\{7\}$, $\{10\}$ from $\{1, 4, 7, 10\}$, we identify the class $\{1, 4\}$ which has the most elements as "big", and $\{7\}$, $\{10\}$ are therefore "small".

Of course, the objective is not to compute the levels and the classes. The algorithm computes maximal repetitions. To that end, a gap list function $gap()$ is maintained and the natural order of indices in the classes are maintained (though the order of classes themselves cannot be maintained without worsening the complexity). On level L, the gap list function contains the following information: let $gap(p) = \langle i_1, ..., i_k \rangle$, then for each i_t from $gap(k)$, $i_t - p$ is an immediate predecessor of i_t in the class of equivalence $\underset{L}{\sim}$. Thus, $gap(L)$ will give us all primitively rooted squares of period L. Note, that if $p < L$, then $gap(p)$ will give us all repeats that overlap with the size of the overlap being p, while if $p > L$, we get all the repeats that do not touch and do not overlap.

Note that the process of refinement terminates once all classes are refined into *singletons*, i.e. classes of size 1. Figure 2.1 is an example of the complete partitions on all the possible levels. It is customary not to repeat the occurrences of singletons to make the diagrams more readable.

2.2 Franek and Jiang's algorithms

Franek and Jiang extended Crochemore's algorithm to compute the runs in 2009. Basically their approach has two parts: first part is computing and

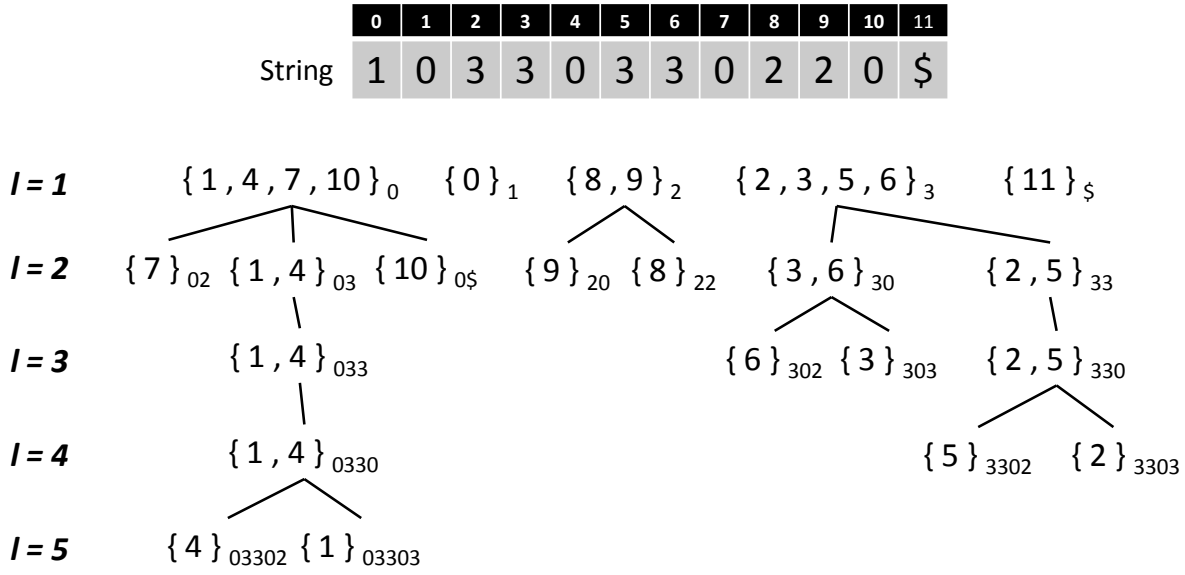

Figure 2.1: Refining the positions into different classes in Crochemore's repetition algorithm

storing all the maximal repetitions using Crochemore's partitioning algorithm, and the second is consolidating the stored repetitions into runs. In [5, 6] they introduced three algorithms A, B and C. Algorithms B and C are of the complexity of $O(n \log n)$ while algorithm A has complexity $O(n \log^2 n)$, all of them requiring $O(n \log n)$ additional memory.

2.2.1 Algorithm A

The underlying Crochemore's partitioning algorithm computes the maximal repetitions level by level. Thus, the maximal repetitions of the same period p are reported once the level p is computed. Algorithm A stores the runs (of

course, their encoding as discussed in section 1.2.2) in a binary search tree ordered by the the starting positions of the runs. When a new repetition is computed, the algorithm traverses the binary search tree and matches is against all the previously computed runs. Either it is found to overlap with an existing run, as explained below, and the repetition is used to possibly extend the run, or not and then the repetition is recorded in the binary tree as a new run. This process had been referred to previously as "consolidation" of repetitions into runs.

In Figure 2.2, we show all possible combinations of a stored run and a new repetition. Since the overlapping substring of case (a) and (b) is shorter than the period, it is impossible to join them together. In case (c), the starting position of the run is to the left of the repetition, having an overlap greater or equal to the period. In other words, the run could be extended to the right, then we update the tail of the run by the repetition. On the other hand, if the run could be extended to the left like case (d), and the overlapping length is greater or equal than the period, then we also update the starting position of the run by the repetition. When discussing the repetition and the run contains each other, in case (e), the run is a substring of the repetition, we use the repetition the replace the run. In case (f) the repetition is a substring of the run, since the run is already represented the repetition, we just ignore the repetition.

The repeated traversal of the search tree leads to a poor performance and complexity of $O(n \log^2 n)$. The additional memory required is of $O(n \log n)$, but can be really implemented using $5(n \log n)$ integers of memory.

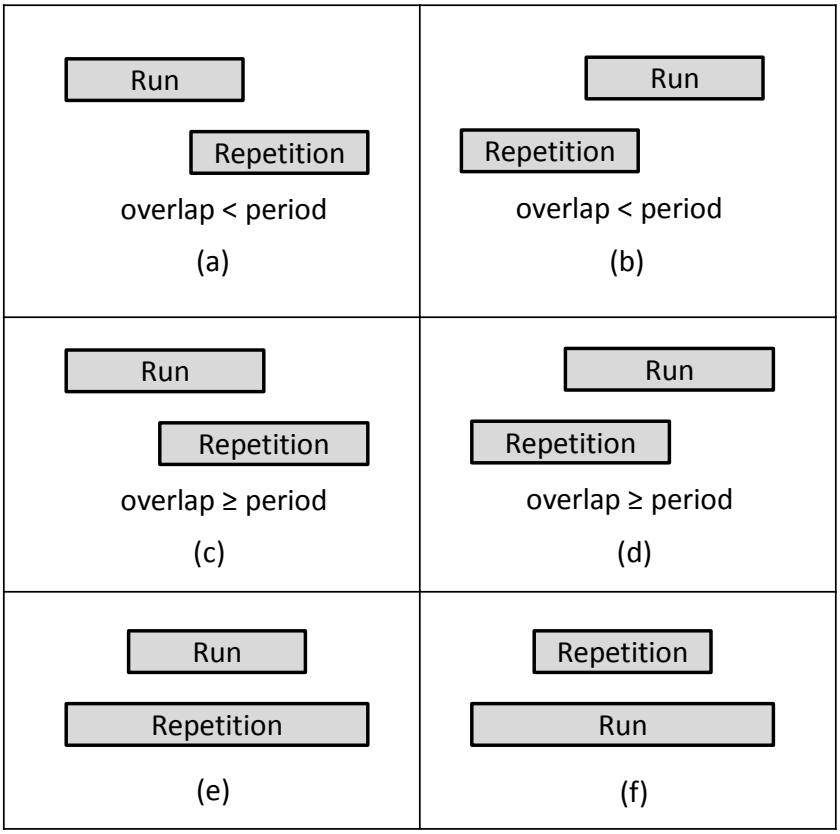

Figure 2.2: Possible combinations for runs and repetitions

2.2.2 Algorithm B

Algorithm B uses the same concept of Algorithm A. It also uses a binary search tree during the computation of the levels. However, Algorithm B builds a separate search tree for each level and stores does not store the runs for all levels, it build a separate search tree to store them at each level and report it right after each level. The only difference between Algorithm A and B is that

the temporary tree structure in Algorithm B could be overwritten. In other words, Algorithm B reuses the tree to save some memory space. According to their experimental result, the performance of this algorithm is better than Algorithm A.

2.2.3 Algorithm C

Unlike building the tree as what Algorithm A and B do, Algorithm C stores the repetitions in the array of buckets. We collect the repetitions computed by Crochemore's algorithm into a linked list first. The repetitions are classified by the starting position, and store their period and the ending position in the bucket. After all the repetitions are computed, the algorithm traverse from left to right of the bucket array. We will test if the repetition is a new run or could be joined with the right most run at that period.

Since we traverse the bucket from the left to the right, it's only possible to get case (a), (c) and (f). If the repetition and the existing run are not overlapped, we will just make the repetition to be a new run and set it as the right most run at that period. In case (a), the repetition and the existing run are overlapped, and the overlapping length is smaller than the period, we will also make the repetition as a new run and set it as the right most run. In case (c), the repetition and the existing run are overlapped, and the overlapping length is greater or equal than the period, we could extend the existing run to the right, so we change the ending position of the existing run as the repetition's. In case (f), the repetition is a substring of the existing run, we just ignore the repetition.

2.3 The new FJW algorithm and its implementation

As Franek and Jiang concluded after the experiments in [5, 6], the performance for Algorithm C is much better than Algorithms A and B and its memory requirement is also the best of the three variants. Nevertheless, the increase of the memory requirement from $15n$ integers to $(15n \log n)$ integers was too a big price to pay. Thus, in 2011, Franek, Jiang and Weng designed and implemented another extension of the Crochemore's partitioning algorithm which does not use the two-step approach and hence computes the runs directly. That not only saved the extra overhead required for collecting the maximal repetitions, it allowed to slightly improve the memory requirement for the underlying partitioning algorithm to $13n$ integers.

Based again on the Crochemore's partitioning algorithm, Franek, Jiang and Weng's algorithm (FJW for short) is more efficient than the three extensions discussed in the previous section. The program not only computes the maximal repetitions (which the original Crochemore's algorithm does), but it also computes the runs and the number of primitively rooted distinct squares.

Note that when we compute runs, we compute all occurrences of runs. In the problem of *the number of primitively rooted distinct squares* we do not compute the occurrences of squares, but their types. For instance, the string $x = x[0..6] = aabaaba$ has 1 run (0,3,2,1), there are 2 occurrences of the square aa (at positions 0 and 3), 1 occurrence of the square $aabaab$ (at position 0) and 1 occurrence of the square $abaaba$ (at position 1), hence it has 4 different squares, but only three distinct squares aa, $aabaab$, and $abaaba$.

The C++ implementation can downloaded from Prof. Franek's web-

site [18]. The website also contains all the versions giving the full genesis of the program and indicating the particular memory saving techniques used.

Unlike Franek and Jiang's algorithms in 2009, this algorithm does not wait until all the repetitions are computed, it computes the the repetitions, or the runs, or the number of distinct level by level based on the information in the gap list function (see section 2.1). The corresponding

At level L, the gap list $gap(L)$ is traversed. As explained in section 2.1, each element i of the gap list $Gap(L)$ at level L identifies a primitively rooted square with period L starting at the position $i - L$. If the entry is marked as *done*, it is ignored and the next entry from the gap list is visited.

1. If the algorithm computes the number of primitively rooted distinct squares, if i is the first entry from a particular class, the square is counted and the class is marked as *done*, otherwise it is ignored. This processing is performed by the procedure *traceSquares*.

2. If the algorithm computes the maximal repetitions, the procedure *traceMaxReps* is used. Once a square is identified, the procedure checks whether it can be extended (as a repetition) to the left, and if so, the entry corresponding to the extension in the gap list is marked as *done*. The new square is than tried if it can be extended yet again to the left and so on as long as the extensions exist. Similarly, the original square is traced to the right. This process builds the maximal repetition from the squares identified by the gap list. The fact that the newly established squares are marked as *done* prevent the algorithm from re-building the repetition once another square of this repetition is encountered in the gap list.

3. If the algorithm computes the runs, the procedure *traceRuns* is used. Once a square is identified, the procedure checks whether it can be shifted to the left, and if so, the entry corresponding to the extension in the gap list is marked as *done*. The new square is then tried if it can be shifted yet again to the left and so on as long as the extensions exist. Similarly, the original square is traced to the right. This process builds the run from the squares identified by the gap list. The fact that the newly established squares are marked as *done* prevent the algorithm from re-building the run once another square of this run is encountered in the gap list.

The fact that runs can be actually built directly from the information in the gap list without first building the maximal repetitions and then consolidating them into runs allowed us to eliminate all the overhead that was so detrimental in Franek and Jiang's previous algorithms. The memory saving techniques discussed in the next section thus apply to the implementation of the underlying partitioning algorithm.

2.3.1 The genesis of the program – the memory saving techniques

1st version – crochB

In the first version `crochB`, the memory requirement is $19n$ integers and no memory saving techniques are used. We use 7 integer arrays dealing with classes, 4 integer arrays dealing with families, 4 integer arrays dealing with the refinement process, and 4 integer arrays dealing with the gap list function.

The following 7 integer arrays represent the classes:

1. $CStart[0..n-1]$ stores the first element of a class. It emulates a pointer to the beginning of the class. For example, $CStart[i] = j$ means that j is the first element of class i.

2. $CEnd[0...n-1]$ stores the last element of a class. It emulates a pointer to the end of the class. For example, $CEnd[i] = j$ means that j is the last element of class i.

3. $CNext[0...n-1]$ stores the next element in the class. It emulates the forward links in the class. For example, $CNext[i] = j$ means that i and j are in the same class, and j is the successor of i. If $CNext[i]$ is *null*, then i is the last element in the class. Note that *null* is a value is emulated by the value of n as this value never occurs in any of the arrays.

4. $CPrev[0...n-1]$ stores the previous element in the class. It emulates the backward links in the class. For example, $CPrev[i] = j$ means that i and j are in the same class, and j is the predecessor of i. If $CPrev[i]$ is *null*, then i is the first element in the class.

5. $CMember[0...n-1]$ stores the class which the elements belong. For example, $CMember[i] = j$ means that the element i belongs to the class j.

6. $CSize[0...n-1]$ stores the size of the classes. For example, $CSize[i] = j$ means that the class i has j elements.

7. $CEmpty[0...n-1]$ is used as a stack. It is a list of empty classes that can be used as destination for refinement.

The following 4 integer arrays represent the families:

1. $FStart[0...n-1]$ is used as a stack. It stores the first class of each family. For example, $FStart[i] = j$ means that the class j is the first class of the family i.

2. $FNext[0...n-1]$ emulates the forward links in the list of classes in the family. For example, $FNext[i] = j$ means that i and j are in the same family, and j is the successor of i. If $FNext[i]$ is *null*, then i is the last class in the family.

3. $FPrev[0...n-1]$ emulates the backward links in the list of classes in the family. For example, $FPrev[i] = j$ means that i and j are in the same family, and j is the predecessor of i. If $FPrev[i]$ is *null*, then i is the first class in the family.

4. $FMember[0...n-1]$ stores the family which the classes belongs to. For example, $FMember[i] = j$ means that the class i belongs to the family j.

The following 4 integer arrays are data structures used in the refinement process:

1. $Refine[0...n-1]$ stores the destination class. For example, $Refine[i] = j$ means that any element from class i is to be moved to class j.

2. $RStack[0...n-1]$ used as a stack. It remembers the items used in $Refine[]$, so it will be able to clean $Refine[]$ (set with *null*s) without traversing the whole array which will destroy the complexity.

3. $Sel[0...n-1]$ is used as a queue. It stores the elements of all small classes.

4. $Sc[0...n-1]$ is used as a queue. It represents a new small class start, so we can use the data from $Sel[]$ and $Sc[]$ to store the information of every small class and its elements.

The following 4 integer arrays represent the gap list function:

1. $Gap[0...n-1]$ stores the first element in the gap lists. For example, $Gap[i] = j$ means that j is the first element in the gap list i, and thus the predecessor of j in the class is $j - i$.

2. $GMember[0...n-1]$ stores the gap membership of the elements. For example, $GMember[i] = j$ means that i belongs to the gap list j.

3. $GNext[0...n-1]$ emulates the forward links in the gap lists. For example, $GNext[i] = j$ means that i and j are in the same gap list, and j is the successor of i. If $GNext[i]$ is *null*, then i is the last element in the gap list.

4. $GPrev[0...n-1]$ emulates the backward links in the gap lists. For example, $GPrev[i] = j$ means that i and j are in the same gap list, and j is the predecessor of i. If $GNext[i]$ is *null*, then i is the first element in the gap list.

2nd version – crochB1

In the next version, `crochB1`, a $GMember()$ function replaces the $GMember[]$ array, and thus the memory requirement is reduced to $18n$ integers. Below is the formula used to compute $GMember()$.

$$GMember(i) = \begin{cases} null & \text{if } i \text{ is not member of any class,} \\ null & \text{if } i \text{ is a member of a class of size 1,} \\ i - Prev[i] & \text{otherwise.} \end{cases}$$

3rd version – crochB2

This version, `crochB2`, eliminates classes `STACK` and `FIFO` and replaces their methods by macros and inlined functions in preparation for their eventual sharing of the same memory segment. The overall memory requirement stays $18n$ integers.

4th version – crochB3

The next version, `crochB3`, virtualizes the $FMember[]$ array over the arrays $FStart[]$ and $FNext[]$ and the access to this virtual array is provided by a function $FMember()$. Moreover a virtual array $FEnd$ is added virtualized over the arrays $FStart[]$ and $FPrev[]$ with an access function $FEnd()$. The memory requirement is thus reduced to $17n$ integers. The access functions are defined below.

$$FMember(i) = \begin{cases} FStart[i] & \text{if } FStart \text{ stack is empty,} \\ FNext[FPrev[FStart[i]]] & \text{if } i \leq \text{the } FStart \text{ stack pointer,} \\ FStart[i] & \text{otherwise.} \end{cases}$$

$FEnd(i) = FPrev[FStart[i]]$.

5th version – crochB4

Since the stack $CEmpty$ and the queue Sc can share the same memory segment, this version `crochB4` implements them so lowering the memory requirement to $16n$ integers.

6th version – crochB5

The version `crochB5` virtualizes the array $CEnd[]$ over $CStart[]$ and $CNext[]$ with an access function $CEnd()$ given below, and the arrays $CSize[]$ over

$CStart[]$ and $CPrev[]$ with an access function $CSize()$ also given below. Thus the memory needed is reduced to $14n$ integers.

$CEnd(i) = CPrev[CStart[i]]$

$CSize(i) = CNext[CPrev[CStart[i]]]$

7th version – crochB6

This version, `crochB6` use a trick to make do with the array $CMember[]$ by virtualizing it over $Gap[]$, $GNext[]$, and $GPrev[]$. In order to be able to distinguish whether the data stored in $Gap[]$ represent the gap list data or the $CMember$ data, positive integers represent the former and negative represent the latter. Of course, this requires that both type of integers stored in $Gap[]$ must be treated as signed integers, which in turn necessitates to limit the length of the input strings to half the size, e.g. for a 32-bit machine from $2^{32} - 1$ to $2^{31} - 1$, which is still more than acceptable for practical purposes.

Note that this positive/negative distinction of the values stored in $Gap[]$ may pose a problem if the value 0 is to be stored there. Thus the $CMember$ data stored in $Gap[]$ are not only negative,m but also shifted by 1, thus -1 really represents 0, -2 really represents 1, etc. This final trick reduces the overall memory to $13n$ integers.

The code given in Algorithm 1 defines the function that sets the value of $CMember$ element e to c, while the code given in Algorithm 2 fetches the value from the e-th element of $CMember$.

Figure 2.3 shows a table representing the memory usage for different FJW implementations.

In order to maintain the running complexity of $O(n \log n)$, the gap list function is updated every time an element is removed from or added to a class.

Algorithm 1: Set the value of CMember(e) to c

```
if Gap[e] = null AND Gap[e] < 0 then
  if c = null then
    Gap[e] = null
  else
    Gap[e] = 0 − 1 − c
  end if
else
  if c = null then
    GNext[GPrev[Gap[e]]] = null
  else
    GNext[GPrev[Gap[e]]] = 0 − 1 − c
  end if
end if
```

Algorithm 2: Get the value of CMember(e)

```
if Gap[e] = null then
  return null
else
  if Gap[e] < 0 then
    return 0 − 1 − Gap[e]
  else
    if GNext[GPrev[Gap[e]]] = null then
      return null
    else
      return GNext[GPrev[Gap[e]]] = 0 − 1 − c
    end if
  end if
end if
```

Type	Array	CrochB	CrochB1	CrochB3	CrochB4	CrochB5	CrochB6	CrochB7
Class	CStart	√	√	√	√	√	√	√
	CEnd	√	√	√	√			
	CNext	√	√	√	√	√	√	√
	CPrev	√	√	√	√	√	√	√
	CMember	√	√	√	√	√		
	CSize	√	√	√	√			
	CEmpty	√	√	√	√ (Share with Sc)			
Family	FStart	√	√	√	√	√	√	√
	FNext	√	√	√	√	√	√	√
	FPrev	√	√	√	√	√	√	√
	FMember	√	√					
Refinement Process	Refine	√	√	√	√	√	√	√
	RStack	√	√	√	√	√	√	√
	Sel	√	√	√	√	√	√	√
	Sc	√	√	√	√ (Share with CEmpty)			
Gap	Gap	√	√	√	√	√	√	√
	GMember	√						
	GNext	√	√	√	√	√	√	√
	GPrev	√	√	√	√	√	√	√
Total Memory Usage		19n	18n	17n	16n	14n	13n	13n

Figure 2.3: Compare the memory usage for different FJW implementation versions

As explained in the section 2.3, when traversing the gap list and tracing either distinct squares, or maximal repetitions, or runs, we need to mark some of the squares *done* so we do not retrace the same repetition or run, or do not count another square of the same type. Since the array $Refine[]$ and the *RStack* are just used for the refinement process, and the tracing takes place after the level had been computed, these arrays can be safely used for this purpose. $Refine[]$ is used to store the required information of what is *done*, while *RStack* is used

to indicate which of the items of $Refine[]$ had been modified, so $Refine[]$ can be initialized (all items set to *null*) without traversing the whole array once all the tracing is done. If the array had to be traversed each time when it is to be initialized, the complexity would increase to $O(n^2 \log n)$.

The final version – crochB7

This is the final version based on `crochB6`. The task of what should be performed by the method `Process` is controlled by the compilation options

```
#define _runs
#define _squares
#define _maxreps
```

The task that should be performed should be defined, the other commented out. If two or all three are defined, the order of precedence is `_runs`, then `_squares`, and then `_maxreps`. If no task is specified, computing runs is the default task. If we want to perform a different task, the program must be recompiled. That was not deemed a serious problem, as rarely there is a need to do in one execution two different tasks. However, it would be a simple modification to distinguish the task to be done in run time.

In any case, the method `Process` returns a single value, the number of runs, or squares, or repetitions.

The program may output the runs or repetitions. The output of the program is handled similarly by defining or commenting out the options that control the output:

```
#define _show
#define _output
```

If neither is defined, there is no output and `Process` only returns the appropriate number. If `_show` is defined, the runs or repetitions or so-called *pretty printed*. They are visually displayed withing the string and the repeating parts are distinguished. For instance, the runs of the string *aaaabaababa* will be shown as

```
aaaabaababa
.....aA....
aAaA.......
......abABa
..aabAABa..
```

or the maximal repetitions of the same string will be shown as

```
aaaabaababa
.....aA....
aAaA.......
......abAB.
.......baBA
..aabAAB...
...abaABA..
```

Note that if the `_show` option is used, the input string must be in lower case characters, as the dichotomy of lower case/upper case is used to make the repeating parts to stand out. For no output or `_output` option the input string can consists of any characters.

An example of output for `_output` option for the same string as above for runs:

```
aaaabaababa
(5,1,2,0)
(0,1,4,0)
(6,2,2,1)
(2,3,2,1)
```

and for maximal repetitions:

```
aaaabaababa
(5,1,2,0)
(0,1,4,0)
(6,2,2,0)
(7,2,2,0)
(2,3,2,0)
(3,3,2,0)
```

If both options are specified, `_show` takes precedence over `_output`.

Chapter 3

Implementation of a linear time Algorithm for Computing Runs

In 1984, Main demonstrated a linear time algorithm that computes a set of runs containing all the leftmost runs from the Lempel-Ziv factorization of the input string, [13]. Note that by *leftmost* run we mean the first from the left occurrence of the run in a string. In 1999, Kolpakov and Kucherov in [10] proved that the number of runs is linear in the length of the input string, and proposed an algorithm that computes all the non-leftmost runs from the Lempel-Ziv factorization. A lot of researchers noticed that a linear time algorithm for computing runs could be realized along the following strategy:

1. Compute the suffix array

2. Using the suffix array compute the largest common prefix array (LCP array for short)

3. Using the suffix array and the LCP array build the Lempel-Ziv factorization of the input string

4. From the Lempel-Ziv factorization compute a superset of all the leftmost runs by Main's algorithm

5. From the Lempel-Ziv factorization compute all the remaining runs by Kolpakov and Kucherov's algorithm.

In 2003, several linear algorithms to compute suffix arrays were introduced opening an avenue for computing the Lempel-Ziv factorization in linear time, [12, 9, 8]. Then the all pieces of a possible strategy to compute runs in linear time were together:

In this chapter, we are going to implement this strategy in C++, and use Johannes Fischer's Java implementation as a reference.

In this chapter, a run is represented as a triple (s, p, d), s being the starting position, p the period, and d the end position of the run. This is purely to save some space when runs must be stored.

3.1 Suffix array

The *suffix array* of a string is a data structure introduced by Manber and Myers 1993 as a more succinct data structure than suffix tree, yet providing almost the same information of the structure of the string, [14]. Suffix array is an array of integers giving the starting positions of suffixes of a string in lexicographical order. In Figure 3.1, we can see an example of the suffixes of a string 10330330220 and the corresponding suffix array.

To compute the suffix array of a string really amounts to sorting of the suffixes, i.e. sorting of a set of strings, which in general, can be done at best in $O(n \log n)$ time,where n is the number of strings to be sorted.

	0	1	2	3	4	5	6	7	8	9	10
String	1	0	3	3	0	3	3	0	2	2	0

SA	Suffix
10	0
7	0220
4	0330220
1	0330330220
0	10330330220
9	20
8	220
6	30220
3	30330220
5	330220
2	330330220

Figure 3.1: The suffix array SA of a string "10330330220"

In 2003, first three linear-time algorithms to compute suffix array occurred, [12, 9, 8]. Of these, only [12] and [9] are really practical. Since then several more efficient and also more complex algorithms have been introduced (see for instance [15]. For our project we opted for implementation of the so-called skew algorithm of Kärkkäinen and Sanders, [12]). The main reason was the relative simplicity of the resulting code relying on the recursive solution to the problem.

We first describe the working of the algorithm in general terms. For simplicity, a $s(i)$ denotes the suffix $x[i..n-1]$ of the string $x = x[0..n-1]$.

- The suffixes of $x = x[0..n-1]$ are divided into two groups. The first group SA^{12} consists of suffixes that start at a position i of the string so that $i \neq 0$ $(mod\ 3)$, i.e. $i/3 = 1$ or 2. The second group SA^0 consists of suffixes that start at a position i so that $i = 0$ $(mod\ 3)$, i.e. i is divisible by 3.

- The group SA^{12} is sorted by a recursion.

- The group SA^0 is sorted using the known order of suffixes in SA^{12} based on the following observation:
  if suffixes $s(i_1), s(i_2) \in SA^0$, then $s(i_1+1), s(i_2+1) \in SA^{12}$.
  So to compare $s(i_1)$ and $s(i_2)$, we first compare the letters $x[i_1]$ and $x[i_2]$, and if they are the same, then we compare $s(i_1+1)$ and $s(i_2+1)$. Thus a radix sort with key of length 2 will allow us to sort SA^0 in linear time.

- The two sorted groups SA^{12} and SA^0 are merged together. A simple comparison-based merge can be employed, based on the following observation:
  Let $s(i_1) \in SA^0$ and let $s(i_2) \in SA^{12}$. If suffix $i_2 = 1$ $(mod\ 3)$, then $s(i_1{+}1), s(i_2{+}1) \in SA^{12}$. If $i_2 = 2$ $(mod\ 3)$, then $s(i_1{+}2), s(i_2{+}2) \in SA^{12}$.
  Thus to compare $s(i_1) \in SA^0$ and $s(i_2) \in SA^{12}$ we follow two scenarios:
  (A) if $i_2 = 1$ $(mod\ 3)$, we compare $x[i_1]$ and $x[i_2]$, and if they are equal, we compare $s(i_1+1)$ and $s(i_2+1)$.
  (B) if $i_2 = 2$ $(mod\ 3)$, we compare $x[i_1]$ and $x[i_2]$, and if they are equal, we compare $x[i_1+1]$ and $x[i_2+1]$, and if they are equal, we compare $s(i_1+2)$ and (s_2+2).

  The important detail left out in the above high-level description is the

nature of the recursive call, as the input must be a string, not a set of suffixes SA^{12}. The trick is to create a string such that order of its suffixes is the same as the order of suffixes in SA^{12} ought to be. We shall illustrate the whole process on an example.

Assume a string $x = x[0..n-1]$. For each $i = 1$ or 2 (mod 3) we consider the substring of length 3 (i.e. $x[i..i+2]$). We refer to them as triples. We sort these triples lexicographically in linear time using radix sort with key of length 3. We assign each triple a symbol (the simplest is to use numbers from the interval [1,2n/3]) such that they respect the order of the triples. Figures 3.2 and 3.3 illustrate this for a string 103303330220.

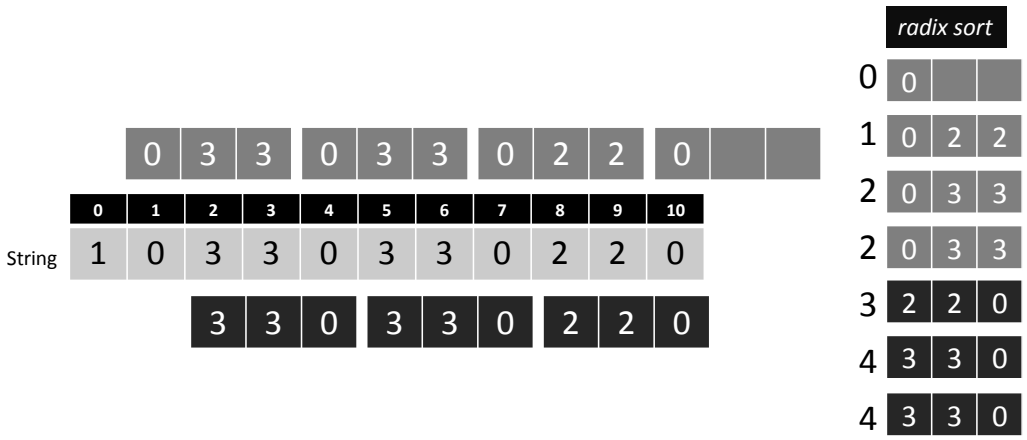

Figure 3.2: Create triples and sort them

The string created by symbols assigned to the suffixes starting at index = 1 (mod 3), concatenated with the string created by symbols assigned to the suffixes starting at index = 2 (mod 3) has the property that all SA^{12}

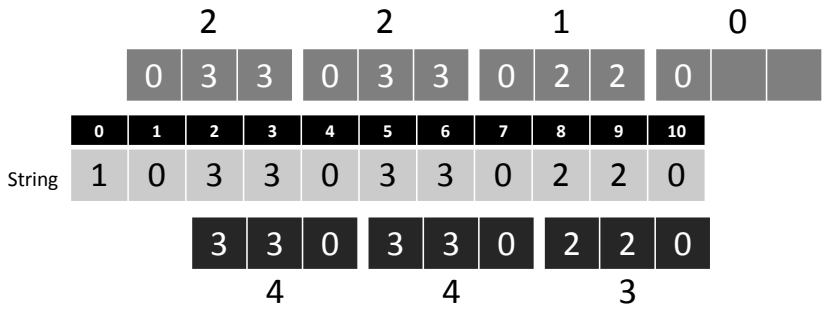

Figure 3.3: Assign names to the triples

suffixes have the same order as the corresponding suffixes of this new string. In Figure 3.3 we can see that this new string is 2210443 and in Figure 3.4 we see how from the suffix array of this new string we determine the order of SA^{12} of the original string.

The second step is sorting the suffixes SA^0 using the SA^{12} array. Here we use radix sort to sort it in linear time using keys of length 2, where the first element of the key is the first letter of the suffix and the second element of the key is the suffix that follows the first character.

The last step is merging the two suffix arrays. In Figure 3.5, both SA^{12} and SA^0 are sorted, and we will do a comparison merge for them. If we did the comparison by brute force, it would have to possibly go through both suffixes until different characters are found, which would lead to $O(n^2)$ complexity. However, as described above, we need keys of length at most 3 (for scenario A we need 2, for scenario B we need 3), and thus can use radix sort preserving the linearity.

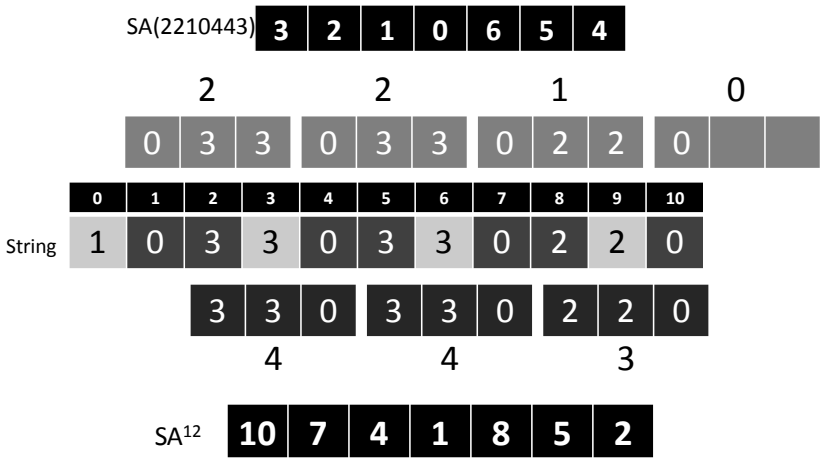

Figure 3.4: Getting SA^{12} from SA(10330330220)

Figure 3.6 gives us another example by using the string 0001000101. We put the indices = 1 (mod 3) in light gray color, 2 in dark gray, and 0 in white. When comparing $SA(S_0)$ with $SA(S_4)$ at the first step, we check the first character. Here both characters are the same, so we check if the order of $SA(S_1)$ and $SA(S_5)$ is already known. Since $SA(S_1)$ and $SA(S_5)$ are both belong to SA^{12}, we can just determine the order of $SA(S_0)$ and $SA(S_4)$ by their following substrings $SA(S_1)$ and $SA(S_5)$. Comparing $SA(S_2)$ and $SA(S_6)$ is similar. However, since the following substrings $SA(S_3)$ and $SA(S_7)$ is not determined yet ($SA(S_3)$ belongs to SA^0, we will look up the next substrings which are $SA(S_4)$ and $SA(S_8)$. Then we can find the result.

We conclude the situation of having the same first character by two cases.

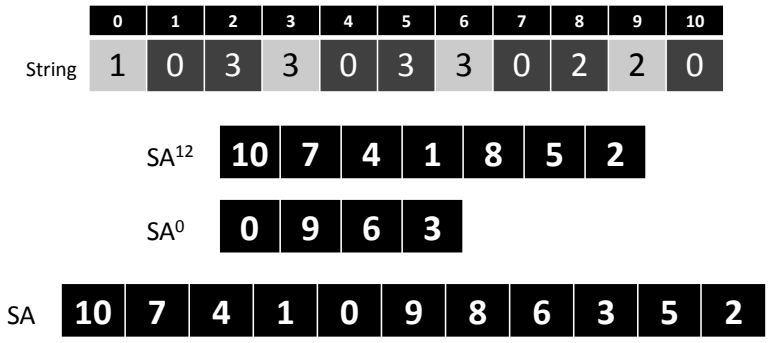

Figure 3.5: Merging SA^{12} and SA^0 arrays

First one starts at the white (SA^0) and the dark gray (SA^0) indices. In this case, we'll look up the next which starts at the light gray (SA^1) and the white (SA^0) ones, if they cannot be determined, then we check the next. The other case is the substrings start at white (SA^0) and the light gray (SA^1) indices. The result could be easily determined by the following light gray (SA^1) and dark gray (SA^2) ones.

After merging and tracing back level by level, we will get the suffix array for the original string. The running time of the skew algorithm is O(n) since the recurrence relation describing the number of steps is $T(n) = O(n) + T(\lceil 2n/3 \rceil)$, T(n)=O(1) for $n < 3$. Hence, $T(n) = O(n)$.

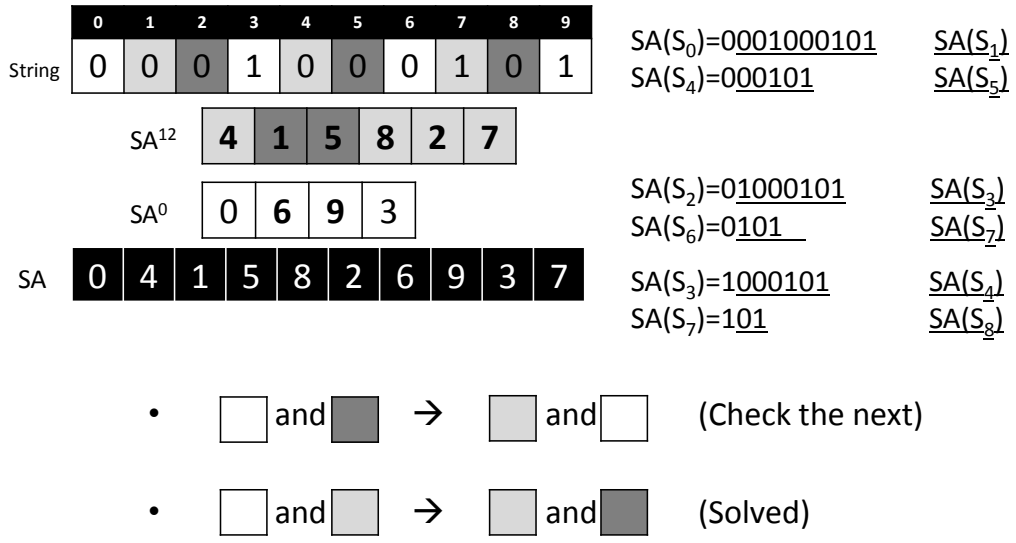

Figure 3.6: Merging SA^{12} and SA^0 arrays, another example

3.2 LCP array

For a given string $x = x[0..n-1]$, we define $lcp(i, j)$ as the length of the longest common prefix of the suffixes $s(i)$ and $s(j)$. Notice that it is a number from the range $0..n-1$. The *LCP array* of x is defined as $LCP[i] = lcp(SA[i-1].SA[i])$., where $SA[]$ is the suffix array of the string x.

Figure 3.7 shows an LCP array example: While calculating $LCP[8]$, we need to look into $SA[7]$ and $SA[8]$, which is 6 and 3. Then we will find the length of the longest common prefix starting from 6 and 3 is 2.

The direct approach to computing LCP array leads to an $O(n^2)$ algorithm, but it can be done in linear time using for instance an algorithm of Kasai, Lee,

	0	1	2	3	4	5	6	7	8	9	10
String	1	0	3	3	0	3	3	0	2	2	0
SA	10	7	4	1	0	9	8	6	3	5	2
LCP	0	1	1	4	0	0	1	0	2	1	3

Figure 3.7: The suffix array and longest common prefix array of the string "10330330220"

Arimura, Arikawa and Park, [11]. Its pseudocode is given in Algorithm 3 below.

They introduced a space efficient algorithm based on the simulation of the bottom-up traversal of a suffix tree. Let us remark that from a given suffix tree the corresponding suffix array can be determined simple from the order of the leafs by the depth-first traversal, while suffix array can give rise to the corresponding suffix tree using minimum range queries. A minimu range query $MRQ(i, j, A)$ returns the index k such that $i \leq k \leq j$ and $A[k]$ is the minimal value on the interval $i..j$ (i.e. $A[k] \leq A[t]$ for any $i \leq t \leq j$). Figure 3.8 shows a suffix array and Figure 3.9 shows the corresponding suffix tree.

We are going to illustrate the working of the algorithm. First we have to compute the intermediate array $Rank[]$ which is defined as the inverse function of the suffix array. That is, if $SA[k]$ is i, then $Rank[i]$ is k and vice-versa.

Then we observe two facts about the LCP arrays that can help us reduce the complexity. First, the *lcp* of a pair of adjacent suffixes in the suffix array is greater than any pair of the suffixes surrounding them. Since the suffixes are lexicographically sorted in the suffix array, the *lcp* between any two suffixes will

	0	1	2	3	4	5	6	7	8	9	10
String	1	0	3	3	0	3	3	0	2	2	0
SA	10	7	4	1	0	9	8	6	3	5	2

0	0	0	0	1	2	2	3	3	3	3
	2	3	3	0	0	2	0	0	0	3	3
	2	3	3	3	3	0		2	3	3	0
	0	0	3	3	3			2	3	0	3
		2	2	3	0			2	0	2	3
		0	2	3	3			0	2	2	0
			0	0	3				2	0	2
				3	3				0		2
				0	0						0
					2						
					2						
					0						

Figure 3.8: The suffix array example, the suffixes are listed vertically

be the minimum of the *lcp* of every adjacent suffix pair between the intervals. We can also consider it as an intersection of those pairs of adjacent suffixes.

Second, if $lcp(String_{SA[x-1]}, String_{SA[x]}) = h > 1$, then $lcp(String_{SA[x-1]+1}, String_{SA[x]+1}) = h - 1$. We can think that the substring starting at $SA[x] + 1$ is same as the substring starting at $SA[x]$ and remove the first element. Since $lcp(String_{SA[x-1]}, String_{SA[x]}) = h$ is greater than 1 and the rest of both substrings are the same, we can say the *lcp* between the substrings starting at $SA[x-1] + 1$ and $SA[x] + 1$ is 1 less than h.

From the above to facts we know that
if $lcp(String_{SA[i-1]}, String_{SA[i]}) = h > 1$, then $lcp(String_{SA[i-1]+1}, String_{SA[i]+1}) = h - 1$ and

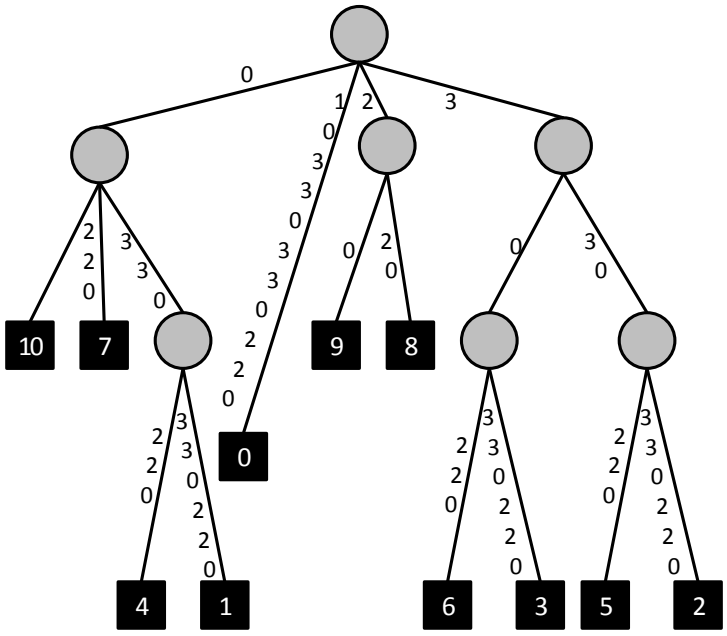

Figure 3.9: The suffix tree corresponding to the suffix array in Figure 3.8

$lcp(String_{SA[Rank[SA[i]+1]-1]}, String_{SA[i]+1}) \geq lcp(String_{SA[i-1]+1}, String_{SA[i]+1})$. Thus we get a formula:
$lcp(String_{SA[Rank[SA[i]+1]-1]}, String_{SA[i]+1}) \geq lcp(String_{SA[i-1]+1}, String_{SA[i]+1}) = lcp(String_{SA[i-1]}, String_{SA[i]}) - 1$.

Using this formula, once we get an *lcp* which is greater than 1, we can use it as a base to compute other substrings. The complexity will be $O(n)$ since h increases at most $2n$ times, and decreases at most n times. We show it in the pseudocode Algorithm 3.

For the algorithm, an LCP query function $LCPQuery(l, r)$ to represent the $lcp(l, r)$ function which we just talked about must be implemented. It

finds the minimum value between the index range from $l+1$ to r. That is: $LCPQuery(String_{SA[l]}, String_{SA[r]}) = \min_{l<k\leq r}\{lcp(String_{SA[k-1]}, String_{SA[k]})\}$.

Note that the $LCPQuery()$ is a form pf range minimum query. The range minimum query can be implemented with linear time preprocessing and constant time querying. However, since such algorithm is much more complicated an would increase the scope of this project too much, here we use as a temporary substitute a linear time algorithm to perform the queries. Replacing this component with a proper range minimum query is the task for near future.

The *longest common suffix array* (LCS) which is used in next step is easy to compute as the LCP array of the reversal of the original string.

Algorithm 3: Computing LCP array in linear time

```
LCP[0]=0, h=0
for i = 0 to Length − 1 do
  if Rank[i] > 0 then
    j=SA[Rank[i]-1]
    while String[i+h] = String[j+h] do
      h=h+1
    end while
    LCP[Rank[i]]=h
    if h > 0 then
      h=h-1
    end if
  end if
end for
```

3.3 Lempel-Ziv factorization

The Lempel-Ziv factorization of a string x is a sequence of substrings of x (often referred to as *factors*) $\langle u_1, ..., u_n\rangle$ so that

- $x = u_1 u_2 ... u_n$, x is concatenation of the factors
- each u_i is either a first occurrence of a letter (so-called new letter), or the longest previously occurring factor in $u_1 ... u_i$.

Introduced by Lempel and Ziv in 1978, see [17], for the purpose of data compression, it found also a place in the investigation of periodicities in strings. As mentioned previously, Lempel-Ziv factorization is the input into Main's algorithm and that is why it is of interest to us in this project.

The algorithm we implemented for Lempel-Ziv factorization is based on the Crochemore, Ilie and Smyth's paper [3]. The approach consists of two steps. First we compute the *longest previous factor* (LPF) array, then we use the LPF array to compute the Lempel-Ziv factorization. During the process of computing the LPF array, we also keep track of the previous occurrences (in $PrevO$) of the longest previous factor which will be used in Kolpakov and Kucherov's algorithm later.

The LPF construction is based on Crochemore and Ilie's paper [2]. The proper definition of the LPF array:
$LPF[i] = max(\{l \mid String[i...i+l-1]$ is a factor of $String[0...i+l-2]\} \cup \{0\})$

We also need two stacks to construct the *previous smaller value* (PSV) array and the *next smaller value* (NSV) array. The PSV array stores the closest index to the left of the suffix array which has a smaller suffix order, and the NSV array stores the closest index to the right of the suffix array which has a smaller suffix order. We put -1 on PSV/NSV array if we cannot find any smaller value to the left/right side.

In Figure 3.10, we can see an example of PSV, NSV, and LPF arrays. For example, $SA[7] = 6$, if we look backward (to the left) of the suffix array, the closest smaller value is $SA[4] = 0$, so we put 4 in $PSV[7]$. When looking forward (to the right), the closest smaller value is $SA[8] = 3$, so we put 8 in $NSV[8]$. The $LPF[5] = 3$ means the first 3 elements 330 of the substring 330220 (which starts at position 5 of the original string) has appeared before. Then we also store the index of the previous occurrence of 330 which is 2 in $PrevO[5]$.

The idea of the algorithm is quite simple. While computing the LPF value for the index x, we just look backward and forward to find the two nearest smaller values of the suffix array. Since the suffix array is lexicographically sorted, the substrings starting from these two smaller values will be the most similar substrings. We use $LCPQuery$ to compute the longest length of the common prefix between those two substrings and the substring starting at x, then choose the larger one and set it and its location as the LPF and $PrevO$ values. Algorithm 4 shows the pseudocode for the LPF and $PrevO$ array construction.

The PSV and NSV arrays can be computed in $O(n)$ time, provided we can implement the range minimum querying in constant time.

After we finish computing of the LPF and $PrevO$ arrays, we can use a simple algorithm by Crochemore, Ilie and Smyth to compute the Lempel-Ziv factorization [3, 2].

Figure 3.10 is an example of Lempel-Ziv factorization construction.

In order to make it easier to compute in the rest of the program, we use LZB array to represent the beginning indices of the LZ blocks, LZE array to

Algorithm 4: Computing LPF array and PrevO array

```
for i = 0 to LZLength − 1 do
  if PSV[i] = −1 then
    p=0
  else
    p=LCPQuery(SA[PSV[i]],SA[i])
  end if
  if NSV[i] = length then
    n=0
  else
    n=LCPQuery(SA[NSV[i]],SA[i])
  end if
  LPF[SA[i]]=MAX(p,n)
  if LPF[SA[i]] = 0 then
    PrevO[SA[i]]=-1
  else
    if p > n then
      PrevO[SA[i]]=SA[PSV[i]]
    else
      PrevO[SA[i]]=SA[NSV[i]]
    end if
  end if
end for
```

Algorithm 5: Lempel-Ziv factorization

```
LZ[0]=0
while LZ[i] < length do
  LZ[i+1]=LZ[i]+max(1,LPF[LZ[i]])
  i=i+1
end while
```

	0	1	2	3	4	5	6	7	8	9	10
String	1	0	3	3	0	3	3	0	2	2	0
SA	10	7	4	1	0	9	8	6	3	5	2
LCP	0	1	1	4	0	0	1	0	2	1	3
LPF	0	0	0	1	4	3	2	1	0	1	1
PSV	-1	-1	-1	-1	-1	4	4	4	4	8	4
NSV	1	2	3	4	-1	6	7	8	10	10	-1
PrevO	-1	-1	-1	2	1	2	3	4	-1	8	7
LZ	0	1	2	3	4	8	9	10	11		

Figure 3.10: The Lempel-Ziv factorization and the LPF array example

represent the ending indices of the LZ blocks (Figure 3.11), and $LZLength$ as the number of Lempel-Ziv blocks.

3.4 Main's algorithm

Main's algorithm uses the Lempel-Ziv factorization as its input, and computes some runs of the input string that are guaranteed to include all the leftmost runs.

To discuss the algorithm and its approach, we classify the runs into two disjoint categories:

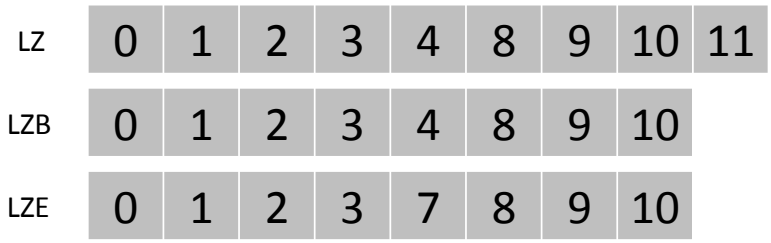

Figure 3.11: LZB and LZE arrays

Class I : The run occurs within one Lempel-Ziv block.

Class II : The run occurs across different Lempel-Ziv blocks.

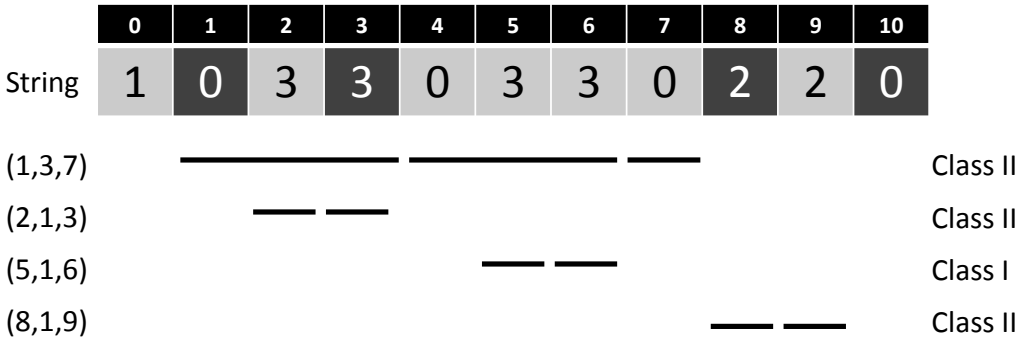

Figure 3.12: Class I and Class II runs

In Figure 3.12, we use different shading to show different Lempel-Ziv blocks. For example, the run (2,1,3) is across two Lempel-Ziv blocks, so it is a

Class II run. The run (5,1,6) is only bounded in one Lempel-Ziv block, so it's defined as a Class I run.

In our program, we use Main's algorithm to compute Class II runs. Let the string s be composed of many Lempel-Ziv blocks w, that is $s = w_0 w_1 ... w_k$. According to Main (see[13]), the Class II run from block w_{h-1} to the block w_h will be bounded in its length by $2|w_{h-1}w_h|$.

For each h between 2 to k, we let t_h be the substring which precedes w_h with length $2|w_{h-1}w_h|$ (if it reaches the beginning, then just start from the index 0). After the bounded substring $t_h w_h$ is found, we could calculate the rightmax and the leftmax runs which start in t_h and end in w_h. The pseudocode is shown in Algorithm 6. According to Main's paper, the runs in the string s can be computed in $\theta(n)$ time.

We store the Class II runs in a *LinkedList* array temporarily. We are going to sort the results, and to remove non-primitively rooted runs later.

3.5 Kolpakov and Kucherov's algorithm

The last step in our program is computing the Class I runs from the leftmost runs which we computed at the previous step. The algorithm we used was introduced by Kolpakov and Kucherov in [10]. For its pseudocode see Algorithm 7 below.

For every Lempel-Ziv blocks of length greater than 1, we get the offset from its previous occurrence. Within the current block, we read every run of the previous occurrence, and check if the ending index of the previous occurrence added the offset is still in the block itself. If it fulfills the condition, it is a new run of Class I.

Algorithm 6: Main's algorithm

```
for i = 0 to length − 1 do
  len=min(LZE[i]-LZB[i-1],LZE[i-1]+1)
  for l = 1 to len do
    if LZE[i − 1] − l < 0 then
      s=0
    else
      s=LCSQuery(LZE[i-1]-l,LZE[i-1])
    end if
    p=LCPQuery(LZB[i]-l,LZB[i])
    if s + p >= l AND (p > 0 OR LZB[i] − l − s > LZB[i − 1]) then
      add run (LZB[i]-l-s,l,LZE[i-1]+p) to LinkedList[LZE[i-1]+p]
    end if
  end for
  for l = 1 to LZE[i] − LZB[i] do
    s=LCSQuery(LZE[i-1]+l,LZE[i-1])
    p=LCPQuery(LZB[i]+l,LZB[i])
    if s + p >= l AND LZE[i − 1] + l + p <= LZE[i] AND s < l then
      add run (LZB[i]-s,l,LZE[i-1]+l+p) to LinkedList[LZE[i-1]+p]
    end if
  end for
end for
```

Algorithm 7: Kolpakov and Kucherov's algorithm

```
for h = 1 to LZLength − 1 do
  if | LZ[h] |> 1 then
    delta=LZB[h] - PrevO[LZB[h]]
    for i = LZB[h] to LZB[h + 1] − 1 do
      for all j ∈ LinkedList[i − delta] do
        if (j.e + delta) < LZB[h + 1] then
          add run (i, j.e+delta, j.l) to LinkedList[i]
        end if
      end for
    end for
  end if
end for
```

This algorithm might find some runs which can be extended outside the block (i.e. Class II), which were already computed by Main's algorithm. We need to look forward and backward one element to check if the run could be extended. In figure 3.13, we can see the example. The Class I run (4,1,5) could be extended to (4,1,6) which is a Class II run. As we look forward to the next element and checking it, we can avoid the duplication.

Figure 3.14 shows the steps of storing the runs. In order to store the results of runs, we implement two *LinkedList* arrays of length n.

First we use *LinkedList* array $Run1[]$ to store the results of Main's algorithm. Then we classify them by the start index and put them to *LinkedList* array $Run2[]$. For example, we put the run (4,1,5) to $Run2[4]$. After that, we use *LinkedList* array $Run2[]$ to find other runs in Kolpakov and Kucherov's algorithm. Finally, we move them back to *LinkedList* array $Run1[]$ with checking if they are primitively rooted. Since Main's algorithm not only calculates the primitively rooted runs but also the non-primitively rooted ones (thus technically speaking not really runs), we use *LinkedList* array $Run2[]$ to store the original result, then we identify if it is a primitively rooted run, and put it to *LinkedList* array $Run1[]$. In this part, we use a temporary array $end[]$ and a stack $s3$ to save the space and make it in linear time.

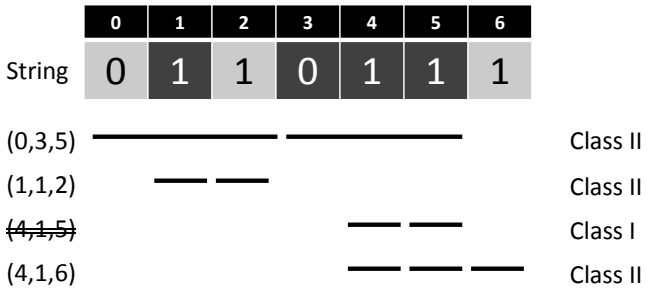

Figure 3.13: Remove the duplication with Class II runs

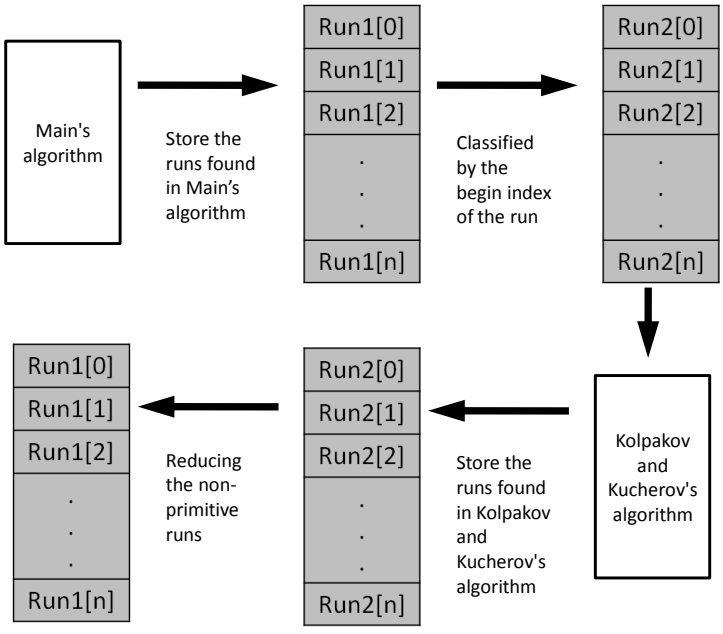

Figure 3.14: The steps of storing runs in Run1[] and Run2[]

Chapter 4

Conclusion

We presented two C++ implementations of algorithms computing primitively rooted runs in a string. The first, FJW, due to Franek, Jiang, and Weng is an extension of Crochemore's repetitions algorithm and also computes the number of primitively rooted distinct squares and, of course, primitively rooted maximal repetitions. The complexity of the algorithm is $O(n \log n)$ where n is the length of the input string. Franek and Jiang introduced three algorithms in 2009 that compute runs. However, Franek, Jiang and Weng's algorithm is faster and more memory efficient requiring $13n$ integers of memory in comparison to the Franek and Jiang algorithms requiring $O(n \log n)$ integers of memory, or Franke, Smyth, and Xiao most memory efficient implementation of Crochemore's algorithm requiring $14n$ integers.

The second one is an implementation of a linear time algorithm to compute runs, based on Main, Kolpakov and Kucherov's algorithms. Main's algorithm computes every Class II runs which across different Lempel-Ziv blocks and some other runs. Kolpakov and Kucherov's algorithm uses the result of Main's algorithm to find all the other runs. In order to provide the essential

data for these two algorithms, we computes the suffix array, the longest common prefix array, the Lempel-Ziv factorization in linear time. This implementation uses some *LinkedList* arrays to store the runs.

In the future work, before mutually benchmarking and comparisons of these two algorithms, the proper constant time range minimum querying must be implemented. With many succinct data structures, the constant time of querying could be achieved by heavy – though linear – preprocessing. Moreover, the whole implementation can be optimized as far as memory requirements are concerned by the technique of "allocation from arena" which helps significantly the efficiency of dynamic memory allocation, [4]. After these modifications are done, we will have a true linear time implementation to compare with the FJW.

Bibliography

[1] M. Crochemore: *An optimal algorithm for computing the repetitions in a word*, Inform. Process. Lett. 5 (5) 1981, pp. 297–315

[2] M. Crochemore and L. Ilie *Computing longest previous factor in linear time and applications* Inform. Proc. Lett. 106 (2008) 75 - 80

[3] M. Crochemore, L. Ilie, and W.F. Smyth, *A simple algorithm for computing the Lempel-Ziv factorization*, Proc. Data Compression Conference (dcc 2008), 2008, pp.482–488

[4] F. Franek *Memory as a programming concept in C and C++* Cambridge (2004)

[5] F. Franek and M. Jiang, *Crochemore's repetitions algorithm revisited - computing runs*, to appear in Int. Jour. of Foundations of Comp. Sci. (2009)

[6] F. Franek and M. Jiang, *Crochemore's repetitions algorithm revisited - computing runs*, AdvOL Report 2009/01, Advanced Optimization Laboratory, Dept. of Comp. and Software, McMaster University, Hamilton, Ontario (2009)

[7] F. Franek, W.F. Smyth, and X. Xiao: *A note on Crochemore's repetitions algorithm, a fast space-efficient approach*, Nordic J. Computing 10-1 (2003), pp. 21–28

[8] D. Kim, J. Sim, H. Park, and K. Park *Linear-time construction of suffix arrays* In Proceedings of the 14th annual conference on Combinatorial pattern matching (CPM'03), Ricardo Baeza-Yates, Edgar Chávez, and Maxime Crochemore (Eds.). Springer-Verlag, Berlin, Heidelberg, 186-199.

[9] P. Ko and S. Aluru *Space efficient linear time construction of suffix arrays* In Combinatorial Pattern Matching (CPM 03). LNCS 2676, Springer (2003), pp 203210.

[10] R. Kolpakov and G. Kucherov, *Finding maximal repetitions in a word in linear time*, Proc. 1999 Symposium on Foundations of Computer Science (FOCS'99), New York (USA), pp. 596–604. IEEE Computer Society, New-York, October 17-19 1999.

[11] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park., *Linear-time longest-common-prefix computation in suffix arrays and its applications.*, Proc. 12th Symposium on Combinatorial Pattern Matching (CPM 01), pp. 181–192., Springer-Verlag LNCS n. 2089 (2001).

[12] J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, Proc. 30th Internat. Colloq. Automata, Languages & Programming (2003), pp. 943–955

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

[14] U. Manber and G. Myers *Suffix arrays: a new method for on-line string searches* SIAM Journal on Computing, Volume 22, Issue 5 (October 1993), pp. 935948.

[15] G. Nong, S. Zhang, and W. Chan *Linear suffix array construction by almost pure induced-sorting* In Proceedings of the 2009 Data Compression Conference (DCC '09). IEEE Computer Society, Washington, DC, USA, 193-202. DOI=10.1109/DCC.2009.42 http://dx.doi.org/10.1109/DCC.2009.42

[16] W. Smyth *Computing patterns in strings* Addison-Wesley Pearson (2003)

[17] J. Ziv and A. Lempel *Compression of individual sequences via variable-rate coding* IEEE Transactions on Information Theory, September 1978.

[18] `http://www.cas.mcmaster.ca/~franek`

[19] `http://www.cas.mcmaster.ca/~wengc2`

[20] `http://www.bio.ifi.lmu.de/~fischer`